



Programmation Python

Une version à jour et éditable de ce livre est disponible sur
Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
http://fr.wikibooks.org/wiki/Programmation_Python

Vous avez la permission de copier, distribuer et/ou

modifier ce document selon les termes de la Licence de documentation libre GNU, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans Texte de dernière page de couverture. Une copie de cette licence est incluse dans l'annexe nommée « Licence de documentation libre GNU ».

Avant-propos

Grace Hopper, inventeur du compilateur :

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

Choix d'un premier langage de programmation

Ce livre est censé vous permettre d'apprendre à programmer à partir du langage Python. Pourquoi Python ? Il existe en effet plusieurs centaines de langages de programmation, chacun avec ses avantages et ses inconvénients. Parmi ces derniers, on compte :

- Ils sont liés à des environnements de programmation (c'est-à-dire des logiciels) propriétaires.

Cela signifie donc, non seulement que l'institution scolaire désireuse de les utiliser devrait acheter une licence de ces logiciels pour chaque poste de travail (ce qui risque de se révéler assez coûteux), mais surtout que les utilisateurs souhaitant exercer leurs compétences de programmation ailleurs qu'à l'école seraient implicitement forcés d'en acquérir eux aussi des licences.

- Ce sont des langages spécifiquement liés au seul système d'exploitation Windows. Ils ne sont pas « portables » sur d'autres systèmes (Unix, MacOS, etc.). Cela ne cadre pas avec un projet pédagogique qui ambitionne d'inculquer une formation générale (et donc diversifiée) dans laquelle les invariants de l'informatique seraient autant que possible mis en évidence.

Nous avons alors décidé d'examiner l'offre alternative, c'est-à-dire celle qui est proposée gratuitement dans la mouvance de l'informatique libre (open source). 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 bon niveau de qualité technique. Le plus célèbre des logiciels libres est le système d'exploitation GNU/Linux, dont la popularité ne cesse de s'accroître de jour en jour.

Ce que nous avons trouvé nous a enthousiasmé : non seulement il existe dans le monde de l'open source des interpréteurs et des compilateurs gratuits pour toute une série de langages, mais le véritable cadeau réside dans le fait que ces langages sont modernes, performants, portables (c'est-à-dire utilisables sur différents systèmes d'exploitation tels que Windows, Linux, ou MacOS), et fort bien documentés.

Le langage dominant y est sans conteste *C*. Ce langage s'impose comme une

référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante. La mise au point d'un gros logiciel écrit en langage *C* est longue et pénible. (Les mêmes remarques valent aussi dans une large mesure pour le langage *Java*).

Pour nos débuts dans l'étude de la programmation, il nous semble préférable d'utiliser un langage de plus haut niveau, c'est-à-dire plus proche du langage humain que du langage machine, donc plus lisible et moins contraignant.

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.

La démarche du programmeur

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

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

Remarque : 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 très grandes satisfactions.

Langage machine, langage de programmation

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

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

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les "bits") souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64. Ce « langage machine » est évidemment presque 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).

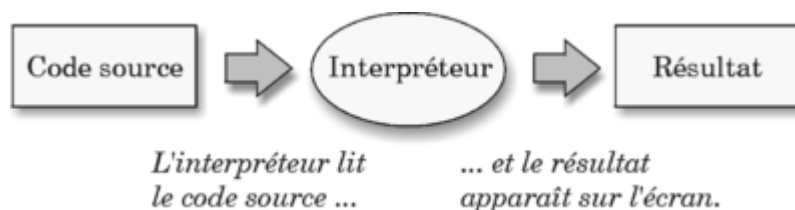
Suivant son niveau d'abstraction, on pourra dire d'un langage qu'il est « de bas niveau » (ex : *Assembleur*) ou « de haut niveau » (ex : *Pascal*, *Perl*, *Smalltalk*, *Clarion*, *Java*...). Un langage de bas niveau est constitué d'instructions très élémentaires, très « proches de la machine ». Un langage 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'est-à-dire l'apport de modifications ultérieures) et la recherche des erreurs (les « bugs ») sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent *portable*, c'est-à-dire que l'on pourra le faire fonctionner sans grosses modifications sur des machines ou des systèmes d'exploitation différents. Un programme écrit dans un langage de bas niveau ne peut jamais fonctionner que sur un seul type de machine : pour qu'une autre l'accepte, il faut le réécrire entièrement.

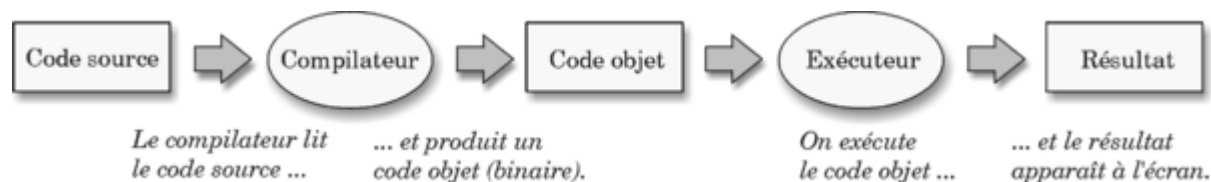
Compilation et interprétation

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

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



- 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 »).



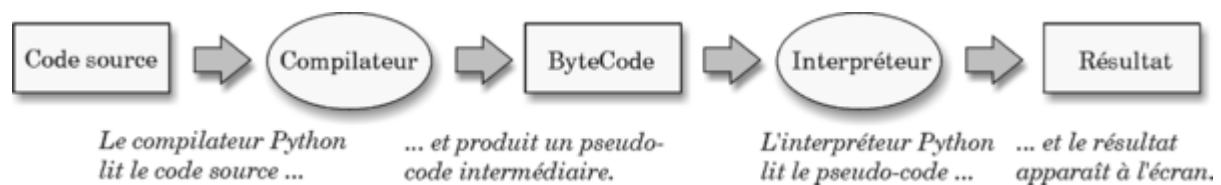
Chacune de ces deux techniques a ses avantages et ses inconvénients :

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 *bytecode*, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le bytecode est très facile à interpréter en langage machine. Cette interprétation sera donc beaucoup plus rapide que celle d'un code source.



Les avantages de cette méthode sont appréciables :

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

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

Mise au point d'un programme - Recherche des erreurs (« debug »)

La programmation est une démarche très complexe, et comme c'est le cas dans toute activité humaine, on y commet de nombreuses erreurs. Pour des raisons anecdotiques, les erreurs de programmation s'appellent des « *bugs* » (ou « bogues », en France)^[1], et l'ensemble des techniques que l'on met en œuvre pour les détecter et les corriger s'appelle *debug* (ou déboguage).

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

Erreurs de syntaxe

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

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

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

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

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

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

Erreurs sémantiques

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

En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le

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

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

Erreurs à l'exécution

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

Recherche des erreurs et expérimentation

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

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 fautive, alors il vous faut émettre une nouvelle hypothèse. Comme l'a bien dit Sherlock Holmes : « Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité » (A. Conan Doyle, *Le signe des quatre*).

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

mesure les erreurs, afin d'avoir de toute façon à chaque étape du processus un programme qui fonctionne.

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

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

Diverses méthodes existent pour effectuer cette tâche d'*analyse*, mais leur étude sort du cadre de ce livre.

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.

Remarque : les langages de programmation sont des langages formels qui ont été développés pour décrire des algorithmes et des structures de données. On appelle algorithme la séquence détaillée de toutes les opérations à effectuer pour résoudre un problème

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 $Zq3G2$.

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 ».

Habitué 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 maîtresses.

Un texte en prose

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

Un programme d'ordinateur

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

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

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

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

Notes

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

Introduction

Définition du langage

Python est un langage de script de haut niveau, structuré et open source. Il est multi-paradigme et multi-usage.

Développé à l'origine par Guido van Rossum en 1989^[1], il est, comme la plupart des applications et outils open source, maintenu par une équipe de développeurs un peu partout dans le monde.

Conçu pour être orienté objet, il n'en dispose pas moins d'outils permettant de se livrer à la programmation fonctionnelle ou impérative ; c'est d'ailleurs une des raisons qui lui vaut son appellation de « langage agile ».

Parmi les autres raisons, citons la rapidité de développement (qualité propre aux langages interprétés), la grande quantité de modules fournis dans la distribution de base ainsi que le nombre d'interfaces disponibles avec des bibliothèques écrites en C, C++ ou Fortran. Il est également apprécié pour la clarté de sa syntaxe, ce qui l'oppose au langage Perl.

Caractéristiques du langage

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

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

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

Utilisation

Comme mentionné plus haut, Python se prête à un grand nombre de tâches. La distribution de base permet, entre autre, des développements réseau, la création d'interfaces graphiques (via tcl/tk), de la programmation cgi, de traiter du XML, etc... Sa relative facilité d'interfaçage avec des bibliothèques écrites en d'autres langages en fait un outil de choix pour des applications de calcul scientifique. Il est également de plus en plus utilisé comme langage de prototypage.

Python est aussi remarquable pour le nombre de bibliothèques accessibles via l'installation des modules appropriés. Que ce soit la connexion avec une base de donnée, l'utilisation de bibliothèques d'interface graphique (wxPython, PyQt, pyGTK), la manipulation avancée de XML (pyXML), le traitement d'image (Python Imaging Library), le développement de jeu vidéo (pygame), OpenGL, la grande majorité des technologies actuelles dispose de son extension python.

Quelques exemples d'utilisation de Python

- Le serveur d'application Zope
- Administration du moteur de recherche Google
- Administration des fermes de rendu de la société d'effets spéciaux ILM
- l'application de dessin vectoriel Skencil (anciennement Sketch)
- Boa constructor, outil de développement rapide d'applications wxPython<

D'autres exemples sont disponibles sur Python success stories (anglais)

Version 2 et version 3

Il est à noter que les deux principales familles de python sont la famille version 2 et la famille version 3.

Lorsqu'il est nécessaire de modifier des anciens code, ou d'utiliser des anciennes librairies, il peut être utile d'apprendre la version 2.

Mais la version 3 devant à terme remplacer la version 2, il peut être d'ores et déjà utile de commencer à apprendre la version 3.

Installation

Exécution sans installation

Certains sites tels que <http://pythonfiddle.com/> permettent d'exécuter du code Python sur un serveur distant, sans avoir à l'installer chez soi. Toutefois pour développer des applications dignes de ce nom, il convient de l'avoir à disposition sur sa machine.

Installation locale

Il est possible d'installer Python sur la plupart des systèmes d'exploitation en le téléchargeant sur <http://www.python.org/download>.

Si vous n'avez pas les droits d'administrateurs de votre machine, il existe aussi une version portable sur <http://www.portablepython.com>.

Pour les systèmes Windows, vous pouvez télécharger le fichier MSI. Python est le plus souvent automatiquement installé avec la plupart des distributions Linux ou Mac OS, mais il est recommandé (nécessaire) de télécharger une version actualisée :

Linux

Vous êtes probablement chanceux et l'interpréteur Python est déjà installé sur votre machine. pour le tester, tapez `python3` dans un terminal. Si vous voyez quelque chose comme dans la prochaine section, vous êtes bon.

Si vous devez installer Python, essayez d'abord d'utiliser le gestionnaire de paquet de votre système d'exploitation ou allez visiter le dépôt où ces paquets sont disponibles, et récupérez Python 3. Python 3.0 est sorti en décembre 2008 ; toutes les distributions (BSD, GNU/Linux-Hurd-Linux Libre)devraient avoir Python 3 disponible, donc vous ne devriez pas avoir besoin de compiler Python 3 "from scratch" après avoir téléchargé le code source. Debian et Fedora possèdent des paquets binaires Python3 disponibles, mais ne le sont pas encore pas défaut, il faudra donc les installer par vous-même.

Voici donc les étapes pour compiler Python sur une unixerie (GNU/Linux, BSD...) :

- Téléchargez le fichier d'archive contenant les sources avec `wget|curl -O|fetch http://www.python.org/ftp/python/3.3.3/Python-3.3.3.tar.xz` (utilisez la commande de votre choix dans votre terminal)
- Décompressez le fichier tar (mettez-le au préalable dans le dossier voulu) :

```
$ tar -xJvf Python-3.3.3.tar.xz
... liste des fichiers décompressés
```


- Déplacez-vous dans le dossier nouvellement créé et dites à votre système de compiler et installer le programme

```

$ cd Python-3.3.3/
$ ./configure
... plein de messages, prêtez attention aux erreurs ...
$ make
... encore plus de messages...
$ make install
    
```

Les commandes ci-dessus installeront Python 3 dans votre dossier `/usr/local`. Si vous voulez utiliser IDLE, vous devez vous assurer que tk et tcl ainsi que leurs fichiers de développement sont installés sur le système. Vous verrez des avertissements durant la phase de compilation (make) si ces derniers ne sont pas disponibles.

Variable d'environnement :

Exécuter un script python

```

> set PATH=%PATH%;C:\<Repertoire de python>
> python MonScript.py
    
```

Utilisateurs de Mac

Depuis Mac OS X (Tiger), Python est embarqué par défaut avec l'OS, mais vous devrez mettre à jour vers Python 3 de la manière décrite ci-dessus, jusqu'à ce qu'OS X incluse Python 3 (vérifiez la version en tapant `python3` dans un terminal).

Remarque : depuis l'apparition de la version 2.3, il est vivement recommandé aux francophones que nous sommes d'inclure l'un des pseudo-commentaires suivant au début de tous nos scripts Python (à la 1e ou 2e ligne) :

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

Ou encore mieux :

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

Mode Interactif

Python dispose d'une interface interactive appelée IDLE, qui permet de tester les commandes de base. Pour appeler l'interface, il suffit de saisir `python` en

ligne de commande (dans un « shell » *Linux*, ou bien dans une fenêtre DOS sous Windows) .

Si vous utilisez une interface graphique telle que *Windows*, *Gnome*, *WindowMaker* ou *KDE*, vous préférerez vraisemblablement travailler dans une « fenêtre de terminal », ou encore dans un environnement de travail spécialisé tel que IDLE. Voici par exemple ce qui apparaît dans une fenêtre de terminal KDE (sous Linux)^[2] :

```

$ python
Python 2.7.12 (default,
[GCC 5.4.0 20160609] on
Type "help", "copyright"
>>> print("Hello World")
Hello World
>>> exit()
$
    
```

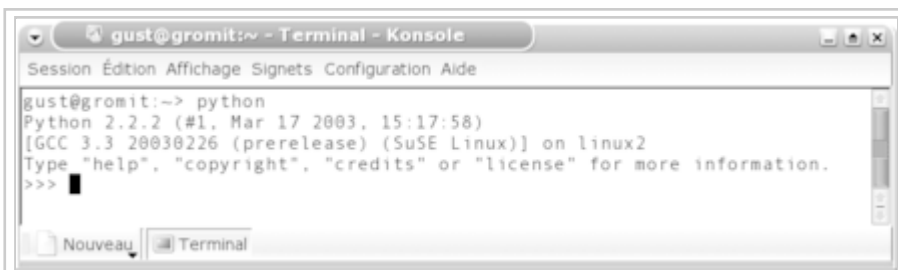
```

Python 3.0 (r30:67503, Dec 29 2008, 21:31:07)
[GCC 4.3.2 20081105 (Red Hat 4.3.2-7)] on linux2
Type "copyright", "credits" or "license()" for more information.

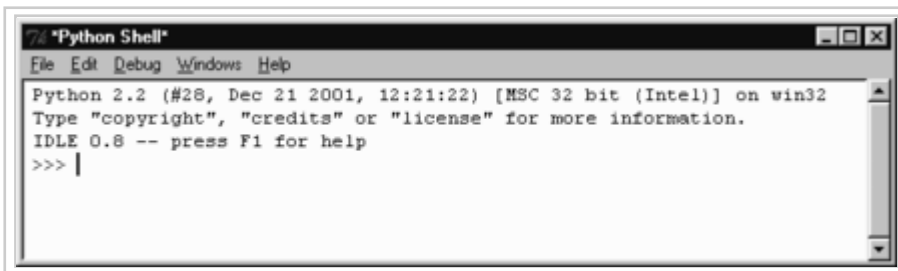
*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 3.0
>>>
    
```

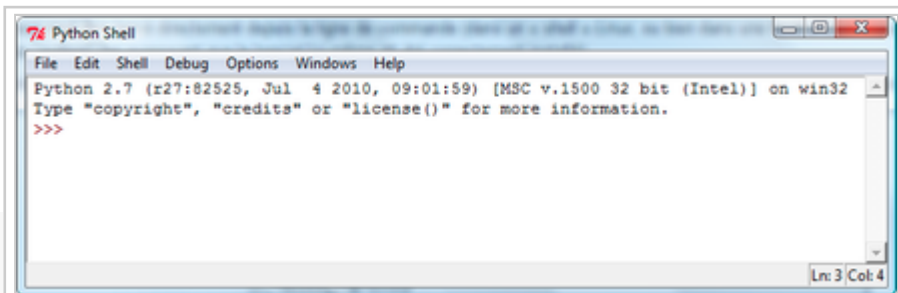
Les trois caractères « >>> » constituent le signal d'invite, ou *prompt principal*, lequel vous indique que Python est prêt à exécuter une commande. En mode interactif, ce que vous tapez est immédiatement exécuté. Mais le plus employé par les programmeurs est d'exécuter un script Python contenant plusieurs



IDLE sous KDE



IDLE 0.8 sous Windows 2000



IDLE sous Windows 7

instructions, en lançant un fichier .py sauvegardé sur le disque dur.

Calculer avec Python

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 :

```
>>> 5+3          # 8
>>> 2 - 9        # -5 : les espaces sont optionnels
>>> 7 + 3 * 4    # 19 : la hiérarchie des opérations mathématiques est bien respectée
>>> (7+3)*4     # 40
>>> 20 / 3      # 6 : arrondi
```

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, mais dans les versions de python 3.* la division n'est plus tronquée : 5/2 donne bien 2.5), comme dans le dernier exemple ci-dessus. Si vous voulez qu'un argument soit compris par Python comme étant un nombre réel, il faut le lui faire savoir, en fournissant au moins un point décimal^[3].

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
```

Aide

Lorsque vous souhaitez obtenir de l'aide sur une fonction ou une librairie, il suffit d'utiliser la commande « help » dans l'interpréteur interactif. Pour savoir comment fonctionnent les expressions régulières sous python par exemple, exécutez

Obtenir de l'aide sur une librairie python

```
> import re
> help(re)
```

pip

pip est le gestionnaire de paquets fourni avec Python. Pour l'utiliser :

```
pip install nom-du-paquet
```

Références

1. <http://python-history.blogspot.fr/2009/01/brief-timeline-of-python.html>
2. Sous *Windows*, vous aurez surtout le choix entre l'environnement *IDLE* développé par Guido Van Rossum, auquel nous donnons nous-même la préférence, et *PythonWin*, une interface de développement développée par Mark Hammond. D'autres environnements de travail plus sophistiqués existent aussi, tels l'excellent *Boa Constructor* par exemple (qui fonctionne de façon très similaire à *Delphi*), mais nous estimons qu'ils ne conviennent guère aux débutants. Pour tout renseignement complémentaire, veuillez consulter le site Web de Python. Sous *Linux*, nous préférons personnellement travailler dans l'environnement graphique *WindowMaker* (plutôt que *KDE* ou *Gnome* trop gourmands en ressources), en ouvrant une simple fenêtre de terminal pour lancer l'interpréteur Python ou l'exécution des scripts, et en faisant appel à un logiciel tel que *Nedit* ou *SciTE* pour l'édition de ces derniers.
3. Dans les langages de programmation, les notations numériques de base sont souvent celles des pays anglophones : le séparateur décimal est le point, et non une virgule comme en français. 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* ».

Éditeurs

Utilité des éditeurs

Si le mode interactif a permis d'apprendre très rapidement les bases du langage, par expérimentation directe, cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions écrites disparaissent irrémédiablement après fermeture de l'interpréteur. Il convient donc de sauvegarder les programmes dans des fichiers, sur une mémoire morte, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, il est déconseillé d'utiliser de simples éditeurs de texte tels que Edit, Notepad ou WordPad, car ils ne sont pas ANSI "intelligent", ni muni de la fonction de coloration syntaxique du code source pour Python, qui aide à relire rapidement et éviter les fautes de syntaxe.

Par conséquent, il vaut mieux en utiliser des spécialisés en programmation, appelés environnement de développement (EDI). Par exemple :

- SciTE
- DrPython
- Eclipse
- NetBeans^[1]
- NEdit
- PythonWin^[2]
- Python scripter^[3] : auto-complétion, navigation entre classe avec 'CTRL', génération et exécution des tests unitaires, debugger...

```

/dos_data/python/essais/fibo1.py DOS 486 bytes
# 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.

print "Suite de Fibonacci :"

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
    
```

NEdit sous *Gnome* (Linux).

SciTE

SciTE (acronyme de *Scintilla Text Editor*) est un logiciel éditeur de texte graphique, gratuit et *open source* fonctionnant sous les environnements Linux et Windows. Il est capable d'effectuer la coloration syntaxique, l'auto-complétion et surtout le pliage de code (**code folding**), c'est à dire le masquage à volonté de différents blocs d'instructions (contenu d'une classe,

d'une fonction, d'une boucle, etc.) : cette fonctionnalité se révèle extrêmement pratique lorsque vos scripts commencent à s'allonger... Il intègre également une fenêtre de terminal ainsi qu'un raccourci pour lancement des scripts.

Cet éditeur est disponible pour Windows et pour Linux sur <http://www.scintilla.org/SciTE.html>.

Scintilla étant une plateforme d'édition de texte qui propose par exemple des outils spécifiques pour corriger du code écrit avec SciTE. Ces deux logiciels sont principalement l'œuvre de leur créateur Neil Hodgson, qui les a placés sous une licence libre peu connue, l'*Historical Permission Notice and Disclaimer*.

Langages supportés

Les langages compris par SciTE. Il est actuellement capable d'appliquer une syntaxe de style à ces langages :

Liste
<ul style="list-style-type: none">■ Ada■ ANS.1 MIB definition files*■ APDL■ Assembler (NASM, MASM)■ AutoIt*■ Avenue*■ Batch files (MS-DOS)■ Baan*■ Bash*■ BlitzBasic*■ Bullant*■ C/C++/C#*■ Clarion*■ conf (Apache)*■ CSound*■ CSS*■ diff files*■ E-Script*■ Eiffel*■ Erlang*■ Flagship (Clipper / XBase)*■ Flash (ActionScript)*■ Fortran*■ Forth*■ Haskell■ HTML*■ HTML with embedded JavaScript, VBScript, PHP and ASP*■ Gui4Cli*■ IDL - both MSIDL and XPIDL*■ INI, properties* and similar

- InnoSetup*
- Java*
- JavaScript*
- LISP*
- LOT*
- Lout*
- Lua*
- Make
- Matlab*
- Metapost
- MMIXAL
- MSSQL
- nnCron
- NSIS*
- Objective Caml*
- Opal
- Octave*
- Pascal/Delphi*
- Perl, most of it except for some ambiguous cases*
- PostScript*
- POV-Ray*
- PowerBasic*
- PureBasic*
- Python*
- Rebol*
- Ruby*
- Scheme*
- scriptol*
- Specman E*
- Spice
- Smalltalk
- SQL and PLSQL
- TADS3*
- TeX and LaTeX
- Tcl/Tk*
- VB and VBScript*
- Verilog*
- VHDL*
- XML*
- YAML*

Les langages marqués par un astérisque doivent être vérifiés pour marcher pour un nombre important de personnes.

Installation sous Linux

L'éditeur Scintilla fait dorénavant partie des paquetages fournis d'office avec les distributions récentes de Linux. Sinon, téléchargez-le au départ du site web mentionné ci-dessus. Sinon :

- téléchargez l'archive *gscite***.tgz* puis l'extraire avec tar ;

- installez l'exécutable SciTE dans */usr/local/bin* ;
- installez tout le reste (fichiers **.properties*) dans */usr/share/scite* (et non */usr/share/gscite* !).

Installation sous Windows

- Téléchargez l'archive *wscite***.zip* puis l'extraire dans *\Program files* ;
- installez une icône de lancement pour l'exécutable *SciTe.exe*.

Pour les deux versions

On peut personnaliser beaucoup de choses (polices, etc.) en éditant le fichier des propriétés globales (Menu *Options* → *Open global options file*).

Par exemple, pour activer de jolis symboles pour replier/déplier, dans la marge de gauche :

- `fold.symbols = 2` # pour de belles icônes + et - cerclées
- `fold.on.open = 1` # ainsi tout est plié au départ
- `margin.width =0` # pour supprimer la marge inutile

Pour forcer le remplacement automatique des tabulations par des groupes de 4 espaces :

- `tabsize = 4`
- `indent.size = 4`
- `use.tabs = 0`

Traductions

La traduction doit être téléchargée^[4], renommée en "locale.properties" et déplacée dans le même répertoire que les propriétés globales.

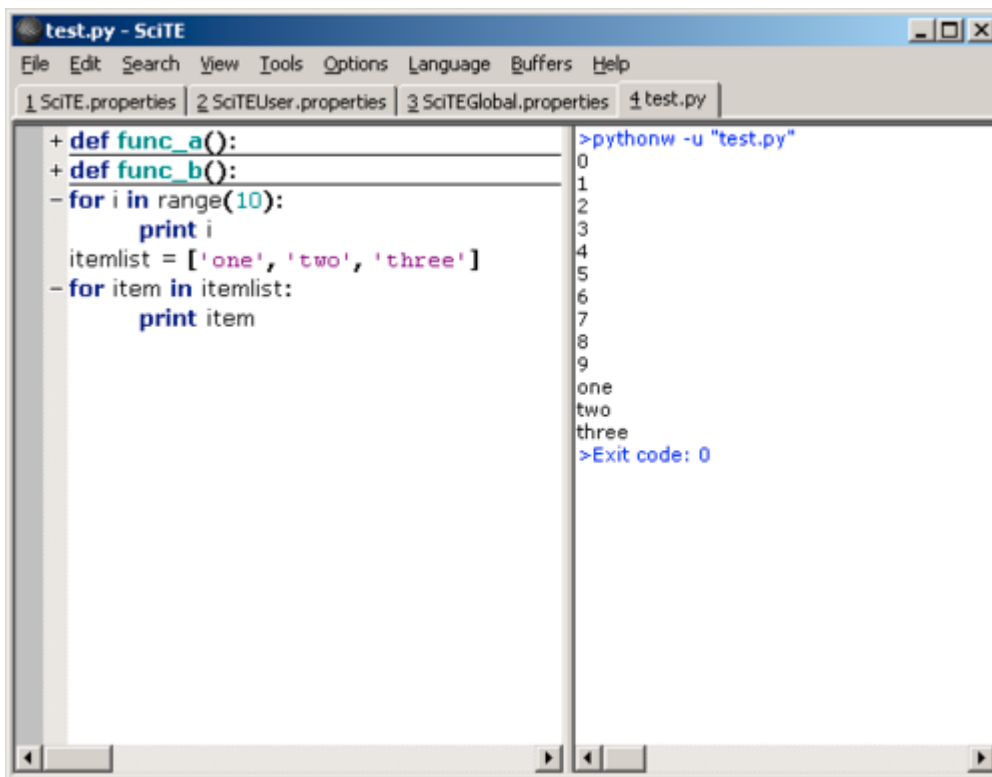
Il faut donc passer en root :

```
sudo nautilus
```

et aller dans le répertoire par défaut :

```
/usr/share/scite
```

Utilisation



DrPython

DrPython est un environnement d'exploration pour le langage Python. (Il y a des autres logiciels comme ça pour les autres langues, par exemple DrJava et DrScheme.) DrPython est un logiciel Python, donc il faut installer Python d'abord.

Téléchargez les suivantes :

- Python soi-même : du *site web Python*. L'installateur est disponible en cliquant "Windows installer", ou *ici* pour Python 2.7.
- La version de bibliothèque *wxPython* correspondant à la version de Python -- ça veut dire wxPython pour Python 2.7. En générale, *c'est ici*.
- *DrPython*; télécharger seulement le paquet du site Sourceforge *ici*.

Maintenant il faut installer Python et wxPython. Pour le faire, double-cliquez sur les fichiers vous avez téléchargé, commençant avec *python-2.7.msi*. Les installations sont en anglais. Il suffit de cliquer sur «Next» plusieurs fois, mais je vais aussi essayer de traduire les écrans.

Python 2.7

Anglais	Français
----------------	-----------------

<p>Select whether to install Python 2.7 for all users of this computer. Install for all users/Install for just me (not available on Windows Vista)</p>	<p>Choisissez à installer Python 2.7 pour tout utilisateur du ordinateur. Installer pour tout utilisateur/Installer pour seulement moi (pas disponible sur Windows Vista) [Le défaut, «Installer pour tout utilisateur», est probablement le plus facile.]</p>
<p>Select destination directory. Please select a directory for the Python 2.7 files.</p>	<p>Choisissez dossier destination. Veuillez sélectionner un dossier pour les fichiers de Python 2.7. [Le défaut est bien.]</p>
<p>Customize Python 2.7. Select the way you want features to be installed.</p>	<p>Customizez Python 2.7. Choisissez la façon d'installer les traits que vous préférez. [Le default est d'installer tout trait, qui est acceptable.]</p>
<p>Please wait while the installer installs Python 2.7. This may take several minutes.</p>	<p>Veillez patienter pendant que l'installateur installe Python 2.7. Ça peut prendre plusieurs minutes.</p>
<p>Completing the Python 2.7 installer. Special thanks to Mark Hammond, without whose years of freely shared Windows experience, Python for Windows would still be Python for DOS. Click the Finish button to exit the installer.</p>	<p>Termine l'installateur Python 2.7. Remerciements spéciale à Mark Hammond, sans les années d'experience duquel Python pour Windows serait toujours Python pour DOS. Cliquez sur «Finish» pour quitter l'installateur.</p>

Après vous avez fait ça, vous allez voir dans le menu Démarrer un groupe qui s'appelle Python 2.7, et dedans vous allez voir «IDLE (Python GUI)», «Module Docs», «Python (command line)» («Invite des commandes Python»), «Python Manuals», et «Uninstall Python» («Déinstaller Python»). C'est bien, mais on va continuer avec la bibliothèque *wxPython*.

wxPython 2.8

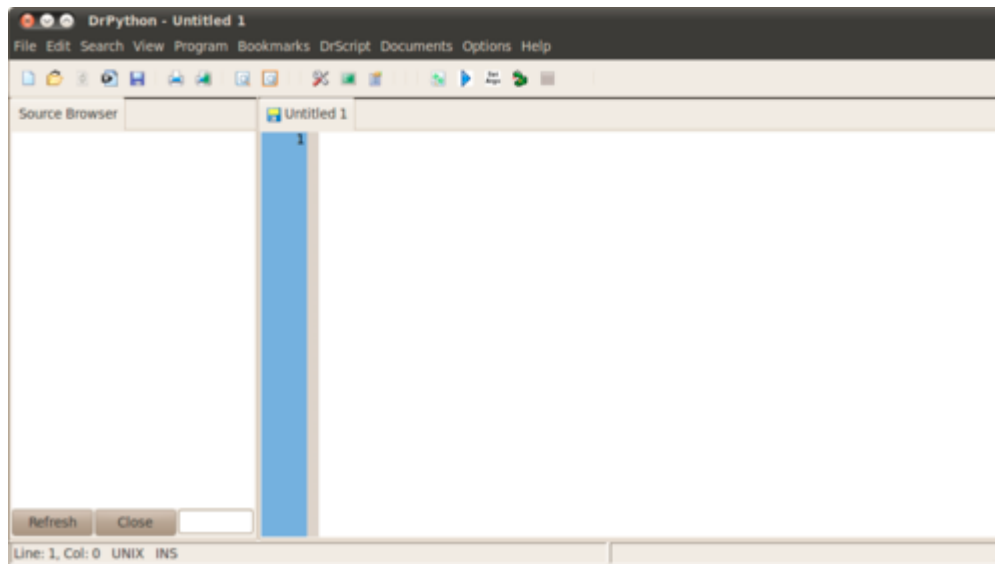
Anglais	Français
<p>Welcome to the wxPython2.8-unicode-py27 Setup Wizard. This will install wxPython 2.8.11.0 (unicode) for Python 2.7 on your computer. It is recommended that you close all other applications before continuing. Click Next to continue, or Cancel to exit Setup.</p>	<p>Bienvenue à l'installateur wxPython2.8-unicode-py27. Ça va installer wxPython 2.8.11.0 (unicode) pour Python 2.7 sur votre ordinateur. C'est conseillé que vous fermez toute autre application avant de continuer. Cliquez «Next» pour continuer, ou «Cancel» pour quitter l'installateur. [Le plupart de celui-ci est commun dans les installateurs.]</p>
<p>License agreement. (The full wxWindows library license.) I accept the agreement./I do not accept the agreement.</p>	<p>Accord de la licence [Ceux qui suit est le texte complet de la licence de la bibliothèque wxWindows.] J'accepte la licence./Je n'accepte pas la licence. [On peut accepter ; la licence est très douce.]</p>
<p>Select destination location. Where should wxPython be installed?</p>	<p>Choisissez location destination. Où devrait wxPython être installé ? [Le défaut est le meilleur lieu.]</p>
<p>Select components. Which components should be installed?</p>	<p>Choisissez composants. Quels composants devrait être installé ? [Il y a seulement un composant, qu'il faut installer. Alors, le défaut et parfait.]</p>
<p>Installing. Please wait while Setup installs wxPython2.8-unicode-py27 on your computer.</p>	<p>Veillez patienter pendant que l'installateur installe wxPython2.8-unicode-py27 sur votre ordinateur.</p>
<p>Completing the wxPython Setup Wizard. Setup has finished installing wxPython on your computer. Click Finish to</p>	<p>Termine l'installateur wxPython. L'installation a fini à installer wxPython sur votre ordinateur. Cliquez «Finish» pour</p>

install Setup. View
 README
 win32.txt./Compile Python
 .py files to .pyc./Create
 batch files for tool scripts.

quitter. Afficher README
 win32.txt./Compiler les
 fichiers Python .py à
 .pyc./Crèer fichiers pour
 [...]. [Décocher la
 première case, parce que
 c'est un fichier anglais,
 mais les autres sont utile.]

Après ça, vous allez voir une fenêtre noire qui va afficher un peu de texte et se ferme.

Maintenant, dépaquetez le dernier fichier que vous avez téléchargé : DrPython_3.11.3.zip. Vous allez trouver dedans un dossier avec beaucoup de fichiers. Cherchez un qui s'appelle drpython. (Il y aura au moins deux.) Double-cliquez le. Après ça vous allez voir une fenêtre qui paraît un peu comme cela :



Félicitations ! Vous avez réussi à installer DrPython.

N.B. Il doit avoir une façon d'installer DrPython pour qu'on puisse l'ouvrir plus facilement, par exemple en créant un raccourci, mais je n'ai pas encore trouvé.

Eclipse

Installation de Eclipse

Eclipse est un environnement de développement intégré (EDI), générique et extensible (site officiel <http://www.eclipse.org>).

Initialement prévu pour développer en Java, il peut maintenant également gérer des projets développés avec d'autres langages de programmation tels que :

- Le C et le C++ grâce à l'ensemble de plugins CDT^[5] (compilateur non intégré).
- Le Python via PyDev^[6].

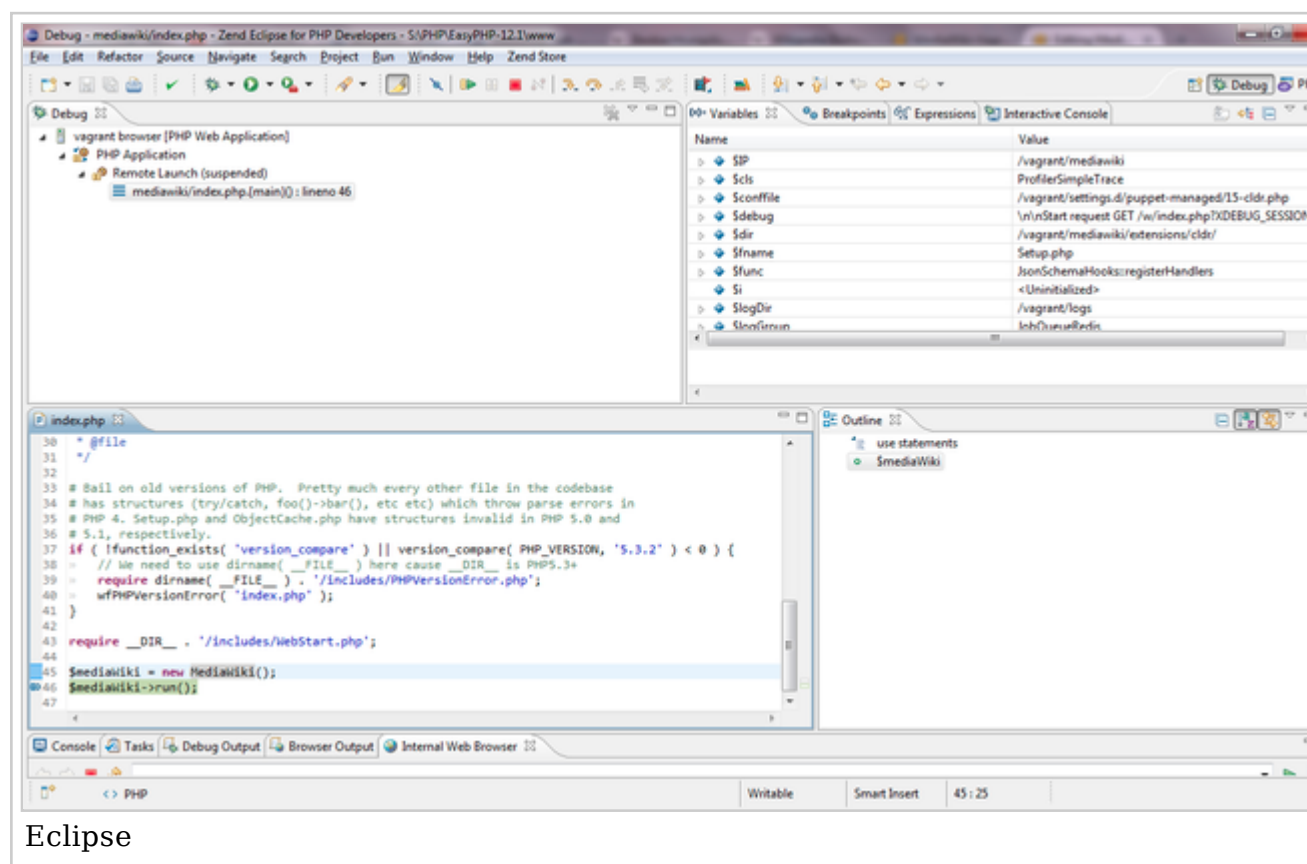
La page de téléchargement d'Eclipse permet de récupérer une version déjà adaptée au langage ciblé sur <http://www.eclipse.org/downloads/>. Mais pour installer un plugin manuellement, il faut :

- Lancer Eclipse, puis dans le menu déroulant : `Help>Software Updates>Find and Install...`
- Cocher *Search for new features to install*, bouton *Next*. Bouton *New Remote Site...*, entrer l'adresse de téléchargement :

Name: Nom du plugin
 URL: adresse du plugin, ex : <http://www.eclipse.org/cdt/downloads.php>

- Bouton *Finish*, choisir un miroir proche puis continuer l'installation.

Utilisation de Eclipse



PyDev

PyDev est un plugin Eclipse pour le développement d'un projet Python (et

Jython).

Il a été créé en Juillet 2003 par Aleks Totic et est maintenu depuis Octobre 2004 par Fabio Zadrozny. Il propose entre autres les fonctionnalités suivantes :

- complétion de code,
- analyse et mise en évidence de la syntaxe,
- debug
- ...

Installation du plugin PyDev

Sous Eclipse, voici la procédure à suivre pour installer le plugin :

- Menu "Help" / "Software Updates" / "Find and install ..." / "Search for new feature to install"
- "New Remote Site..." / Name : Python Dev, URL : <http://pydev.org/updates/> / Finish

Une fois le plug-in installé, il faut configurer le compilateur Python :

- Menu "Window" / "Preferences" / "PyDev" + "Interpreter Python" / "New"

Il vous faudra ensuite choisir l'exécutable python : `"/usr/bin/python"` sous Linux, `"C:\Python\python.exe"` sous Windows et valider, puis sélectionner les répertoires à inclure dans le PYTHONPATH (en cas de doute, prenez ceux qui vous sont proposés).

Créer un projet "Hoo hoo World" avec le plugin PyDev

Sous Eclipse, une fois le plug-in installé, choisir de créer un nouveau projet

- Menu "File" / "New" / "Project"
- Sélectionner `Pydev Project` / "Next"
- Donner un nom et choisir la version correspondant à python (ex: 2.4).
- Valider ("Finish") : *Vous devez avoir une nouvelle entrée correspondant au projet*
- Clic droit sur le projet pour ajouter un nouveau module ("New" / "Pydev module")
- donner lui un nom (ex: `monScript`)
- saisir le code suivant dans le fichier :

```
print("Hoo hoo World")
```

- sauvegarder (CTRL + S, ou clic sur la disquette, ou menu "File" / "Save")
- exécuter : Bouton droit sur le fichier `monScript` / "Run as" / "Python run"

Pour les prochaines exécutions du script, utiliser la barre d'outil (symbole lecture blanc dans un rond vert) ou CTRL + F11.

Complétion auto

Pour voir la complétion automatique de code, utilisez CTRL + Espace.

Par exemple pour le code suivant :

```
x = "Bonjour"  
x.
```

si l'on place le curseur après le point, et que l'on tape CTRL + Espace, l'ensemble des méthodes de la classe String seront proposées (ex: `.upper()` qui passe en majuscule la chaîne de caractère `x`).

Lancement des scripts

Certains EDI permettent de lancer les scripts en cliquant. Toutefois d'une manière générale, il faut les exécuter en lançant en ligne de commande l'interpréteur Python, en lui fournissant comme premier argument le nom du fichier qui contient le script, voire des paramètres supplémentaires comme autres arguments.

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

```
python MonScript.py
```

ou

```
python MonScript.py argument2 argument3
```

Par ailleurs, quand le fichier porte bien l'extension conventionnelle Python (`.py`), il a un icône Python et il est possible de l'exécuter simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c'est-à-dire l'explorateur, sous *Windows*, *Gnome*, ou *Konqueror*, sous *KDE*). En effet ces gestionnaires graphiques « savent » qu'il doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par `.py` (cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs « intelligents » de reconnaître automatiquement les scripts Python et d'adapter leur coloration syntaxique en conséquence.

Évitez cependant de choisir des noms qui risqueraient d'être déjà attribués à des modules python existants, tels que `"math.py"` ou `"Tkinter.py"`.

Si vous travaillez avec *IDLE* sous Windows, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches `<Ctrl-F5>`.

En-tête des fichiers

Enfin, l'en-tête des fichiers .py doit toujours contenir le shebang :

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

suivi par leur encodage, comme :

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

ou

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

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

- Soit le jeu de caractères accentués correspondant aux principales langues de l'Europe occidentale (Français, Italien, Portugais, etc.), codé sur un seul octet suivant la norme ISO-8859 ;
- Soit le système de codage mondial sur deux octets appelé Unicode (dont la variante Utf-8 ne code que les caractères « spéciaux » sur deux octets, les caractères du jeu ASCII standard restant codés sur un seul octet). Ce dernier système commence à se répandre de plus en plus, car il présente l'avantage de permettre la coexistence de caractères de toutes origines dans le même document (caractères grecs, arabes, cyrilliques, japonais, etc.). C'est ce mode qu'il faut privilégier pour plus de compatibilité.

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

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

Si vous n'en indiquez aucun, vous recevrez de temps à autre des messages d'avertissement de la part de l'interpréteur, et vous éprouverez peut-être même quelques difficultés à éditer correctement vos scripts dans l'environnement IDLE (en particulier sous **Windows**).

Références

1. <https://netbeans.org/downloads/index.html>
2. <https://wiki.python.org/moin/PythonWin>

3. <http://www.mmm-experts.com/Downloads.aspx>
4. <http://scintilla.sourceforge.net/SciTETranslation.html>
5. <https://eclipse.org/cdt/>
6. (anglais) <http://pydev.org/>

Programmer en deux minutes

1. Ouvrir Python (command line). *Comment ?* En exécutant python ou en recherchant python.exe

Une invite de commande s'affiche.

```

Hello World !

Saisissez « print("Bonjour tout le monde.") » puis pressez Entrée/Retour.
Vous avez à l'écran :

>>> print("Bonjour tout le monde.")
Bonjour tout le monde.
    
```

2. Saisissez « a=3 » puis pressez Entrée/Retour. Dans l'invite suivante saisissez « print(a) ». Vous avez à l'écran :

```

>>> a=3
>>> print(a)
3
    
```

3. Saisissez ces commandes :

```

>>> a=0 # Shift + Enter pour sauter à la ligne suivante
>>> while (a<4): # ouvre un process (1), donc...
...     a = a+1 # ajouter une indentation (espace) au début pour signifier l'appartenance
...     print("semaine"), a # demande d'afficher "semaine" et a.
...
semaine 1 # 1er résultat
semaine 2
semaine 3
semaine 4 # etc, votre programme s'exécute en boucle tant que a<4
    
```

4. Ouvrez un éditeur de texte, écrivez...

```

a = 0
while (a<20):
    a = a + 1
    print ("semaine"), a, "2009"
    
```

...et enregistrez ce script sous la forme *.py

5. Ouvrez la ligne de commande de votre système d'exploitation (par exemple : bash GNU/Linux, Terminal pour Mac OS X, ou cmd pour Windows), placez votre invite de commande dans le répertoire de votre fichier *.py, affichez votre

fichier *.py dans votre invite de commande et exécutez-le en pressant Enter.

Félicitation, vous avez écrit et exécuté votre premier script !

Programmer en deux minutes/une messagerie instantanée

Nous allons écrire deux scripts en Python et les exécuter chacun dans une console. Les deux programmes vont communiquer afin d'envoyer un message et d'attendre la réponse.

Écoute

1). Ouvrir un éditeur de texte, et coller le script suivant (sans caractère spéciaux comme "é")...

```
-----  
ecoute.py  
-----  
#!/usr/bin/python  
# -*- coding: latin-1 -*-  
import socket          # importe un ensemble d'instructions pour connecter les programmes.  
                        # Cet ensemble est disponible a l'installation de Python, dans la bibl  
# Creation du connecteur d'ecoute par l'instruction 'socket'  
# de la bibliotheque socket precedemment importee.  
Connecteur = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
  
Hote = '127.0.0.1' # Adresse locale de l'ordinateur.  
Port = 80         # Choix d'un port d'ecoute.  
Connecteur.bind((Hote,Port)) # instruction 'bind' de la bibliotheque du connecteur  
print "Le programme est a l'ecoute d'une eventuelle discussion, vous en serez averti." #  
Connecteur.listen(1)          # ecoute...  
client, adresse = Connecteur.accept() # accepte...  
print "L'ordinateur",adresse," veut discuter ! J'attends son message." # Rajoutez des par  
  
# Creation du connecteur de reponse  
Reponse = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
Portreponse = 234  
Reponse.connect((Hote,Portreponse ))  
print "Note : je me suis connecte a",adresse," pour lui repondre" # Rajoutez des parenthe  
  
while 1:  
    Message = client.recv(255) # reception de la reponse, 255 caracteres max ; Python  
    if not Message:  
        break  
    print "\nMessage : ",Message,"\a" + "\n\nVotre reponse :" # Rajoutez des parenthe  
    msgR = raw_input('>> ') # votre message ? Python 3 : msgR = bytes(input('>> '),  
    Reponse.send(msgR)      # envoi.  
  
client.close() # ferme la connexion lorsque le client est parti : [ctrl+C] pour abandonne  
-----
```

...enregistrez ce script (par exemple ecoute.py) et exécutez-le.

Discussion

2. Ouvrir l'éditeur de texte, écrire le script de discussion...

```
-----  
discussion.py  
-----  
#!/usr/bin/python  
# -*- coding: latin-1 -*-  
import socket  
Discuter = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
Hote = '127.0.0.1'  
Port = 80  
Port_de_reponse = 234  
Discuter.connect((Hote,Port))      # Se connecte au programme ecoute.py  
  
Reponse = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
Reponse.bind((Hote,Port_de_reponse))  
Reponse.listen(1)  
client, adresse = Reponse.accept() # Creation du connecteur pour la reponse de ecoute.py  
print "L'adresse",adresse," vous a entendu et attend votre message." # Rajoutez des paren  
while 1:  
    msg = raw_input('>> ') # votre message ? Python 3 : msg = bytes(input('>> '), 'm  
    Discuter.send(msg)      # envoi.  
    print "Attente de la reponse..." # Rajoutez des parentheses pour Python 3 !  
    reponseaumessage = client.recv(255) # reception de la reponse, 255 caracteres max  
    if not reponseaumessage:  
        break  
    print "\n",adresse,":",reponseaumessage,"\n\n" # affiche la reponse # Rajoutez d  
client.close() # ferme la connexion lorsque le client quitte.  
-----
```

...enregistrez ce script (par exemple `discussion.py`) et exécutez-le dans une nouvelle console.

Félicitation, vos deux consoles communiquent !

Programmer en deux minutes/l'interface de Wikipédia pour programmer

Nous allons écrire un script en Python et l'exécuter dans une console. Le script va utiliser deux ensembles de commandes définis dans la bibliothèque fournie à l'installation du langage.

L'interface avec Wikipédia se fait via des requêtes à :

```
https://fr.wikipedia.org/w/api.php?
```

Par exemple :

```
https://fr.wikipedia.org/w/api.php?action=query&prop=info%7Crevisions&titles=Accueil
```

Réviseur de la page

1. Ouvrir un éditeur de texte, coller le script suivant (sans caractère spéciaux comme "é" si le fichier est en ASCII au lieu de Unicode)...

```

reviseur_de_la_page.py
-----
#!/usr/bin/python
# -*- coding: latin-1 -*-
# Pour Python 3, remplacez la ligne ci-dessous par import urllib.request, re
import urllib
import re      # import des modules a partir de la bibliotheque d'instructions de base,
# 'urllib' pour URL library et 're' pour regular expression.
nom_de_page = "Accueil"

url = "http://fr.wikipedia.org/w/api.php?action=query&prop=info|revisions&titles=%s&forma
# affichage
# Python 3 : page = urllib.request.urlopen(url)
page = urllib.urlopen(url)
# Python 3 : infos = str(page.read()), 'utf_8')
infos = page.read()      # lit le resultat de la requete a l'url ci-dessus
page.close()
print("Les informations demandees concernant" + nom_de_page + "sont les suivantes, en XML

# extraction
print("\n...recherche l'expression rationnelle...") # Rajoutez des parenthèses pour Pyth
reviseur = re.findall(' user="(.*?)" ',infos) # recherche l'expression rationnelle
print("\nDernier reviseur : " + reviseur) # Rajoutez des parenthèses pour Python 3 !
-----

```

...enregistrez ce script (par exemple `reviseur_de_la_page.py`) et exécutez-le. Le script utilise cette requête pour afficher le dernier réviseur de la page d'accueil.

Note : il s'agit de la méthode utilisant les bibliothèques natives de Python. Une alternative est d'utiliser le framework Pywikibot, qui s'occupe de l'API de MediaWiki à la place du développeur, rendant la tâche beaucoup plus simple pour des scripts plus perfectionnés.

Boucle réviseur bistro

2. Obtenir la liste des derniers réviseurs des Bistros du mois dernier. Ouvrir l'éditeur de texte, écrire ce script utilisant plusieurs fois cette requête... Si vous souhaitez utiliser le code suivant avec Python 3, faites les mêmes modifications que dans le script précédent. C'est-à-dire : rajoutez des parenthèses aux `print` ; chargez la classe `urllib.request` (au lieu d'`urllib` tout court) ; utilisez la fonction `urllib.request.urlopen` (au lieu de `urllib.urlopen`) ; transformez le résultat de `read` en chaîne de caractères (`infos = str(url.read(), 'utf_8')`).

```

boucle_reviseur_bistro.py
-----
#!/usr/bin/python
# -*- coding: latin-1 -*-
import urllib # import des modules de la bibliotheque d'instructions fournie a l'install
import re

a = 0
space = ' '
nospace = ''

while a < 31:
    a = a+1 # a est un nombre entier (integer)
    b = str(int(a)) # et b une chaine de caractere (string)
    nom = "Wikipedia:Le_Bistro/"+b+"_avril_2009"
    nom = nom.replace(space, nospace) # supprime les espace
    url = urllib.urlopen("http://fr.wikipedia.org/w/api.php?action=query" +
        "&prop=info|revisions&titles=%s&format=xml" % nom )
    infos = url.read()
    url.close()
    reviseur = re.findall(' user="(.*?)" ', infos) # recherche l'expression rationnelle
    for chacun in reviseur:
        print("\nDernier reviseur du bistro du " + b + " avril 2009 : " + chacun)
-----

```

...enregistrez ce script (par exemple `boucle_reviseur_bistro.py`) et exécutez-le.

Liste des réviseurs

3. La liste des réviseurs de la page d'accueil entre deux dates, et les

commentaires de révisions : ouvrir l'éditeur de texte, écrire ce script, faire les mêmes modifications pour Python 3 le cas échéant... Ce script utilise cette requête.

```

liste_des_reviseurs.py
-----
#!/usr/bin/python
# -*- coding: latin-1 -*-
import urllib # import des modules de la bibliotheque d'instructions fournie a l'installa
import re

debut = str(int(20090311000000)) # date pour commencer a lister, ici 11 mars 2009.
fin = str(int(20090511000000))

nom = "Accueil"

url = urllib.urlopen("http://fr.wikipedia.org/w/api.php?action=query" +
    "&prop=revisions&rvstart=%s&revent=%s&titles=%s&format=xml" % (debut, fin, nom))
infos = url.read()
url.close()

# recherche et affiche les reviseurs
#
reviseur = re.findall(' user="(.*?)" ',infos)
for chacun in reviseur:
    print("Reviseur : " + chacun)

# recherche et affiche les commentaires
commentaire = re.findall(' comment="(.*?)" ',infos)
for comment in commentaire:
    print("\nCommentaire de revision : " + comment)
-----

```

Félicitations, vous utilisez Wikipédia via son **API** !

Vous pouvez poursuivre cet exercice en programmant du python sur une base vierge, ou alors utiliser la librairie d'instructions Pywikipedia et les scripts Pywikipedia hébergés par Wikimédia. Par exemple, vous devriez être capable de lire un script tel que :

```

statistics_in_wikitable.py
-----
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
\03{lightyellow}This bot renders statistics provided by [[Special:Statistics]] in a table
Thus it creates and updates a Statistics wikitable.

The following parameters are supported:
-----

```



```

\03{lightred}-screen\03{default}      If True, doesn't do any changes, but only shows the
\03{lightgreen}-page\03{default}     On what page statistics are rendered.
                                     If not existing yet, it is created.
                                     If existing, it is updated.
"""
__version__ = '$Id$'
import wikipedia
import pagegenerators
import query
import time

# This is the title of the wikipage where to render stats.
your_page = "Logstats"

summary_update = {
    'en':u'Updating some statistics.',
}
summary_creation = {
    'en':u'Creating statistics log page.',
}

class StatisticsBot:
    def __init__(self, screen, your_page):
        """
        Constructor. Parameter:
            * screen      - If True, doesn't do any real changes,
                           but only shows some stats.
        """
        self.screen = screen
        self.your_page = your_page
        self.dict = self.getdata() # Try to get data.
        self.site = wikipedia.getSite()

    def run(self):
        if self.screen:
            wikipedia.output("Bot is running to output stats.")
            self.idle(1) # Run a function to idle
            self.outputall()
        if not self.screen:
            self.outputall() # Output all datas on screen.
            wikipedia.output("\nBot is running. " +
                "Going to treat \03{lightpurple}%s\03{default}..." %
                self.your_page )
            self.idle(2)
            self.treat()

# getdata() returns a dictionary of the query to
# api.php?action=query&meta=siteinfo&siprop=statistics
def getdata(self):
    # This method return data in a dictionary format.
    # View data with: api.php?action=query&meta=siteinfo&siprop=statistics&format=json
    params = {
        'action'    : 'query',
        'meta'      : 'siteinfo',
        'siprop'    : 'statistics',
    }
    wikipedia.output("\nQuerying api for json-formatted data...")
    try:
        data = query.GetData(params,self.site, encodeTitle = False)
    except:
        url = self.site.protocol() + '://' + self.site.hostname() + self.site.api_add
        wikipedia.output("The query has failed. Have you check the API? Cookies are w

```

```

        wikipedia.output(u"\n>> \03{lightpurple}%s\03{default} <<" % url)
    if data != None:
        wikipedia.output("Extracting statistics...")
        data = data['query'] # "query" entry of data.
        dict = data['statistics'] # "statistics" entry of "query" dict.
        return dict

def treat(self):
    page = wikipedia.Page(self.site, self.your_page)
    if page.exists():
        wikipedia.output(u'\nWikitable on ' +
            u'\03{lightpurple}%s\03{default} will be completed with:\n' % self.your_p
        text = page.get()
        newtext = self.newraw()
        wikipedia.output(newtext)
        choice = wikipedia.inputChoice(
            u'Do you want to add these on wikitable?', ['Yes', 'No'], ['y', 'N'], 'N')
        text = text[:-3] + newtext
        summ = wikipedia.translate(self.site, summary_update)
        if choice == 'y':
            try:
                page.put(u''.join(text), summ)
            except:
                wikipedia.output(u'Impossible to edit. It may be an edit conflict...')
    else:
        wikipedia.output(
            u'\nWikitable on \03{lightpurple}%s\03{default} will be created with:\n'
        newtext = self.newtable()+self.newraw()
        wikipedia.output(newtext)
        summ = wikipedia.translate(self.site, summary_creation)
        choice = wikipedia.inputChoice(
            u'Do you want to accept this page creation?', ['Yes', 'No'], ['y', 'N'],
        if choice == 'y':
            try:
                page.put(newtext, summ)
            except wikipedia.LockedPage:
                wikipedia.output(u"Page %s is locked; skipping." % title)
            except wikipedia.EditConflict:
                wikipedia.output(u'Skipping %s because of edit conflict' % title)
            except wikipedia.SpamfilterError, error:
                wikipedia.output(
                    u'Cannot change %s because of spam blacklist entry %s' % (title,

def newraw(self):
    newtext = ('\n|----\n!\'\'+ self.date() +'\'\') # new raw for date and stats
    for name in self.dict:
        newtext += '\n|'+str(abs(self.dict[name]))
    newtext += '\n|----\n|}'
    return newtext

def newtable(self):
    newtext = ('\n{ class=wikitable style=text-align:center\n!' + "date") # create
    for name in self.dict:
        newtext += '\n|'+name
    return newtext

def date(self):
    return time.strftime('%Y/%m/%d', time.localtime(time.time()))

def outputall(self):
    list = self.dict.keys()
    list.sort()
    for name in self.dict:

```

```

        wikipedia.output("There are "+str(self.dict[name])+" "+name)

def idle(self, retry_idle_time):
    time.sleep(retry_idle_time)
    wikipedia.output(u"Starting in %i second..." % retry_idle_time)
    time.sleep(retry_idle_time)

def main(your_page):
    screen = False # If True it would not edit the wiki, only output statistics
    _page = None

    wikipedia.output("\nBuilding the bot...")
    for arg in wikipedia.handleArgs(): # Parse command line arguments
        if arg.startswith('-page'):
            if len(arg) == 5:
                _page = wikipedia.input(u'On what page do you want to add statistics?')
            else:
                _page = arg[6:]
        if arg.startswith("-screen"):
            screen = True
    if not _page:
        _page = your_page
    if not screen:
        wikipedia.output("The bot will add statistics on %s.\n" % _page )
    bot = StatisticsBot(screen, _page) # Launch the instance of a StatisticsBot
    bot.run() # Execute the 'run' method

if __name__ == "__main__":
    try:
        main(your_page)
    finally:
        wikipedia.stopme()

```

Le script `statistics_in_wikitable.py` importe quelques librairies d'instructions dont `Pywikipedia`, définit trois variables, définit l'objet `StatisticsBot`, puis définit une fonction principale qui est exécutée à la fin du script (par l'instruction `try: main(your_page)`).

Programmer en deux minutes/un serveur Web

Nous allons écrire un script en Python et l'exécuter dans une console. Le programme va utiliser deux ensemble d'instructions, ces ensembles étant définis dans la bibliothèque de base.

Serveur HTTP

1. Ouvrir un éditeur de texte, écrire le script...

```

serveur_http.py
-----
#!/usr/bin/python
# -*- coding: latin-1 -*-
import SimpleHTTPServer # importe un ensemble d'instructions pour servir les requêtes htt
import SocketServer      # importe un ensemble d'instructions pour connecter le programme.
                        # Ces deux ensembles sont disponibles à l'installation de Python
## Python 3 :
# import http.server
# import socketserver
# Serveur http de base delivrant le contenu du repertoire courant via le port indique.
PORT = 5432
Handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer("", PORT), Handler)
print "serving at port", PORT
## Python 3 :
# Handler = http.server.SimpleHTTPRequestHandler
# httpd = socketserver.TCPServer("",PORT), Handler)
# print("à l'écoute sur le port :", PORT)
httpd.serve_forever()
-----

```

...enregistrez ce script (par exemple `serveur_http.py`) dans un répertoire que vous voulez partager (par exemple `/PUBLIC_web`) et exécutez-le dans une console.

Félicitation, votre programme répond aux requêtes `http://localhost:5432` [Note 1] et votre répertoire public [Note 2] peut être téléchargé.

1. « `http://localhost:5432` » est une requête de protocole HTTP à votre machine locale via le port 5432
2. Nommé « `/PUBLIC_web` » dans l'exemple

Index.html

2. Ouvrez un éditeur de texte et créez l'index de votre site statique :

index.html

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//FR"
  "http://www.w3.org/TR/html4/strict.dtd">
<html lang="fr-FR">
  <head>
    Bonjour, bienvenue sur le serveur de mon ordinateur perso.
    C'est un <a href="serveur_http.py">programme en Python</a> qui vous sert.
  </head>
  <body>
    <title>Salut le Web</title>
    <h2>Sommaire</h2>
    <p><a href="page1.html">Page 1</a> - <a href="page2.html">Page 2</a>

    <p>Fichier partagé : <a href="rapportXX.odt">rapportXX</a> et lien web : <a href="http:/
  </body>
</html>
```

...enregistrez ce fichier sous le nom `index.html` dans le répertoire que vous partagez (Nommé « `/PUBLIC_web` » dans cet exemple).

Félicitation, votre programme répond aux requêtes `http://localhost:5432` en servant l'index de votre site web !

Afficher un texte

Ce chapitre détaille les différentes manières d'afficher un texte en console, par exemple pour générer des logs. Les interfaces graphiques plus élaborées seront traitées avec des bibliothèques.

print()

La fonction *print* sert à afficher des données sur la sortie standard, qui est l'écran. Exemple :

fonction *print*

```
>>> print('Hello World!')
Hello World!
```

print formaté

Il est possible de changer une partie du texte en y incorporant des *marqueurs de conversion* précédés de l'opérateur "%". Il s'agit alors d'un "print formaté" (comme avec printf dans d'autres langages).

Les paramètres placés entre parenthèses à droite de la chaîne à afficher seront insérés dedans, en lieu et place des marqueurs. Cette technique a l'avantage de convertir automatiquement les variables pour qu'elles s'inscrivent dans le texte et qu'elles s'affichent.

Parmi les marqueurs disponibles^[1], on trouve :

- %s représente un paramètre de type "string" (texte).
- %d un "digit" (nombre entier).
- %f un "floating" (flottant : nombre réel).

Exemple :

```
>>> int1, int2 = 1, 2
>>> print(u"le résultat de " + int1 + ' + ' + int2 + " est : " + int1 - int2)
le résultat de 1 + 2 est 3
>>> print(u"le résultat de", int1 , '+', int2, "est :", int1 - int2)
le résultat de 1 + 2 est 3
>>> print(u"et le résultat de %s + %d est %f" % (int1, int2, int1 + int2))
et le résultat de 1 + 2 est 3.000000
```

input()

input fait comme "print", mais stoppe le programme en attente que l'utilisateur presse "entrée". De plus, si on stocke son résultat une variable, elle aura le type et contenu de ce que l'utilisateur a tapé avant de valider. Si ce texte est vide, le programme plante. Exemple :

```
variable = input('Quel est votre nom ? ') # répondre avec un texte entre apostrophe
print('Bonjour ' + variable)
```

raw_input()

raw_input() fait comme "input", mais renvoie toujours une chaîne de caractère. Elle ne plante donc pas si le texte est vide.

Attention !



A partir de Python 3.1, la fonction "raw_input()" n'existe plus, et la fonction input() la remplace et renvoi systématiquement une chaîne de caractère.

Afficher des caractères spéciaux

Normalement, les symboles ne faisant pas partie du code ASCII sont traités comme les autres. Ex :

```
print('à â ç é è ê ë ï ô ù ÿ')
```

```
à â ç é è ê ë ï ô ù ÿ
```

Toutefois selon l'encodage, il peut être nécessaire de les convertir en Unicode en plaçant un "u" avant :

```
print(u'à â ç é è ê ë ï ô ù ÿ')
```

Pour les retours à la ligne, utiliser "\n".

Antislash

Les caractères affichés par les fonctions précédentes doivent être encadrés par

des apostrophes ou des guillemets, mais que se passe-t-il s'ils contiennent eux-mêmes ces symboles ?

Testons la fonction *print* :

ce qui marche :

```
>>> print("ça marche aussi")
ça marche aussi
>>> print('et ça ? "ça marche" ou pas')
et ça ? "ça marche" ou pas
>>> print("et on 'essaye' l'inverse")
et on 'essaye' l'inverse
```

ce qui ne marche pas :

```
>>> print('un simple guillemet ' encadré par du simple quote ')
[message d'erreur]
>>> print(" pareil pour le " double ")
[message d'erreur]
>>> print("double " deux " fois dans double")
[message d'erreur]
```

En Python, le simple quote peut encadrer le double, et le double peut encadrer le simple, car la chaîne commence au premier guillemet et finit au même guillemet.

Pour s'affranchir de ces limitations, on utilise un caractère d'échappement, qui est, en Python, l'antislash (« \ »). Il permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes n'importe quel type de commande. Pratique pour éviter de dépasser les 120 caractères pouvant nécessiter l'utilisation d'un ascenseur latéral pour être lus.
- À l'intérieur d'une chaîne de caractères, l'*antislash* permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> print('une chaîne avec des \' guillemets \' simples dans des simples, c\'est protégé '
une chaîne avec des ' guillemets ' simples dans des simples, c'est protégé')
>>> 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.

```

On remarque que la séquence `\'` permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes, et `\"` par des guillemets. De même, `"\"` affiche `"\"`.

D'autres caractères peuvent aussi être introduits par antislash :

- Retour chariot : `\n`
- Tabulation : `\t`

Triple quotes

Si le nombre d'antislash nuit à la lisibilité du code, on peut délimiter la chaîne à l'aide de *triples guillemets* ou de *triples apostrophes* :

```

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

```

Commentaires

Tout ce qui suit un dièse (`#`) jusqu'à un retour à la ligne est un commentaire : ce n'est pas pris en compte par l'interpréteur. Mais cette règle ne s'applique pas lorsque le dièse est positionné dans une chaîne de caractères (entre quotes).

Le but reste de produire du code compréhensible sans commentaire, mais ce texte affiché uniquement dans le code source, peut servir à expliquer à celui qui relit le code (y compris soi-même), les subtilités les moins évidentes de celui-ci.

```

# Toute cette première ligne est un commentaire.
print("Bonjour le monde") # Ceci est également un commentaire

```

```
print("Bonjour"); print("Le monde"); # Ceci est une ligne comportant
# plusieurs instructions
print("Cette ligne ne contient pas de #commentaire")
```

Exemple 1 : On notera que la fonction print affiche son argument.

Commentaires de bloc

Les commentaires de bloc (*block comments* en anglais) permettent de commenter plusieurs lignes. En Python, ils sont assurés par des triples apostrophes ou guillemets :

```
print('début')
'''
Le paragraphe suivant est commenté
sur plusieurs lignes
'''
print('fin')
```

Couleurs

Il est possible d'écrire en couleurs dans le terminal en précisant leurs codes^[2]. Exemple :

```
print('\x1b[6;31;40m' + ' Texte rouge ' + '\x1b[0m')
print('\x1b[6;30;42m' + ' Fond vert ' + '\x1b[0m')
```

Références

1. <https://docs.python.org/2.4/lib/typesseq-strings.html>
2. <https://i.stack.imgur.com/6otvY.png>

Structure d'un programme

Les instructions

Un programme Python est composé d'**instructions**. Une instruction est un ordre unitaire donné à un programme. Par exemple *afficher Bonjour* est une instruction, de même que *calculer un plus un*.

La structure d'un programme Python est certainement ce qui étonne le plus le programmeur ayant l'habitude d'un langage plus traditionnel comme le C : en Python les limites des instructions et des blocs sont définies par la mise en page.

- Le point-virgule (;) à la fin de chaque instruction est remplacé par le caractère de fin de ligne (symbolisé par "\n"). Ce qui simplifie la syntaxe car traditionnellement les points-virgules étaient toujours suivis de retours à la ligne.
- Les accolades ({ }) fréquemment utilisées pour définir les blocs sont remplacées par les niveaux d'indentations. En effet, il était déjà conseillé aux programmeurs de se servir *aussi* des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs, donc c'est également un allègement du code.

En définitive, Python oblige à écrire du code lisible, et à prendre de bonnes habitudes conservables dans d'autres langages, ce qui est fait un langage idéal pour apprendre la programmation.

Les espaces sont ignorés

A part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions sont ignorés, sauf s'ils font partie d'une chaîne de caractères.

L'interpréteur Python commence par analyser la première ligne :

- si celle-ci contient une instruction, alors il l'**exécute**
- si l'instruction n'est pas une **instruction de contrôle**, alors, il passe à la ligne suivante, l'analyse et l'exécute
- si le programme Python arrive à la fin du fichier à exécuter, alors, il **sort** du programme et en arrête l'exécution.

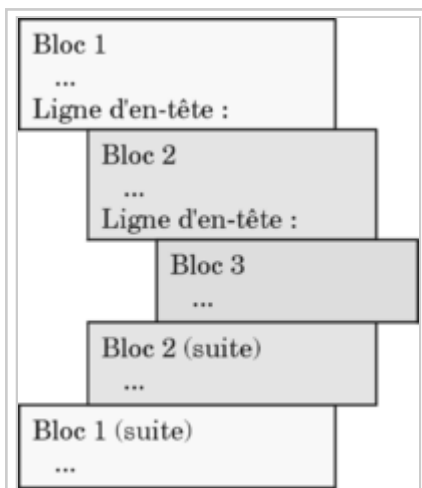
```
print "Bonjour"
1+1
c = 3e5
e = m*c**2
a, b, c = 1, 2, 3
discriminant = b**2-4*a*c
```

```

lambda x,y : x + y
dicol = {'prénom':'Eric', 'nom':'tartempion'}
print ("Bonjour %s" % dicol['nom'])
    
```

Exemple 1 : quelques instructions

Notion de bloc d'instructions



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).

Un **bloc d'instructions** est une suite d'instructions qui est alignée sur la même tabulation. Les blocs d'instructions sont créés par les instructions de contrôles comme *if*, *while* et *for*, ainsi que par les instructions permettant de déclarer des fonctions.

Sous Python, toutes les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête.

```

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

Il y a deux solutions pour indenter : utiliser quatre espaces ou un seul caractère tabulation, mais jamais un mélange des deux sous peine d'erreurs *IndentationError: unindent does not match any outer indentation level*. 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.

La tabulation est notamment la convention utilisée pour la librairie standard de Python, mais les espaces ont l'avantage de fonctionner avec tous les éditeurs (et ainsi, de pouvoir être plusieurs sur le même projet avec des éditeurs différents). On préférera donc 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", qui modifie ce que produit la touche tab.

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau*. Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête.

```
#Ce bloc d'instruction est collé contre le bord gauche du fichier
print "Je suis dans le premier bloc"
print "Je suis toujours dans le premier bloc"

if (a == 12) : #L'instruction 'if' initie un nouveau bloc
    #Ce bloc est a quatre espace du bord
    print "Je suis dans le second bloc"
    print "Je suis encore dans le second bloc"

    if (b == 13 ) :
        #Il est possible d'imbriquer des blocs dans des blocs
        print "Je suis dans un troisième bloc"
        print "et ici aussi"

    print "Je reviens dans le second bloc"

print "Je suis revenue dans le premier bloc"
```

Exemple 2 : Les blocs

Variables

Affectations

Pour les non-programmeurs

Une **variable** est un espace mémoire dans lequel il est possible de mettre une **valeur**. Par exemple, si en français je dis *x est égal à 1*, j'utilise la variable dont le nom est *x* pour lui fixer la valeur *1*. Pour faire la même chose en Python, je note simplement : `x = 1`.

Cette opération est appelée "affectation" (ou "assignation"), et consiste à stocker une valeur en mémoire vive de l'ordinateur. On dit donc que l'on procède à l'affectation de la variable "x" avec la valeur "1".

En Python comme dans de nombreux autres langages, le symbole `=` est l'opérateur d'affectation. Cette valeur restera accessible jusqu'à la fin de l'exécution du programme (vérifiable avec `print(x)`).

La valeur de la variable peut être un littéral, c'est-à-dire une constante, ou bien une expression (ex : une autre variable ou une instruction comme `"1+1"`). L'expression est évaluée avant d'être affectée à la variable.

Réaffectation

Si la variable affectée n'existe pas, l'ordinateur la crée, sinon il écrase sa valeur. En effet, il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite.

```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
```

Ceci nous amène à attirer votre attention sur le fait que le symbole *égale* utilisé sous Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il

est tentant d'interpréter l'instruction `altitude = 320` comme une affirmation d'égalité, mais ce n'en n'est pas une !

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

En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```

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

Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple d'affectation multiple :

```

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

Affectations parallèles

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

```

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

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

Exercices

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

```
Python>>> largeur = 20
Python>>> hauteur = 5 * 9.3
Python>>> largeur * hauteur
930.0
```

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 ?

Solution (cliquez)

Réfléchissez !

Principe de fonctionnement

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

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

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

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

Exemple

Prenons l'exemple suivant :


```

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

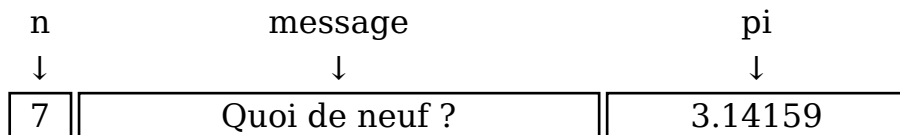
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

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

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

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

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



Les trois noms de variables sont des *références*, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres.

Les références

Formellement, les variables Python sont des **références**, c'est à dire que écrire "a = ((1,0,0),(0,1,0),(0,0,1))" ne signifie pas que "a" vaut "((1,0,0),(0,1,0),(0,0,1))" mais que "a" référence le n-uplet "((1,0,0),(0,1,0),(0,0,1))". La différence est que ensuite, une autre variable peut référencer le même n-uplet, simplement avec "b = a". Si on modifie "b", alors "a" sera également modifié.

L'utilisation des références

```

>>> a = [(1,0,0), (0,1,0), (0,0,1)]
>>> b = a
>>> print (a)
[(1,0,0), (0,1,0), (0,0,1)]
>>> print (b)
[(1,0,0), (0,1,0), (0,0,1)]
    
```

```
>>> b[0] = (1,1,0)
>>> print (b)
[(1,1,0), (0,1,0), (0,0,1)]
>>> print (a)
[(1,1,0), (0,1,0), (0,0,1)]
```

Notez que a et b ne sont pas "liés", ils référencent simplement le même objet.

Si ensuite on fait référence à un nouvel objet (par l'intermédiaire de l'opérateur d'affectation "="), b référencera toujours l'ancien objet, et une modification de l'un des objets ne modifiera pas l'autre.

Indépendances des variables

```
>>> a = [1,2,3]
>>> b = a
>>> print (a)
[1,2,3]
>>> print (b)
[1,2,3]
>>> a = [4,5,6]
>>> print (b)
[1,2,3]
>>> print (a)
[4,5,6]
>>> a.append(7)
>>> print (a)
[4,5,6,7]
>>> print (b)
[1,2,3]
```

Idem pour les nombres :

Références et nombres

```
>>> a = 5
>>> b = a
>>> print (a)
5
>>> print (b)
5
>>> b += 5
>>> print (b)
10
>>> print (a)
5
```

Noms des variables

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que "longitude" et "latitude" conviennent mieux que "x" et "y", car un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

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

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- La casse est significative (les caractères majuscules et minuscules sont distingués). Donc "Joseph", "joseph", et "JOSEPH" sont des variables différentes.
- les 29 « mots réservés » du langage sont déjà pris (ex : "print").
- Seules les lettres du code ASCII sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc. sont interdits, à l'exception du caractère `_` (souligné).

Remarque : sous Python 3 les règles de nommage ont été assouplies, car il devient possible d'utiliser n'importe quel caractère Unicode non opérateur dans les noms.

Il faut également noter que les variables dont le nom commence par le caractère `_` ont une signification particulière :

- les noms commençant par un `_` ne sont pas exportés lorsqu'ils se trouvent dans un module ;
- les noms commençant par deux `_` et finissant par deux `_` sont réservés par le langage lui même, notamment pour la programmation orientée objet.

Les noms suivants ne peuvent pas être utilisés comme nom de variable

```
41toto # On ne commence pas par un chiffre
élément # Contient un caractère accentué, qui ne sont pas autorisés
```

De plus, il est recommandé d'écrire les noms de variables avec une minuscule, car l'usage veut qu'on le réserve plutôt la majuscule aux noms de classes.

Si le nom d'une variable doit comporter plusieurs mots, il y a deux possibilités d'écrire le nom de la variable :

- En snake_case (à la C), c'est à dire en séparant les mots par le caractère `"_"`.
Exemple : `marge_brut`.
- En CamelCase (à la Java), c'est à dire en séparant les mots par un passage en

haut de casse (lettre majuscule). Exemple : *margeBrut*.

Il convient aussi d'éviter autant que possible l'énumération de variables (toto1, toto2, toto3, ...), cela rend le programme parfaitement incompréhensible et sujet à des erreurs.

Il est possible de préfixer le nom de la variable par son type. Par exemple `int_margeBrut`, `str_message_de_bienvenue`, mais cela alourdit très fortement le programme. On pourra par exemple s'inspirer de la notation hongroise qui formalise ce mécanisme.

Sinon, d'autres notations existent ou peuvent être imposées en fonction d'un projet, des habitudes d'une entreprise, etc.

Références

Opérateurs

Définition

Un *opérateur* est un symbole utilisé pour effectuer une *opération* entre des *opérandes*.

Une opérande est une variable, un littéral ou bien une expression.

Une expression est une suite valide d'opérateurs et d'opérandes.

Par exemple, dans l'expression :

```
x = y + 1
```

Il y a deux opérateurs (= et +) et trois opérandes (x, y et 1).

Certains opérateurs peuvent avoir des comportements différents en fonction des types d'opérandes sur lesquels ils agissent : on parle alors de *surcharge des opérateurs*. Exemple :

- "+" additionne des nombres, mais concatène des chaînes de caractères.
- "*" multiplie des nombres entre eux, mais duplique des chaînes de caractères.

Il existe différentes catégories d'opérateur :

Présentation des différents opérateurs

les opérateurs d'affectation

- '='
- Affectation multiple, e.g. $x = y = z = 3$
- Affectation parallèle e.g. $x, y = 1, 0,5$

les opérateurs logiques

Les expressions avec un opérateur logique sont évaluées à "True" ou "False".

- **X or Y** : OU logique, si X évalué à True, alors l'expression est True et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **X and Y** : ET logique, si X est évalué à False, alors l'expression est False et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
- **not X** : évalué à la valeur booléenne opposée de X.

les opérateurs de comparaisons

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient

une valeur booléenne "True" ou "False". Les opérateurs de comparaisons s'appliquent sur tous les types de base.

- < strictement inférieur
- > strictement supérieur
- <= inférieur ou égal
- >= supérieur ou égal
- == égal
- != différent
- <> différent, on utilisera de préférence !=
- X **is** Y : X et Y représentent le même objet.
- X **is not** Y : X et Y ne représentent pas le même objet

Il est possible d'enchaîner les opérateurs : $X < Y < Z$, dans ce cas, c'est Y qui est pris en compte pour la comparaison avec Z et non pas l'évaluation de $(X < Y)$ comme on pourrait s'y attendre dans d'autres langages.

les opérateurs mathématiques

symbole	effet	exemple
+	addition	$6+4 == 10$
-	soustraction	$6-4 == 2$
*	multiplication	$6*4 == 24$
/	division (entière ou réelle en fonction du type de donnée)	$6/4 == 1$ ou bien $6./4 == 1.5$
**	élévation à la puissance	$12**2 == 144$
//	division entière	$6//4 == 1$
%	reste de la division entière	$6\%4 == 2$

Priorité des opérations

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

- P pour *parenthèses*. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.
Ainsi $2*(3-1) = 4$, et $(1+1)**(5-2) = 8$.

- E pour *exposants*. Les exposants sont évalués ensuite, avant les autres

opérations.

Ainsi $2^{**}1+1 = 3$ (et non 4), et $3*1^{**}10 = 3$ (et non 59049 !).

- \ast et \div pour *multiplication* et *division*, qui ont la même priorité. Elles sont évaluées avant l'*addition* $+$ et la *soustraction* $-$, 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*).

- $+$ et $-$ pour *addition* et *soustraction*.

Précédence des opérateurs (du plus au moins prioritaire^[1])

Symbole ◊	Nom ◊
{}	Dictionnaire
()	Argument
[]	Partie (opérateur d'indilage)
.	Attribut
**	Puissance
~	inversion de bit
+	Positif
-	Négatif
*	Multiplier
/	Diviser
//	Résultat entier d'une division
%	Modulo
+	Plus
-	Moins
<<	Décalage à gauche
>>	Décalage à droite
&	et logique
^	ou exclusif
	ou logique
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
==	est égal
!=	est différent
is	est
in	appartient
not	non booléen
and	et booléen
or	ou booléen
lambda	expression lambda

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". Python assigne automatiquement le type « réel » à la variable "a", et le type « entier » à la variable "b".

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

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

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

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

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

Cet opérateur pourra être utile 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.

Exercices

1. Testez les lignes d'instructions suivantes :

```
>>> r, pi = 12, 3.14159
>>> s = pi * r**2
```

```
>>> print(s)
452.38896
>>>print(type(r), type(pi), type(s))
(<type 'int'>, <type 'float'>, <type 'float'>)
```

Solution On constate qu'une variable appartenant à l'ensemble des nombres entiers, multipliée par un nombre décimal, donne un décimal.

Composition

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

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

```
>>> print(17 + 3)
>>> 20
```

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

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

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

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

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

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

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

Références

1. https://swaroop.wordpress.com/notes/python_fr-op%C3%A9rateurs_et_expressions/

Structures de contrôle

Définition

Pour les non-programmeurs

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

Les *structures de contrôle* sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois :

1. la *séquence* (ex : chaînes de caractères, tuples et listes) ;
2. la *sélection* ;
3. la *répétition*.

Séquence d'instructions

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

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". Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des 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", qui permet de tester une condition et de n'exécuter les instructions que si cette condition est vérifiée. Exemple :

```
a = 11
if a > 10 :
    print("a est plus grand que dix")
```

En exécutant ce programme, on voit "a est plus grand que dix" apparaître à l'écran. On peut perfectionner le programme pour prendre en compte le cas où "a" est plus petit que dix :

```
if a > 10 :
    print("a est plus grand que dix")
else:
    print("a n'est pas plus grand que dix")
```

On utilise aussi parfois "elif" (contraction de "else if") :

```
if a > 10 :
    print("a est plus grand que dix")
elif a == 10:
    print("a est égal à dix")
else:
    print("a est plus petit que dix")
```

Remarque : en exécutant un "if" dans l'IDLE, on constate que le *prompt principal* (>>>) est maintenant remplacé par un prompt secondaire constitué de trois points (du moins sur Linux).

```
>>> if (a > 10):
...

```

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

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. Mais on peut enchaîner autant de conditions que nécessaires en utilisant avant l'instruction "elif" (contraction

de « else if »).

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")
...
...

```

Dans cet exemple, les deux lignes d'instructions indentées sous la ligne contenant l'instruction "if" constituent un même bloc logique : ces deux lignes ne sont exécutées - toutes les deux - que si la condition testée avec l'instruction "if" se révèle vraie, c'est-à-dire si le reste de la division de "a" par "2" est nul.

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 :

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

```

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

Pour que la phrase « c'est en tous cas un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 6) 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.

pass

De manière générale, un bloc contient tout le code avec une même indentation.

```
ceci est le bloc principal
if condition:
    bloc 2
    if condition2:
        bloc 3
    fin du bloc 2
fin du bloc 1
```

Si à un endroit on a syntaxiquement besoin d'un bloc mais qu'il n'y a rien à faire, on peut utiliser l'instruction `pass`, qui justement ne fait rien.

```
if condition:
    pass
else:
    instruction
```

Instructions répétitives

L'instruction "while"

Cette instruction est une boucle, c'est-à-dire qu'elle permet de répéter plusieurs fois un bloc d'instructions (en boucle).

"while" exécute des commandes **tant qu'**une ou plusieurs conditions sont vraies.

```
while condition:
    commandes
```

par exemple :

```
i = 0
while i < 5:
    i = i + 1
    print(i)
```

donne à l'exécution :

```
1 2 3 4 5
```

En effet, la seconde ligne indique à Python qu'il lui faut *répéter continuellement le bloc d'instructions qui suit, tant que* le contenu de la variable "a" reste inférieur à "5".

Comme l'instruction "if", 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.

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

- Avec l'instruction "while", Python commence par évaluer la validité de la condition fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*, c'est-à-dire :
 - l'instruction `a = a + 1` qui *incrémente* d'une unité le contenu de la variable "a" (ce qui signifie que l'on affecte à la variable "a" une nouvelle valeur, qui

est égale à la valeur précédente augmentée d'une unité).

- l'instruction "print" qui affiche la valeur courante de la variable "a".
- Lorsque ces deux instructions ont été exécutées, nous avons assisté à une première *itération*, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction "while". La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.
Dans notre exemple, si la *condition* $a < 5$ est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.

Remarques

- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner), en risquant de saturer les processeurs (si cela survient, presser CTRL + C pour annuler le lancement). 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 donc que la boucle se termine.

Exemple de boucle infinie (à éviter) :

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

Exemple : élaboration d'une table de multiplication.

```
>>> a = 0
>>> while a < 12:
...     a = a + 1
...     print(a), a**2 , a**3
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
11 121 1331
12 144 1728
```

On obtient donc la liste des carrés et des cubes des nombres de 1 à 12.

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,                # print ( b , end=" " ) avec python3
...     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.

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

VARIABLES	a	b	c
Valeurs initiales	1	1	1
Valeurs prises successivement, au cours des itérations	1	2	2
	2	3	3
	3	5	4
	5	8	5

Expression de remplacement	b	a+b	c+1

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

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

Exercices

1. Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.
2. Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :

```
1 euro(s) = 1.65 dollar(s)
2 euro(s) = 3.30 dollar(s)
4 euro(s) = 6.60 dollar(s)
8 euro(s) = 13.20 dollar(s)
etc. (S'arrêter à 16384 euros)
```

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

Solution (cliquez)

```
1. >>> c = 0
>>> while c < 20:
...     c = c + 1
...     print c, "x 7 =", c*7
```

ou encore :

```
>>> c = 1
>>> while c <= 20:
...     print c, "x 7 =", c*7
...     c = c + 1
```

```
2. >>> s = 1
>>> while s <= 16384:
...     print s, "euro(s) =", s *1.65, "dollar(s)"
...     s = s *2
```

```
3. >>> a, c = 1, 1
>>> while c < 13:
...     print a,
...     a, c = a *3, c+1
```

Exercices

1. Ecrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.
2. Ecrivez un programme qui convertisse un nombre entier de secondes fourni au départ, en un nombre d'années, de mois, de jours, de minutes et de secondes.
(Utilisez l'opérateur modulo : %).
3. Ecrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
Exemple : 7 14 21 * 28 35 42 * 49
4. Ecrivez un programme qui calcule les 50 premiers termes de la table de multiplication par 13, mais n'affiche que ceux qui sont des multiples de 7.
5. Ecrivez un programme qui affiche la suite de symboles suivante :

```

|*
|**
|***
|****
|*****
|*****
|*****
|*****

```

Solution (cliquez)

1. Réfléchissez !

```

2. # Le nombre de secondes est fourni au départ :
# (un grand nombre s'impose !)
'nsd = 12345678912
# Nombre de secondes dans une journée :
'nspj = 3600 * 24
# Nombre de secondes dans un an (soit 365 jours -
# on ne tiendra pas compte des années bissextiles) :
'nspa = nspj * 365
# Nombre de secondes dans un mois (en admettant
# pour chaque mois une durée identique de 30 jours) :
'nspm = nspj * 30
# Nombre d'années contenues dans la durée fournie :
'na = nsd / nspa      # division <entière>
'nsr = nsd % nspa     # n. de sec. restantes
# Nombre de mois restants :
'nmo = nsr / nspm     # division <entière>
'nsr = nsr % nspm     # n. de sec. restantes
# Nombre de jours restants :
'nj = nsr / nspj      # division <entière>
'nsr = nsr % nspj     # n. de sec. restantes
# Nombre d'heures restantes :
'nh = nsr / 3600      # division <entière>
'nsr = nsr % 3600     # n. de sec. restantes
# Nombre de minutes restantes :

```

```

nmi = nsr /60          # division <entière>
nsr = nsr % 60        # n. de sec. restantes

print "Nombre de secondes à convertir :", nsd
print "Cette durée correspond à", na, "années de 365 jours, plus"
print nmo, "mois de 30 jours,",
print nj, "jours,",
print nh, "heures,",
print nmi, "minutes et",
print nsr, "secondes."

```

3.

```

# affichage des 20 premiers termes de la table par 7,
# avec signalement des multiples de 3 :

i = 1          # compteur : prendra successivement les valeurs de 1 à 20
while i < 21:
    # calcul du terme à afficher :
    t = i * 7
    # affichage sans saut à la ligne (utilisation de la virgule) :
    print t,
    # ce terme est-il un multiple de 3 ? (utilisation de l'opérateur modulo) :
    if t % 3 == 0:
        print "*",      # affichage d'une astérisque dans ce cas
    i = i + 1          # incrémentation du compteur dans tous les cas

```

4. Réfléchissez !

5. Réfléchissez !

L'instruction "for"

La boucle "for" permet d'exécuter une itération un certain nombre de fois :

```

>>> for i in range(5) :
...     print(i)
1
2
3
4
5

```

Pour le "foreach" (répétition pour chaque élément d'un itérable) :

```

>>> for champ in ['champ1', 'champ2', 'champ3']:
...     print(champ)
champ1
champ2
champ3

```

Les instructions "break", "continue"

L'instruction "break" permet d'arrêter une boucle avant sa fin. L'instruction "continue" est similaire, mais au lieu d'interrompre la boucle, elle permet de

passer à l'itération suivante.

```
for i in range(5):
    if i==3:
        break
    print i
```

affichera

```
0 1 2
```

tandis que

```
for i in range(5):
    if i==3:
        continue
    print i
```

affichera

```
0 1 2 4
```

Types

Typage des variables

Pour les non-programmeurs

Pour distinguer les uns des autres ces divers contenus possibles des variables, le langage de programmation fait usage de différents types de données (ex : le type 'entier', le type 'réel', le type 'chaîne de caractères', le type 'liste', etc.).

Python est un langage à typage dynamique. Cela signifie que bien que gérant différents types, lorsqu'une variable est affectée, l'interpréteur trouvera automatiquement son type sans que l'utilisateur soit contraint de le préciser. Ceci constitue une particularité intéressante de Python, qui le rattache à une famille particulière de langages où l'on trouve aussi par exemple *Lisp*, *Scheme*, et quelques autres.

Par opposition, le typage statique est préférable dans le cas des langages compilés comme 'C++ et Java, *parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire « figé »).* Dans ces langages, il faut toujours - par des instructions distinctes - d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

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

Liste des types

Liste des types	
<i>int</i>	Nombre entier optimisé
<i>long</i>	Nombre entier de taille arbitraire
<i>float</i>	Nombre à virgule flottante
<i>complex</i>	Nombre complexe

<i>str</i>	Chaîne de caractère
<i>unicode</i>	Chaîne de caractère unicode
<i>tuple</i>	Liste de longueur fixe
<i>list</i>	Liste de longueur variable
<i>dict</i>	dictionnaire
<i>file</i>	Fichier
<i>bool</i>	Booléen
<i>NoneType</i>	Absence de type
<i>NotImplementedType</i>	Absence d'implémentation
<i>function</i>	fonction
<i>module</i>	module

Tableau 2 : Liste des types prédéfinis en Python

Tableau des types Python 3.x ayant changé de nom depuis 2.x

nom	description	équivalent 2.x
int	nombre entier de longueur arbitraire	long
byte	chaîne de caractères ASCII	str
str	chaîne de caractères Unicode	unicode

Détermination d'un type

La fonction `type()` permet de connaître le type d'une variable.

```
>>> a=3
>>> type(a)
<type 'int'>
```

Exemple 1 : utilisation de la fonction `type`

Conversion des types

Il existe plusieurs fonctions qui permettent de forcer le type d'une variable en un autre type.

- `int()` : permet de modifier une variable en entier. Provoque une erreur si cela n'est pas possible.
- `long()` : transforme une valeur en long.
- `str()` : permet de transformer la plupart des variables d'un autre type en chaînes de caractère.
- `float()` : permet la transformation en flottant.
- `repr()` : similaire à "str". Voir la partie sur les objets
- `eval()` : évalue le contenu de son argument comme si c'était du code Python.
- `split()` : sépare une chaîne en liste.
- `join()` : transforme une liste en chaîne, dont le séparateur est en préfixe ("`.join(MaListe)`").
- `datetime.strptime('20170727222900', '%Y%m%d%H%M%S')` : change une chaîne en date.
- `maDate.strftime('%Y%m%d%H%M%S')` : change une date en chaîne.

Numériques

Il existe deux types pour définir des nombres entiers : le type **int** et le type **long**. Il existe également un type pour représenter des nombres à virgule : le type **float**.

Les nombres entiers de type *int*

Les int représentent le type le plus facilement représentable sur une architecture donnée. Par exemple, sur une machine 32-bits, la taille d'un int sera de 32-bit, donc un int permettra de représenter des nombres entre -2^{31} et $2^{31} - 1$, soit entre -2 147 483 648 et 2 147 483 647.

Un littéral int s'écrit tout simplement avec les chiffres de 0 à 9, précédé éventuellement du symbole -. Il est possible d'écrire ce littéral suivant trois bases :

- la base décimale : le littéral devra commencer par un chiffre entre 1 et 9
- la base octale (base 8) : le littéral devra commencer par 0 suivi de chiffres de 0 à 7
- la base hexadécimale (base 16): le littéral devra commencer par 0x suivi de chiffres de 0 à 9 et de lettres de A à F (en minuscule ou majuscule)

```
x=1
x=0
x=-33
x=4566
x=2147483647
x=076 #équivalent à x=62
x=0xFF #équivalent à x=255
x=0xa1 #équivalent à x=161
```

Exemple 2 : Quelques entiers int'

Les nombres entiers de type *long*

Un entier long est un entier dont la taille n'est limitée que par la mémoire allouée par l'ordinateur à l'interpréteur Python. C'est à dire qu'un long permet d'écrire des entiers aussi grands que l'on veut.

Il existe deux manières d'utiliser des long :

- il faut rajouter L ou l à la fin d'un littéral entier pour qu'il soit automatiquement long
- lorsque le résultat d'une opération dépasse la capacité de stockage d'un int, alors, ce résultat est automatiquement convertit en long

```
x=1L
x=-45l
x=1212121212121212121212121212121 #Automatiquement converti en long
x=2147483647+1
```

Exemple 3 : Quelques entiers long'

Il n'est pas nécessaire d'utiliser le type long de manière systématique : pour les entiers de taille raisonnable, le type int est beaucoup plus optimisé.

Pour convertir un long en int (et inversement), il est possible d'utiliser les fonctions int() et long().

```
x = int(1L) #x est un int
x = long(1) #x est un long
x = int(1212121212121212121) #x est quand même un long
```

Exemple 4 : Utilisation des fonctions int() et long()

Limite entre « integer » et « long »

 Pour les non-programmeurs

Les lignes incluses dans le paragraphe d'une boucle while sont exécutées répétitivement, tant que sa condition est remplie.

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

```
>>> a, b, c = 1, 1, 1
>>> while c<49:
    print c, " : ", b, type(b)
    a, b, c = b, a+b, c+1
...
... (affichage des 43 premiers termes)
...
44 : 1134903170 <type 'int'>
```

```

45 : 1836311903 <type 'int'>
46 : 2971215073 <type 'long'>
47 : 4807526976 <type 'long'>
48 : 7778742049 <type 'long'>

```

Que pouvons-nous constater ?

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

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

L'exercice que nous venons de réaliser indique cependant qu'il se passe « quelque chose » lorsque ces nombres deviennent très grands. Au début du programme, les variables `a`, `b` et `c` sont définies implicitement comme étant du type *integer*. C'est ce qui se passe toujours avec Python lorsqu'on affecte une valeur entière à une variable, à condition que cette valeur ne soit pas trop grande. Dans la mémoire de l'ordinateur, ce type de donnée est en effet encodé sous la forme d'un bloc de 4 octets (ou 32 bits). Or la gamme de valeurs décimales qu'il est possible d'encoder sur 4 octets seulement s'étend de -2147483648 à + 2147483647 (Voir cours d'informatique générale).

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

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

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

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

Ce type *long* permet l'encodage de valeurs entières avec une précision quasi infinie : une valeur définie sous cette forme peut en effet posséder un nombre

de chiffres significatifs quelconque, ce nombre *n'étant limité que par la taille de la mémoire disponible sur l'ordinateur utilisé !*

Exemple :

```

>>> a, b, c = 3, 2, 1
>>> while c < 15:
    print c, ": ", b
    a, b, c = b, a*b, c+1
1 : 2
2 : 6
3 : 12
4 : 72
5 : 864
6 : 62208
7 : 53747712
8 : 3343537668096
9 : 179707499645975396352
10 : 600858794305667322270155425185792
11 : 107978831564966913814384922944738457859243070439030784
12 : 64880030544660752790736837369104977695001034284228042891827649456186234
582611607420928
13 : 70056698901118320029237641399576216921624545057972697917383692313271754
88362123506443467340026896520469610300883250624900843742470237847552
14 : 45452807645626579985636294048249351205168239870722946151401655655658398
64222761633581512382578246019698020614153674711609417355051422794795300591700
96950422693079038247634055829175296831946224503933501754776033004012758368256
>>>

```

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

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

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

Les nombres à virgule flottante (*float*)

Un nombre à virgule flottante est un nombre décimal qu'il est possible de représenter par sa mantisse et son exposant. Par exemple, le nombre 125,789 est représentable par le couple (mantisse = 1,25789, exposant = 2). La mantisse étant toujours comprise entre -10 et 10 exclus.

Les nombres sont traduits par la formule *nombre = mantisse * 10^{exposant}*.

Les limites dépendent de l'architecture de la machine et sont équivalentes au type de donnée double du langage C.

Les littéraux peuvent s'écrire avec les chiffres, le caractère point pour indiquer la séparation entre la partie entière et la partie décimale et la lettre 'e' ou 'E' pour spécifier l'exposant.

```
x = 1.234
x = 1.0 #Notons qu'un entier peut être un flottant
x = 1. #Même résultat que précédemment
x = 1.234e54 #C'est à dire <math>1.234*10^{54}</math>
x = 1.234E54 #idem
x = -1.454e-2 #La mantisse et l'exposant peuvent être négatifs;
```

Exemple 5 : nombres à virgules

Essayons donc ce type de données dans un nouveau petit programme :

```
>>> a, b, c = 1., 2., 1 # => a et b seront du type 'float'
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print b
2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
      Inf
      Inf
```

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

Le type *float* utilisé dans notre exemple permet de manipuler des nombres (positifs ou négatifs) compris entre 10e-308 et 10e+308 avec une précision de

12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

Exercices

1. Écrivez un programme qui convertisse en radians un angle fourni au départ en degrés, minutes, secondes.
2. Écrivez un programme qui convertisse en degrés, minutes, secondes un angle fourni au départ en radians.
3. Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse. La formule de conversion est : $T_F = T_C * 1,8 + 32$
4. Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %
5. 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 de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :
 - le nombre exact de grains (nombre entier)
 - le nombre de grains en notation scientifique (nombre réel)

Solution (cliquez)

```

1. # Conversion degrés -> radians
# Rappel : un angle de 1 radian est un angle qui correspond à une portion
# de circonférence de longueur égale à celle du rayon.
# Puisque la circonférence vaut 2 pi R, un angle de 1 radian correspond
# à 360° / 2 pi , ou encore à 180° / pi
.
# Angle fourni au départ en degrés, minutes, secondes :
'deg, min, sec = 32, 13, 49
.
# Conversion des secondes en une fraction de minute :
# (le point décimal force la conversion du résultat en un nombre réel)
'fm = sec/60.
# Conversion des minutes en une fraction de degré :
'fd = (min + fm)/60
# Valeur de l'angle en degrés "décimalisés" :
'ang = deg + fd
# Valeur de pi :
'pi = 3.14159265359
# Valeur d'un radian en degrés :
.

```

```
rad = 180 / pi
# Conversion de l'angle en radians :
arad = ang / rad
# Affichage :
print deg, "°", min, "'", sec, "' =", arad, "radian(s)"
```

2. Réfléchissez !

3. # Conversion °Fahrenheit <-> °Celsius

```
# A) Température fournie en °C :
tempC = 25
# Conversion en °Fahrenheit :
tempF = tempC * 1.8 + 32
# Affichage :
print tempC, "°C =", tempF, "°F"

# B) Température fournie en °F :
tempF = 25
# Conversion en °Celsius :
tempC = (tempF - 32) / 1.8
# Affichage :
print tempF, "°F =", tempC, "°C"
```

4. Réfléchissez !

```
5. >>> a, b = 1, 1 # variante : a, b = 1., 1
>>> while b<65:
...     print b, a
...     a,b = a*2, b+1
... 
```

Les nombres complexes

Python est un des rares langages à proposer un type de base pour les nombres complexes \mathbb{C} . Un nombre complexe est un nombre composé d'une partie réelle et d'une partie imaginaire. On note i une des racines du polynôme $x^2 = -1$.

Dans certains contextes, (comme la physique), la racine de -1 est notée j . C'est ce qui se fait en Python.

Un littéral complexe s'écrit donc : $a + bj$, avec a et b des variables de type float. Attention, j doit être précédé d'un nombre car sinon, Python l'interprétera comme étant la variable j . Il ne doit pas y avoir d'espace entre ce nombre et j .

```
x = 1 + 1j
x = 1.2e3 + 1.5e7j
x = 5j + 4
x = 1 + x*1j
```


Exemple 5 : quelques nombres complexes

Booléens

Définition

Un booléen est un type de données qui ne peut prendre que deux valeurs : vrai ou faux. En Python, les constantes littérales sont notées *True* et *False*.

Tous les types de variables peuvent être interprétés de manière booléenne. Par exemple, pour les entiers (int), la valeur "0" correspond à "faux" et les autres valeurs à "vrai". Il en est de même pour tous les autres types : une valeur particulière vaut *False* et le reste des valeurs *True*. Le tableau suivant présente les valeurs "faux" pour les principaux type de données.

Exemple d'expressions booléennes

```
a = 6
b = 7
c = 42
print(1, a == 6)
print(2, a == 7)
print(3, a == 6 and b == 7)
print(4, a == 7 and b == 7)
print(5, not a == 7 and b == 7)
print(6, a == 7 or b == 7)
print(7, a == 7 or b == 6)
print(8, not (a == 7 and b == 6))
print(9, not a == 7 and b == 6)
```

Affiche :

```
1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False
```

Que se passe-t-il ? Le programme consiste en un tas de `print`. Chaque `print` affiche un nombre et une expression. Le nombre sert à savoir de quel ligne on parle. Notez bien comme chaque expression finit soit par `False` ou `True`. En python `False` et `True` peuvent aussi être écrits 0 et 1.

And

Voyez ce code :

```
print(1, a == 6)
print(2, a == 7)
```

Cela affiche respectivement a True et a False comme attendu, le premier est vrai, et le second est faux. La troisième ligne, `print(3, a == 6 and b == 7)`, est un peu différente. L'opérateur `and` signifie que si les deux expressions qui l'entourent sont vraies alors toute l'expression l'est également, autrement elle est fausse. Le comportement de `and` peut être résumé comme suit :

Expression	Résultat
True and True	True
True and False	False
False and True	False
False and False	False

Notez que si la première expression est fausse Python ne vérifie pas la seconde car il sait que l'expression est fausse. Essayez de lancer `False and print("Salut")` et comparez le à `True and print("Salut")`. Le terme technique pour ceci est *short-circuit evaluation* (littéralement "évaluation en circuit court").

Not

La ligne suivante, `print(5, not a == 7 and b == 7)`, utilise l'opérateur `not`. `not` retourne l'opposé d'une expression (l'expression pourrait être réécrite `print(5, a != 7 and b == 7)`). Voici la table de vérité :

Expression	Résultat
not True	False
not False	True

Or

Les deux lignes suivantes, `print(7, a == 7 or b == 7)` et `print(7, a == 7 or b == 6)` utilisent l'opérateur `or`. `or` retourne vrai si la première expression est vraie ou si la seconde expression est vraie, si la seconde expression est vraie ou si les deux le sont. Si aucune n'est vraie il retourne faux. Voici sa table de vérité :

Expression	Résultat
True or True	True
True or False	True
False or True	True
False or False	False

Notez que si la première expression est vraie Python ne vérifie pas la seconde expression car il sait que toute l'expression est vraie. Cela marche car `or` est vrai si au moins une des expressions est vraie. La première partie est vraie donc la seconde pourrait être vraie ou fausse, mais l'expression est toujours vraie.

Les deux lignes suivantes, `print(8, not (a == 7 and b == 6))` et `print(9, not a == 7 and b == 6)`, montrent que les parenthèses peuvent être utilisées pour regrouper des expressions et forcer l'évaluation d'une partie en premier. Notez que les parenthèses ont changé l'expression de "faux" à "vrai". La raison est que les parenthèses ont forcé le `not` à s'appliquer à toute l'expression au lieu de seulement la partie `a == 7`.

bool()

La valeur booléenne d'une variable "x" peut être obtenue avec `bool(x)`. Voici la table de vérité selon le type de la variable :

Vrai	Faux
True	False
1	0
Nombre (positif ou négatif)	None
Chaine non vide	Chaine vide
Liste non vide	Liste vide
Dictionnaire non vide	Dictionnaire vide

Pour tester si une variable "x" n'est pas nulle, on utilise : `not x is None`.

Note sur les opérateurs booléens



Attention !

`x == ('a' or 'b')` n'est pas équivalent à `x == 'a' or x == 'b'`. En effet, cela ne vérifie pas si x est équivalent aux caractères 'a' ou 'b', car l'expression booléenne s'arrête avant la fin par *short-circuit evaluation*.

Exemple :

```

>>> 'a' == ('a' or 'b') # 'a' = True, donc ('a' or 'b') = 'a' sans avoir à évaluer 'b'.
True
>>> 'b' == ('a' or 'b') # Revient à : 'b' == 'a', de la même manière
False
    
```

```
>>> 'a' == ('a' and 'b') # 'a' = True et 'b' = True, donc ('a' and 'b') = 'b' (dernière à False
False
>>> 'b' == ('a' and 'b') # Revient à : 'b' == 'b', de la même manière
True
```

Véracité/fausseté d'une expression

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

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

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

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

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

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

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

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

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

Ces explications ne sont qu'une première information à propos d'un système de représentation des opérations logiques de l'*algèbre de Boole*.

Exemple

Créer le fichier *password1.py* :

```
#!/usr/bin/python
# -*- coding:Utf-8 -*-
# Ce programme demande à l'utilisateur son nom et son mot de passe, puis il les vérifie

name = input("What is your name? ")
password = input("What is the password? ")
if name == "Josh" and password == "Friday":
    print("Welcome Josh")
elif name == "Fred" and password == "Rock":
    print("Welcome Fred")
else:
    print("I don't know you.")
```

Résultats :

```
What is your name? Josh
What is the password? Friday
Welcome Josh

What is your name? Bill
What is the password? Money
I don't know you.
```

Exercices

Écrire un programme devine votre nom, avec seulement trois chances avant de terminer.

Solution (cliquez)

```
#!/usr/bin/python
# -*- coding:Utf-8 -*-

print("Try to guess my name!")
count = 1
name = "guilherme"
guess = input("What is my name? ")
while count < 3 and guess.lower() != name:    # .lower permet ici d'ignorer les majuscules
    print("You are wrong!")
    guess = input("What is my name? ")
    count = count + 1
```

```
if guess.lower() != name:
    print("You are wrong!")
    print("You ran out of chances.")
else:
    print("Yes! My name is"), name + "!"
```

Chaines de caractères

Les données alphanumériques

Pour les non-programmeurs

Les chaînes de caractères ("string" en anglais) sont des listes de caractères. On parle de chaînage car les caractères se suivent et chaque caractère a sa place, comme les maillons d'une chaîne. Il est ainsi possible de faire apparaître plusieurs fois le même caractère dans une chaîne, afin de manipuler des phrases par exemple.

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 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*.

Le type « string »

Sous Python, une donnée de type *string* est une suite quelconque de caractères délimitée soit par des apostrophes (*simple quotes*), soit par des guillemets (*double quotes*), soit par des *triples quotes* (''' ou ''').

Exemples :

```
>>> phrase1 = 'les œufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print(phrase2), phrase3, phrase1
'"Oui", répondit-il, j'aime bien les œufs durs.
print("""1, " 2", \n 3,
4""")
```

Résultat :


```
1, " 2",
3,
4
```

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

Accès aux caractères individuels d'une chaîne

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

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

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

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

Exemple :

```
>>> chaine = "Stéphanie"
>>> print(chaine[0], chaine[3])
S p
```

Taille des chaînes

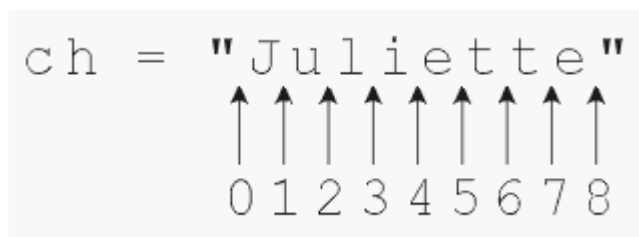
- Déterminer la longueur (c'est-à-dire le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée `len()` :

```
>>> print(len(c))
29
```

Sous-chainnes

L'opérateur d'indiaçage ([]) permet aussi de sélectionner des sous-chainnes selon leurs indices. On appelle cette technique le *slicing* (« découpage en tranches »).

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`.

De plus, si la borne de départ est le premier caractère (indice 0) ou si celle d'arrivée est la dernière (indice égal à la longueur de la chaîne), il devient facultatif. Exemple :

```

>>> chain = "123456789"
>>> print(chain[1:3])
23
>>> print(chain[0:-1])
12345678
>>> print(chain[: -1])
12345678
>>> print(chain[1:len(chain)])
23456789
>>> print(chain[1:])
23456789
  
```

Pour vérifier le début et la fin d'une chaîne sans avoir à déterminer sur quelle longueur, on peut utiliser `startswith()` et `endswith()`, mais c'est moins rapide car faisant appel à une fonction de plus haut niveau. (Référence nécessaire)

```

if chain.startswith('1234'):
    print('ok')
if chain.endswith('89'):
    print('ok')
  
```

Concaténation

L'opération d'assembler plusieurs petites chaînes pour en construire une plus grande s'appelle *concaténation* et on la réalise sous Python à l'aide de l'opérateur "+". Exemple :

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

```
petit poisson deviendra grand
```

Pour convertir en nombre véritable une chaîne de caractères qui représente un nombre. Exemple :

```
>>> chaine = '8647'  
>>> print(chaine + 45)  
==> *** erreur *** on ne peut pas additionner une chaîne et un nombre  
  
>>> print(int(chaine) + 65)  
8712 # OK : on peut additionner deux nombres  
  
>>> print(chaine + str(65))  
864765 # OK : on peut concaténer deux chaînes
```

Dans cet exemple, la fonction intégrée `int()` convertit la chaîne en nombre entier, et `str()` convertit l'entier en chaîne.

Parcours d'une séquence : for... in...

Il arrive très souvent que l'on doive traiter l'intégralité d'une chaîne caractère par caractère, du premier jusqu'au dernier, pour effectuer à partir de chacun d'eux une opération quelconque. Nous appellerons cette opération un *parcours*. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours 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 effectivement traité tous les caractères de la chaîne, puisque ceux-ci sont

indités de *zéro* à 9.

Le parcours d'une séquence est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée, basée sur le couple d'instructions `for ... in ...` :

Avec ces instructions, le programme ci-dessus devient :

```
nom = 'Jacqueline'
for character in nom:
    print(character + ' *'),
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter une variable spécifique (un « compteur ») pour gérer l'indice du caractère que vous voulez traiter à chaque itération. 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 traite successivement tous les éléments d'une séquence donnée*. Dans l'exemple ci-dessus, la séquence était une chaîne de caractères. L'exemple ci-après démontre que l'on peut appliquer le même traitement aux listes :

```
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 donc 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'est-à-dire qu'il est inutile de la définir au préalable), et son type est automatiquement adapté à celui de l'élément de la séquence qui est en cours de traitement.

Exemple :

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

```
print(e)
```

L'exécution de ce script donne :

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

Bien que les éléments de la liste divers 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

1. Écrivez un script qui détermine si une chaîne contient ou non le caractère « e ».
2. Écrivez un script qui compte le nombre d'occurrences du caractère « e » dans une chaîne.
3. Écrivez un script qui recopie une chaîne (dans une nouvelle variable), en insérant des astérisques entre les caractères.
Ainsi par exemple, « gaston » devra devenir « g*a*s*t*o*n »
4. Écrivez un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant.
Ainsi par exemple, « zorglub » deviendra « bulgroz ».
5. En partant de l'exercice précédent, écrivez un script qui détermine si une chaîne de caractères donnée est un palindrome (c'est-à-dire une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple « radar » ou « s.o.s ».

Solution (cliquez)

1. *# Recherche d'un caractère particulier dans une chaîne*

```
# Chaîne fournie au départ :
chain = "Monty python flying circus"
# Caractère à rechercher :
letter = "e"
# Recherche proprement dite :
i = 0 # indice du caractère en cours d'examen
found = False # "drapeau" à lever si le caractère recherché est présent
while i < len(chain):
    if chain[i] == letter:
        found = True
    i = i + 1

# Affichage :
print(u"Le caractère"), letter # Le caractère e
```

```

if t == 1:
    print(u"est présent")
else:
    print(u"est introuvable")
print(u"dans la chaîne"), chain # dans la chaîne Monty python flying circus
    
```

2. Réfléchissez !

3. # Insertion d'un caractère d'espacement dans une chaîne

```

# Chaîne fournie au départ :
ch = "Gaston"
# Caractère à insérer :
cr = "*"
# Le nombre de caractères à insérer est inférieur d'une unité au
# nombre de caractères de la chaîne. On traitera donc celle-ci à
# partir de son second caractère (en omettant le premier).
lc = len(ch) # nombre de caractères total
i = 1 # indice du premier caractère à examiner (le second, en fait)
nch = ch[0] # nouvelle chaîne à construire (contient déjà le premier caractère)
while i < lc:
    nch = nch + cr + ch[i]
    i = i + 1
# Affichage :
print(nch)
    
```

4. # Inversion d'une chaîne de caractères

```

# Chaîne fournie au départ :
ch = "zorglub"
lc = len(ch) # nombre de caractères total
i = lc - 1 # le traitement commencera à partir du dernier caractère
nch = "" # nouvelle chaîne à construire (vide au départ)
while i >= 0:
    nch = nch + ch[i]
    i = i - 1
# Affichage :
print(nch)
    
```

5. Réfléchissez !

Exercices

1. 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'
 - suffixe = 'ack'
 Si vous utilisez une instruction `for... in...`, votre script ne devrait comporter que deux lignes.
2. Rechercher le nombre de mots contenus dans une phrase donnée.

Solution (cliquez)

```

1. prefixes, suffixe = "JKLMNOP", "ack"

for p in prefixes:
    print p + suffixe

2.
# Comptage du nombre de mots dans "chain"
chain = "Les petits ruisseaux font les grandes rivières"
if len(chain) == 0:
    print(0)
nm = 1 # la chaîne comporte au moins un mot
for c in chain:
    if c == " ": # il suffit de compter les espaces
        nm = nm + 1
print(nm)

```

Appartenance d'un élément à une séquence : **if.. in...**

L'instruction `in` peut être utilisée indépendamment de `for`, pour vérifier si un élément donné fait partie ou non d'une séquence. Vous pouvez par exemple vous servir de `in` pour vérifier si tel caractère alphabétique fait partie d'un groupe bien déterminé :

```

car = "e"
voyelles = "aeiouyAEIOUY"
if car in voyelles:
    print car, "est une voyelle"

```

D'une manière similaire, vous pouvez vérifier l'appartenance d'un élément à une liste :

```

n = 5
nombrePremiers = [1, 2, 3, 5, 7, 11, 13, 17]
if n in nombrePremiers:
    print n, "fait partie de notre liste de nombres premiers"

```

Cette instruction très puissante effectue donc à elle seule un véritable parcours de la séquence. À titre d'exercice, écrivez les instructions qui effectueraient le même travail à l'aide d'une boucle classique utilisant l'instruction `while`.

Exercices

1. Écrivez un script qui affiche « vrai » si un caractère est un chiffre.
2. Écrivez un script qui affiche « vrai » si un caractère est une majuscule.

Solution (cliquez)

```
if character in "0123456789":
    print True
else:
    print False
```

1.

```
if character in "ABCDEFGHIJKLMNPOQRSTUVWXYZ":
    print True
else:
    print False
```

Les chaînes sont des séquences non modifiables

Vous ne pouvez pas modifier le contenu d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur [] dans la partie gauche d'une instruction d'affectation. Essayez par exemple d'exécuter le petit script suivant (qui cherche à remplacer une lettre dans une chaîne) :

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

Au lieu d'afficher « Bonjour à tous », ce script « lève » une erreur du genre : `TypeError: object doesn't support item assignment`. Cette erreur est provoquée à la deuxième ligne du script. On y essaie de remplacer une lettre par une autre dans la chaîne, mais cela n'est pas permis.

Par contre, le script ci-dessous fonctionne :

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

Dans cet autre exemple, en effet, nous ne modifions pas la chaîne "salut". Nous en recréons une nouvelle avec le même nom à la deuxième ligne du script (à partir d'un morceau de la précédente, soit, mais qu'importe : il s'agit bien d'une *nouvelle* chaîne).

Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (c'est-à-dire les instructions `if ... elif ... else`) fonctionnent aussi avec les chaînes de caractères. Cela 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.

Encodage

Pour encoder une chaîne en Unicode, il faut la préfixer par la lettre "u"^[1] :

```
>>> string = u"il était une fois"  
>>> print(string)  
'il était une fois'
```

Au sujet des chaînes de caractères, il est important de savoir que de notions différentes de chaînes de caractères existent en Python.

Pour des raisons historiques dans les versions de Python inférieures à Python 2, on considérait une chaîne de caractère comme une chaîne d'octets. Ceci permettait d'avoir un seul type pour traiter de deux notions différentes, mais pouvait parfois engendrer de la confusion sur le type traité.

- Le type String contenait à la fois une séquence d'octet et une chaîne de caractères ASCII/UTF-8 (exemple : "Hellow World").

Depuis la version 3 de Python, les types chaînes d'octet et chaîne de caractères sont clairement différenciées.

- Le type chaîne de caractères (String) contient des caractères Unicode (exemple: 'xyzzzy', "frobozz")
- Les octets et tableaux d'octets (objets Bytes et bytearray) ne contiennent que des octets (l'un est immutable alors que l'autre est mutable). Exemple : b'xyzzzy'.

Classement des caractères

Il est souvent utile de pouvoir déterminer si tel caractère extrait d'une chaîne est une lettre majuscule ou minuscule, ou plus généralement encore, 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. Une première possibilité consiste à utiliser l'instruction "in". Mais puisque nous savons désormais que les caractères forment une suite bien ordonnée dans le code ASCII, nous pouvons exploiter d'autres méthodes. Par exemple, pour déterminer si on a affaire à une minuscule :

```
if 'a' <= character <= 'z' :  
    print('bas de casse')  
else:  
    print('haut de casse')
```

Exercices

1. Écrivez un script qui affiche « vrai » si l'argument transmis est un chiffre (avec une autre méthode que celles exploitées précédemment)
2. Écrivez un script qui affiche « vrai » si un caractère est une majuscule

Solution (cliquez)

```
1. if character >= "0" and character <= "9":  
    print True  
else:  
    print False
```

```
2. if character >= "A" and character <= "Z":  
    print True  
else:  
    print False
```

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(character)` accepte n'importe quel caractère comme argument. En retour, elle fournit le code ASCII correspondant à ce caractère. Ainsi `ord('A')` renvoie la valeur 65, et `ord('é')` 233.
- La fonction `chr(number)` 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)` renvoie le caractère A.

Exercices

1. 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.
2. À partir des relations trouvées, écrivez un script qui convertit tous les

- caractères d'une phrase donnée en capitales.
3. À partir des mêmes relations, écrivez un script qui convertit tous les caractères d'une phrase en capitales.
 4. Écrivez un script qui compte le nombre de fois qu'apparaît tel caractère (fourni en argument) dans une phrase donnée.
 5. Écrivez un script qui affiche le nombre de voyelles contenues dans une phrase donnée.

Solution (cliquez)

```
1. # Table des codes ASCII :
c = 32          # Premier code ASCII <imprimable>
while c < 128 : # caractères non accentués seulement
    print("Code"), c, ":", chr(c), " ",
    c = c + 1
```

2. Réfléchissez !

```
3. # Convertir majuscules -> minuscules et inversement :
# Échange les majuscules et les minuscules de "chaîne"
chain = "Ferdinand-Charles de CAMARET"
newChain = ""          # chaîne à construire
for car in chain:
    code = ord(car)
    if car >= "A" and car <= "Z":
        code = code + 32
    elif car >= "a" and car <= "z":
        code = code - 32
    newChain = newChain + chr(code)
print(newChain)
```

4. Réfléchissez !

```
5. # Comptage de voyelles :
# Teste si car est une voyelle"
if character in "AEIOUYaeiouy":
    print(True)
else:
    print(False)

# Compte les voyelles présentes dans la chaîne "chaîne"
n = 0
for c in chaîne:
    if voyelle(c):
        n = n + 1
print(n)
```

Méthodes des objets string

Sous Python, les chaînes de caractères sont des objets. On peut donc effectuer de nombreux traitements dessus en utilisant des méthodes appropriées. En voici quelques-unes, choisies parmi les plus utiles. Mais vous pouvez obtenir la

liste complète de toutes les méthodes associées à un objet à l'aide de la fonction intégrée `dir()` *OU* `help(str)` :

```

>>> dir('')
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',

```

- Les fonctions "`str()`" et "`repr()`" (pour *représentation* d'objet) permettent de transformer un objet Python quelconque en chaîne de caractères. Ces deux fonctions sont différentes : "`str('chaîne')`" retournera 'chaîne' tandis que "`repr('chaîne')`" retournera "'chaîne'".

- `index(c)` : retrouve l'index de la première occurrence du caractère "c" dans la chaîne, ou déclenche une erreur si absent (*ValueError: substring not found*).

```

>>> foin = "Portez ce vieux whisky au juge blond qui fume"
>>> print foin.index("w")
16

```

- `find(aiguille)` : cherche la position d'une sous-chaîne `aiguille` dans la chaîne, en partant du début.

```

>>> foin = "Cette leçon vaut bien un fromage, sans doute ?"
>>> aiguille = "fromage"
>>> print foin.find(aiguille)
25

```

- `rfind(aiguille)` : pareil que "find" mais en partant de la fin (reverse).

```

>>> foin = "Cette leçon vaut bien deux fromages, dont un fromage râpé ?"
>>> aiguille = "fromage"
>>> print foin.rfind(aiguille)
46

```

- `count(aiguille)` : compte le nombre de sous-chaînes `aiguille` dans la chaîne.

```

>>> foin = "Le héron au long bec emmanché d'un long cou"
>>> aiguille = 'long'
>>> print foin.count(aiguille)
2

```

- `lower()` : convertit une chaîne en minuscules.

```

>>> phrase ="ATTENTION : Danger !"
>>> print phrase.lower()
attention : danger !

```

- `upper()` : convertit une chaîne en majuscules.

```

>>> phrase = "Merci beaucoup"
>>> print phrase.upper()
MERCI BEAUCOUP
    
```

- `capitalize()` : convertit en majuscule la première lettre d'une chaîne.

```

>>> phrase = "quel beau temps, aujourd'hui !"
>>> print phrase.capitalize()
"Quel beau temps, aujourd'hui !"
    
```

- `swapcase()` : convertit toutes les majuscules en minuscules et vice-versa.

```

>>> phrase = "La CIGALE et la FOURMI"
>>> print phrase.swapcase()
!a cigale ET LA fourmi
    
```

- `strip()` : enlève les espaces éventuels au début et à la fin de la chaîne (trime).

```

>>> phrase = "  Monty Python  "
>>> print phrase.strip()
'Monty Python'
    
```

- `replace(old, new)` : remplace tous les caractères `old` par des caractères `new` dans la chaîne.

```

>>> phrase = "Si ce n'est toi c'est donc ton frère"
>>> print phrase.replace(" ", "_")
'Si_ce_n'est_toi_c'est_donc_ton_frère'
    
```

Dans la plupart de ces méthodes, il est possible de préciser quelle portion de la chaîne doit être traitée, en ajoutant des arguments supplémentaires.

```

>>> print ch9.index("e")           # cherche à partir du début de la chaîne
4                                 # et trouve le premier 'e'
>>> print ch9.index("e",5)        # cherche seulement à partir de l'indice 5
8                                 # et trouve le second 'e'
>>> print ch9.index("e",15)       # cherche à partir du caractère n° 15
29                                # et trouve le quatrième 'e'
    
```

Références

1. <https://docs.python.org/2/tutorial/introduction.html#unicode-strings>

Listes

Déclaration

Les listes sont des séquences, des collections ordonnées d'objets séparés par des virgules. On les déclare avec l'opérateur d'indiciage (`[]`) :

```
>>> maListe = ['a', 'b', 'c']
>>> maListe
['a', 'b', 'c']
```

Lecture

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, à partir de zéro). De plus, le *slicing* (« découpage en tranches ») permet aussi de dégager une sous-liste en précisant deux index correspondant aux bornes de la plage.

Une sortie indicée donne :

```
>>> maListe[0]
'a'
>>> maListe[-1]
'c'
```

Une séquence donne par slicing :

```
>>> maListe[0:2]
['a', 'b']
>>> maListe[1:]
['b', 'c']
```

Types

Les éléments qui constituent une liste peuvent être de types variés :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.142]
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.142]
```

Dans cet exemple, en effet, les premiers éléments sont des chaînes de caractères, puis il y a un entier et un réel. A cet égard, le concept de liste est donc différent des tableaux (*array*) que l'on rencontre les langages de

programmation de bas niveau, où tous les éléments doivent être du même type.

Recherche

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction `in` :

```
print("a" in ["a", "b", "c"])      # True
print(not "a" in ["a", "b", "c"]) # False
```

Modification

Affectation

À la différence de ce qui se passe pour les chaînes, qui constituent un type de données non-modifiables, il est possible de changer les éléments individuels d'une liste :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.142]
>>> jour[5] = jour[5] + 1016
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 1492, 3.142]
```

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.

Exemple en deux dimensions :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', ['samedi matin', 'samedi ap
>>> jour[5][0] = "samedi midi"
>>> jour
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', ['samedi midi', 'samedi ap
```

Ainsi, dans l'exemple ci-dessus on remplace le premier élément d'une liste (n° 0), qui est elle-même l'élément n° 5 d'une autre liste.

Opérations

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]
```

Techniques de slicing avancée pour modifier une liste

Insertion d'un ou plusieurs éléments n'importe où dans une liste :

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] = ["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> mots[5:5] = ['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson', 'ketchup']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur `[]` à la gauche du signe égale pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une « tranche » dans la liste cible (c'est-à-dire deux index réunis par le symbole ":"), et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égale doit lui-même être une liste. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément `mots[1]` n'est pas une liste (c'est la chaîne `fromage`), alors que l'élément `mots[1:3]` en est une.

Suppression / remplacement d'éléments :

```
1 >>> mots[2:5] = [] # [] désigne une liste vide
2 >>> mots
3 ['jambon', 'fromage', 'saucisson', 'ketchup']
4 >>> mots[1:3] = ['salade']
5 >>> mots
6 ['jambon', 'salade', 'ketchup']
7 >>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
8 >>> mots
9 ['jambon', 'mayonnaise', 'poulet', 'tomate']
```

- À la première ligne de cet exemple, nous remplaçons la tranche `[2:5]` par une

liste vide, ce qui correspond à un effacement.

- À 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).
- À la 7e ligne, nous remplaçons une tranche de deux éléments par une autre qui en comporte 3.

Copie

Considérons que vous disposez d'une liste `fable` que vous souhaitez recopier dans une nouvelle variable que vous appellerez `phrase`. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

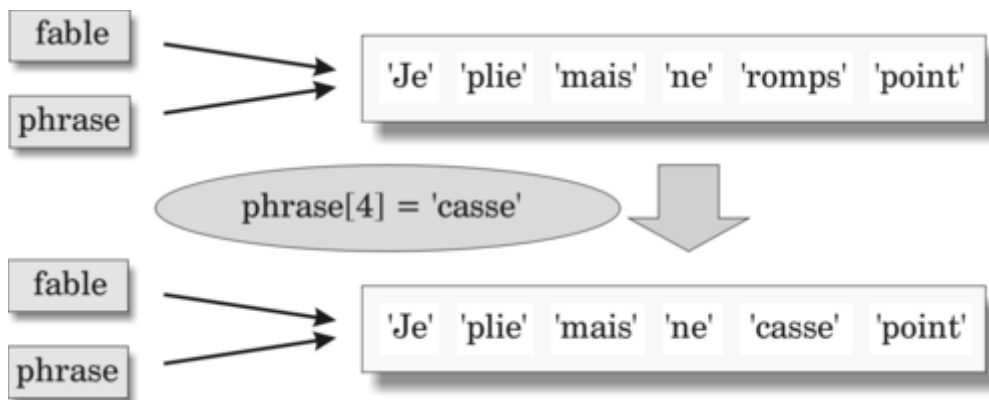
```
>>> phrase = fable
```

En procédant ainsi, sachez que vous ne créez pas une véritable copie. À la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement *une nouvelle référence vers cette liste*. Essayez par exemple :

```
>>> fable = ['Je', 'plie', 'mais', 'ne', 'romps', 'point']
>>> phrase = fable
>>> fable[4] = 'casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si la variable `phrase` contenait une véritable copie de la liste, cette copie serait indépendante de l'original et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable `fable`. Vous pouvez encore expérimenter d'autres modifications, soit au contenu de `fable`, soit au contenu de `phrase`. Dans tous les cas, vous constaterez que les modifications de l'une sont répercutées dans l'autre, et vice-versa.

En fait, les noms `fable` et `phrase` désignent tous deux un seul et même objet en mémoire. Pour décrire cette situation, les informaticiens diront que le nom `phrase` est un *alias* du nom `fable`.



Nous verrons plus tard l'utilité des *alias*. Pour l'instant, nous voudrions disposer d'une technique pour effectuer une véritable copie d'une liste. Avec les notions vues précédemment, vous devriez pouvoir en trouver une par vous-même.

Remarque : python vous autorise à « étendre » une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthésées, ou encore la définition de longues listes, de grands tuples ou de grands dictionnaires. Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction, là où la paire syntaxique est refermée.

Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes. Exemple :

```
couleurs = ['noir', 'brun', 'rouge',
            'orange', 'jaune', 'vert',
            'bleu', 'violet', 'gris', 'blanc']
```

Méthodes

Sous Python, les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de *méthodes* particulièrement efficaces, après l'opérateur "." :

```
>>> 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]

```

append

`append()` (ajouter) : l'exemple suivant qui fait appel à la méthode "append" de l'objet "liste" (déclarée vide) illustre ce concept.

Il s'agit de remplir la hotte du Père Noël.

```

#!/usr/bin/python
# -*- coding: iso8859-1 -*-

# on prépare une liste encore vide
cadeaux = []

# on va ajouter des éléments à la liste
unCadeau = ""

# la suite des éléments à introduire dans la liste est définie par l'utilisateur
# lorsqu'il a terminé, l'utilisateur indique le mot "fin" qui ne sera pas introduit dans
while unCadeau <> "fin":
    unCadeau = raw_input("Père Noël, je voudrais que tu m'apportes ")
    # si l'utilisateur n'a pas frappé "fin", le mot est ajouté à la liste
    if unCadeau <> "fin":
        cadeaux.append(unCadeau)

# finalement, on écrit la suite des éléments introduits dans la liste
for chaqueCadeau in cadeaux:
    print chaqueCadeau

```

split

`split()` : convertit une chaîne en une liste de sous-chaînes. On peut choisir le caractère séparateur en le fournissant comme argument, sinon c'est un espace, par défaut.

```

>>> chaine ="Votez pour moi"
>>> tableau = chaine.split()
>>> print tableau
['Votez', 'pour', 'moi']

>>> chaine = "Cet exemple, parmi d'autres, peut encore servir"
>>> chaine.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']

```

join

`join(liste)` : rassemble une liste de chaînes en une seule (Cette méthode fait donc l'inverse de la précédente). Attention : la chaîne à laquelle on applique cette méthode est celle qui servira de séparateur (un ou plusieurs caractères) ; l'argument transmis est la liste des chaînes à rassembler.

```
>>> tableau = ["Salut", "les", "copains"]
>>> print " ".join(tableau)
Salut les copains
>>> print "__".join(tableau)
Salut__les__copains
```

pop

`pop(i)` retire et renvoie l'élément de la liste d'index *i*. Si *i* est vide, il traite le dernier élément.

```
>>> list = [1, 2, 3, 4]
>>> print(list.pop(0))
1
>>> list
[2, 3, 4]
>>> print(list.pop())
4
>>> list
[2, 3]
```

Fonctions

len

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

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

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)`.

del

Une autre *fonction intégrée* permet de supprimer d'une liste un élément ou plusieurs éléments quelconques à partir de leurs index. Il s'agit de la fonction `del()` :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 476, 3.142]
>>> del(jour[5:7])
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> del(jour[0])
>>> print(jour)
['mardi', 'mercredi', 'jeudi', 'vendredi']
```

Remarque : notez bien la différence entre la méthode `remove()` et la fonction `del()` : `del` travaille avec un indice ou une tranche d'indices, tandis que `remove()` recherche *une valeur* (si plusieurs éléments de la liste possèdent la même valeur, seul le premier est effacé).

min et max

- `min(liste)` : minimum d'une liste (élément ayant la plus petite valeur).
- `max(liste)` : maximum d'une liste.

```
>>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> min(jour)
'jeudi'
>>> max(jour)
'vendredi'
```

range

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de cette fonction :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La fonction `range()` génère une liste de nombres entiers de valeurs croissantes. Si vous appelez `range()` avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant à partir de zéro (*c'est-à-dire que `range(n)` génère les nombres de 0 à $n-1$*).

Notez bien que l'argument fourni n'est jamais dans la liste générée.

On peut aussi utiliser `range()` avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques :

```
>>> range(5,13)
[5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3,16,3)
[3, 6, 9, 12, 15]
```

Si vous avez du mal à assimiler l'exemple ci-dessus, considérez que `range()` attend toujours trois arguments, que l'on pourrait intituler FROM, TO et STEP. FROM est la première valeur à générer, TO est la dernière (ou plutôt la dernière + un), et STEP le « pas » à sauter pour passer d'une valeur à la suivante. S'ils ne sont pas fournis, les paramètres FROM et STEP prennent leurs valeurs par défaut, qui sont respectivement 0 et 1.

Parcours d'une liste à l'aide de for, range() et len()

L'instruction `for` est l'instruction idéale pour parcourir une liste :

```
>>> prov = ['La', 'raison', 'du', 'plus', 'fort', 'est', 'toujours', 'la', 'meilleure']
>>> for mot in prov:
    print mot,
La raison du plus fort est toujours la meilleure
```

Il est très pratique de combiner les fonctions `range()` et `len()` pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne). Exemple :

```
fable = ['Maître', 'Corbeau', 'sur', 'un', 'arbre', 'perché']
for index in range(len(fable)):
    print index, fable[index]
```

Mais il existe également une syntaxe adaptée à cela :

```
for (cle, valeur) in enumerate(fable): {retour à la ligne}{tab}print cle, valeur
```

L'exécution de ce script donne le résultat :

```
0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

Tableau à trois lignes :

```
monTableau = range(1, 5)
monTableau[1] = u'ligne 1'
monTableau[2] = u'ligne 2'
monTableau[3] = u'ligne 3'

for ligne in range(1, 4):
```

```
print(monTableau[ligne])
```

Tableau à deux dimensions :

```
ligne = 3
colonne = 2
monTableau = [[0] * (colonne+1) for _ in range(ligne+1)]
monTableau[1][1] = u'1.1'
monTableau[1][2] = u'1.2'
monTableau[2][1] = u'2.1'
monTableau[2][2] = u'2.2'
monTableau[3][1] = u'3.1'
monTableau[3][2] = u'3.2'

for l in range(1, ligne + 1):
    for c in range(1, colonne + 1):
        print(monTableau[l][c])
```

Une conséquence du typage dynamique

Le type de la variable utilisée avec l'instruction "for" est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de "for" sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture. Exemple :

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers:
        print item, type(item)
```

```
3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
Linux is not Windoze <type 'str'>
```

random

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'est-à-dire qui produise un résultat totalement imprévisible). Il existe cependant des techniques mathématiques permettant de *simuler* plus ou moins bien l'effet du hasard. Des livres entiers ont été écrits sur les moyens de produire ainsi un hasard « de

bonne qualité ». Nous n'allons évidemment pas développer ici une telle question, mais rien ne vous empêche de consulter à ce sujet votre professeur de mathématiques.

Dans son module *random*, Python propose toute une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques. Nous n'examinerons ici que quelques-unes d'entre elles. Veuillez consulter la documentation en ligne pour découvrir les autres. Vous pouvez importer toutes les fonctions du module par :

```
>>> from random import *
```

Pour créer une liste de nombres réels aléatoires, de valeur comprise entre zéro et un :

```
>>> s = [0]*tailleListe
      for i in range(tailleListe):
          s[i] = random()
      print(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

1. 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).
2. Écrivez 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.

Solution (cliquez)

1. Réfléchissez !
2. Réfléchissez !

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.

Exercices

1. 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'est-à-dire 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 1/10e 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 œuvre un parcours de la liste des valeurs tirées au hasard (boucle), et effectuer un test sur chacune d'elles pour déterminer dans quelle fraction de l'intervalle 0-1 elle se situe. Incrémenter de une unité le compteur correspondant. f) Lorsque c'est terminé, afficher l'état de chacun des compteurs.

Solution (cliquez)

```
1. # Test du générateur de nombres aléatoires :
from random import random          # tire au hasard un réel entre 0 et 1

n = raw_input("Nombre de valeurs à tirer au hasard (défaut = 1000) : ")
if n == "":
    nVal = 1000
else:
    nVal = int(n)

n = raw_input("Nombre de fractions dans l'intervalle 0-1 (entre 2 et "
              + str(nVal/10) + ", défaut =5) : ")
if n == "":
    nFra = 5
else:
    nFra = int(n)

if nFra < 2:
    nFra = 2
elif nFra > nVal/10:
    nFra = nVal/10

print "Tirage au sort des", nVal, "valeurs ..."
```

```
listVal = [0]*nVal          # créer une liste de zéros
for i in range(nVal):      # puis modifier chaque élément
    listVal[i] = random()

print "Comptage des valeurs dans chacune des", nFra, "fractions ..."
listCompt = [0]*nFra      # créer une liste de compteurs

# parcourir la liste des valeurs :
for valeur in listVal:
    # trouver l'index de la fraction qui contient la valeur :
    index = int(valeur*nFra)
    # incrémenter le compteur correspondant :
    listCompt[index] = listCompt[index] +1

# afficher l'état des compteurs :
for compt in listCompt:
    print compt,
```

Exemple de résultats affichés par un programme de ce type

```
Nombre de valeurs à tirer au hasard (défaut = 1000) : 100
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 10, défaut =5) : 5
Tirage au sort des 100 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
11 30 25 14 20
Nombre de valeurs à tirer au hasard (défaut = 1000) : 10000
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 1000, défaut =5) : 5
Tirage au sort des 10000 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
1970 1972 2061 1935 2062
```

Une bonne approche de ce genre de problème consiste à essayer d'imaginer quelles fonctions simples vous pourriez écrire pour résoudre l'une ou l'autre partie du problème, puis de les utiliser dans un ensemble plus vaste.

Par exemple, vous pourriez chercher à définir d'abord une fonction `numeroFraction()` qui servirait à déterminer dans quelle fraction de l'intervalle 0-1 une valeur tirée se situe. Cette fonction attendrait deux arguments (la valeur tirée, le nombre de fractions choisi par l'utilisateur) et fournirait en retour l'index du compteur à incrémenter (c'est-à-dire le n° de la fraction correspondante). Il existe peut-être un raisonnement mathématique simple qui permette de déterminer l'index de la fraction à partir de ces deux arguments. Pensez notamment à la fonction intégrée `int()`, qui permet de convertir un nombre réel en nombre entier en éliminant sa partie décimale.

Si vous ne trouvez pas, une autre réflexion intéressante serait peut-être de construire d'abord une liste contenant les valeurs « pivots » qui délimitent les fractions retenues (par exemple 0 - 0,25 - 0,5 - 0,75 - 1 dans le cas de 4 fractions). La connaissance de ces valeurs faciliterait peut-être l'écriture de la fonction `numeroFraction()` que nous souhaitons mettre au point.

Si vous disposez d'un temps suffisant, vous pouvez aussi réaliser une version graphique de ce programme, qui présentera les résultats sous la forme d'un histogramme (diagramme « en bâtons »).

Tirage au hasard de nombres entiers

Lorsque vous développerez des projets personnels, il vous arrivera fréquemment de souhaiter pouvoir disposer d'une fonction qui permette de tirer au hasard un nombre entier entre certaines limites. Par exemple, si vous voulez écrire un programme de jeu dans lequel des cartes à jouer sont tirées au hasard (à partir d'un jeu ordinaire de 52 cartes), vous aurez certainement l'utilité d'une fonction capable de tirer au hasard un nombre entier compris entre 1 et 52.

Vous pouvez pour ce faire utiliser la fonction `randrange()` du module `random`.

Cette fonction peut être utilisée avec 1, 2 ou 3 arguments.

Avec un seul argument, elle renvoie un entier compris entre zéro et la valeur de l'argument diminué d'une unité. Par exemple, `randrange(6)` renvoie un nombre compris entre 0 et 5.

Avec deux arguments, le nombre renvoyé est compris entre la valeur du premier argument et la valeur du second argument diminué d'une unité. Par exemple, `randrange(2, 8)` renvoie un nombre compris entre 2 et 7.

Si l'on ajoute un troisième argument, celui-ci indique que le nombre tiré au hasard doit faire partie d'une série limitée d'entiers, séparés les uns des autres par un certain intervalle, défini lui-même par ce troisième argument. Par exemple, `randrange(3, 13, 3)` renverra un des nombres de la série 3, 6, 9, 12 :

```
>>> for i in range(15):
      print random.randrange(3,13,3),
3 12 6 9 6 6 12 6 3 6 9 3 6 12 12
```

Exercices

- Écrivez un script qui tire au hasard des cartes à jouer. Le nom de la carte tirée doit être correctement présenté, « en clair ». Le programme affichera par exemple :
 Frappez <Enter> pour tirer une carte :
 Dix de Trèfle
 Frappez <Enter> pour tirer une carte :
 As de Carreau
 Frappez <Enter> pour tirer une carte :
 Huit de Pique
 Frappez <Enter> pour tirer une carte :
 etc ...

Solution (cliquez)

- `# Tirage de cartes`

```

from random import randrange

couleurs = ['Pique', 'Trèfle', 'Carreau', 'Coeur']
valeurs = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as']

# Construction de la liste des 52 cartes :
carte = []
for coul in couleurs:
    for val in valeurs:
        carte.append("%s de %s" % (str(val), coul))

# Tirage au hasard :
while 1:
    k = raw_input("Frappez <c> pour tirer une carte, <Enter> pour terminer ")
    if k == "":
        break
    r = randrange(52)
    print carte[r]

```

Exercices

Exercices

- Soient les listes suivantes : $t_1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]$
 $t_2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']$ 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 : $['Janvier', 31, 'Février', 28, 'Mars', 31, \text{etc...}]$.
- Créez une liste A contenant quelques éléments. Effectuez une vraie copie de cette liste dans une nouvelle variable B. Suggestion : créez d'abord une liste B de même taille que A mais ne contenant que des zéros. Remplacez ensuite tous ces zéros par les éléments tirés de A.
- Même question, mais autre suggestion : créez d'abord une liste B vide. Remplissez-la ensuite à l'aide des éléments de A ajoutés l'un après l'autre.
- Même question, autre suggestion encore : pour créer la liste B, découpez dans la liste A une tranche incluant tous les éléments (à l'aide de l'opérateur [:]).
- Un *nombre premier* est un nombre qui n'est divisible que par un et par lui-même. Écrivez un programme qui établisse la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du *crible d'Ératosthène* :
 - Créez une liste de 1000 éléments, chacun initialisé à la valeur 1.
 - Parcourez cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés. En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices

pairs : 4, 6, 8, 10, etc. Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Solution (cliquez)

```

1. # Insertion de nouveaux éléments dans une liste existante

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

c, d = 1, 0
while d < 12 :
    t2[c:c] = [t1[d]]          # ! l'élément inséré doit être une liste
    c, d = c+2, d+1
</pre>
</li>
<li>Réfléchissez !</li>
<li>Réfléchissez !</li>
<li>Réfléchissez !</li>
<li>
<source lang=python>
# Crible d'Ératosthène pour rechercher les nombres premiers de 1 à 999

# Créer une liste de 1000 éléments 1 (leurs indices vont de 0 à 999) :
lst = [1]*1000
# Parcourir la liste à partir de l'élément d'indice 2:
for i in range(2,1000):
    # Mettre à zéro les éléments suivants dans la liste,
    # dont les indices sont des multiples de i :
    for j in range(i*2, 1000, i):
        lst[j] = 0

# Afficher les indices des éléments restés à 1 (on ignore l'élément 0) :
for i in range(1,1000):
    if lst[i]:
        print i,

```

Exercices

1. 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 est plus petit que le premier, par exemple, ou bien si le second indice est plus grand que la taille de la chaîne).
2. Découpez une grande chaîne en fragments de cinq caractères chacun. Rassemblez ces morceaux dans l'ordre inverse.
3. Tâchez d'écrire un script qui fera exactement le contraire de ce que fait l'opérateur d'indexage ([]) : au lieu de partir d'un index donné pour retrouver le caractère correspondant, il devra retrouver l'index correspondant à un caractère donné, à partir du nom de la chaîne à traiter et du caractère à trouver. Exemple : pour "Juliette & Roméo", "&" il

devra afficher : 9, ou -1 si le caractère est introuvable.

4. Améliorez le script en lui ajoutant une troisième variable : l'index à partir duquel la recherche doit s'effectuer dans la chaîne. Ainsi par exemple, pour : "César & Cléopâtre", "r", 5, il devra afficher : 15 (et non 4 !)
5. Écrivez un script qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Ainsi, pour "ananas au jus", "a" il devra afficher : 4.

Solution (cliquez)

1. *IndexError: list index out of range*

```

2. # Découpage d'une chaîne en fragments :
chain = "abcdefghijklmnopqrstuvwxyz123456789"
n = 5

# Découpage de "chain" en une liste de fragments de n caractères"
d, f = 0, n # indices de début et de fin de fragment
tt = [] # liste à construire
while d < len(chain):
    if f > len(chain): # on ne peut pas découper au-delà de la fin
        f = len(chain)
    fr = chain[d:f] # découpage d'un fragment
    tt.append(fr) # ajout du fragment à la liste
    d, f = f, f + n # indices suivants
print(tt)

# Rassemble les éléments de la liste tt dans l'ordre inverse"
chain = "" # chaîne à construire
i = len(tt) # on commence par la fin de la liste
while i > 0 :
    i = i - 1 # le dernier élément possède l'indice n - 1
    chain = chain + tt[i]
print(chain)
    
```

```

3. # Rechercher l'indice d'un caractère dans une chaîne
chain, car = "Coucou c'est moi", "z"
start = 0

# Trouve l'indice du caractère car dans "chain"
i = start
while i < len(chain):
    if chain[i] == car:
        print(i) # le caractère est trouvé -> on termine
    i = i + 1
print(-1) # toute la chaîne a été scannée sans succès

# Autres tests :
chain, car = "Juliette & Roméo", "&"
chain, car, start = "César & Cléopâtre", "r", 5
    
```

4. Réfléchissez !
5. Réfléchissez !

Exercices

1. Soient les listes suivantes :

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

Écrivez un petit programme qui crée une nouvelle liste t3. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant : ['Janvier',31,'Février',28,'Mars',31, etc...].

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

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

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

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

4. Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs. Par exemple, si la liste initiale est celle de l'exercice précédent, le programme devra construire une liste pairs qui contiendra [32, 12, 8, 2], et une liste impairs qui contiendra [5, 3, 75, 15]. Astuce : pensez à utiliser l'opérateur modulo (%) déjà cité précédemment.

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

Solution (cliquez)

```
1. # Combinaison de deux listes en une seule

# Listes fournies au départ :
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
      'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
# Nouvelle liste à construire (vide au départ) :
t3 = []
# Boucle de traitement :
i = 0
while i < len(t1):
    t3.append(t2[i])
```



```

        t3.append(t1[i])
        i = i + 1

# Affichage :
print t3

```

2. # Affichage des éléments d'une liste

```

# Liste fournie au départ :
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
      'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
# Affichage :
i = 0
while i < len(t2):
    print t2[i],
    i = i + 1

```

3. # Recherche du plus grand élément d'une liste

```

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
# Au fur et à mesure du traitement de la liste, on mémoriserà dans
# la variable ci-dessous la valeur du plus grand élément déjà trouvé :
max = 0
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] > max:
        max = tt[i]           # mémorisation d'un nouveau maximum
    i = i + 1
# Affichage :
print "Le plus grand élément de cette liste a la valeur", max

```

4. # Séparation des nombres pairs et impairs

```

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
pairs = []
impairs = []
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] % 2 == 0:
        pairs.append(tt[i])
    else:
        impairs.append(tt[i])
    i = i + 1
# Affichage :
print "Nombres pairs :", pairs
print "Nombres impairs :", impairs

```

5. Réfléchissez !

Exercices

1. Écrivez un script qui génère la liste des carrés et des cubes des nombres de 20 à 40.

2. Écrivez un script qui crée automatiquement la liste des sinus des angles de 0° à 90° , par pas de 5° . Attention : la fonction `sin()` du module `math` considère que les angles sont fournis en radians ($360^\circ = 2\pi$ radians)
3. Écrivez un script qui permette d'obtenir à l'écran les 15 premiers termes des tables de multiplication par 2, 3, 5, 7, 11, 13, 17, 19 (ces nombres seront placés au départ dans une liste) sous la forme d'une table similaire à la suivante :


```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
etc.
```
4. Soit la liste suivante :


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

 Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.
5. Vous disposez d'une liste de nombres entiers quelconques, certains d'entre eux étant présents en plusieurs exemplaires. Écrivez un script qui recopie cette liste dans une autre, en omettant les doublons. La liste finale devra être triée.
6. Écrivez un script qui recherche le mot le plus long dans une phrase donnée (l'utilisateur du programme doit pouvoir entrer une phrase de son choix).
7. Écrivez un script capable d'afficher la liste de tous les jours d'une année imaginaire, laquelle commencerait un Jeudi. Votre script utilisera lui-même trois listes : une liste des noms de jours de la semaine, une liste des noms des mois, et une liste des nombres de jours que comportent chacun des mois (ne pas tenir compte des années bissextiles).
Exemple de sortie :


```
Jeudi 1 Janvier Vendredi 2 Janvier Samedi 3 Janvier Dimanche 4 Janvier
... et ainsi de suite jusqu'au Jeudi 31 Décembre.
```
8. Vous avez à votre disposition un fichier texte qui contient un certain nombre de noms d'élèves. Écrivez un script qui effectue une copie triée de ce fichier.
9. Écrivez un script permettant de trier une liste. Il ne pourra pas utiliser la méthode intégrée "`sort()`" de Python : vous devez donc définir vous-même l'algorithme de tri.

Solution (cliquez)

1. Réfléchissez !
2. Réfléchissez !

```
3. # Affichage de tables de multiplication
m, n = 3, 15
chain = ""
for i in range(n):
    v = m * (i+1) # calcul d'un des termes
    chain = chain + "%4d" % (v) # formatage à 4 caractères
print(chain)
```

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
```

```
4. # Simple parcours d'une liste :
lst = ['Jean-Michel', 'Marc', 'Vanessa', 'Anne',
       'Maximilien', 'Alexandre-Benoît', 'Louise']

for e in lst:
    print("%s : %s caractères" % (e, len(e)))
```

```
5. # Élimination de doublons

lst = [9, 12, 40, 5, 12, 3, 27, 5, 9, 3, 8, 22, 40, 3, 2, 4, 6, 25]
lst2 = []

for el in lst:
    if el not in lst2:
        lst2.append(el)

lst2.sort()
print(lst2)
```

6. Réfléchissez !

```
7. # Afficher tous les jours d'une année :
## Cette variante utilise une liste de listes ##
## (que l'on pourrait aisément remplacer par deux listes distinctes)

# La liste ci-dessous contient deux éléments qui sont eux-mêmes des listes.
# l'élément 0 contient les nombres de jours de chaque mois, tandis que
# l'élément 1 contient les noms des douze mois :
mois = [[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
        ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
         'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']]

jour = ['Dimanche', 'Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi']

ja, jm, js, m = 0, 0, 0, 0

while ja < 365:
    ja, jm = ja + 1, jm + 1    # ja = jour dans l'année, jm = jour dans le mois
    js = (ja + 3) % 7        # js = jour de la semaine. Le décalage ajouté
                            # permet de choisir le jour de départ

    if jm > mois[0][m]:      # élément m de l'élément 0 de la liste
        jm, m = 1, m + 1

    print jour[js], jm, mois[1][m] # élément m de l'élément 1 de la liste
```

8. Réfléchissez !

9. Réfléchissez !

Tuples

Description d'un tuple

Vous devez vous rappeler une autre différence importante entre chaînes et listes : il n'est pas possible de changer les caractères au sein d'une chaîne existante, alors que vous pouvez modifier les éléments d'une liste. En d'autres termes, les listes sont des séquences modifiables, alors que les chaînes sont des séquences non-modifiables. Exemple :

```
>>> liste = ['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> liste[1:3] = ['salade']
>>> print liste
['jambon', 'salade', 'confiture', 'chocolat']
>>> chaine = 'Roméo préfère Juliette'
>>> chaine[14:] = 'Brigitte'
***** ==> Erreur: object doesn't support slice assignment *****
```

Nous essayons de modifier la fin de la chaîne, mais cela ne marche pas. La seule possibilité d'arriver à nos fins est de créer une nouvelle chaîne et d'y recopier ce que nous voulons changer :

```
>>> chaine = chaine[:14] + 'Brigitte'
>>> print chaine
Roméo préfère Brigitte
```

Python propose un type de données appelé *tuple* (anglicisme informatique signifiant "Table UPLet"), qui est assez semblable à une liste mais qui n'est pas modifiable. Les tuples sont donc préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les tuples sont moins « gourmands » en ressources système (ils occupent moins de place en mémoire).

Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules :

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
>>> print(tuple)
('a', 'b', 'c', 'd', 'e')
```

Pour améliorer la lisibilité du code, il est vivement conseillé de déclarer le tuple en évidence en l'enfermant dans une paire de parenthèses, comme l'instruction "print" de Python le fait elle-même.

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

Les opérations que l'on peut effectuer sur des tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuples ne sont pas modifiables :

```
>>> print(tuple[2:4])
>('c', 'd')

>>> tuple[1:3] = ('x', 'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> tuple = ('André',) + tuple[1:]
>>> print(tuple)
('André', 'b', 'c', 'd', 'e')
```

Remarquez qu'il faut toujours au moins une virgule pour définir un tuple (le dernier exemple ci-dessus utilise un tuple contenant un seul élément : 'André').

Dictionnaires

Déclarations

Le dictionnaire ou tableau associatif n'est pas une séquence mais un autre type composite. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles), mais les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une clé, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

Remarque : comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type (valeurs numériques, chaînes, listes, etc.).

Création d'un dictionnaire

A titre d'exemple, nous allons créer un dictionnaire de langue, pour la traduction de termes informatiques anglais en français. Dans ce dictionnaire, les index seront des chaînes de caractères.

Puisque le type dictionnaire est un type modifiable, nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue syntaxique, on reconnaît une structure de données de type dictionnaire au fait que ses éléments sont enfermés dans *une paire d'accolades*. Un dictionnaire vide sera donc noté `{ }` :

```
-----  
dico = {}  
dico['computer'] = 'ordinateur'  
dico['mouse'] = 'souris'  
dico['keyboard'] = 'clavier'  
print dico  
{'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}  
-----
```

Comme vous pouvez l'observer dans la ligne ci-dessus, un dictionnaire apparaît comme une série d'éléments séparés par des virgules (le tout étant enfermé entre deux accolades}. Chacun de ces éléments est constitué d'une paire d'objets : un index et une valeur, séparés par un double point.

Dans un dictionnaire, les index s'appellent des *clés*, et les éléments peuvent donc s'appeler *des paires clé-valeur*. Vous pouvez constater que l'ordre dans lequel les éléments apparaissent à la dernière ligne ne correspond pas à celui dans lequel nous les avons fournis. Cela n'a strictement aucune importance : nous n'essaierons jamais d'extraire une valeur d'un dictionnaire à l'aide d'un index numérique. Au lieu de cela, nous utiliserons les clés :

```
>>> print dico['mouse']
souris
```

Contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

Opérations sur les dictionnaires

Pour en enlever, vous utiliserez l'instruction `del`. Créons pour l'exemple un autre dictionnaire, destiné cette fois à contenir l'inventaire d'un stock de fruits. Les index (ou clés) seront les noms des fruits, et les valeurs seront les masses de ces fruits répertoriées dans le stock (il s'agit donc cette fois de valeurs de type numérique).

```
>>> invent = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires' : 137}
>>> print invent
{'oranges': 274, 'pommes': 430, 'bananes': 312, 'poires': 137}
```

Si le patron décide de liquider toutes les pommes et de ne plus en vendre, nous pouvons enlever cette entrée dans le dictionnaire :

```
>>> del invent['pommes']
>>> print invent
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

La fonction `len()` est utilisable avec un dictionnaire : elle en renvoie le nombre d'éléments.

Les dictionnaires sont des objets

On peut appliquer aux dictionnaires un certain nombre de *méthodes* spécifiques :

keys et *values*

La méthode `keys()` renvoie la liste des clés utilisées dans le dictionnaire :

```
>>> print dico.keys()
['computer', 'keyboard', 'mouse']
```

La méthode `values()` renvoie la liste des valeurs mémorisées dans le dictionnaire :

```
>>> print invent.values()
```

```
{274, 312, 137}
```

has_key

La méthode `has_key()` permet de savoir si un dictionnaire comprend une clé déterminée. On fournit la clé en argument, et la méthode renvoie une valeur 'vraie' ou 'fausse' (en fait, 1 ou 0), suivant que la clé est présente ou pas :

```
>>> print invent.has_key('bananes')
1
>>> if invent.has_key('pommes'):
    print 'nous avons des pommes'
else:
    print 'pas de pommes, sorry'
pas de pommes, sorry
```

items

La méthode `items()` extrait du dictionnaire une liste équivalente de tuples :

```
>>> print invent.items()
[('oranges', 274), ('bananes', 312), ('poires', 137)]
```

Il est donc utilisé pour les "foreach" :

```
for key, value in invent.items():
    print "Clé : %s, valeur : %s." % (key, value)
```

copy

La méthode `copy()` permet d'effectuer une vraie copie d'un dictionnaire. Il faut savoir en effet que la simple affectation d'un dictionnaire existant à une nouvelle variable crée seulement une nouvelle référence vers le même objet, et non un nouvel objet (*aliasing*). Par exemple, l'instruction ci-dessous ne définit pas un nouveau dictionnaire (contrairement aux apparences) :

```
>>> stock = invent
>>> print stock
{'oranges': 274, 'bananes': 312, 'poires': 137}
>>> del invent['bananes']
>>> print stock
{'oranges': 274, 'poires': 137}
```

Si nous modifions "invent", alors `stock` aussi est modifié, et vice-versa (ces deux noms désignent en effet le même objet dictionnaire dans la mémoire de l'ordinateur).

Pour obtenir une vraie copie (indépendante) d'un dictionnaire préexistant, il faut employer la méthode `copy()` :

```
>>> magasin = stock.copy()
>>> magasin['prunes'] = 561
>>> print magasin
{'oranges': 274, 'prunes': 561, 'poires': 137}
>>> print stock
{'oranges': 274, 'poires': 137}
>>> print invent
{'oranges': 274, 'poires': 137}
```

update

Il est possible de concaténer deux dictionnaires avec cette méthode :

```
>>> d = {'apples': 1, 'oranges': 3, 'pears': 2}
>>> ud = {'pears': 4, 'grapes': 5, 'lemons': 6}
>>> d.update(ud)
>>> d
{'grapes': 5, 'pears': 4, 'lemons': 6, 'apples': 1, 'oranges': 3}
>>>
```

Remarque : les doublons sont automatiquement fusionnés.

Parcours d'un dictionnaire

Vous pouvez utiliser une boucle `for` pour traiter successivement tous les éléments contenus dans un dictionnaire, mais attention :

- Au cours de l'itération, ce sont les clés utilisées dans le dictionnaire qui seront successivement affectées à la variable de travail, et non les valeurs.
- L'ordre dans lequel les éléments seront extraits est imprévisible (puisque'un dictionnaire n'est pas une séquence).

Exemple :

```
>>> invent = {"oranges":274, "poires":137, "bananes":312}
>>> for clef in invent:
...     print clef

poires
bananes
oranges
```

Si vous souhaitez effectuer un traitement sur les valeurs, il vous suffit alors de récupérer chacune d'elles à partir de la clé correspondante :

```
for clef in invent:
```

```
print clef, invent[clef]

poires 137
bananes 312
oranges 274
```

Cette manière de procéder n'est cependant pas idéale, ni en termes de performances ni même du point de vue de la lisibilité. Il est recommandé de plutôt faire appel à la méthode `items()` décrite à la section précédente :

```
for clef, valeur in invent.items():
    print clef, valeur

poires 137
bananes 312
oranges 274
```

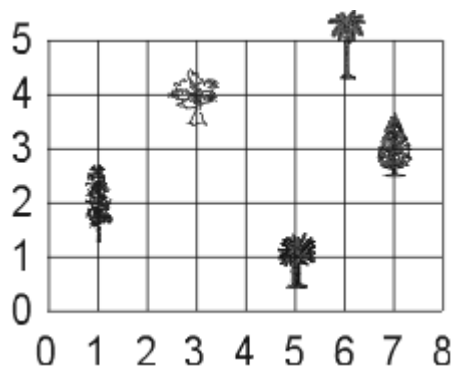
Les clés ne sont pas nécessairement des chaînes de caractères

Jusqu'à présent nous avons décrit des dictionnaires dont les clés étaient à chaque fois des valeurs de type *string*. En fait nous pouvons utiliser en guise de clés n'importe quel type de donnée non modifiable : des entiers, des réels, des chaînes de caractères, et même des tuples.

Considérons par exemple que nous voulions répertorier les arbres remarquables situés dans un grand terrain rectangulaire. Nous pouvons pour cela utiliser un dictionnaire, dont les clés seront des tuples indiquant les coordonnées *x,y* de chaque arbre :

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[(6,5)] = 'Palmier'
>>> arb[(5,1)] = 'Cycas'
>>> arb[(7,3)] = 'Sapin'

>>> print arb
{(3, 4): 'Platane', (6, 5): 'Palmier', (5, 1): 'Cycas', (1, 2): 'Peuplier', (7, 3): 'Sapin'}
>>> print arb[(6,5)]
palmier
```



Vous pouvez remarquer que nous avons allégé l'écriture à partir de la troisième ligne, en profitant du fait que les parenthèses délimitant les tuples sont facultatives (à utiliser avec prudence !).

Dans ce genre de construction, il faut garder à l'esprit que le dictionnaire contient des éléments seulement pour certains couples de coordonnées. Ailleurs, il n'y a rien. Par conséquent, si nous voulons interroger le dictionnaire pour savoir ce qui se trouve là où il n'y a rien, comme par exemple aux coordonnées (2,1), nous allons provoquer une erreur :

```

>>> print arb[1,2]
Peuplier
>>> print arb[2,1]
***** Erreur : KeyError: (2, 1) *****
    
```

Pour résoudre ce petit problème, nous pouvons utiliser la méthode `get()` :

```

>>> print arb.get((1,2), 'néant')
Peuplier
>>> print arb.get((2,1), 'néant')
néant
    
```

Le premier argument transmis à cette méthode est la clé de recherche, le second argument est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

Les dictionnaires ne sont pas des séquences

Comme vous l'avez vu plus haut, les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des opérations comme la concaténation et l'extraction (d'un groupe d'éléments contigus) ne peuvent donc tout simplement pas s'appliquer ici. Si vous essayez tout de même, Python lèvera une erreur lors de l'exécution du code :

```

>>> print arb[1:3]
***** Erreur : KeyError: slice(1, 3, None) *****
    
```

Vous avez vu également qu'il suffit d'affecter un nouvel indice (une nouvelle

clé) pour ajouter une entrée au dictionnaire. Cela ne marcherait pas avec les listes :

```

>>> invent['cerises'] = 987
>>> print invent
{'oranges': 274, 'cerises': 987, 'poires': 137}

>>> liste = ['jambon', 'salade', 'confiture', 'chocolat']
>>> liste[4] = 'salami'
***** IndexError: list assignment index out of range *****
    
```

Du fait qu'ils ne sont pas des séquences, les dictionnaires se révèlent donc particulièrement précieux pour gérer des ensembles de données où l'on est amené à effectuer fréquemment des ajouts ou des suppressions, dans n'importe quel ordre. Ils remplacent avantageusement les listes lorsqu'il s'agit de traiter des ensembles de données numérotées, dont les numéros ne se suivent pas.

Exemple :

```

>>> client = {}
>>> client[4317] = "Dupond"
>>> client[256] = "Durand"
>>> client[782] = "Schmidt"
    
```

etc.

Exercices

1. Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, et dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille. Votre script devra comporter deux fonctions : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation. Dans la fonction de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur. Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel). La fonction de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple « âge, taille » correspondant. Le résultat de la requête devra être une ligne de texte bien formatée, telle par exemple : « Nom : Jean Dhoute - âge : 15 ans - taille : 1.74 m ».
2. Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). (On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques).

Solution (cliquez)

```

1. #!/usr/bin/env python
# coding: utf-8

# Création du dictionnaire
dico = {}
while 1:
    choix = raw_input("Choisissez : (R)emplir - (C)onsulter - (T)erminer : ")
    if choix.upper() == 'T':
        break

    elif choix.upper() == 'R':
        nom = raw_input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "":
            break
        age = int(raw_input("Entrez l'âge (nombre entier !) : "))
        taille = float(raw_input("Entrez la taille (en mètres) : "))
        dico[nom] = (age, taille)

    elif choix.upper() == 'C':
        # Consultation
        nom = raw_input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "":
            break
        if dico.has_key(nom):           # le nom est-il répertorié ?
            item = dico[nom]           # consultation proprement dite
            age, taille = item[0], item[1]
            print "Nom : %s - âge : %s ans - taille : %s m." \
                % (nom, age, taille)
        else:
            print "*** nom inconnu ! ***"

```

```

2. # Échange des clés et des valeurs dans un dictionnaire
dico = {'Computer': 'Ordinateur',
        'Mouse': 'Souris',
        'Keyboard': 'Clavier',
        'Hard disk': 'Disque dur',
        'Screen': 'Ecran'}

print(dico)

dic_inv = {}
for cle in dico:
    item = dico[cle]
    dic_inv[item] = cle

print(dic_inv)

```

Construction d'un histogramme à l'aide d'un dictionnaire

Les dictionnaires constituent un outil très élégant pour construire des histogrammes.

Supposons par exemple que nous voulions établir l'histogramme qui représente

la fréquence d'utilisation de chacune des lettres de l'alphabet dans un texte donné. L'algorithme permettant de réaliser ce travail est extraordinairement simple si on le construit sur base d'un dictionnaire :

```

>>> texte = "les saucisses et saucissons secs sont dans le saloir"
>>> lettres = {}
>>> for c in texte:
    lettres[c] = lettres.get(c, 0) + 1

>>> print lettres
{'t': 2, 'u': 2, 'r': 1, 's': 14, 'n': 3, 'o': 3, 'l': 3, 'i': 3, 'd': 1,
'e': 5, 'c': 3, ' ': 8, 'a': 4}
    
```

Nous commençons par créer un dictionnaire vide : `lettres`. Ensuite, nous allons remplir ce dictionnaire en utilisant les caractères de l'alphabet en guise de clés. Les valeurs que nous mémoriserons pour chacune de ces clés seront les fréquences des caractères correspondants dans le texte. Afin de calculer celles-ci, nous effectuons un parcours de la chaîne de caractères `texte`. Pour chacun de ces caractères, nous interrogeons le dictionnaire à l'aide de la méthode `get()`, en utilisant le caractère en guise de clé, afin d'y lire la fréquence déjà mémorisée pour ce caractère. Si cette valeur n'existe pas encore, la méthode `get()` doit renvoyer une valeur nulle. Dans tous les cas, nous incrémentons la valeur trouvée, et nous la mémorisons dans le dictionnaire à l'emplacement qui correspond à la clé (c'est-à-dire au caractère en cours de traitement).

Pour figurer notre travail, nous pouvons encore souhaiter afficher l'histogramme dans l'ordre alphabétique. Pour ce faire, nous pensons immédiatement à la méthode `sort()`, mais celle-ci ne peut s'appliquer qu'aux listes. Qu'à cela ne tienne ! Nous avons vu plus haut comment nous pouvons convertir un dictionnaire en une liste de tuples :

```

>>> lettres_triees = lettres.items()
>>> lettres_triees.sort()
>>> print lettres_triees
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3),
('n', 3), ('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
    
```

Exercices

1. Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte (on ne tiendra pas compte du problème des lettres accentuées)..
2. Modifiez le script ci-dessus afin qu'il établisse une table des occurrences de chaque mot dans le texte. Conseil : dans un texte quelconque, les mots ne sont pas seulement séparés par des espaces, mais également par divers signes de ponctuation. Pour simplifier le problème, vous pouvez commencer par remplacer tous les caractères non-alphabétiques par des espaces, et convertir la chaîne résultante en une liste de mots à l'aide de la méthode `split()`.

3. Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui analyse ce texte, et mémorise dans un dictionnaire l'emplacement exact de chacun des mots (compté en nombre de caractères à partir du début). Lorsqu'un même mot apparaît plusieurs fois, tous ses emplacements doivent être mémorisés : chaque valeur de votre dictionnaire doit donc être une liste d'emplacements.

Solution (cliquez)

```
1. # histogramme
nFich = raw_input('Nom du fichier : ')
fi = open(nFich, 'r')
texte = fi.read()      # conversion du fichier en une chaîne de caractères
fi.close()

print texte
dico = {}
for c in texte:
    c = c.upper()      # conversion de toutes les lettres en majuscules
    dico[c] = dico.get(c, 0) + 1

liste = dico.items()
liste.sort()
print liste
```

```
2. nFich = raw_input('Nom du fichier à traiter : ')
fi = open(nFich, 'r')
texte = fi.read()
fi.close()

# afin de pouvoir aisément séparer les mots du texte, on commence
# par convertir tous les caractères non-alphabétiques en espaces :
alpha = "abcdefghijklmnopqrstuvwxyzèèàùçâêîôûäëïöü"

lettres = ''          # nouvelle chaîne à construire
for c in texte:
    c = c.lower()     # conversion de chaque caractère en minuscule
    if c in alpha:
        lettres = lettres + c
    else:
        lettres = lettres + ' '

# conversion de la chaîne résultante en une liste de mots :
mots = lettres.split()

# construction de l'histogramme :
dico = {}
for m in mots:
    dico[m] = dico.get(m, 0) + 1

liste = dico.items()

# tri de la liste résultante :
liste.sort()
```

```

# affichage en clair :
for item in liste:
    print item[0], ":", item[1]

```

3. # encodage d'un texte dans un dictionnaire

```

nFich = raw_input('Nom du fichier à traiter : ')
fi = open(nFich, 'r')
texte = fi.read()
fi.close()

# On considère que les mots sont des suites de caractères faisant partie
# de la chaîne ci-dessous. Tous les autres sont des séparateurs :

alpha = "abcdefghijklmnopqrstuvwxyzèèàùçâêîôûäëïöü"

# construction du dictionnaire :
dico = {}
# parcours de tous les caractères du texte :
i = 0 # indice du caractère en cours de lecture
mot = "" # variable de travail : mot en cours de lecture
for c in texte:
    c = c.lower() # conversion de chaque caractère en minuscule

    if c in alpha: # car. alphas. => on est à l'intérieur d'un mot
        mot = mot + c
    else: # car. non-alphabétique => fin de mot
        if mot != "": # afin d'ignorer les car. non-alphabétiques successifs
            # pour chaque mot, on construit une liste d'indices :
            if dico.has_key(mot): # mot déjà répertorié :
                dico[mot].append(i) # ajout d'un indice à la liste
            else: # mot rencontré pour la 1e fois :
                dico[mot] = [i] # création de la liste d'indices
            mot = "" # préparer la lecture du mot suivant
        i = i+1 # indice du caractère suivant

# Affichage du dictionnaire, en clair :
for clef, valeur in dico.items():
    print clef, ":", valeur

```

Contrôle du flux d'exécution à l'aide d'un dictionnaire

Il arrive fréquemment que l'on ait à diriger l'exécution d'un programme dans différentes directions, en fonction de la valeur prise par une variable. Vous pouvez bien évidemment traiter ce problème à l'aide d'une série d'instructions `if - elif - else`, mais cela peut devenir assez lourd et inélégant si vous avez affaire à un grand nombre de possibilités. Exemple :

```

matériau = raw_input("Choisissez le matériau : ")

if matériau == 'fer':
    fonctionA()
elif matériau == 'bois':
    fonctionC()

```



```
elif materiau == 'cuivre':
    fonctionB()
elif materiau == 'pierre':
    fonctionD()
elif ... etc ...
```

Les langages de programmation proposent souvent des instructions spécifiques pour traiter ce genre de problème, telles les instructions `switch` ou `case` du C ou du Pascal. Python n'en propose aucune, mais vous pouvez vous tirer d'affaire dans bien des cas à l'aide d'une liste, ou mieux encore à l'aide d'un dictionnaire. Exemple :

```
materiau = raw_input("Choisissez le matériau : ")

dico = {'fer':fonctionA,
        'bois':fonctionC,
        'cuivre':fonctionB,
        'pierre':fonctionD,
        ... etc ...}
dico[materiau]()
```

Les deux instructions ci-dessus pourraient être condensées en une seule, mais nous les laissons séparées pour bien détailler le mécanisme :

- La première instruction définit un dictionnaire `dico` dans lequel les clés sont les différentes possibilités pour la variable `materiau`, et les valeurs, les noms des fonctions à invoquer en correspondance. Notez bien qu'il s'agit seulement des noms de ces fonctions, qu'il ne faut surtout pas faire suivre de parenthèses dans ce cas (Sinon Python exécuterait chacune de ces fonctions au moment de la création du dictionnaire).
- La seconde instruction invoque la fonction correspondant au choix opéré à l'aide de la variable `materiau`. Le nom de la fonction est extrait du dictionnaire à l'aide de la clé, puis associé à une paire de parenthèses. Python reconnaît alors un appel de fonction tout à fait classique et l'exécute.

Vous pouvez encore améliorer la technique ci-dessus en remplaçant cette instruction par sa variante ci-dessous, qui fait appel à la méthode `get()` afin de prévoir le cas où la clé demandée n'existerait pas dans le dictionnaire (vous obtenez de cette façon l'équivalent d'une instruction `else` terminant une longue série de `elif`) :

```
dico.get(materiau, fonctAutre)()
```

Lorsque la la valeur de la variable "materiau "ne correspond à aucune clé du dictionnaire, c'est la fonction `fonctAutre()` qui est invoquée.

Dates

Module datetime

Les dates se manipulent à l'aide du module `datetime`^[1].

Syntaxe

```
>>> import datetime
>>> maDate = datetime.datetime.now()
>>> print (maDate)
2017-04-22 01:49:41.230711
>>>
```

Impossible de tronquer une date complète comme une chaîne de caractères, il faut utiliser les fonctions du module :

```
>>> print (maDate.year)
2017
>>> print (maDate.month)
4
>>> print (maDate.day)
22
>>> print (maDate.hour)
1
>>> print (maDate.minute)
49
>>> print (maDate.second)
41
>>> print (maDate.microsecond)
230711
>>>
```

Les variables de ce type sont très faciles à additionner et soustraire :

```
>>> maDateDeFin = datetime.datetime.now()
>>> print (maDateDeFin - maDate)
0:04:38.266647
```

Références

1. <https://docs.python.org/2/library/datetime.html>

Ensembles

Définition

En Python, les ensembles sont définis par le mot "set()"^[1] depuis Python 2.3, d'abord en important le module du même nom, puis depuis nativement Python 2.6, avec "frozenset()".

Un ensemble est une collection non ordonnée d'objets, contrairement aux séquences comme les listes et les tuples dans lesquels chaque élément est indexé. Un ensemble ne peut pas contenir de doublon : on ne peut y trouver des éléments que zéro ou une fois. Tous les membres d'un ensemble doivent être hachable, comme les clés des dictionnaires. A titre d'exemple, les scalaires comme les entiers, flottants, tuples, et chaînes sont hachables ; par contre les dictionnaires, listes, et ensembles ne le sont pas.

Exemple :

```

set1 = set() # Nouvel ensemble vide
set1.add("cat") # Ajout d'un membre
set1.update(["dog", "mouse"]) # Ajout de plusieurs membres
if "cat" in set1: # Recherche d'un membre
    set1.remove("cat") # Retrait d'un membre
#set1.remove("elephant") - Erreur de retrait d'un membre introuvable
set1.discard("elephant") # Aucune erreur de retrait d'un membre introuvable

print(set1) # Affichage d'un ensemble
for item in set1: # Itération pour chaque élément
    print(item)
print("Item count:", len(set1)) # Compte des éléments

#Istitem = set1[0] # Erreur d'index introuvable
isempty = len(set1) == 0 # Test si l'ensemble est vide
set1 = set(["cat", "dog"]) # Initialisation de l'ensemble depuis une liste de membre
set2 = set(["dog", "mouse"])
set3 = set1 & set2 # Intersection
set4 = set1 | set2 # Union
set5 = set1 - set2 # Différence
set6 = set1 ^ set2 # Différence symétrique
issubset = set1 <= set2 # Test de sous-ensemble
issuperset = set1 >= set2 # Test de sur-ensemble
set7 = set1.copy() # Copie d'un ensemble
set7.remove("cat")
set8 = set1.copy()
set8.clear() # Effacement d'un ensemble
print(set1, set2, set3, set4, set5, set6, set7, set8, issubset, issuperset)

```

Construction d'ensembles

Une première méthode consiste à fournir un objet séquentiel en paramètre :

```

>>> set([0, 1, 2, 3])

```

```
set([0, 1, 2, 3])
>>> set("obtuse")
set(['b', 'e', 'o', 's', 'u', 't'])
```

Ajout de chaque membre un par un :

```
>>> s = set([12, 26, 54])
>>> s.add(32)
>>> s
set([32, 26, 12, 54])
```

Ajout de groupes de membres :

```
>>> s.update([26, 12, 9, 14])
>>> s
set([32, 9, 12, 14, 54, 26])
```

Remarque : si on ajoute un doublon avec "add()" ou "update()", cela n'a aucun effet.

Ajout par copie d'un autre ensemble :

```
>>> s2 = s.copy()
>>> s2
set([32, 9, 12, 14, 54, 26])
```

Recherche de membre

Pour chercher si un élément existe dans un ensemble, on utilise "in" :

```
>>> 32 in s
True
>>> 6 in s
False
>>> 6 not in s
True
```

Si un sous-ensemble existe dans un ensemble, c'est "issubset()" :

```
>>> s.issubset(set([32, 8, 9, 12, 14, -4, 54, 26, 19]))
True
```

Si un sur-ensemble contient un ensemble, c'est "issuperset()" :

```
>>> s.issuperset(set([9, 12]))
True
```

```
# Équivalent à :
>>> s.issuperset([9, 12])

# Équivalent à :
>>> s >= [9, 12]
```

Retrait de membre

Il existe quatre fonctions pour retirer des membres à un ensemble :

1. "pop" : retire un membre non précisé.
2. "remove" : retire le membre existant précisé.
3. "discard" : retire un membre précisé.
4. "clear" : retire tous les éléments.

```
>>> s = set([1,2,3,4,5,6])
>>> s.pop()
1
>>> s
set([2,3,4,5,6])
>>> s.remove(3)
>>> s
set([2,4,5,6])
>>> s.remove(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 9
>>> s.clear()
>>> s
set([])
```

Itération des ensembles

Les éléments n'étant pas ordonnés, il n'y a qu'une boucle possible :

```
>>> s = set("blerg")
>>> for n in s:
...     print n,
...
r b e l g
```

Opérations sur les ensembles

Python offre les mêmes opérations sur les ensembles qu'en mathématiques, applicables par soit par des opérateurs, soit par des fonctions équivalentes.

Intersection

Les éléments communs à deux ensembles.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.intersection(s2)
set([6])
>>> s1 & s2
set([6])

>>> s1.intersection_update(s2)
>>> s1
set([6])
```

Union

Somme des éléments de deux ensembles.

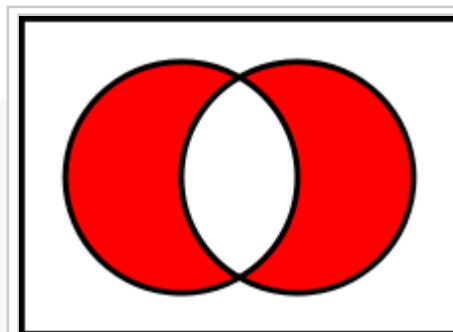
```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.union(s2)
set([1, 4, 6, 8, 9])
>>> s1 | s2
set([1, 4, 6, 8, 9])
```

Différence symétrique

Éléments contenu dans un seul ensemble à la fois, parmi deux.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.symmetric_difference(s2)
set([8, 1, 4, 9])
>>> s1 ^ s2
set([8, 1, 4, 9])

>>> s1.symmetric_difference_update(s2)
>>> s1
set([8, 1, 4, 9])
```



La différence symétrique des deux cercles apparait en rouge.

Différence

Éléments non contenu dans un des deux ensembles.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.difference(s2)
set([9, 4])
>>> s1 - s2
set([9, 4])

>>> s1.difference_update(s2)
```

```
>>> s1
set([9, 4])
```

Opérations non binaires

Depuis Python 2.6, les fonctions vues précédemment acceptent plus de deux arguments :

1. `.intersection()`
2. `.union()`
3. `.symetic_difference()`
4. `.difference()`.

Exemple :

```
>>> s1 = set([3, 6, 7, 9])
>>> s2 = set([6, 7, 9, 10])
>>> s3 = set([7, 9, 10, 11])
>>> set.intersection(s1, s2, s3)
set([9, 7])
```

frozenset

Un "frozenset" (ensemble figé) se comporte comme un ensemble, sauf qu'il est immuable, c'est-à-dire qu'il fois créé, on ne peut pas le mettre à jour. Il dispose donc des mêmes fonctions que le type "set", mais sans "add", "update", "pop", "remove", "discard" et "clear".

De plus, ils sont hachables, ce qui leur permet de faire partie d'ensembles.

```
>>> fs = frozenset([2, 3, 4])
>>> s1 = set([fs, 4, 5, 6])
>>> s1
set([4, frozenset([2, 3, 4]), 6, 5])
>>> fs.intersection(s1)
frozenset([4])

>>> fs.add(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Exercices

1. Créer un set {'cat', 1, 2, 3}, appelé "s".
2. Créer un set {'c', 'a', 't', '1', '2', '3'}.
3. Créer un frozen set {'cat', 1, 2, 3}, appelé "fs".
4. Créer un set contenant {frozenset({'cat', 2, 3, 1})}.

Références

1. <https://docs.python.org/2/library/stdtypes.html#set>

- Python Tutorial, section "Data Structures", subsection "Sets" -- python.org
- Python Library Reference on Set Types -- python.org
- PEP 218 -- Adding a Built-In Set Object Type, python.org, a nice concise overview of the set type

Fonctions

Définir une fonction

Pour les non-programmeurs

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales. De plus, créer ses propres fonctions permet de factoriser le code redondant en pouvant être appelées depuis plusieurs endroits.

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.

Une fonction Python est définie par le spécificateur "def" suivi du nom de la fonction et de ses paramètres :

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

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, 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 d'utiliser surtout des lettres minuscules, notamment au début du nom.
- Comme les instructions `if` et `while`, 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 (avec leurs éventuelles valeurs par défaut) 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.
- Une fonction Python ne renvoie pas obligatoirement de résultat : le mot "return" est facultatif. S'il est absent, en termes de programmation on parlera alors plutôt de procédure que de fonction, et elle renverra "None".
- 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
```

Fonctionnement

```
def factorielle(n):
    f = 1
    i = 1
    while i <= n:
        f = f * i
        i = i + 1
    return f # la valeur retournée

factorielle(7) # 5040
```

Récurtivité

Une première fonction peut appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Mais elle peut aussi s'appeler elle-même :

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

```

    else:
        return n * factorielle(n-1)
factorielle(7) # 5040

```

Passage d'argument

Un fonction accepte entre zéro et 255 d'arguments :

```

>>> def f(x, y):
        return x*y
f(3, 4) # 12

```

La signature est ici "x" et "y" en paramètre.

Arguments facultatifs

Il suffit de définir une valeur par défaut à un argument pour le rendre facultatif. Naturellement, cette valeur est écrasée si l'argument est précisé :

```

>>> def f(x = None):
        if x:
            print(x)
print f() # None
print f(1) # 1 None

```

Arguments nommés

Pour ne pas être obligé de remplir tous les paramètres facultatifs dans l'ordre, il est possible de n'en n'appeler que quelques-uns s'ils sont nommés :

```

>>> def f(p1 = 0, p2 = 0, p3):
        ...
f(p3 = 1) # 1

```

Fonction lambda

Une fonction lambda est une fonction anonyme : elle n'est pas définie par *def*.

```

>>> def f(x):
        return x*2
>>> f(3)
6
>>> g = lambda x: x*2 # 1
>>> g(3)

```

```
6
>>> (lambda x: x*2)(3) # 2
6
```

1 et 2 sont des fonctions lambda.

Récupérer les arguments de la ligne de commande

La variable `sys.argv` contient les arguments de la ligne de commande, sous forme d'une liste dont le premier élément est le nom du script invoqué.

Exemple :

Si le script `truc.py` contient

```
#!/usr/bin/python
#-*- coding: utf-8 -*-
import sys
print("Arguments : ", sys.argv)
```

alors l'invocation :

```
$ python truc.py -a rien -n=nervures
```

produira la sortie :

```
Arguments : ['truc.py', '-a', 'rien', '-n=nervures']
```

Si on veut récupérer l'argument n° 2 :

```
#!/usr/bin/python
#-*- coding: utf-8 -*-
import sys
print("Argument 2 : ", sys.argv[2])
```

produira la sortie :

```
Argument 2 : 'rien'
```

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.

En effet, chaque fois que la fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) *un nouvel espace de noms*. Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. De plus, cet espace de noms est automatiquement détruit dès que la fonction a terminé son travail, donc si on l'appelle deux fois de suite elle recommence à zéro.

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

```

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

```

Analysons attentivement cet exemple

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

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

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

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

On pourrait croire d'abord que la fonction `mask()` a simplement modifié le contenu de la variable globale *p* (puisque'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.

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.

Autre exemple

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
>>>
```

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

À 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()`.

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

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : d'abord la définition des fonctions, et ensuite le corps principal du programme. Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source. Dans 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 entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé `__main__` (le mot *main* signifie « principal », en anglais. Il est encadré par des caractères « souligné » en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité `__main__`, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu. Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction *avant* l'entité `__main__`, et celle-ci sera donc placée en général à la fin du script.

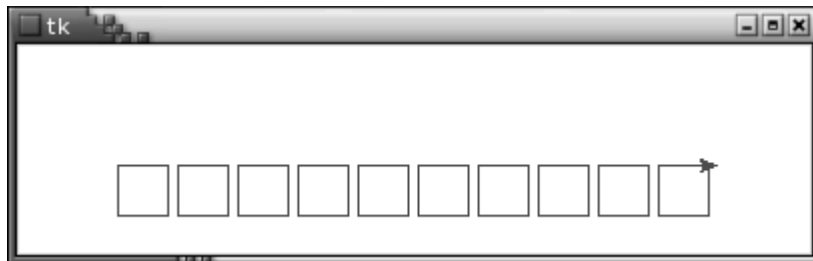
Dans notre exemple, l'entité `__main__` appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première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

Modules de fonctions

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

Exemple

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



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

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

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

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

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

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

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


Cette fonction est bien documentée mais ne fait presque rien.

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

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

```
from dessins_tortue import *
up()                # relever le crayon
goto(-150, 50)     # reculer en haut à gauche
# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down()          # abaisser le crayon
    carre(25, 'red') # tracer un carré
    up()            # relever le crayon
    forward(30)     # avancer + loin
    i = i + 1
a = input()        # attendre
```

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

Résumé : Structure d'un programme Python type

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

#####
# Importation de fonctions externes :

from math import sqrt

#####
# Définition locale de fonctions :

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

    nc = 0

    i = 0
    while i < len(ch):
        if ch[i] == car:
            nc = nc + 1
        i = i + 1
    return nc

#####
# Corps principal du programme :

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

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

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

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

Un programme Python contient en général les blocs suivants, dans l'ordre :

- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales)
- Les définitions locales de fonctions et/ou de classes
- Le corps principal du programme.

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

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

Une boucle de répétition de type 'while' doit en principe inclure les 4 éléments suivants :

- l'initialisation d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrément du compteur.

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

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

Exercices

1. 1.2. Définissez une fonction `ligneCar(n, ca)` qui renvoie une chaîne de n

caractères `ca`.

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

```
print surfCercle(2.5)
```

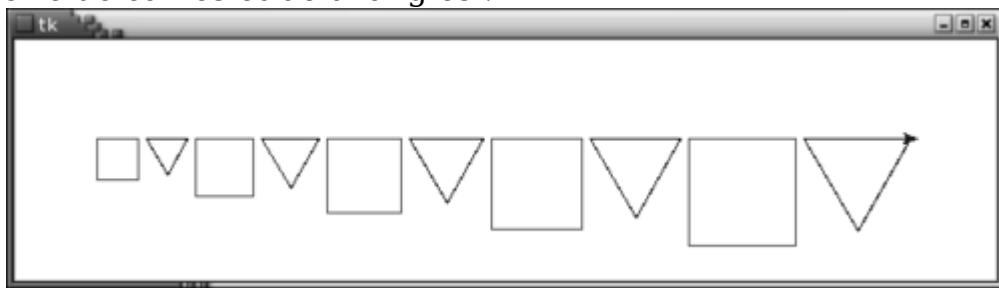
 doit donner le résultat `19.635`
3. Définissez une fonction `volBoite(x1,x2,x3)` qui renvoie le volume d'une boîte parallélépipédique dont on fournit les trois dimensions `x1`, `x2`, `x3` en arguments. Par exemple, l'exécution de l'instruction :

```
print volBoite(5.2, 7.7, 3.3)
```

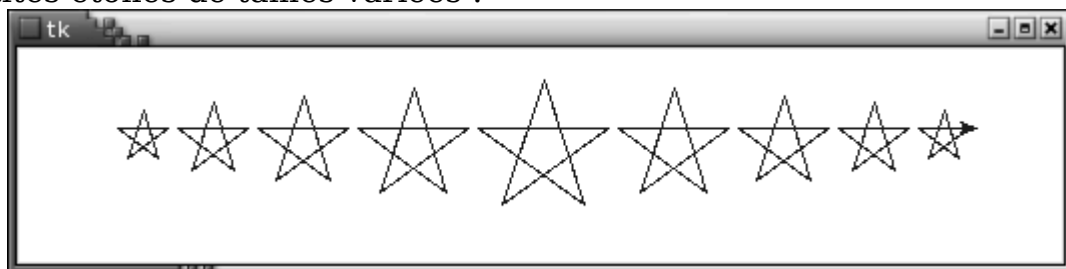
 doit donner le résultat : `132.13`
4. Définissez une fonction `maximum(n1,n2,n3)` qui renvoie le plus grand de 3 nombres `n1`, `n2`, `n3` fournis en arguments. Par exemple, l'exécution de l'instruction :

```
print maximum(2,5,4)
```

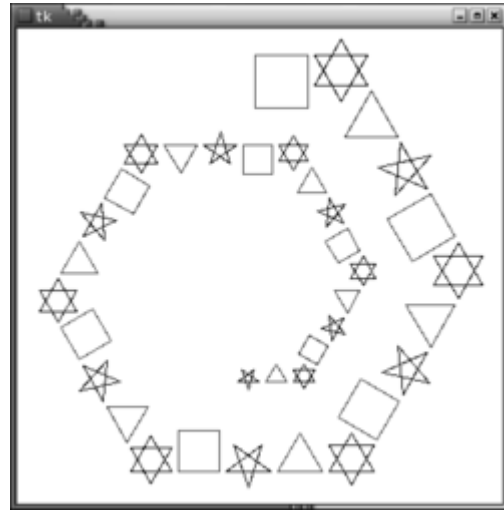
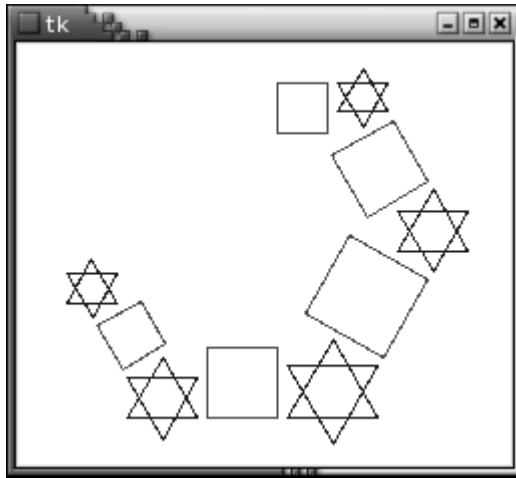
 doit donner le résultat : `5`
5. Complétez le module de fonctions graphiques `dessins_tortue.py`. Commencez par ajouter un paramètre `angle` à la fonction `carre()`, de manière à ce que les carrés puissent être tracés dans différentes orientations. Définissez ensuite une fonction `triangle(taille, couleur, angle)` capable de dessiner un triangle équilatéral d'une taille, d'une couleur et d'une orientation bien déterminées. Testez votre module à l'aide d'un programme qui fera appel à ces fonctions à plusieurs reprises, avec des arguments variés pour dessiner une série de carrés et de triangles :



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



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



8. Définissez une fonction `compteCar(ca,ch)` qui renvoie le nombre de fois que l'on rencontre le caractère `ca` dans la chaîne de caractères `ch`. Par exemple, l'exécution de l'instruction : `print compteCar('e','Cette phrase est un exemple')` doit donner le résultat : 7
9. Définissez une fonction `indexMax(liste)` qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument.
Exemple d'utilisation :

```
serie = [5, 8, 2, 1, 9, 3, 6, 7]
print indexMax(serie)
4
```
10. Définissez une fonction `nomMois(n)` qui renvoie le nom du ne mois de l'année.
Par exemple, l'exécution de l'instruction : `print nomMois(4)` doit donner le résultat : Avril
11. Définissez une fonction `inverse(ch)` qui permette d'inverser les l'ordre des caractères d'une chaîne quelconque. (La chaîne inversée sera renvoyée au programme appelant).
12. Définissez une fonction `compteMots(ph)` qui renvoie le nombre de mots contenus dans la phrase `ph` (On considère comme mots les ensembles de caractères inclus entre des espaces).

Solution (cliquez)

1. Réfléchissez !

```
2. from math import pi

def surfCercle(r):
    "Surface d'un cercle de rayon r"
    return pi * r**2

# test :
print surfCercle(2.5)
```

```
3. def volBoite(x1, x2, x3):
    "Volume d'une boîte parallépipédique"
```

```

    return x1 * x2 * x3

# test :
print volBoite(5.2, 7.7, 3.3)

```

4.

```

def maximum(n1, n2, n3):
    "Renvoie le plus grand de trois nombres"
    if n1 >= n2 and n1 >= n3:
        return n1
    elif n2 >= n1 and n2 >= n3:
        return n2
    else:
        return n3

# test :
print maximum(4.5, 5.7, 3.9)

```

5. Réfléchissez !
6. Réfléchissez !
7. Réfléchissez !

8.

```

def compteCar(ca, ch):
    "Renvoie le nombre de caractères ca trouvés dans la chaîne ch"
    i, tot = 0, 0
    while i < len(ch):
        if ch[i] == ca:
            tot = tot + 1
            i = i + 1
    return tot

# test :
print compteCar("e", "Cette chaîne est un exemple")

```

9.

```

def indexMax(tt):
    "renvoie l'indice du plus grand élément de la liste tt"
    i, max = 0, 0
    while i < len(tt):
        if tt[i] > max :
            max, imax = tt[i], i
            i = i + 1
    return imax

# test :
serie = [5, 8, 2, 1, 9, 3, 6, 4]
print indexMax(serie)

```

10.

```

def nomMois(n):
    "renvoie le nom du n-ième mois de l'année"
    mois = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
            'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
    return mois[n - 1] # les indices sont numérotés à partir de zéro

# test :
print nomMois(4)

```

11. Réfléchissez !

12. Réfléchissez !

Exercices

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

Par exemple :

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

2. Modifiez la fonction `volBoite(x1,x2,x3)` ci-dessus de manière à ce qu'elle puisse être appelée avec un, deux, ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique (l'argument étant l'arête de ce cube). Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée. (Dans ce cas le premier argument est le côté du carré, et le second la hauteur du prisme). Si trois arguments sont utilisés, la boîte est considérée comme un parallélépipède. Par exemple :

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

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

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

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

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

```
i7
>>> print eleMax(serie, 2)
i8
>>> print eleMax(serie, fin =3, debut =1)
i6
```

Solution (cliquez)

```
1. def volBoite(x1 =10, x2 =10, x3 =10):
    "Volume d'une boîte parallépipédique"
    return x1 * x2 * x3
```

```
# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
```

```
2. def volBoite(x1 =-1, x2 =-1, x3 =-1):
    "Volume d'une boîte parallépipédique"
    if x1 == -1 :
        return x1 # aucun argument n'a été fourni
    elif x2 == -1 :
        return x1**3 # un seul argument -> boîte cubique
    elif x3 == -1 :
        return x1*x1*x2 # deux arguments -> boîte prismatique
    else :
        return x1*x2*x3
```

```
# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
print volBoite(5.2, 3, 7.4)
```

```
3. def changeCar(ch, ca1, ca2, debut =0, fin =-1):
    "Remplace tous les caractères ca1 par des ca2 dans la chaîne ch"
    if fin == -1:
        fin = len(ch)
    nch, i = "", 0 # nch : nouvelle chaîne à construire
    while i < len(ch) :
        if i >= debut and i <= fin and ch[i] == ca1:
            nch = nch + ca2
        else :
            nch = nch + ch[i]
        i = i + 1
    return nch
```

```
# test :
print changeCar("Ceci est une toute petite phrase", " ", "*")
print changeCar("Ceci est une toute petite phrase", " ", "*", 8, 12)
print changeCar("Ceci est une toute petite phrase", " ", "*", 12)
```

```
4. def eleMax(lst, debut =0, fin =-1):
    "renvoie le plus grand élément de la liste lst"
    if fin == -1:
```

```
        fin = len(lst)
    max, i = 0, 0
    while i < len(lst):
        if i >= debut and i <= fin and lst[i] > max:
            max = lst[i]
            i = i + 1
    return max

# test :
serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
print eleMax(serie)
print eleMax(serie, 2)
print eleMax(serie, 2, 5)
```


Modules

Définition

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. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des *modules*.

Les modules sont donc des fichiers qui regroupent des ensembles de fonctions. En effet, il peut être commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance et le travail collectif. Une application Python typique sera alors constituée d'un programme principal accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.

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

Un module peut être appelé depuis plusieurs programmes, il s'agit d'un fichier `.py` commençant par son identité (qui ne contient pas de point).

N'importe quel fichier `.py` peut donc être appelé depuis un autre comme un module^[1]. Il peut contenir :

- du script
- des fonctions
- des classes
- ...

Importer un module

Pour utiliser des fonctions de modules dans un programme, il faut au début du fichier importer ceux-ci.

Pour ce faire, utiliser la commande "import" :

```
import os
import codecs
```

Sur la même ligne :

```
import os, codecs
```

Ou encore en sélectionnant tous les éléments d'un fichier :

```
from pywikibot import *
```

A noter : cette dernière méthode est dangereuse, car des objets du module portant le même nom que des objets du programme peuvent s'écraser l'un l'autre. Mieux vaut donc les nommer explicitement :

```
from pywikibot import pagegenerators
```

Le module *math*, par exemple, est fournit avec Python, et 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 *
```

Dans le corps du script lui-même, vous écrirez par exemple :

racine = sqrt(nombre) pour assigner à la variable "racine" la racine carrée de nombre, sinusx = sin(angle) pour assigner à la variable "sinusx" le sinus de angle (en radians !), etc.

Modules personnalisés

Pour créer son propre module et l'importer dans un autre fichier, il faut :

- Soit que le module existe sous la forme d'un fichier .py situé dans le même dossier que le fichier qui l'importe. Exemple :

```
import Fichier1
Fichier1.fonction1()
```

Pour changer le préfixe :

```
import Fichier1 as F1
F1.fonction1()
```

Pour supprimer le préfixe :

```
from Fichier1 import *
```

```
fonction1()
```

- Soit qu'il soit dans un autre dossier du programme, contenant un fichier `__init__.py` jouant le rôle de relais en important tous fichiers situés à côté de lui, ce qui permettra à toutes les fonctions de ce dossier d'être accessibles par le nom du dossier. Exemple dans dans un dossier "lib" :

```
from Fichier1 import *
from Fichier2 import *
from Fichier3 import *

__all__ = ["Fichier1", "Fichier2", "Fichier3"]
```

Dans le dossier parent :

```
import lib
fonction1()
```

Liste des modules standards

Les modules standards les plus importants sont^[2] :

- cgi
- math
- os
- pickle
- random
- re
- socket
- sys
- time
- urllib

Exemple du module "math"

```
# Démo : utilisation des fonctions du module <math>
from math import *
nombre = 121
angle = pi/6 # soit 30° (la bibliothèque math inclut aussi la définition de pi)
print 'racine carrée de', nombre, '=', sqrt(nombre)
print 'sinus de', angle, 'radians', '=', sin(angle)
```

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

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

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

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

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

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

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

Exercices

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

1. Écrivez un programme qui convertisse en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (Rappel : 1 mile = 1609 mètres)
2. Écrivez un programme qui calcule le périmètre et l'aire d'un triangle quelconque dont l'utilisateur fournit les 3 côtés. (Rappel : l'aire d'un triangle quelconque se calcule à l'aide de la formule :

$$S = \sqrt{d \times (d - a) \times (d - b) \times (d - c)}$$
 dans laquelle d désigne la longueur du demi-périmètre, et a , b , c celles des trois côtés).

3. Écrivez un programme qui calcule la période d'un pendule simple de longueur donnée. La formule qui permet de calculer la période d'un

pendule simple est $T = 2\pi\sqrt{\frac{l}{g}}$, l représentant la longueur du pendule

et g la valeur de l'accélération de la pesanteur au lieu d'expérience.

4. Écrivez un programme qui permette d'encoder des valeurs dans une liste. Ce programme devrait fonctionner en boucle, l'utilisateur étant invité à entrer sans cesse de nouvelles valeurs, jusqu'à ce qu'il décide de terminer en frappant <enter> en guise d'entrée. Le programme se

terminerait alors par l'affichage de la liste. Exemple de fonctionnement :

```

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

Solution (cliquez)

1. *# Conversion de miles/heure en km/h et m/s*

```

print "Veillez entrer le nombre de miles parcourus en une heure : ",
ch = raw_input()          # en général préférable à input()
mph = float(ch)           # conversion de la chaîne entrée en nombre réel
mps = mph * 1609 / 3600   # conversion en mètres par seconde
kmph = mph * 1.609       # conversion en km/h
# affichage :
print mph, "miles/heure =", kmph, "km/h, ou encore", mps, "m/s"
    
```

2. *# Périmètre et Aire d'un triangle quelconque*

```

from math import sqrt

print "Veillez entrer le côté a : "
a = float(raw_input())
print "Veillez entrer le côté b : "
b = float(raw_input())
print "Veillez entrer le côté c : "
c = float(raw_input())
d = (a + b + c)/2          # demi-périmètre
s = sqrt(d*(d-a)*(d-b)*(d-c)) # aire (suivant formule)

print "Longueur des côtés =", a, b, c
print "Périmètre =", d*2, "Aire =", s
    
```

3. Réfléchissez !

4. *# Entrée d'éléments dans une liste*

```

tt = []                  # Liste à compléter (vide au départ)
ch = "start"            # valeur quelconque (mais non nulle)
while ch != "":
    print "Veillez entrer une valeur : "
    ch = raw_input()
    if ch != "":
        tt.append(float(ch))          # variante : tt.append(ch)

# affichage de la liste :
print tt
    
```

Références

1. (anglais) <http://docs.python.org/2/tutorial/modules.html>
2. <http://www.python.org/doc/>

Exercices sur les bases du langage

Révision

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

Contrôle du flux - Utilisation d'une liste simple

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

```
-----  
# Utilisation d'une liste et de branchements conditionnels  
  
print ("Ce script recherche le plus grand de trois nombres")  
print ('Veillez entrer trois nombres séparés par des virgules : ')  
# Note : la fonction list() convertit en liste la séquence de données qu'on  
# lui fournit en argument. L'instruction ci-dessous convertira donc les  
# données fournies par l'utilisateur en une liste nn :  
nn = list(input())  
max, index = nn[0], 'premier'  
if nn[1] > max:           # ne pas omettre le double point !  
    max = nn[1]  
    index = 'second'  
if nn[2] > max:  
    max = nn[2]  
    index = 'troisième'  
print ("Le plus grand de ces nombres est", max)  
print ("Ce nombre est le", index, "de votre liste.")  
-----
```

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

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

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

Boucle while - Instructions imbriquées

Continuons dans cette voie en imbriquant d'autres structures :

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

Nous retrouvons ici une boucle `while`, associée à un groupe d'instructions `if`, `elif` et `else`.

L'instruction `while` est utilisée ici pour relancer le questionnement après chaque réponse de l'utilisateur (du moins jusqu'à ce que celui-ci décide de « quitter » en entrant une valeur nulle : rappelons à ce sujet que l'opérateur de comparaison `!=` signifie « est différent de »). Dans le corps de la boucle, nous trouvons le groupe d'instructions `if`, `elif` et `else` (de la ligne 6 à la ligne 16), qui aiguille le flux du programme vers les différentes réponses, ensuite une instruction `print` et une instruction `input()` (lignes 17 & 18) qui seront exécutées dans tous les cas de figure : notez bien leur niveau d'indentation, qui est le même que celui du bloc `if`, `elif` et `else`. Après ces instructions, le programme boucle et l'exécution reprend à l'instruction `while` (ligne 5). Les deux dernières instructions `print` (lignes 19 & 20) ne sont exécutées qu'à la sortie de la boucle.

Exercices

1. 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'
```


2. Que font ces programmes ?

```
a = 5
b = 2
if (a==5) & (b<2):
    print '"&" signifie "et"; on peut aussi utiliser le mot "and"'
```

```
a, b = 2, 4
if (a==4) or (b!=4):
    print 'gagné'
elif (a==4) or (b==4):
    print 'presque gagné'
```

```
a = 1
if not a:
    print 'gagné'
elif a:
    print 'perdu'
```

3. Reprendre le programme c) avec $a = 0$ au lieu de $a = 1$. Que se passe-t-il? Conclure !
4. Écrire un programme qui, étant données deux bornes entières a et b , additionne les nombres multiples de 3 et de 5 compris entre ces bornes. Prendre par exemple $a = 0$, $b = 32$ ® le résultat devrait être alors $0 + 15 + 30 = 45$.
Modifier légèrement ce programme pour qu'il additionne les nombres multiples de 3 ou de 5 compris entre les bornes a et b . Avec les bornes 0 et 32, le résultat devrait donc être : $0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225$.
5. Déterminer si une année (dont le millésime est introduit par l'utilisateur) est bissextile ou non. (Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400).
6. Demander à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher « Cher Monsieur » ou « Chère Mademoiselle » suivi du nom de l'élève.
7. 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.
8. 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.
9. 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
$N \geq 80 \%$	A
$80 \% > N \geq 60 \%$	B

- 60 % > N >= 50 % C
- 50 % > N >= 40 % D
- N < 40 % E

10. Soit la liste suivante :

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

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

11. 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.

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

$$F = 6,67 \times 10^{-11} \times \frac{m \times m'}{d^2}$$

Exemple d'affichage :

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

Solution (cliquez)

1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !

4. *# Traitement de nombres entiers compris entre deux limites*

```
print "Veuillez entrer la limite inférieure :",
a = input()
print "Veuillez entrer la limite supérieure :",
b = input()
s = 0 # somme recherchée (nulle au départ)
# Parcours de la série des nombres compris entre a et b :
n = a # nombre en cours de traitement
while n <= b:
    if n % 3 ==0 and n % 5 ==0: # variante : 'or' au lieu de 'and'
        s = s + n
    n = n + 1
print "La somme recherchée vaut", s
```

5. # Années bissextiles

```

print "Veuillez entrer l'année à tester :",
a = input()

if a % 4 != 0:
    # a n'est pas divisible par 4 -> année non bissextile
    bs = 0
else:
    if a % 400 == 0:
        # a divisible par 400 -> année bissextile
        bs = 1
    elif a % 100 == 0:
        # a divisible par 100 -> année non bissextile
        bs = 0
    else:
        # autres cas ou a est divisible par 4 -> année bissextile
        bs = 1
if bs == 1:
    ch = "est"
else:
    ch = "n'est pas"
print "L'année", a, ch, "bissextile."

Variante (proposée par Alex Misbah) :
a=input('entrée une année:')

if (a%4==0) and ((a%100!=0) or (a%400==0)):
    print a,"est une année bissextile"
else:
    print a,"n'est pas une année bissextile"

Variante (de Mik)
a=input('année:')
if (a%4==0 and a%100!=0) or (a%400==0):
    print "bi6"
else:
    print "nbi6"

```

6. Réfléchissez !

```

7. from sys import exit      # module contenant des fonctions système

print """
Veuillez entrer les longueurs des 3 côtés
(en séparant ces valeurs à l'aide de virgules) :"""
a, b, c = input()
# Il n'est possible de construire un triangle que si chaque côté
# a une longueur inférieure à la somme des deux autres :
if a < (b+c) and b < (a+c) and c < (a+b) :
    print "Ces trois longueurs déterminent bien un triangle."
else:
    print "Il est impossible de construire un tel triangle !"
    exit()      # ainsi l'on n'ira pas plus loin.

f = 0
if a == b and b == c :
    print "Ce triangle est équilatéral."
    f = 1
elif a == b or b == c or c == a :
    print "Ce triangle est isocèle."

```

```

    f = 1
if a*a + b*b == c*c or b*b + c*c == a*a or c*c + a*a == b*b :
    print "Ce triangle est rectangle."
    f = 1
if f == 0 :
    print "Ce triangle est quelconque."

Variante (de Mik)
a,b= input('a:'),input('b:')
if b>a:
    print '"c":>(' ,b-a, ');(' ,a+b, '['
else:
    print '"c":>(' ,a-b, ');(' ,a+b, '['
c=input('c:')
if a < (b+c) and b < (a+c) and c < (a+b) :
    print 'bien un triangle'
else:
    print 'impossible construction triangle'
    exit()
if a>b and a>c:
    max=a
    x=c*c+b*b
elif b>a and b>c:
    max=b
    x=c*c+a*a
elif c>b and c>a:
    max=c
    x=a*a+b*b
if a==b and b==c :
    print "triangle équilatéral."
elif a==b or b==c or c==a :
    print "triangle isocèle."
elif x==max**2 :
    print "triangle rectangle."
else:
    print "triangle quelconque."

```

8. Réfléchissez !
9. Réfléchissez !
10. Réfléchissez !

11. *# Notes de travaux scolaires*

```

notes = []          # liste à construire
n = 2              # valeur positive quelconque pour initier la boucle
while n >= 0 :
    print "Entrez la note suivante, s.v.p. : ",
    n = float(raw_input())      # conversion de l'entrée en un nombre réel
    if n < 0 :
        print "OK. Terminé."
    else:
        notes.append(n)        # ajout d'une note à la liste
        # Calculs divers sur les notes déjà entrées :
        # valeurs minimale et maximale + total de toutes les notes.
        min = 500              # valeur supérieure à toute note
        max, tot, i = 0, 0, 0
        nn = len(notes)        # nombre de notes déjà entrées
        while i < nn:
            if notes[i] > max:
                max = notes[i]
            if notes[i] < min:

```

```

        min = notes[i]
        tot = tot + notes[i]
        moy = tot/nn
        i = i + 1
    print nn, "notes entrées. Max =", max, "Min =", min, "Moy =", moy
    
```

12. Réfléchissez !

Exercices

Exercice 1

Écrire un programme qui affiche "Bonjour le monde".

Solution
<pre>print ("Bonjour le monde")</pre>

Exercice 2

Écrire un programme qui permet de saisir le nom de l'utilisateur et de renvoyer "Bonjour", suivi de ce nom

Solution	
<pre>nom = input ("Quel est votre nom ? ") print ("Bonjour ",nom)</pre>	<pre>ou print ("Bonjour ", input ("Quel est votre nom</pre>

En version 3.1 cela donne
<pre>name = input("Quel est votre nom? ") print("Bonjour " + name + "!")</pre>

Exercice 3

3. Écrire un programme qui demande à l'utilisateur la saisie de a et b et affiche la somme de a et de b.

Solution

```
a = input ("Valeur de a : ")
b = input ("Valeur de b : ")
a= int (a)
b= int (b)
print ("Somme de a+b = ",a+b)
```

En version 3.1 cela pourrait être

```
var_a = int(input("Donnez moi la valeur de a: "))
var_b = int(input("Donnez moi la valeur de b: "))

print("Voici le résultat de a + b : " ,var_a + var_b, "!")
```

Exercice 4

4. Écrire un programme qui demande à l'utilisateur son année de naissance et qui affiche son âge. L'année courante sera mise dans une variable.

Solution

```
# Ce script ne gère pas si l'anniversaire est passé ou non d'où le "environ"
annee_courante = 2016
print ("Quelle est votre année de naissance ?")
reponse = raw_input()
print ("Vous avez environ", annee_courante- int(reponse),"ans")
```

Pour Python 3.1

```
# Ce script gère si l'anniversaire est passé
from datetime import datetime
dn=input('Votre date de naissance ? (format jj/mm/aaaa) :')
dn=dn.split('/') ;jn=int(dn[0]);mn=int(dn[1]);an=int(dn[2])
da=datetime.now()
if mn<da.month or (mn==da.month and jn<=da.day):
    age=da.year-an
else:
    age=da.year-an-1
print('Vous avez',age,'ans.')
```

Exercice 5

5. Écrire un programme qui demande à l'utilisateur les coordonnées de deux points dans le plan et qui calcule puis affiche la distance entre ces deux points

selon la formule : $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Solution

```
import math
print ("xA?")
xA = int(raw_input())
print ("yA?")
yA = int(raw_input())
print ("xB?")
xB = int(raw_input())
print ("yB?")
yB = int(raw_input())
print ("distance entre A et B:", math.sqrt( (xA-xB)**2 + (yA-yB)**2))
```

En version 3.1

```
import math
xA = int(input("Donnez moi Xa"))
yA = int(input("Donnez moi Ya"))
xB = int(input("Donnez moi Xb"))
yB = int(input("Donnez moi Yb"))

distance = math.sqrt((xB - xA)**2 + (yB - yA)**2)

print("Bonjour, la distance qui sépare les deux points est de " + str(distance))
```

Regex

Syntaxe

Expressions rationnelles courantes

Caractère	Type	Explication
.	Point	n'importe quel caractère
[...]	classe de caractères	tous les caractères énumérés dans la classe
[^...]	classe complétée	Tous les caractères sauf ceux énumérés
^	circonflexe	marque le début de la chaîne, la ligne...
\$	dollar	marque la fin d'une chaîne, ligne...
	barre verticale	alternative - ou reconnaît l'un ou l'autre
(...)	parenthèse	utilisée pour limiter la portée d'un masque ou de l'alternative
*	astérisque	0, 1 ou plusieurs occurrences
+	le plus	1 ou plusieurs occurrence
?	interrogation	0 ou 1 occurrence

Les expressions régulières en Python nécessitent le module *re*^[1].

Types

Les types de caractères sont représentés par :

- `\n` : fin de ligne.
- `\b` : fin de mot.
- `\w` : caractère alphanumérique.
- `\d` : caractère numérique.
- `\s` : espace.

Certains ont leurs opposés en capitale :

- `\W` : caractère non-alphanumérique.
- `\D` : caractère non-numérique.
- `\S` : non-espace.

Recherche

- `compile()` renvoie `None` si l'expression rationnelle n'est pas trouvée dans la chaîne.
- `search()` renvoie la position des chaînes recherchées.


```
#!/usr/bin/env python
import re
chaine = "Test regex Python pour Wikibooks francophone."
if re.compile('Wikibooks').search(chaine):
    print "Position du mot Wikibooks : "
    print re.search(u'Wikibooks', chaine).start()
        # Affiche "23"
    print re.search(u'Wikibooks', chaine).end()
        # Affiche "32"
```

- findall() trouve toutes les correspondances dans un tableau.
- finditer() trouve toutes les correspondances dans un itérateur.

```
#!/usr/bin/env python
# Affiche tous les mots qui commencent par "Wiki"
import re
chaine = "Wikilivre regex Python pour Wikibooks francophone."
print (re.findall(r"Wiki\w+", chaine))
# Affiche ['Wikilivre', 'Wikibooks']
```

Les parenthèses imbriquées permettent d'indiquer des mots facultatifs au sein d'un groupe de capture. Ex :

```
#!/usr/bin/env python
# Trouve à un mot prêt
import re
chaine = "Wikilivre regex Python pour Wikibooks francophone."

regex = ur'(Python pour Wikibooks)'
print re.search(regex, chaine).start() # 16

regex = ur'(Python (pour)*Wikibooks)'
print re.search(regex, chaine).start() # 16

regex = ur'(Python pour (les)*Wikibooks)'
print re.search(regex, chaine).start() # 16
```

Les méthodes de la bibliothèque regex sont limitées à une seule ligne par défaut, pour étendre cette limite il faut les appeler avec le paramètre 2 :
re.MULTILINE.

En multiligne, un "^" en début de re.search() considérera le début de chaque ligne. Pour partir uniquement du début de la chaîne, utiliser "re.match()"^[2].

Pour accéder aux résultats des groupes de capture, utiliser group() en partant de 1 :

```
#!/usr/bin/env python
import re
chaine = "Wikilivre regex Python pour Wikibooks francophone."
```


```
s = re.search(ur'(Wiki[a-z]*).*(Wiki[a-z]*)', chaine)
if s:
    print s.group(0)
    # Affiche 'Wikilivre regex Python pour Wikibooks'
    print s.group(1)
    # Affiche 'Wikilivre'
    print s.group(2)
    # Affiche 'Wikibooks'
```

Remplacement

```
#!/usr/bin/env python
# Remplace tous les espaces par des underscores
import re
chaine = "Test regex Python pour Wikibooks francophone."
chaineTrie = re.sub(r' ', "_", chaine)
print chaineTrie
# Affiche "Test_regex_Python_pour_Wikibooks_francophone."
```

Pour remplacer certains éléments en conservant ceux placés entre parenthèses, il faut les désigner par \1, \2, \3...

```
#!/usr/bin/env python
# Ajoute des guillemets à tous les mots suivent "livre"
import re
chaine = "Test regex Python pour le livre Python de Wikibooks francophone."
chaineTrie = re.sub(r'(.*)livre (\w+)(.*)', r'\1livre "\2"\3', chaine)
print chaineTrie
# Affiche "Test regex Python pour le livre "Python" de Wikibooks francophone."
```

Remarque : si les paramètres (\1, \2...) sont remplacés par le symbole , vérifier que la chaîne regex est bien encodée avec r.



Attention !

Les différents contenus d'un même groupe de capture sont remplacés par le premier \1. Pour éviter cela, il faut les traiter un par un avec "finditer()".

Exemple : remplacement de la balise "font color=" par "span style=font-size:".

```
text = ur'<font color=green>Vert</font> / <font color=red>rouge</font>'
regex = ur'<font color=(\[^\>]*)>'
pattern = re.compile(regex, re.UNICODE)
for match in pattern.finditer(text):
    print u'Replacement de ' + match.group(0) + u' par <span style="font-color:' + ma
    text = text.replace(match.group(0), u'<span style="font-color:' + match.group(1) +
    text = text.replace('</font>', u'</span>')
raw_input(text)
```

Exemples de formules

- Récupérer le premier modèle 1 wiki non imbriquée dans un autre modèle :

```
page = u'{{Modèle2|Paramètre2, {{Modèle1|Paramètre3}} }}, {{Modèle1|Paramètre4}}'
regex = ur'({{(.*)}}|.)*[{}]*'
raw_input(re.sub(regex, ur'\2', page).encode(config.console_encoding, 'replace'))
```

Pour indiquer un nombre précis d'occurrences, utiliser "{nombre}". Ex :

```
#!/usr/bin/env python
import re
chaine = 'w.1, ww.2, www.3, www.4'
print re.sub(ur' w{3}\.', ' http://www.', chaine)
```

```
w.1, ww.2, http://www.3, www.4
```

Idem pour une plage de nombres : {min,max}.



Attention !

Quand on injecte une variable dans un pattern, il faut échapper ses caractères interprétables avec `re.escape()`.

Références

1. <https://docs.python.org/2/howto/regex.html>
2. <https://docs.python.org/2/library/re.html#search-vs-match>

Programmation orientée objet pour les non-programmeurs

Le concept objet

Dans les environnements de développement informatique, il a fallu attendre assez longtemps pour voir émerger le concept de l'objet. Son apparition a permis la création de systèmes beaucoup plus complexes mais aussi très empreints de mimétisme. En effet, dans notre monde réel, nous sommes tous entourés d'objets qui ont très souvent deux critères d'appréciation.

Le critère descriptif

Ce premier est universel, il contient toutes les caractéristiques qui décrivent l'objet. Nous prendrons comme exemple un dé, si nous avons à le décrire, nous dirions qu'il possède 6 faces avec un chiffre allant de 1 à 6 sur chacune d'elles, que la somme de deux valeurs étant sur des faces opposées vaut 7, que chaque chiffre entre un et six y est repris une et une seule fois, qu'il est (souvent) de couleur rouge et de petite taille. Il serait possible de le décrire plus précisément, mais en réalité, indiquer qu'il est fait de bois, que les nombres sont représentés par une quantité de point qui leur est égal, qu'il dispose de coin arrondi... n'aurait pas été plus éloquent.

Le critère d'interaction

Le deuxième critère est celui d'interaction, il indique l'utilité de l'objet, les possibilités qu'il vous offre. Pour le dé nous pourrions indiquer que celui-ci peut rouler, mais ce n'est pas son rôle. De même, dans certaines circonstances, celui-ci peut vous servir de cale, mais ici encore, nous nous éloignons du sujet. Objectivement, le dé a pour rôle de donner un nombre compris entre son minimum et son maximum (inclus) au hasard. D'ailleurs, on peut ajouter que cela arrive après l'avoir lancé.

L'héritage et l'implémentation

Ici, nous avons décrit un objet, et il nous suffit de faire de même en informatique. Mais nous pourrions approfondir la description en indiquant aussi que le dé est en fait dérivé d'un objet de base : le cube. Ainsi nous pourrions dire que le dé :

- est un **cube**.
- est de couleur rouge.
- peut être lancé pour renvoyer un nombre compris entre 1 et 6 (le nombre de face qui le compose).

puis expliquer que le cube :

- est un volume géométrique à trois dimensions.
- est constitué de 6 **carrés**.

puis bien sûr qu'un **carré** :

- est une figure géométrique à deux dimensions.

Et nous pourrions continuer précisant le terme **dimension** mais dans notre cas ce n'est pas utile. Nous pouvons ainsi établir le schémas suivant : le dé hérite des caractéristiques du cube (c'est un cube). Mais on ne peut pas dire que le cube hérite des caractéristiques du carré. En effet, on indique bien qu'il est **constitué de** mais pas qu'il **est** et c'est la toute la différence, vous êtes constitué de deux bras musclés mais vous n'êtes pas deux bras musclés (sauf si vous êtes déménageur... *c'est une blague bien entendu, les déménageurs sont suffisamment allègre pour ne pas lancer un avis de recherche sur ma tête*) ! Nous dirons donc que :

- l'objet **cube** implémente l'objet **surface carré**
- l'objet **dé** hérite de l'objet **cube**

Un objet est une entité que l'on construit par *instanciation* à partir d'une *classe* (c'est-à-dire en quelque sorte une « catégorie » ou un « type » d'objet). Par exemple, on peut trouver dans la bibliothèque *Tkinter*, une classe `Button()` à partir de laquelle on peut créer dans une fenêtre un nombre quelconque de boutons.

Nous allons à présent examiner comment vous pouvez vous-mêmes définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais vous l'aborderez de manière très progressive, en commençant par définir des classes d'objets très simples, que vous perfectionnerez ensuite. Attendez-vous cependant à rencontrer des objets de plus en plus complexes par après.

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

Utilité des classes

Les classes sont les principaux outils de la programmation orientée objet ou POO (*Object Oriented Programming* ou *OOP* en anglais). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.

Le premier bénéfice de cette approche de la programmation consiste dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept

d'*encapsulation* : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermés » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies.

En particulier, l'utilisation de classes dans vos programmes vous permettra - entre autres choses - *d'éviter au maximum l'emploi de variables globales*. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, surtout dans les programmes volumineux, parce qu'il est toujours possible que de telles variables soient modifiées ou même redéfinies n'importe où dans le corps du programme (et ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de *construire de nouveaux objets à partir d'objets préexistants*, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de *dérivation* et de *polymorphisme*.

- La dérivation est le mécanisme qui permet de construire une classe « enfant » au départ d'une classe « parente ». L'enfant ainsi obtenu *hérite* toutes les propriétés et toute la fonctionnalité de son ancêtre, auxquelles on peut ajouter ce que l'on veut.
- Le polymorphisme permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte.

La programmation orientée objet est optionnelle sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que les classes constituent des outils pratiques et puissants. Une bonne compréhension des classes vous aidera notamment à maîtriser le domaine des interfaces graphiques (*Tkinter*, *wxPython*), et vous préparera efficacement à aborder d'autres langages modernes tels que *C++* ou *Java*.

Objet = [attributs + méthodes]

Cette façon d'associer dans une même « capsule » les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel qui nous entoure.

Considérons par exemple un *widget* « bouton ». Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons ainsi ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel. Or la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est bien intégrée dans l'objet lui-même (au même titre que d'autres propriétés telles que sa taille, sa couleur, etc.) De la même

manière, nous souhaiterons que les différentes caractéristiques de notre bouton logiciel (sa taille, son emplacement, sa couleur, le texte qu'il supporte), mais aussi la définition de ce qui se passe lorsque l'on effectue différentes actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, de manière telle qu'il n'y ait pas de confusion avec un autre bouton ou d'autres entités.

Classes

Définition d'une classe élémentaire

Pour créer une nouvelle classe d'objets Python, donc un nouveau type de donnée, on utilise l'instruction `class`. 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).

Par exemple, nous allons maintenant créer un nouveau type composite : le type `Point`. Ce type correspondra au concept de point en mathématiques. Dans un espace à deux dimensions, un point est caractérisé par deux nombres (ses coordonnées suivant x et y). En notation mathématique, on représente donc un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point $(25, 17)$. Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type `float`. Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une classe `Point()` :

```
>>> class Point:
    "Définition d'un point mathématique"
```

Remarquons d'emblée que :

- L'instruction `class` est un nouvel exemple d'*instruction composée*. Ce bloc doit contenir au moins une ligne. Dans notre exemple, 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 sera considérée comme un commentaire et 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.
- 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 encore 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 pouvons dès à présent nous servir de cette classe pour créer des objets de ce type, par *instanciation*. Créons par exemple un nouvel objet `p9`.

Remarque : sous Python, on peut donc instancier un objet à l'aide d'une simple instruction d'affectation. D'autres langages imposent l'emploi d'une instruction spéciale, souvent appelée "new" pour bien montrer que l'on crée un nouvel objet à partir d'un moule. Exemple : `p9 = new Point()`.


```
>>> 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()`.

Remarque : 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). Remarquez bien cependant que la *définition* d'une classe ne nécessite pas de parenthèses (contrairement à ce qui est de règle lors de la définition des fonctions), sauf si nous souhaitons que la classe en cours de définition dérive d'une autre classe préexistante.

Attributs (ou variables) d'instance

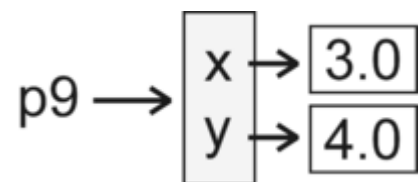
L'objet que nous venons de créer est une coquille vide. Nous pouvons ajouter des composants à cet objet par simple assignation, en utilisant le système de qualification des noms par points.

Remarque : ce système de notation est similaire à celui que nous utilisons pour désigner les variables d'un module, comme par exemple `"math.pi"` ou `"string.uppercase"`. Les modules peuvent en effet contenir des fonctions, mais aussi des classes et des variables. Essayez par exemple :

```
>>> import string
>>> print string.uppercase
>>> print string.lowercase
>>> print string.hexdigits
```

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

Les variables ainsi définies sont des *attributs* de l'objet `p9`, ou encore des *variables d'instance*. Elles sont incorporées, ou plutôt *encapsulées* dans l'objet. Le diagramme d'état ci-contre montre le résultat de ces affectations : la variable `p9` contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs `x` et `y`.



On peut utiliser les attributs d'un objet dans n'importe quelle expression, comme toutes les variables ordinaires :

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

Du fait de leur encapsulation dans l'objet, les attributs sont des variables distinctes d'autres variables qui pourraient porter le même nom. Par exemple, l'instruction `x = p9.x` signifie : « extraire de l'objet référencé par `p9` la valeur de son attribut `x`, et assigner cette valeur à la variable `x` ».

Il n'y a pas de conflit entre la variable `x` et l'attribut `x` de l'objet `p9`. L'objet `p9` contient en effet son propre espace de noms, indépendant de l'espace de nom principal où se trouve la variable `x`.

Remarque : nous venons de voir qu'il est très aisé d'ajouter un attribut à un objet en utilisant une simple instruction d'assignation telle que `p9.x = 3.0`. On peut se permettre cela sous Python (c'est une conséquence de l'assignation dynamique des variables), mais *cela n'est pas vraiment recommandable*. En effet, nous n'utiliserons cette façon de faire uniquement dans le but de simplifier nos explications concernant les attributs d'instances.

Passage d'objets comme arguments lors de l'appel d'une fonction

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
```

Exercices

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

Solution (cliquez)

1. Réfléchissez !

Similitude et unicité

Dans la langue parlée, les mêmes mots peuvent avoir des significations fort différentes suivant le contexte dans lequel on les utilise. La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs manières différentes (expressions ambiguës).

Le mot « même », par exemple, a des significations différentes dans les phrases : « Charles et moi avons la même voiture » et « Charles et moi avons la même mère ». Dans la première, ce que je veux dire est que la voiture de Charles et la mienne sont du même modèle. Il s'agit pourtant de deux voitures distinctes. Dans la seconde, j'indique que la mère de Charles et la mienne constituent en fait une seule et unique personne.

Lorsque nous traitons d'objets logiciels, nous pouvons rencontrer la même ambiguïté. Par exemple, si nous parlons de l'égalité de deux objets `Point()`, cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ? Considérez par exemple les instructions suivantes :

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> print (p1 == p2)
0
```

Ces instructions créent deux objets `p1` et `p2` qui restent distincts, même s'ils ont des contenus similaires. La dernière instruction teste l'égalité de ces deux objets (double signe égale), et le résultat est zéro (ce qui signifie que l'expression entre parenthèses est fausse : il n'y a donc pas égalité).

On peut confirmer cela d'une autre manière encore :

```
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C50F9C>
```

L'information est claire : les deux variables `p1` et `p2` référencent bien des objets différents.

Essayons autre chose, à présent :

```
>>> p2 = p1
>>> print (p1 == p2)
```

```
1
```

Par l'instruction `p2 = p1`, nous assignons le contenu de `p1` à `p2`. Cela signifie que désormais ces deux variables référencent le même objet. Les variables `p1` et `p2` sont des alias l'une de l'autre.

Le test d'égalité dans l'instruction suivante renvoie cette fois la valeur 1, ce qui signifie que l'expression entre parenthèses est vraie : `p1` et `p2` désignent bien toutes deux un seul et unique objet, comme on peut s'en convaincre en essayant encore :

```
>>> p1.x = 7
>>> print p2.x
7
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C2CBEC>
```

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 :

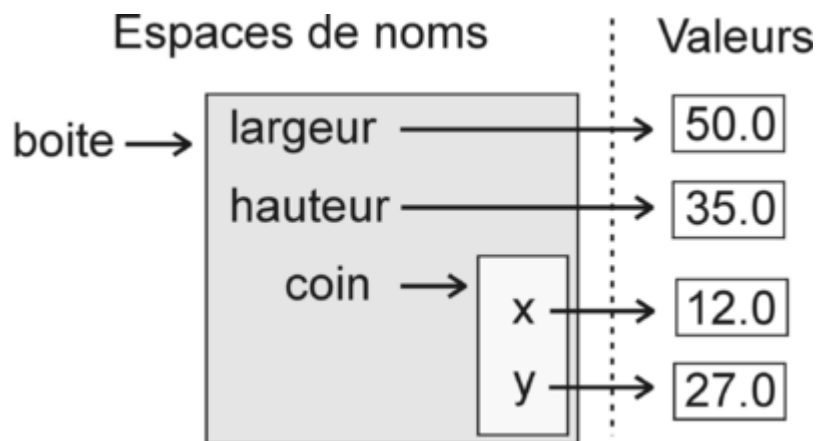
```
>>> boite = Rectangle()
>>> boite.largeur = 50.0
>>> boite.hauteur = 35.0
```

Nous créons ainsi un nouvel objet `Rectangle()` et deux attributs. 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. Ainsi l'expression `boite.coin.y` signifie « Aller à l'objet référencé dans la variable `boite`. Dans cet objet, repérer l'attribut `coin`, puis aller à l'objet référencé dans cet attribut. Une fois cet autre objet trouvé, sélectionner son attribut `y`. »

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



Le nom « boîte » se trouve dans l'espace de noms principal. Il référence un autre espace de noms réservé à l'objet correspondant, dans lequel sont mémorisés les noms « largeur », « hauteur » et « coin ». Ceux-ci référencent à leur tour, soit d'autres espaces de noms (cas du nom « coin »), soit des valeurs bien déterminées. Python réserve des espaces de noms différents pour chaque module, chaque classe, chaque instance, chaque fonction. Vous pouvez tirer parti de tous ces espaces bien compartimentés afin de réaliser des programmes robustes, c'est-à-dire des programmes dont les différents composants ne peuvent pas facilement interférer.

Objets comme valeurs de retour d'une fonction

Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour. Par exemple, la fonction `trouveCentre()` ci-dessous doit être appelée avec un argument de type `Rectangle()` et elle renvoie un objet `Point()`, lequel contiendra les coordonnées du centre du rectangle.

```
>>> def trouveCentre(box):
```

```
p = Point()
p.x = box.coin.x + box.largeur/2.0
p.y = box.coin.y + box.hauteur/2.0
return p
```

Pour appeler cette fonction, vous pouvez utiliser l'objet `boite` comme argument :

```
>>> centre = trouveCentre(boite)
>>> print centre.x, centre.y
37.0 44.5
```

Les objets sont modifiables

Nous pouvons changer les propriétés d'un objet en assignant de nouvelles valeurs à ses attributs. Par exemple, nous pouvons modifier la taille d'un rectangle (sans modifier sa position), en réassignant ses attributs hauteur et largeur :

```
>>> boite.hauteur = boite.hauteur + 20
>>> boite.largeur = boite.largeur - 5
```

Nous pouvons faire cela sous Python, parce que dans ce langage les propriétés des objets sont toujours publiques (du moins dans la version actuelle 2.0). D'autres langages établissent une distinction nette entre attributs publics (accessibles de l'extérieur de l'objet) et attributs privés (qui sont accessibles seulement aux algorithmes inclus dans l'objet lui-même).

Comme nous l'avons déjà signalé plus haut (à propos de la définition des attributs par assignation simple, depuis l'extérieur de l'objet), *modifier de cette façon les attributs d'une instance n'est pas une pratique recommandable*, parce qu'elle contredit l'un des objectifs fondamentaux de la programmation orientée objet, qui vise à établir une séparation stricte entre la fonctionnalité d'un objet (telle qu'elle a été déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

Plus concrètement, nous devons veiller désormais à ce que les objets que nous créons ne soient modifiables en principe que par l'intermédiaire de méthodes mises en place spécifiquement dans ce but, comme nous allons l'expliquer dans le chapitre suivant.

Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle classe *Time*, qui

nous permettra d'effectuer toute une série d'opérations sur des instants, des durées, etc. :

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

Créons à présent un objet de ce type, et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
>>> instant = Time()
>>> instant.heure = 11
>>> instant.minute = 34
>>> instant.seconde = 25
```

À titre d'exercice, écrivez maintenant vous-même une fonction `affiche_heure()`, qui serve à visualiser le contenu d'un objet de classe `Time()` sous la forme conventionnelle « heure:minute:seconde ».

Appliquée à l'objet `instant` créé ci-dessus, cette fonction devrait donc afficher 11:34:25 :

```
>>> print affiche_heure(instant)
11:34:25
```

Votre fonction ressemblera probablement à ceci :

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

(Notez au passage l'utilisation de la fonction `str()` pour convertir les données numériques en chaînes de caractères). 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 `affiche_heure()` dans la classe `Time()` elle-même, 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*.

Définition concrète d'une méthode

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

- *La définition d'une méthode est toujours placée à l'intérieur de la définition d'une classe*, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.
- Le premier paramètre utilisé par une méthode doit toujours être une référence d'instance. Vous pourriez en principe utiliser un nom de variable quelconque pour ce paramètre, mais il est vivement conseillé de respecter la convention qui consiste à toujours lui donner le nom `self`. Le paramètre `self` désigne donc l'instance à laquelle la méthode sera associée, dans les instructions faisant partie de la définition. (De ce fait, la définition d'une méthode comporte toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Voyons comment cela se passe en pratique :

Pour ré-écrire la fonction `affiche_heure()` comme une méthode de la classe `Time()`, il nous suffit de déplacer sa définition à l'intérieur de celle de la classe, et de 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.

Essai de la méthode dans une instance

Nous pouvons dès à présent instancier un objet de notre nouvelle classe `Time()` :

```
>>> 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 attributs d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
```



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

Nous avons cependant déjà signalé à plusieurs reprises qu'il n'est pas recommandable de créer ainsi les attributs d'instance en dehors de l'objet lui-même, ce qui conduit (entre autres désagréments) à des erreurs comme celle que nous venons de rencontrer, par exemple.

Voyons donc à présent comment nous pouvons mieux faire.

Méthodes prédéfinies

Certaines méthodes de classe Python existent automatiquement dans toutes les classes sans être déclarées, et certaines se lancent automatiquement lors de certains événements. Ces méthodes spéciales sont nommées entre deux underscores (`__`)^[1].

`__doc__`

Comme pour les fonctions, la chaîne de documentation est définie dans cette méthode.

```
>>> print p9.__doc__
Définition d'un point mathématique
```

`__contains__`

Cette méthode permet de lancer des recherches dans une classes comme dans un objet composite comme la liste, avec "in". En effet, il suffit d'y placer les getters sur les attributs :

```
class MyClass:
    attribut1 = 'ok'

    def __contains__(self, attribute):
        if self.attribut1: return True

MaClassel = MyClass()
print attribut1 in MaClassel # True
print attribute2 in MaClassel # False
```

`__del__`

Destructeur : se lance quand l'objet est détruit.

`__enter__` et `__exit__`

Respectivement constructeur et destructeur des classes instanciées avec `with`, exécutés respectivement après et avant `__init__` et `__del__`. Exemple :

```
class Test:
    def __enter__(self):
        print 'enter'

    def __exit__(self, exc_type, exc_value, traceback):
        print 'exit'

with Test():
    pass
```

```
enter
exit
```

`__init__`

Cette méthode se lance lors du premier accès à la classe.

Exemple :

```
class Complexe:
    def __init__(self, r, i):
        self.reel = r
        self.imaginaire = i

Complexel = Complexe(1, 2)
print Complexel.reel # Affiche 1
```

L'erreur que nous avons rencontrée au paragraphe précédent est-elle évitable ? Elle ne se produirait effectivement pas, si nous nous étions arrangés pour que la méthode `affiche_heure()` puisse toujours afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable aucune manipulation sur l'objet nouvellement créé. En d'autres termes, *il serait judicieux que les variables d'instance soient prédéfinies elles aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur « par défaut ».*

Pour obtenir cela, nous allons faire appel à une méthode particulière, que l'on appelle un constructeur. Une méthode constructeur est une méthode qui est exécutée automatiquement lorsque l'on instancie un nouvel objet à partir de la classe. On peut y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée.

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)

>>> tstart = Time()
>>> tstart.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
```

La méthode `__init__()` comporte à présent 3 paramètres, avec pour chacun une valeur par défaut. Pour lui transmettre les arguments correspondants, il suffit de placer ceux-ci dans les parenthèses qui accompagnent le nom de la classe, lorsque l'on écrit l'instruction d'instanciation du nouvel objet.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet `Time()` :

```
>>> recreation = Time(10, 15, 18)
>>> recreation.affiche_heure()
10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi bien créer de tels objets `Time()` en omettant un ou plusieurs arguments :

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

__main__

```
>>> print p9
<__main__.Point instance at 0x403e1a8c>
```

Le message renvoyé par Python indique, comme vous l'aurez certainement bien compris tout de suite, que "p9" est une instance de la classe "Point()", qui est définie elle-même au niveau principal du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale.

__new__

Constructeur de métaclasse.

__repr__

Cette méthode renvoie une représentation de l'objet quand on l'appelle directement (sans chercher à accéder à ses attributs ou méthodes). Exemple :

```
class Bar:
    def __init__(self, iamthis):
        self.iamthis = iamthis

    def __repr__(self):
        return "Bar('%s')" % self.iamthis

bar = Bar('apple')
print bar
```

```
Bar('apple')
```

Dans cette méthode, cet objet renverrait :

```
<__main__.Bar instance at 0x7f282bbf2a28>
```

__str__

Renvoie une chaîne de caractères quand on traite l'objet comme tel. Exemple :

```
class Bar:
    def __init__(self, iam_this):
        self.iam_this = iam_this
```

```

def __str__(self):
    return self.iam_this

bar = Bar('apple')
print bar

```

apple

__unicode__

Réservé à Python 2.x.

Opérateurs binaires

Fonction	Opérateur
<code>__add__</code>	A + B
<code>__sub__</code>	A - B
<code>__mul__</code>	A * B
<code>__truediv__</code>	A / B
<code>__floordiv__</code>	A // B
<code>__mod__</code>	A % B
<code>__pow__</code>	A ** B
<code>__and__</code>	A & B
<code>__or__</code>	A B
<code>__xor__</code>	A ^ B
<code>__eq__</code>	A == B
<code>__ne__</code>	A != B
<code>__gt__</code>	A > B
<code>__lt__</code>	A < B
<code>__ge__</code>	A >= B
<code>__le__</code>	A <= B
<code>__lshift__</code>	A << B
<code>__rshift__</code>	A >> B
<code>__contains__</code>	A in B A not in B

Opérateurs unaires

Fonction	Opérateur
<code>__pos__</code>	<code>+A</code>
<code>__neg__</code>	<code>-A</code>
<code>__inv__</code>	<code>~A</code>
<code>__abs__</code>	<code>abs(A)</code>
<code>__len__</code>	<code>len(A)</code>

Gestion des attributs

Getters et setters.

Fonction	Forme indirecte	Forme directe
<code>__getattr__</code>	<code>getattr(A, B)</code>	<code>A.B</code>
<code>__setattr__</code>	<code>setattr(A, B, C)</code>	<code>A.B = C</code>
<code>__delattr__</code>	<code>delattr(A, B)</code>	<code>del A.B</code>

Gestion des indices

Se déclenchent lorsque l'on manipule un objet comme un dictionnaire^[2].

Fonction	Opérateur
<code>__getitem__</code>	<code>C[i]</code>
<code>__setitem__</code>	<code>C[i] = v</code>
<code>__delitem__</code>	<code>del C[i]</code>
<code>__getslice__</code>	<code>C[s:e]</code>
<code>__setslice__</code>	<code>C[s:e] = v</code>
<code>__delslice__</code>	<code>del C[s:e]</code>

Divers

Fonction	Opérateur
<code>__cmp__</code>	<code>cmp(x, y)</code>
<code>__hash__</code>	<code>hash(x)</code>
<code>__nonzero__</code>	<code>bool(x)</code>
<code>__call__</code>	<code>f(x)</code>
<code>__iter__</code>	<code>iter(x)</code>
<code>__reversed__</code>	<code>reversed(x)</code> (2.6+)
<code>__divmod__</code>	<code>divmod(x, y)</code>
<code>__int__</code>	<code>int(x)</code>
<code>__long__</code>	<code>long(x)</code>
<code>__float__</code>	<code>float(x)</code>
<code>__complex__</code>	<code>complex(x)</code>
<code>__hex__</code>	<code>hex(x)</code>
<code>__oct__</code>	<code>oct(x)</code>
<code>__index__</code>	
<code>__copy__</code>	<code>copy.copy(x)</code>
<code>__deepcopy__</code>	<code>copy.deepcopy(x)</code>
<code>__sizeof__</code>	<code>sys.getsizeof(x)</code> (2.6+)
<code>__trunc__</code>	<code>math.trunc(x)</code> (2.6+)
<code>__format__</code>	<code>format(x, ...)</code> (2.6+)

Espaces de noms des classes et instances

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 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.

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 sont 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 précédemment. À 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

```

Veillez 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 aa, Espaces.aa, essai.aa               # 9
12 33 67

```

Dans cet exemple, le même nom `aa` est utilisé pour définir trois variables différentes : une dans l'espace de noms de la classe (à la ligne 2), une autre dans l'espace de noms principal (à la ligne 5), et enfin une dernière dans l'espace de nom de l'instance (à la ligne 7).

La ligne 4 et la ligne 9 montrent comment vous pouvez accéder à ces trois espaces de noms (de l'intérieur d'une classe, ou au niveau principal), en utilisant la qualification par points. Notez encore une fois l'utilisation de `self` pour désigner l'instance.

Héritage

Les classes constituent le principal outil de la programmation orientée objet ou POO (*Object Oriented Programming* ou *OOP* en anglais), 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 différentes ou 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 toutes des 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 toutes les caractéristiques de la classe `Mammifere` 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 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 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 :

```
1 >>> mirza.caract2 = "son corps est couvert de poils"
2 >>> print mirza.caract2
3 son corps est couvert de poils
4 >>> fido = Chien()
```

```
5 >>> print fido.caract2
6 il se nourrit de la chair de ses proies ;
```

Dans ce nouvel exemple, la ligne 1 ne modifie pas l'attribut `caract2` de la classe `Carnivore()`, contrairement à ce que l'on pourrait penser au vu de la ligne 3. Nous pouvons le vérifier en créant une nouvelle instance `fido` (lignes 4 à 6).

Si vous avez bien assimilé les paragraphes précédents, vous aurez compris que l'instruction de la ligne 1 crée une nouvelle variable d'instance associée seulement à l'objet `mirza`. Il existe donc dès ce moment deux variables avec le même nom `caract2` : l'une dans l'espace de noms de l'objet `mirza`, et l'autre dans l'espace de noms de la classe `Carnivore()`.

Comment faut-il alors interpréter ce qui s'est passé aux lignes 2 et 3 ? Comme nous l'avons vu plus haut, l'instance `mirza` peut accéder aux variables situées dans son propre espace de noms, mais aussi à celles qui sont situées dans les espaces de noms de toutes les classes parentes. S'il existe des variables aux noms identiques dans plusieurs de ces espaces, laquelle sera-t-elle sélectionnée lors de l'exécution d'une instruction comme celle de la ligne 2 ?

Pour résoudre ce conflit, Python respecte une règle de priorité fort simple. Lorsqu'on lui demande d'utiliser la valeur d'une variable nommée *alpha*, par exemple, il commence par rechercher ce nom dans l'espace local (le plus « interne », en quelque sorte). Si une variable *alpha* est trouvée dans l'espace local, c'est celle-là qui est utilisée, et la recherche s'arrête. Sinon, Python examine l'espace de noms de la structure parente, puis celui de la structure grand-parente, et ainsi de suite jusqu'au niveau principal du programme.

À la ligne 2 de notre exemple, c'est donc la variable d'instance qui sera utilisée. À la ligne 5, par contre, c'est seulement au niveau de la classe grand-parente qu'une variable répondant au nom `caract2` peut être trouvée. C'est donc celle-là qui est affichée.

Héritage et polymorphisme

Pour bien comprendre ce script, il faut cependant d'abord vous rappeler quelques notions élémentaires de chimie. Dans votre cours de *chimie*, vous avez certainement dû apprendre que les *atomes* sont des entités constituées d'un certain nombre de *protons* (particules chargées d'électricité positive), d'*électrons* (chargés négativement) et de *neutrons* (neutres).

Le type d'atome (ou élément) est déterminé par le nombre de protons, que l'on appelle également *numéro atomique*. Dans son état fondamental, un atome contient autant d'électrons que de protons, et par conséquent il est électriquement neutre. Il possède également un nombre variable de neutrons, mais ceux-ci n'influencent en aucune manière la charge électrique globale.

Dans certaines circonstances, un atome peut gagner ou perdre des électrons. Il acquiert de ce fait une charge électrique globale, et devient alors un *ion* (il

s'agit d'un *ion négatif* si l'atome a gagné un ou plusieurs électrons, et d'un *ion positif* s'il en a perdu). La charge électrique d'un ion est égale à la différence entre le nombre de protons et le nombre d'électrons qu'il contient.

Le script suivant génère des objets `atome` et des objets `ion`. Nous avons rappelé ci-dessus qu'un ion est simplement un atome modifié. Dans notre programmation, la classe qui définit les objets « ion » sera donc une classe dérivée de la classe `atome` : elle héritera d'elle tous ses attributs et toutes ses méthodes, en y ajoutant les siennes propres. On pourra dire également que la méthode `affiche()` a été surchargée.

L'une de ces méthodes ajoutées (la méthode `affiche()`) remplace une méthode de même nom héritée de la classe `atome`. Les classes « `atome` » et « `ion` » possèdent donc chacune une méthode de même nom, mais qui effectuent un travail différent. On parle dans ce cas de polymorphisme. On pourra dire également que la méthode `affiche()` a été *surchargée*.

Il sera évidemment possible d'instancier un nombre quelconque d'atomes et d'ions à partir de ces deux classes. Or l'une d'entre elles (la classe `atome`) doit contenir une version simplifiée du tableau périodique des éléments (tableau de Mendelév), de façon à pouvoir attribuer un nom d'élément chimique, ainsi qu'un nombre de neutrons, à chaque objet généré. Comme il n'est pas souhaitable de recopier tout ce tableau dans chacune des instances, nous le placerons dans *un attribut de classe*. Ainsi ce tableau n'existera qu'en un seul endroit en mémoire, tout en restant accessible à tous les objets qui seront produits à partir de cette classe.

Voyons concrètement comment toutes ces idées s'articulent :

```
class Atome:
    """atomes simplifiés, choisis parmi les 10 premiers éléments du TP"""
    table = [None, ('hydrogène',0), ('hélium',2), ('lithium',4),
             ('béryllium',5), ('bore',6), ('carbone',6), ('azote',7),
             ('oxygène',8), ('fluor',10), ('néon',10)]

    def __init__(self, nat):
        """le n° atomique détermine le n. de protons, d'électrons et de neutrons"""
        self.np, self.ne = nat, nat          # nat = numéro atomique
        self.nn = Atome.table[nat][1]      # nb. de neutrons trouvés dans table

    def affiche(self):
        print
        print "Nom de l'élément :", Atome.table[self.np][0]
        print "%s protons, %s électrons, %s neutrons" % \
              (self.np, self.ne, self.nn)

class Ion(Atome):
    """les ions sont des atomes qui ont gagné ou perdu des électrons"""

    def __init__(self, nat, charge):
        """le n° atomique et la charge électrique déterminent l'ion"""
        Atome.__init__(self, nat)
        self.ne = self.ne - charge
        self.charge = charge
```

```

def affiche(self):
    "cette méthode remplace celle héritée de la classe parente"
    Atome.affiche(self)      # ... tout en l'utilisant elle-même !
    print "Particule électrisée. Charge =", self.charge

### Programme principal : ###

a1 = Atome(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.affiche()
a2.affiche()
a3.affiche()

```

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

```

Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons
Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
Particule électrisée. Charge = 1
Nom de l'élément : oxygène
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2

```

Au niveau du programme principal, vous pouvez constater que l'on instancie les objets `Atome()` en fournissant leur numéro atomique (lequel doit être compris entre 1 et 10). Pour instancier des objets `Ion()`, par contre, on doit fournir un numéro atomique et une charge électrique globale (positive ou négative). La même méthode `affiche()` fait apparaître les propriétés de ces objets, qu'il s'agisse d'atomes ou d'ions, avec dans le cas de l'ion une ligne supplémentaire (polymorphisme).

Commentaires

La définition de la classe `Atome()` commence par l'assignation de la variable `table`. Une variable définie à cet endroit fait partie de l'espace de noms de la classe. C'est donc un *attribut de classe*, dans lequel nous plaçons une liste d'informations concernant les 10 premiers éléments du tableau périodique de Mendeleïev. Pour chacun de ces éléments, la liste contient un tuple : (nom de l'élément, nombre de neutrons), à l'indice qui correspond au numéro atomique. Comme il n'existe pas d'élément de numéro atomique zéro, nous avons placé à l'indice zéro dans la liste, l'objet spécial `None` (*a priori*, nous aurions pu placer à cet endroit n'importe quelle autre valeur, puisque cet indice ne sera pas utilisé. L'objet `None` de Python nous semble cependant particulièrement explicite).

Viennent ensuite les définitions de deux méthodes :

- Le constructeur `__init__()` sert essentiellement ici à générer trois *attributs d'instance*, destinés à mémoriser respectivement les nombres de protons, d'électrons et de neutrons pour chaque objet atome construit à partir de cette

classe (Les attributs d'instance sont des variables liées à `self`).

Notez bien la technique utilisée pour obtenir le nombre de neutrons à partir de l'attribut de classe, en mentionnant le nom de la classe elle-même dans une qualification par points.

- La méthode `affiche()` utilise à la fois les attributs d'instance, pour retrouver les nombres de protons, d'électrons et de neutrons de l'objet courant, et l'attribut de classe (lequel est commun à tous les objets) pour en extraire le nom d'élément correspondant. Veuillez aussi remarquer au passage l'utilisation de la technique de formatage des chaînes.

La définition de la classe `Ion()` comporte des parenthèses. Il s'agit donc d'une *classe dérivée*, sa *classe parente* étant bien entendu la classe `Atome()` qui précède.

Les méthodes de cette classe sont des variantes de celles de la classe `Atome()`. Elles devront donc vraisemblablement faire appel à celles-ci. Cette remarque est importante :

Remarque : comment peut-on, à l'intérieur de la définition d'une classe, faire appel à une méthode définie dans une autre classe ?

Il ne faut pas perdre de vue, en effet, qu'une méthode se rattache toujours à l'instance qui sera générée à partir de la classe (instance représentée par `self` dans la définition). Si une méthode doit faire appel à une autre méthode définie dans une autre classe, il faut pouvoir lui transmettre la référence de l'instance à laquelle elle doit s'associer. Comment faire ? C'est très simple :

Remarque : lorsque dans la définition d'une classe, on souhaite faire appel à une méthode définie dans une autre classe, on doit lui transmettre la référence de l'instance comme premier argument.

C'est ainsi que dans notre script, par exemple, la méthode `affiche()` de la classe `Ion()` peut faire appel à la méthode `affiche()` de la classe `Atome()` : les informations affichées seront bien celles de l'objet-ion courant, puisque sa référence a été transmise dans l'instruction d'appel :

```
Atome.affiche(self)
```

(dans cette instruction, `self` est bien entendu la référence de l'instance courante).

De la même manière, la méthode constructeur de la classe `Ion()` fait appel à la méthode constructeur de sa classe parente, dans :

```
Atome.__init__(self, nat)
```

Résumé : Définition et utilisation d'une classe

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

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

class Rectangle:
    """rectangle"""
    def __init__(self, ang, lar, hau):
        self.ang = ang
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        xc = self.ang.x + self.lar /2
        yc = self.ang.y + self.hau /2
        return Point(xc, yc)

class Carre(Rectangle):
    """carré = rectangle particulier"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self,
            coin, cote, cote)
        self.cote = cote

    def surface(self):
        return self.cote**2

#####
## Programme principal : ##

# coord. de 2 coins sup. gauches :
csgR = Point(40,30)
csgC = Point(10,25)

# "boîtes" rectangulaire et carrée :
boiteR = Rectangle(csgR, 100, 50)
boiteC = Carre(csgC, 40)

# Coordonnées du centre pour chacune :
cR = boiteR.trouveCentre()
cC = boiteC.trouveCentre()

print "centre du rect. :", cR.x, cR.y
print "centre du carré :", cC.x, cC.y

print "surf. du carré :",
print boiteC.surface()
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une instance de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui posséderont seulement un attribut 'x' et un attribut 'y'; ils ne seront dotés d'aucune méthode.

Le paramètre self désigne toutes les instances qui seront produites par cette classe

Les instances de la classe Rectangle() posséderont 3 attributs : le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

Pour créer (ou instancier) un objet, il suffit d'affecter une classe à une variable. Les instructions ci-contre créent donc deux objets de la classe Point()...

... et celles-ci, encore deux autres objets. Note : par convention, le nom d'une classe commence par une lettre majuscule

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de classe Rectangle().

Par contre, la méthode surface() ne peut être invoquée que pour les objets carrés.

Exercices

Exercices

1. Définissez une classe Domino() qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0). Deux autres méthodes seront définies :

- une méthode `affiche_points()` qui affiche les points présents sur les deux faces ;
- une méthode `valeur()` qui renvoie la somme des points présents sur les deux faces.

Exemples d'utilisation de cette classe :

```

>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
face A : 2 face B : 6
>>> d2.affiche_points()
face A : 4 face B : 3
>>> print "total des points :", d1.valeur() + d2.valeur()
total des points : 15
>>> liste_dominos = []
>>> for i in range(7):
    liste_dominos.append(Domino(6, i))

>>> liste_dominos[1].affiche_points()
face A: 6 face B: 1

etc., etc.
    
```

2. Définissez une classe `CompteBancaire()`, qui permette d'instancier des objets tels que `compte1`, `compte2`, etc. Le constructeur de cette classe initialisera deux attributs d'instance "nom" et "solde", avec les valeurs par défaut 'Dupont' et 1000. Trois autres méthodes seront définies :
 - `depot(somme)` permettra d'ajouter une certaine somme au solde
 - `retrait(somme)` permettra de retirer une certaine somme du solde
 - `affiche()` permettra d'afficher le nom du titulaire et le solde de son compte.

Exemples d'utilisation de cette classe :

```

>>> comptel = CompteBancaire('Duchmol', 800)
>>> comptel.depot(350)
>>> comptel.retrait(200)
>>> comptel.affiche()
Le solde du compte bancaire de Duchmol est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
    
```

3. Définissez une classe `Voiture()` qui permette d'instancier des objets reproduisant le comportement de voitures automobiles. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées : `marque = 'Ford'`, `couleur = 'rouge'`, `pilote = 'personne'`, `vitesse = 0`. Lorsque l'on instanciera un nouvel objet `Voiture()`, on pourra choisir sa marque et sa couleur, mais pas sa vitesse, ni le nom de son conducteur. Les méthodes suivantes seront définies : -
 - `choix_conducteur(nom)` permettra de désigner (ou changer) le nom du conducteur
 - `accelerer(taux, duree)` permettra de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit : $\text{taux} \times \text{duree}$. Par exemple, si la voiture accélère au taux de 1,3 m/s² pendant 20

secondes, son gain de vitesse doit être égal à 26 m/s. Des taux négatifs seront acceptés (ce qui permettra de décélérer). La variation de vitesse ne sera pas autorisée si le conducteur est 'personne'. - `affiche_tout()` permettra de faire apparaître les propriétés présentes de la voiture, c'est-à-dire sa marque, sa couleur, le nom de son conducteur, sa vitesse. Exemples d'utilisation de cette classe :

```

>>> a1 = Voiture('Peugeot', 'bleue')
>>> a2 = Voiture(couleur = 'verte')
>>> a3 = Voiture('Mercedes')
>>> a1.choix_conducteur('Roméo')
>>> a2.choix_conducteur('Juliette')
>>> a2.accelerer(1.8, 12)
>>> a3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> a2.affiche_tout()
Ford verte pilotée par Juliette, vitesse = 21.6 m/s.
>>> a3.affiche_tout()
Mercedes rouge pilotée par personne, vitesse = 0 m/s.
    
```

4. Définissez une classe `Satellite()` qui permette d'instancier des objets simulant des satellites artificiels lancés dans l'espace, autour de la terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées : `masse = 100`, `vitesse = 0`. Lorsque l'oninstanciera un nouvel objet `Satellite()`, on pourra choisir son nom, sa masse et sa vitesse. Les méthodes suivantes seront définies : - `impulsion(force, duree)` permettra de faire varier la vitesse du satellite. Pour savoir comment, rappelez-vous votre cours de physique : la variation de vitesse Δv subie par un objet de masse m soumis à l'action d'une force F pendant un temps t vaut $\Delta v = \frac{F \times \Delta t}{m}$. Par exemple : un satellite de 300 kg qui subit une force de 600 Newtons pendant 10 secondes voit sa vitesse augmenter (ou diminuer) de 20 m/s. - `affiche_vitesse()` affichera le nom du satellite et sa vitesse courante. - `energie()` renverra au programme appelant la valeur de l'énergie cinétique du satellite. Rappel : l'énergie cinétique E_c se calcule à l'aide de la formule $E_c = \frac{1}{2}mv^2$ Exemples d'utilisation de cette classe :

```

>>> s1 = Satellite('Zoé', masse =250, vitesse =10)
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 40 m/s.
>>> print s1.energie()
200000
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 70 m/s.
>>> print s1.energie()
612500
    
```

Solution (cliquez)


```

1. class Domino:
    def __init__(self, pa, pb):
        self.pa, self.pb = pa, pb

    def affiche_points(self):
        print "face A :", self.pa,
        print "face B :", self.pb

    def valeur(self):
        return self.pa + self.pb

# Programme de test :

d1 = Domino(2,6)
d2 = Domino(4,3)

d1.affiche_points()
d2.affiche_points()

print "total des points :", d1.valeur() + d2.valeur()

liste_dominos = []
for i in range(7):
    liste_dominos.append(Domino(6, i))

vt = 0
for i in range(7):
    liste_dominos[i].affiche_points()
    vt = vt + liste_dominos[i].valeur()

print "valeur totale des points", vt

```

2. Réfléchissez !

```

3. class Voiture:
    def __init__(self, marque = 'Ford', couleur = 'rouge'):
        self.couleur = couleur
        self.marque = marque
        self.pilote = 'personne'
        self.vitesse = 0

    def accelerer(self, taux, duree):
        if self.pilote == 'personne':
            print "Cette voiture n'a pas de conducteur !"
        else:
            self.vitesse = self.vitesse + taux * duree

    def choix_conducteur(self, nom):
        self.pilote = nom

    def affiche_tout(self):
        print "%s %s pilotée par %s, vitesse = %s m/s" % \
            (self.marque, self.couleur, self.pilote, self.vitesse)

a1 = Voiture('Peugeot', 'bleue')
a2 = Voiture(couleur = 'verte')
a3 = Voiture('Mercedes')
a1.choix_conducteur('Roméo')
a2.choix_conducteur('Juliette')
a2.accelerer(1.8, 12)
a3.accelerer(1.9, 11)

```

```

a2.affiche_tout()
a3.affiche_tout()

```

```

4. class Satellite:
    def __init__(self, nom, masse =100, vitesse =0):
        self.nom, self.masse, self.vitesse = nom, masse, vitesse

    def impulsions(self, force, duree):
        self.vitesse = self.vitesse + force * duree / self.masse

    def energie(self):
        return self.masse * self.vitesse**2 / 2

    def affiche_vitesse(self):
        print "Vitesse du satellite %s = %s m/s" \
              % (self.nom, self.vitesse)

# Programme de test :

s1 = Satellite('Zoé', masse =250, vitesse =10)

s1.impulsions(500, 15)
s1.affiche_vitesse()
print s1.energie()
s1.impulsions(500, 15)
s1.affiche_vitesse()
print s1.energie()

```

Exercices

1. Définissez une classe `Cercle()`. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre `rayon`), vous définirez une méthode `surface()`, qui devra renvoyer la surface du cercle. Définissez ensuite une classe `Cylindre()` dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres `rayon` et `hauteur`. Vous y ajouterez une méthode `volume()` qui devra renvoyer le volume du cylindre. (*Rappel : Volume d'un cylindre = surface de section x hauteur*). Exemple d'utilisation de cette classe :

```

>>> cyl = Cylindre(5, 7)
>>> print cyl.surface()
78.54
>>> print cyl.volume()
549.78

```

2. Complétez l'exercice précédent en lui ajoutant encore une classe `Cone()`, qui devra dériver cette fois de la classe `Cylindre()`, et dont le constructeur comportera lui aussi les deux paramètres `rayon` et `hauteur`. Cette nouvelle classe possédera sa propre méthode `volume()`, laquelle devra renvoyer le volume du cône. (*Rappel : Volume d'un cône = volume du cylindre correspondant divisé par 3*). Exemple d'utilisation de cette classe :

```

>>> co = Cone(5,7)

```

```
>>> print co.volume()
```

183.26

3. Définissez une classe `JeuDeCartes()` permettant d'instancier des objets « jeu de cartes » dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les trois méthodes suivantes :
 - méthode constructeur : création et remplissage d'une liste de 52 éléments, qui sont eux-mêmes des tuples de 2 éléments contenant les caractéristiques de chacune des 52 cartes. Pour chacune d'elles, il faut en effet mémoriser séparément un nombre entier indiquant la valeur (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, les 4 dernières valeurs étant celles des valet, dame, roi et as), et un autre nombre entier indiquant la couleur de la carte (c'est-à-dire 0,1,2,3 pour Cœur, Carreau, Trèfle & Pique). Dans une telle liste, l'élément (11,2) désigne donc le valet de Trèfle, et la liste terminée doit être du type : [(2, 0), (3,0), (3,0), (4,0), (12,3), (13,3), (14,3)]
 - méthode `nom_carte()` : cette méthode renvoie sous la forme d'une chaîne l'identité d'une carte quelconque, dont on lui a fourni le tuple descripteur en argument. Par exemple, l'instruction : `print jeu.nom_carte((14, 3))` doit provoquer l'affichage de : As de pique
 - méthode `battre()` : comme chacun sait, battre les cartes consiste à les mélanger. Cette méthode sert donc à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre.
 - méthode `tirer()` : lorsque cette méthode est invoquée, une carte est retirée du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. On retire toujours la première carte de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer l'objet spécial `None` au programme appelant.
 Exemple d'utilisation de la classe `JeuDeCartes()` :

```
j = JeuDeCartes()           # instantiation d'un objet
j.battre()                  # mélange des cartes
for n in range(53):        # tirage des 52 cartes :
    c = j.tirer()
    if c == None:           # il ne reste plus aucune carte
        print 'Terminé !'  # dans la liste
    else:
        print j.nom_carte(c) # valeur et couleur de la carte
```

4. Complément de l'exercice précédent : Définir deux joueurs A et B. Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger. Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des 2 qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.

Solution (cliquez)

1. Voir ci-dessous.

```

2.  #(classes de cylindres et de cônes) :
    # Classes dérivées - polymorphisme

   class Cercle:
       def __init__(self, rayon):
           self.rayon = rayon

       def surface(self):
           return 3.1416 * self.rayon**2

   class Cylindre(Cercle):
       def __init__(self, rayon, hauteur):
           Cercle.__init__(self, rayon)
           self.hauteur = hauteur

       def volume(self):
           return self.surface()*self.hauteur

            # la méthode surface() est héritée de la classe parente

   class Cone(Cylindre):
       def __init__(self, rayon, hauteur):
           Cylindre.__init__(self, rayon, hauteur)

       def volume(self):
           return Cylindre.volume(self)/3

            # cette nouvelle méthode volume() remplace celle que
            # l'on a héritée de la classe parente (exemple de polymorphisme)

   cyl = Cylindre(5, 7)
   print cyl.surface()
   print cyl.volume()

   co = Cone(5,7)
   print co.surface()
   print co.volume()

```

```

3.  # Tirage de cartes

   from random import randrange

   class JeuDeCartes:
        """Jeu de cartes"""
        # attributs de classe (communs à toutes les instances) :
       couleur = ('Pique', 'Trèfle', 'Carreau', 'Coeur')
       valeur = (0, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as')

       def __init__(self):
            "Construction de la liste des 52 cartes"
           self.carte = []
           for coul in range(4):
               for val in range(13):
                   self.carte.append((val + 2, coul))  # la valeur commence à 2

       def nom_carte(self, c):
            "Renvoi du nom de la carte c, en clair"
           return "%s de %s" % (self.valeur[c[0]], self.couleur[c[1]])

       def battre(self):
            "Mélange des cartes"

```

```

t = len(self.carte)          # nombre de cartes restantes
# pour mélanger, on procède à un nombre d'échanges équivalent :
for i in range(t):
    # tirage au hasard de 2 emplacements dans la liste :
    h1, h2 = randrange(t), randrange(t)
    # échange des cartes situées à ces emplacements :
    self.carte[h1], self.carte[h2] = self.carte[h2], self.carte[h1]

def tirer(self):
    "Tirage de la première carte de la pile"
    t = len(self.carte)      # vérifier qu'il reste des cartes
    if t > 0:
        carte = self.carte[0] # choisir la première carte du jeu
        del(self.carte[0])    # la retirer du jeu
        return carte         # en renvoyer copie au prog. appelant
    else:
        return None         # facultatif

### Programme test :

if __name__ == '__main__':
    jeu = JeuDeCartes()      # instantiation d'un objet
    jeu.battre()            # mélange des cartes
    for n in range(53):     # tirage des 52 cartes :
        c = jeu.tirer()
        if c == None:      # il ne reste aucune carte
            print 'Terminé !' # dans la liste
        else:
            print jeu.nom_carte(c) # valeur et couleur de la carte

```

4. *4. (On supposera que l'exercice précédent a été sauvegardé sous le nom cartes.py)*
Bataille de de cartes

```

from cartes import JeuDeCartes

jeuA = JeuDeCartes()      # instantiation du premier jeu
jeuB = JeuDeCartes()      # instantiation du second jeu
jeuA.battre()            # mélange de chacun
jeuB.battre()
pA, pB = 0, 0             # compteurs de points des joueurs A et B

# tirer 52 fois une carte de chaque jeu :
for n in range(52):
    cA, cB = jeuA.tirer(), jeuB.tirer()
    vA, vB = cA[0], cB[0] # valeurs de ces cartes
    if vA > vB:
        pA += 1
    elif vB > vA:
        pB += 1           # (rien ne se passe si vA = vB)
    # affichage des points successifs et des cartes tirées :
    print "%s * %s ==> %s * %s" % (jeuA.nom_carte(cA), jeuB.nom_carte(cB), pA, pB)

print "le joueur A obtient %s points, le joueur B en obtient %s." % (pA, pB)

```

Exercices

Créer vous-même un nouveau module de classes, en encodant les lignes

d'instruction ci-dessous dans un fichier que vous nommerez formes.py :

```
class Rectangle:
    "Classe de rectangles"
    def __init__(self, longueur = 30, largeur = 15):
        self.L = longueur
        self.l = largeur
        self.nom = "rectangle"

    def perimetre(self):
        return "(%s + %s) * 2 = %s" % (self.L, self.l,
                                       (self.L + self.l)*2)

    def surface(self):
        return "%s * %s = %s" % (self.L, self.l, self.L*self.l)

    def mesures(self):
        print "Un %s de %s sur %s" % (self.nom, self.L, self.l)
        print "a une surface de %s" % (self.surface(),)
        print "et un périmètre de %s\n" % (self.perimetre(),)

class Carre(Rectangle):
    "Classe de carrés"
    def __init__(self, cote =10):
        Rectangle.__init__(self, cote, cote)
        self.nom = "carré"

if __name__ == "__main__":
    r1 = Rectangle(15, 30)
    r1.mesures()
    c1 = Carre(13)
    c1.mesures()
```

Une fois ce module enregistré, vous pouvez l'utiliser de deux manières : soit vous en lancez l'exécution comme celle d'un programme ordinaire, soit vous l'importez dans un script quelconque ou depuis la ligne de commande, pour en utiliser les classes :

```
>>> import formes
>>> f1 = formes.Rectangle(27, 12)
>>> f1.mesures()
Un rectangle de 27 sur 12
a une surface de 27 * 12 = 324
et un périmètre de (27 + 12) * 2 = 78

>>> f2 = formes.Carre(13)
>>> f2.mesures()
Un carré de 13 sur 13
a une surface de 13 * 13 = 169
et un périmètre de (13 + 13) * 2 = 52
```

On voit dans ce script que la classe Carre() est construite par dérivation à partir de la classe Rectangle() dont elle hérite toutes les caractéristiques. En d'autres termes, la classe Carre() est une classe fille de la classe Rectangle().

Vous pouvez remarquer encore une fois que le constructeur de la classe Carre() fait appel au constructeur de sa classe parente (Rectangle.__init__()), en lui transmettant la référence de l'instance (c'est-à-dire self) comme

premier argument.

Quant à l'instruction :

```
if __name__ == "__main__":
```

placée à la fin du module, elle sert à déterminer si le module est « lancé » en tant que programme (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une bibliothèque de classes importée ailleurs. Dans ce cas cette partie du code est sans effet.

Solution (cliquez)

Références

1. <https://docs.python.org/2.7/reference/datamodel.html#special-method-names>
2. <http://farmdev.com/src/secrets/magicmethod/>

Classes et Interfaces graphiques

La programmation orientée objet convient particulièrement bien au développement d'applications avec interface graphique. Des bibliothèques de classes comme *Tkinter* ou *wxPython* fournissent une base de *widgets* très étoffée, que nous pouvons adapter à nos besoins par dérivation. Dans ce chapitre, nous allons utiliser à nouveau la bibliothèque *Tkinter*, mais en appliquant les concepts décrits dans les pages précédentes, et en nous efforçant de mettre en évidence les avantages qu'apporte l'*orientation objet* dans nos programmes.

« Code des couleurs » : un petit projet bien encapsulé

Nous allons commencer par un petit projet qui nous a été inspiré par le cours d'initiation à l'électronique. L'application que nous décrivons ci-après permet de retrouver rapidement le code de trois couleurs qui correspond à une résistance électrique de valeur bien déterminée.

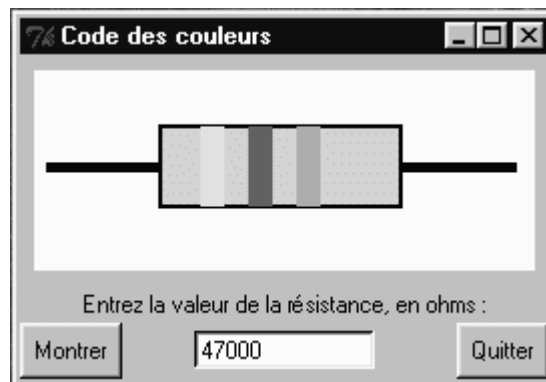
Pour rappel, la fonction des résistances électriques consiste à s'opposer (à résister) plus ou moins bien au passage du courant. Les résistances se présentent concrètement sous la forme de petites pièces tubulaires cerclées de bandes de couleur (en général 3). Ces bandes de couleur indiquent la valeur numérique de la résistance, en fonction du code suivant :

Remarque : chaque couleur correspond conventionnellement à l'un des chiffres de zéro à neuf : Noir = 0 ; Brun = 1 ; Rouge = 2 ; Orange = 3 ; Jaune = 4 ; Vert = 5 ; Bleu = 6 ; Violet = 7 ; Gris = 8 ; Blanc = 9.

On oriente la résistance de manière telle que les bandes colorées soient placées à gauche. La valeur de la résistance - exprimée en ohms (Ω) - s'obtient en lisant ces bandes colorées également à partir de la gauche : les deux premières bandes indiquent les deux premiers chiffres de la valeur numérique ; il faut ensuite accoler à ces deux chiffres un nombre de zéros égal à l'indication fournie par la troisième bande. Par exemple supposons qu'à partir de la gauche, les bandes colorées soient jaune, violette et verte et que la valeur de cette résistance est 4700000Ω , ou $4700 \text{ k}\Omega$, ou encore $4,7 \text{ M}\Omega$.

Ce système ne permet évidemment de préciser une valeur numérique qu'avec deux chiffres significatifs seulement. Il est toutefois considéré comme largement suffisant pour la plupart des applications électroniques « ordinaires » (radio, TV, etc.)

Cahier des charges de notre programme



Notre application doit faire apparaître une fenêtre comportant un dessin de la résistance, ainsi qu'un champ d'entrée dans lequel l'utilisateur peut encoder une valeur numérique. Un bouton « Montrer » déclenche la modification du dessin de la résistance, de telle façon que les trois bandes de couleur se mettent en accord avec la valeur numérique introduite.

Contrainte : Le programme doit accepter toute entrée numérique fournie sous forme entière ou réelle, dans les limites de 10 à $10^{11} \Omega$. Par exemple, une valeur telle que $4.78e6$ doit être acceptée et arrondie correctement, c'est-à-dire convertie en 4800000Ω .

Mise en œuvre concrète

Nous construisons cette application simple sous la forme d'une *classe*. Sa seule utilité présente consiste à nous fournir *un espace de noms commun* dans lequel nous pouvons *encapsuler* nos variables et nos fonctions, ce qui nous permet de nous passer de variables globales. En effet :

- Les variables auxquelles nous souhaitons pouvoir accéder de partout sont déclarées comme des attributs d'instance (nous attachons chacune d'elles à l'instance à l'aide de `self`).
- Les fonctions sont déclarées comme des méthodes, et donc attachées elles aussi à `self`.

Au niveau principal du programme, nous nous contentons d'instancier un objet de la classe ainsi construite (aucune méthode de cet objet n'est activée de l'extérieur).

```

1 class Application:
2     def __init__(self):
3         """Constructeur de la fenêtre principale"""
4         self.root = Tk()
5         self.root.title('Code des couleurs')
6         self.dessineResistance()
7         Label(self.root,
8             text="Entrez la valeur de la résistance, en ohms :").grid(row =2)
9         Button(self.root, text='Montrer',
10             command =self.changeCouleurs).grid(row =3, sticky = W)
11         Button(self.root, text='Quitter',
12             command =self.root.quit).grid(row =3, sticky = E)

```

```

13     self.entree = Entry(self.root, width =14)
14     self.entree.grid(row =3)
15     # Code des couleurs pour les valeurs de zéro à neuf :
16     self.cc = ['black', 'brown', 'red', 'orange', 'yellow',
17              'green', 'blue', 'purple', 'grey', 'white']
18     self.root.mainloop()
19
20     def dessineResistance(self):
21         """Canevas avec un modèle de résistance à trois lignes colorées"""
22         self.can = Canvas(self.root, width=250, height =100, bg = 'ivory')
23         self.can.grid(row =1, pady =5, padx =5)
24         self.can.create_line(10, 50, 240, 50, width =5)           # fils
25         self.can.create_rectangle(65, 30, 185, 70, fill = 'light grey', width =2)
26         # Dessin des trois lignes colorées (noires au départ) :
27         self.ligne =[]           # on mémoriserà les trois lignes dans 1 liste
28         for x in range(85,150,24):
29             self.ligne.append(self.can.create_rectangle(x, 30, x+12, 70, fill='black',
30
31     def changeCouleurs(self):
32         """Affichage des couleurs correspondant à la valeur entrée"""
33         self.vlch = self.entree.get()           # la méthode get() renvoie une chaîne
34         try:
35             v = float(self.vlch)               # conversion en valeur numérique
36         except:
37             err =1                             # erreur : entrée non numérique
38         else:
39             err =0
40         if err ==1 or v < 10 or v > 1e11 :
41             self.signaleErreur()               # entrée incorrecte ou hors limites
42         else:
43             li =[0]*3                           # liste des 3 codes à afficher
44             logv = int(log10(v))                 # partie entière du logarithme
45             ordgr = 10**logv                     # ordre de grandeur
46             # extraction du premier chiffre significatif :
47             li[0] = int(v/ordgr)                 # partie entière
48             decim = v/ordgr - li[0]              # partie décimale
49             # extraction du second chiffre significatif :
50             li[1] = int(decim*10 +.5)           # +.5 pour arrondir correctement
51             # nombre de zéros à accoler aux 2 chiffres significatifs :
52             li[2] = logv -1
53             # Coloration des 3 lignes :
54             for n in range(3):
55                 self.can.itemconfigure(self.ligne[n], fill =self.cc[li[n]])
56
57     def signaleErreur(self):
58         self.entree.configure(bg = 'red')       # colorer le fond du champ
59         self.root.after(1000, self.videEntree) # après 1 seconde, effacer
60
61     def videEntree(self):
62         self.entree.configure(bg = 'white')     # rétablir le fond blanc
63         self.entree.delete(0, len(self.vlch))   # enlever les car. présents
64
65 # Programme principal :
66 from Tkinter import *
67 from math import log10 # logarithmes en base 10
68 f = Application()     # instantiation de l'objet application

```

Commentaires

- Ligne 1 : La classe est définie sans référence à une classe parente (pas de parenthèses). Il s'agira donc d'une nouvelle classe indépendante.

- Lignes 2 à 14 : Le constructeur de la classe instancie les *widgets* nécessaires : pour améliorer la lisibilité du programme, on a placé l'instanciation du canevas (avec le dessin de la résistance) dans une méthode séparée `dessineResistance()`. Les boutons et le libellé ne sont pas mémorisés dans des variables, parce que l'on ne souhaite pas y faire référence ailleurs dans le programme. Le positionnement des *widgets* dans la fenêtre utilise la méthode `grid()`.
- Lignes 15-17 : Le code des couleurs est mémorisé dans une simple liste.
- Ligne 18 : La dernière instruction du constructeur démarre l'application.
- Lignes 20 à 30 : Le dessin de la résistance se compose d'une ligne et d'un premier rectangle gris clair, pour le corps de la résistance et ses deux fils. Trois autres rectangles figureront les bandes colorées que le programme devra modifier en fonction des entrées de l'utilisateur. Ces bandes sont noires au départ ; elles sont référencées dans la liste `self.ligne`.
- Lignes 32 à 53 : Ces lignes contiennent l'essentiel de la fonctionnalité du programme.

L'entrée brute fournie par l'utilisateur est acceptée sous la forme d'une chaîne de caractères.

À la ligne 36, on essaie de convertir cette chaîne en une valeur numérique de type *float*. Si la conversion échoue, on mémorise l'erreur. Si l'on dispose bien d'une valeur numérique, on vérifie ensuite qu'elle se situe effectivement dans l'intervalle autorisé (de 10 Ω à 1011 Ω). Si une erreur est détectée, on signale à l'utilisateur que son entrée est incorrecte en colorant de rouge le fond du champ d'entrée, qui est ensuite vidé de son contenu (lignes 55 à 61).

- Lignes 45-46 : Les mathématiques viennent à notre secours pour extraire de la valeur numérique son ordre de grandeur (c'est-à-dire l'exposant de 10 le plus proche). Veuillez consulter votre cours de mathématiques pour de plus amples explications concernant les logarithmes.
- Lignes 47-48 : Une fois connu l'ordre de grandeur, il devient relativement facile d'extraire du nombre traité ses deux premiers chiffres significatifs. Exemple : Supposons que la valeur entrée soit 31687. Le logarithme de ce nombre est 4,50088... dont la partie entière (4) nous donne l'ordre de grandeur de la valeur entrée (soit 10^4). Pour extraire de celle-ci son premier chiffre significatif, il suffit de la diviser par 10^4 , soit 10000, et de conserver seulement la partie entière du résultat (3).
- Lignes 49 à 51 : Le résultat de la division effectuée dans le paragraphe précédent est 3,1687.

Nous récupérons la partie décimale de ce nombre à la ligne 49, soit 0,1687 dans notre exemple.

Si nous le multiplions par dix, ce nouveau résultat comporte une partie entière qui n'est rien d'autre que notre second chiffre significatif (1 dans notre exemple).

Nous pourrions facilement extraire ce dernier chiffre, mais puisque c'est le dernier, nous souhaitons encore qu'il soit correctement arrondi. Pour ce faire, il suffit d'ajouter une demi unité au produit de la multiplication par dix, avant d'en extraire la valeur entière. Dans notre exemple, en effet, ce calcul donnera donc $1,687 + 0,5 = 2,187$, dont la partie entière (2) est bien la valeur arrondie recherchée.

- Ligne 53 : Le nombre de zéros à accoler aux deux chiffres significatifs correspond au calcul de l'ordre de grandeur. Il suffit de retirer une unité au logarithme.
- Ligne 56 : Pour attribuer une nouvelle couleur à un objet déjà dessiné dans un canevas, on utilise la méthode `itemconfigure()`. Nous utilisons donc cette méthode pour modifier l'option `fill` de chacune des bandes colorées, en utilisant les noms de couleur extraits de la liste `self.cc` grâce à aux trois indices `li[1]`, `li[2]` et `li[3]` qui contiennent les 3 chiffres correspondants.

Exercices

1. Modifiez le script ci-dessus de telle manière que le fond d'image devienne bleu clair ('light blue'), que le corps de la résistance devienne beige ('beige'), que le fil de cette résistance soit plus fin, et que les bandes colorées indiquant la valeur soient plus larges.
2. Modifiez le script ci-dessus de telle manière que l'image dessinée soit deux fois plus grande.
3. Modifiez le script ci-dessus de telle manière qu'il devienne possible d'entrer aussi des valeurs de résistances comprises entre 1 et 10 Ω . Pour ces valeurs, le premier anneau coloré devra rester noir, les deux autres indiqueront la valeur en Ω et dixièmes d' Ω .
4. Modifiez le script ci-dessus de telle façon que le bouton « Montrer » ne soit plus nécessaire. Dans votre script modifié, il suffira de frapper <Enter> après avoir entré la valeur de la résistance, pour que l'affichage s'active.
5. Modifiez le script ci-dessus de telle manière que les trois bandes colorées redeviennent noires dans les cas où l'utilisateur fournit une entrée inacceptable.

Solution

1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !
5. Réfléchissez !

« Petit train » : héritage, échange d'informations entre classes

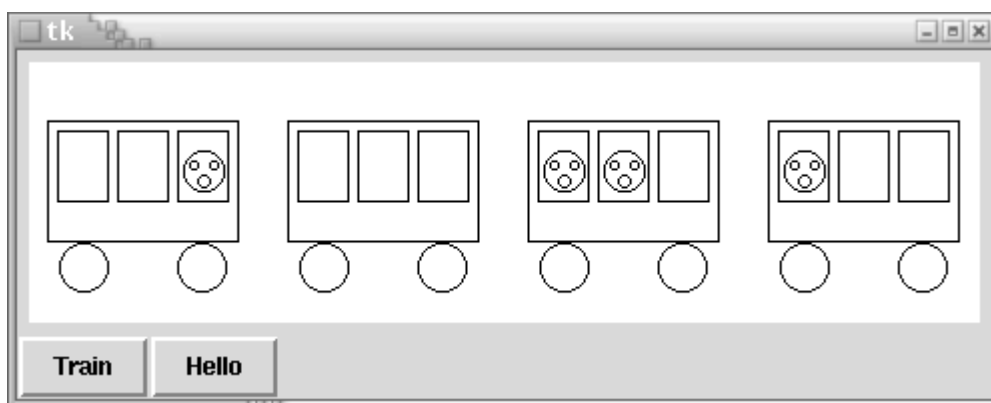
Dans l'exercice précédent, nous n'avons exploité qu'une seule caractéristique

des classes : *l'encapsulation*. Celle-ci nous a permis d'écrire un programme dans lequel les différentes fonctions (qui sont donc devenues des *méthodes*) peuvent chacune accéder à un même *pool* de variables : toutes celles qui sont définies comme étant attachées à `self`. Toutes ces variables peuvent être considérées en quelque sorte comme des variables globales à l'intérieur de l'objet.

Comprenez bien toutefois qu'il ne s'agit pas de véritables variables globales. Elles restent en effet strictement confinées à l'intérieur de l'objet, et il est déconseillé de vouloir y accéder de l'extérieur^[1]. D'autre part, tous les objets que vousinstancierez à partir d'une même classe posséderont chacun leur propre jeu de ces variables, qui sont donc bel et bien *encapsulées* dans ces objets. On les appelle pour cette raison des *attributs d'instance*.

Nous allons à présent passer à la vitesse supérieure et réaliser une petite application sur la base de plusieurs classes, afin d'examiner *comment différents objets peuvent s'échanger des informations par l'intermédiaire de leurs méthodes*. Nous allons également profiter de cet exercice pour vous montrer comment vous pouvez définir la classe principale de votre application graphique par *dérivation* d'une classe *Tkinter* préexistante, mettant ainsi à profit le mécanisme d'héritage.

Le projet développé ici très simple, mais il pourrait constituer une première étape dans la réalisation d'un logiciel de jeu. Il s'agit d'une fenêtre contenant un canevas et deux boutons. Lorsque l'on actionne le premier de ces deux boutons, un petit train apparaît dans le canevas. Lorsque l'on actionne le second bouton, quelques petits personnages apparaissent à certaines fenêtres des wagons.



Cahier des charges

L'application comportera deux classes :

- La classe `Application()` sera obtenue par dérivation d'une des classes de base de *Tkinter* : elle mettra en place la fenêtre principale, son canevas et ses deux boutons.
- Une classe `Wagon()`, indépendante, permettra d'instancier dans le canevas 4 objets-wagons similaires, dotés chacun d'une méthode `perso()`. Celle-ci sera

destinée à provoquer l'apparition d'un petit personnage à l'une quelconque des trois fenêtres du wagon. L'application principale invoquera cette méthode différemment pour différents objets-wagons, afin de faire apparaître un choix de quelques personnages.

Implémentation

```

1 from Tkinter import *
2
3 def cercle(can, x, y, r):
4     "dessin d'un cercle de rayon <r> en <x,y> dans le canevas <can>"
5     can.create_oval(x-r, y-r, x+r, y+r)
6
7 class Application(Tk):
8     def __init__(self):
9         Tk.__init__(self)          # constructeur de la classe parente
10        self.can =Canvas(self, width =475, height =130, bg ="white")
11        self.can.pack(side =TOP, padx =5, pady =5)
12        Button(self, text ="Train", command =self.dessine).pack(side =LEFT)
13        Button(self, text ="Hello", command =self.coucou).pack(side =LEFT)
14
15    def dessine(self):
16        "instanciation de 4 wagons dans le canevas"
17        self.w1 = Wagon(self.can, 10, 30)
18        self.w2 = Wagon(self.can, 130, 30)
19        self.w3 = Wagon(self.can, 250, 30)
20        self.w4 = Wagon(self.can, 370, 30)
21
22    def coucou(self):
23        "apparition de personnages dans certaines fenêtrés"
24        self.w1.perso(3)           # 1er wagon, 3e fenêtré
25        self.w3.perso(1)           # 3e wagon, 1e fenêtré
26        self.w3.perso(2)           # 3e wagon, 2e fenêtré
27        self.w4.perso(1)           # 4e wagon, 1e fenêtré
28
29 class Wagon:
30    def __init__(self, cane, x, y):
31        "dessin d'un petit wagon en <x,y> dans le canevas <canev>"
32        # mémorisation des paramètres dans des variables d'instance :
33        self.canev, self.x, self.y = cane, x, y
34        # rectangle de base : 95x60 pixels :
35        cane.create_rectangle(x, y, x+95, y+60)
36        # 3 fenêtrés de 25x40 pixels, écartées de 5 pixels :
37        for xf in range(x+5, x+90, 30):
38            cane.create_rectangle(xf, y+5, xf+25, y+40)
39        # 2 roues de rayon égal à 12 pixels :
40        cercle(cane, x+18, y+73, 12)
41        cercle(cane, x+77, y+73, 12)
42
43    def perso(self, fen):
44        "apparition d'un petit personnage à la fenêtré <fen>"
45        # calcul des coordonnées du centre de chaque fenêtré :
46        xf = self.x + fen*30 -12
47        yf = self.y + 25
48        cercle(self.canev, xf, yf, 10)    # visage
49        cercle(self.canev, xf-5, yf-3, 2) # oeil gauche
50        cercle(self.canev, xf+5, yf-3, 2) # oeil droit
51        cercle(self.canev, xf, yf+5, 3)  # bouche
52
53 app = Application()
54 app.mainloop()

```

Commentaires

- Lignes 3 à 5 : Nous projetons de dessiner une série de petits cercles. Cette petite fonction nous facilitera le travail en nous permettant de définir ces cercles à partir de leur centre et leur rayon.
- Lignes 7 à 13 : La classe principale de notre application est construite par dérivation de la classe de fenêtres `Tk()` importée du module `Tkinter`^[2]. Comme nous l'avons expliqué au chapitre précédent, le constructeur d'une classe dérivée doit activer lui-même le constructeur de la classe parente, en lui transmettant la référence de l'instance comme premier argument. Les lignes 10 à 13 servent à mettre en place le canevas et les boutons.
- Lignes 15 à 20 : Ces lignesinstancient les 4 objets-wagons, produits à partir de la classe correspondante. Ceci pourrait être programmé plus élégamment à l'aide d'une boucle et d'une liste, mais nous le laissons ainsi afin de ne pas alourdir inutilement les explications qui suivent. Nous voulons placer nos objets-wagons dans le canevas, à des emplacements bien précis : il nous faut donc transmettre quelques informations au constructeur de ces objets : au moins la référence du canevas, ainsi que les coordonnées souhaitées. Ces considérations nous font également entrevoir, que lorsque nous définirons la classe `Wagon()`, nous devons associer à sa méthode constructeur un nombre égal de paramètres pour réceptionner ces arguments.
- Lignes 22 à 27 : Cette méthode est invoquée lorsque l'on actionne le second bouton. Elle invoque elle-même la méthode `perso()` de certains objets-wagons, avec des arguments différents, afin de faire apparaître les personnages aux fenêtres indiquées.

Remarque : ces quelques lignes de code vous montrent donc comment un objet peut communiquer avec un autre en faisant appel à l'une ou l'autre de ses méthodes. Il s'agit là du mécanisme central de la programmation par objets : les objets sont des entités programmées qui s'échangent des messages et interagissent par l'intermédiaire de leurs méthodes.

Idéalement, la méthode `coucou()` devrait comporter quelques instructions complémentaires, lesquelles vérifieraient d'abord si les objets-wagons concernés existent bel et bien, avant d'autoriser l'activation d'une de leurs méthodes. Nous n'avons pas inclus ce genre de garde-fou afin que l'exemple reste aussi simple que possible, mais cela entraîne la conséquence que vous ne pouvez pas actionner le second bouton avant le premier. (Pouvez-vous ajouter un correctif ?)

- Lignes 29-30 : La classe `Wagon()` ne dérive d'aucune autre classe préexistante. Cependant, étant donné qu'il s'agit d'une classe d'objets graphiques, nous devons munir sa méthode constructeur de paramètres, afin de recevoir la référence du canevas auquel les dessins sont destinés, ainsi que les coordonnées de départ de ces dessins. Dans vos expérimentations éventuelles autour de cet exercice, vous pourriez bien évidemment ajouter encore d'autres paramètres : taille du dessin, orientation, couleur, vitesse, etc.

- Lignes 31 à 51 : Ces instructions ne nécessitent guère de commentaires. La méthode `perso()` est dotée d'un paramètre qui indique celle des 3 fenêtres où il faut faire apparaître un petit personnage. Ici aussi nous n'avons pas prévu de garde-fou : vous pouvez invoquer cette méthode avec un argument égal à 4 ou 5, par exemple, ce qui produira des effets incorrects.
- Lignes 53-54 : Pour démarrer l'application, il ne suffit pas d'instancier un objet de la classe `Application()` comme dans l'exemple de la rubrique précédente. Il faut également invoquer la méthode `mainloop()` qu'elle a hérité de sa classe parente. Vous pourriez cependant condenser ces deux instructions en une seule, laquelle serait alors : `Application().mainloop()`

Exercices

1. Perfectionnez le script décrit ci-dessus, en ajoutant un paramètre couleur au constructeur de la classe `Wagon()`, lequel déterminera la couleur de la cabine du wagon. Arrangez-vous également pour que les fenêtres soient noires au départ, et les roues grises (pour réaliser ce dernier objectif, ajoutez aussi un paramètre couleur à la fonction `cercle()`). À cette même classe `Wagon()`, ajoutez encore une méthode `allumer()`, qui servira à changer la couleur des 3 fenêtres (initialement noires) en jaune, afin de simuler l'allumage d'un éclairage intérieur. Ajoutez un bouton à la fenêtre principale, qui puisse déclencher cet allumage. Profitez de l'amélioration de la fonction `cercle()` pour teinter le visage des petits personnages en rose (`pink`), leurs yeux et leurs bouches en noir, et instanciez les objets-wagons avec des couleurs différentes.

Solution

```

1. 1 from Tkinter import *
    2
    3 def cercle(can, x, y, r, coul = 'white'):
    4     "dessin d'un cercle de rayon <r> en <x,y> dans le canevas <can>"
    5     can.create_oval(x-r, y-r, x+r, y+r, fill =coul)
    6
    7 class Application(Tk):
    8     def __init__(self):
    9         Tk.__init__(self)          # constructeur de la classe parente
   10         self.can =Canvas(self, width =475, height =130, bg ="white")
   11         self.can.pack(side =TOP, padx =5, pady =5)
   12         Button(self, text ="Train", command =self.dessine).pack(side =LEFT)
   13         Button(self, text ="Hello", command =self.coucou).pack(side =LEFT)
   14         Button(self, text ="Ecl34", command =self.eclai34).pack(side =LEFT)
   15
   16     def dessine(self):
   17         "instanciation de 4 wagons dans le canevas"
   18         self.w1 = Wagon(self.can, 10, 30)
   19         self.w2 = Wagon(self.can, 130, 30, 'dark green')
   20         self.w3 = Wagon(self.can, 250, 30, 'maroon')
   21         self.w4 = Wagon(self.can, 370, 30, 'purple')
   22
   23     def coucou(self):
   24         "apparition de personnages dans certaines fenêtres"
   25         self.w1.perso(3)          # 1er wagon, 3e fenêtre

```



```

26         self.w3.perso(1)           # 3e wagon, 1e fenêtre
27         self.w3.perso(2)           # 3e wagon, 2e fenêtre
28         self.w4.perso(1)           # 4e wagon, 1e fenêtre
29
30     def eclai34(self):
31         "allumage de l'éclairage dans les wagons 3 & 4"
32         self.w3.allumer()
33         self.w4.allumer()
34
35     class Wagon:
36     def __init__(self, canev, x, y, coul = 'navy'):
37         "dessin d'un petit wagon en <x,y> dans le canevas <canev>"
38         # mémorisation des paramètres dans des variables d'instance :
39         self.canev, self.x, self.y = canev, x, y
40         # rectangle de base : 95x60 pixels :
41         canev.create_rectangle(x, y, x+95, y+60, fill =coul)
42         # 3 fenêtres de 25x40 pixels, écartées de 5 pixels :
43         self.fen =[] # pour mémoriser les réf. des fenêtres
44         for xf in range(x +5, x +90, 30):
45             self.fen.append(canev.create_rectangle(xf, y+5,
46                                                     xf+25, y+40, fill = 'black'))
47         # 2 roues de rayon égal à 12 pixels :
48         cercle(canev, x+18, y+73, 12, 'gray')
49         cercle(canev, x+77, y+73, 12, 'gray')
50
51     def perso(self, fen):
52         "apparition d'un petit personnage à la fenêtre <fen>"
53         # calcul des coordonnées du centre de chaque fenêtre :
54         xf = self.x + fen*30 -12
55         yf = self.y + 25
56         cercle(self.canev, xf, yf, 10, "pink") # visage
57         cercle(self.canev, xf-5, yf-3, 2) # oeil gauche
58         cercle(self.canev, xf+5, yf-3, 2) # oeil droit
59         cercle(self.canev, xf, yf+5, 3) # bouche
60
61     def allumer(self):
62         "déclencher l'éclairage interne du wagon"
63         for f in self.fen:
64             self.canev.itemconfigure(f, fill = 'yellow')
65
66 Application().app.mainloop()

```

« OscilloGraphe » : un widget personnalisé

Le projet qui suit va nous entraîner encore un petit peu plus loin. Nous allons y construire une nouvelle classe de widget, qu'il sera possible d'intégrer dans nos projets futurs comme n'importe quel widget standard. Comme la classe principale de l'exercice précédent, cette nouvelle classe sera construite par dérivation d'une classe *Tkinter* préexistante.

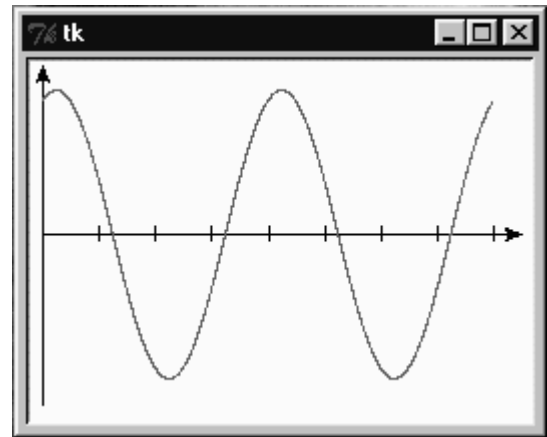
Le sujet concret de cette application nous est inspiré par le cours de physique. Pour rappel :

Un mouvement vibratoire harmonique se définit comme étant la projection d'un mouvement circulaire uniforme sur une droite. Les positions successives d'un mobile qui effectue ce type de mouvement sont traditionnellement repérées par rapport à une position centrale : on les appelle alors des *élongations*. L'équation qui décrit l'évolution de l'élongation d'un tel mobile au

cours du temps est toujours de la forme $e = A \sin(2\pi ft + \phi)$, dans laquelle e représente l'élongation du mobile à tout instant t . Les constantes A , f et ϕ désignent respectivement l'amplitude, la fréquence et la phase du mouvement vibratoire.

Le but du présent projet est de fournir un instrument de visualisation simple de ces différents concepts, à savoir un système d'affichage automatique de graphiques élongation/temps. L'utilisateur pourra choisir librement les valeurs des paramètres A , f et ϕ , et observer les courbes qui en résultent.

Le *widget* que nous allons construire d'abord s'occupera de l'affichage proprement dit. Nous construirons ensuite d'autres *widgets* pour faciliter l'entrée des paramètres A , f et ϕ .



Veillez donc encoder le script ci-dessous et le sauvegarder dans un fichier, auquel vous donnerez le nom `oscillo.py`. Vous réaliserez ainsi un véritable *module* contenant une classe (vous pourrez par la suite ajouter d'autres classes dans ce même module, si le cœur vous en dit).

```

1 from Tkinter import *
2 from math import sin, pi
3
4 class OscilloGraphe(Canvas):
5     "Canevas spécialisé, pour dessiner des courbes élongation/temps"
6     def __init__(self, boss=None, larg=200, haut=150):
7         "Constructeur du graphique : axes et échelle horiz."
8         # construction du widget parent :
9         Canvas.__init__(self) # appel au constructeur
10        self.configure(width=larg, height=haut) # de la classe parente
11        self.larg, self.haut = larg, haut # mémorisation
12        # tracé des axes de référence :
13        self.create_line(10, haut/2, larg, haut/2, arrow=LAST) # axe X
14        self.create_line(10, haut-5, 10, 5, arrow=LAST) # axe Y
15        # tracé d'une échelle avec 8 graduations :
16        pas = (larg-25)/8. # intervalles de l'échelle horizontale
17        for t in range(1, 9):
18            stx = 10 + t*pas # +10 pour partir de l'origine
19            self.create_line(stx, haut/2-4, stx, haut/2+4)
20
21        def traceCourbe(self, freq=1, phase=0, ampl=10, coul='red'):
22            "tracé d'un graphique élongation/temps sur 1 seconde"
23            curve = [] # liste des coordonnées
24            pas = (self.larg-25)/1000. # l'échelle X correspond à 1 seconde
25            for t in range(0,1001,5): # que l'on divise en 1000 ms.
26                e = ampl*sin(2*pi*freq*t/1000 - phase)
27                x = 10 + t*pas
28                y = self.haut/2 - e*self.haut/25
29                curve.append((x,y))
30            n = self.create_line(curve, fill=coul, smooth=1)

```

```

31         return n                                # n = numéro d'ordre du tracé
32
33 ##### Code pour tester la classe : #####
34
35 if __name__ == '__main__':
36     root = Tk()
37     gra = OscilloGraphe(root, 250, 180)
38     gra.pack()
39     gra.configure(bg='ivory', bd=2, relief=SUNKEN)
40     gra.traceCourbe(2, 1.2, 10, 'purple')
41     root.mainloop()

```

Le niveau principal du script est constitué par les lignes 35 à 41. Les lignes de code situées après l'instruction `if __name__ == '__main__':` ne sont pas exécutées si le script est importé en tant que module. Si on lance le script comme application principale, par contre, ces instructions sont exécutées.

Nous disposons ainsi d'un mécanisme intéressant, qui nous permet d'intégrer des instructions de test à l'intérieur des modules, même si ceux-ci sont destinés à être importés dans d'autres scripts.

Lancez donc l'exécution du script de la manière habituelle. Vous devriez obtenir un affichage similaire à celui qui est reproduit à la page précédente.

Expérimentation

Commençons d'abord par expérimenter quelque peu la classe que nous venons de construire. Ouvrez une fenêtre de terminal (*Python shell*), et entrez les instructions ci-dessous directement à la ligne de commande :

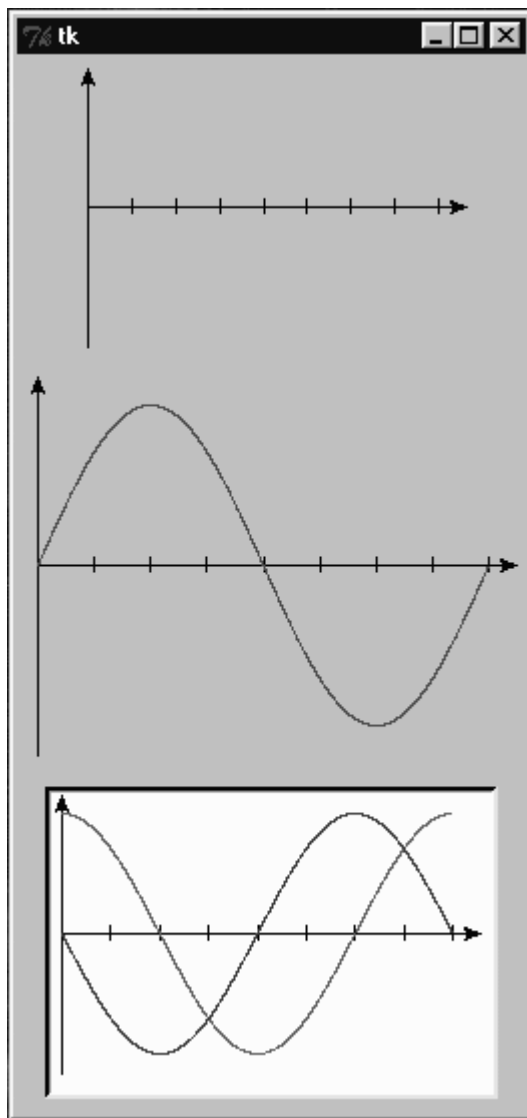
```

>>> from oscillo import *
>>> g1 = OscilloGraphe()
>>> g1.pack()

```

Après importation des classes du module `oscillo`, nousinstancions un premier objet `g1`, de la classe `OscilloGraphe()`.

Puisque nous ne fournissons aucun argument, l'objet possède les dimensions par défaut, définies dans le constructeur de la classe. Remarquons au passage que nous n'avons même pas pris la peine de définir d'abord une fenêtre maître pour y placer ensuite notre *widget*. *Tkinter* nous pardonne cet oubli et nous en fournit une automatiquement !



```

>>> g2 = OscilloGraphe(haut=200, larg=250)
>>> g2.pack()
>>> g2.traceCourbe()
    
```

Par ces instructions, nous créons un second *widget* de la même classe, en précisant cette fois ses dimensions (hauteur et largeur, dans n'importe quel ordre).

Ensuite, nous activons la méthode `traceCourbe()` associée à ce *widget*. Étant donné que nous ne lui fournissons aucun argument, la sinusoïde qui apparaît correspond aux valeurs prévues par défaut pour les paramètres A , f et φ .

```

>>> g3 = OscilloGraphe(larg=220)
>>> g3.configure(bg='white', bd=3, relief=SUNKEN)
>>> g3.pack(padx=5, pady=5)
>>> g3.traceCourbe(phase=1.57, coul='purple')
>>> g3.traceCourbe(phase=3.14, coul='dark green')
    
```

Pour comprendre la configuration de ce troisième *widget*, il faut nous rappeler

que la classe `OscilloGraphe()` a été construite par dérivation de la classe `Canvas()`. Elle hérite donc de toutes les propriétés de celle-ci, ce qui nous permet de choisir la couleur de fond, la bordure, etc., en utilisant les mêmes arguments que ceux qui sont à notre disposition lorsque nous configurons un canevas.

Nous faisons ensuite apparaître deux tracés successifs, en faisant appel deux fois à la méthode `traceCourbe()`, à laquelle nous fournissons des arguments pour la phase et la couleur.

Exercices

1. Créez un quatrième widget, de taille 400 x 300, couleur de fond jaune, et faites-y apparaître plusieurs courbes correspondant à des fréquences et des amplitudes différentes.

Solution

1. Réfléchissez !

Il est temps à présent que nous analysons la structure de la classe qui nous a permis d'instancier tous ces *widgets*. Nous avons enregistré cette classe dans le module `oscillo.py`.

Cahier des charges

Nous souhaitons définir une nouvelle classe de *widget*, capable d'afficher automatiquement les graphiques élongation/temps correspondant à divers mouvements vibratoires harmoniques.

Ce *widget* doit pouvoir être dimensionné à volonté au moment de son instanciation. Il fait apparaître deux axes cartésiens X et Y munis de flèches. L'axe X représente l'écoulement du temps pendant une seconde au total, et il est muni d'une échelle comportant 8 intervalles.

Une méthode `traceCourbe()` est associée à ce *widget*. Elle provoque le tracé du graphique élongation/temps pour un mouvement vibratoire dont on fournit la fréquence (entre 0.25 et 10 Hz), la phase (entre 0 et 2π radians) et l'amplitude (entre 1 et 10 ; échelle arbitraire).

Implémentation

- Ligne 4 : La classe `OscilloGraphe()` est créée par dérivation de la classe `Canvas()`. Elle hérite donc toutes les propriétés de celle-ci : on pourra configurer les objets de cette nouvelle classe en utilisant les nombreuses options déjà disponibles pour la classe `Canvas()`.
- Ligne 6 : La méthode « constructeur » utilise 3 paramètres, qui sont tous optionnels puisque chacun d'entre eux possède une valeur par défaut. Le paramètre `boss` ne sert qu'à réceptionner la référence d'une fenêtre maîtresse éventuelle (voir exemples suivants). Les paramètres `larg` et `haut`

(largeur et hauteur) servent à assigner des valeurs aux options `width` et `height` du canevas parent, au moment de l'instanciation.

- Lignes 9 et 10 : La première opération que doit accomplir le constructeur d'une classe dérivée, c'est activer le constructeur de sa classe parente. En effet : nous ne pouvons hériter toute la fonctionnalité de la classe parente, que si cette fonctionnalité a été effectivement mise en place. Nous activons donc le constructeur de la classe `Canvas()` à la ligne 9 , et nous ajustons deux de ses options à la ligne 10. Notez au passage que nous pourrions condenser ces deux lignes en une seule, qui deviendrait en l'occurrence :

```
Canvas.__init__(self, width=larg, height=haut)
```

Nous devons transmettre à ce constructeur la référence de l'instance présente (`self`) comme premier argument.

- Ligne 11 : Il est nécessaire de mémoriser les paramètres `larg` et `haut` dans des variables d'instance, parce que nous devons pouvoir y accéder aussi dans la méthode `traceCourbe()`.
- Lignes 13 et 14 : Pour tracer les axes X et Y, nous utilisons les paramètres `larg` et `haut`, ainsi ces axes sont automatiquement mis à dimension. L'option `arrow=LAST` permet de faire apparaître une petite flèche à l'extrémité de chaque ligne.
- Lignes 16 à 19 : Pour tracer l'échelle horizontale, on commence par réduire de 25 pixels la largeur disponible, de manière à ménager des espaces aux deux extrémités. On divise ensuite en 8 intervalles, que l'on visualise sous la forme de 8 petits traits verticaux.
- Ligne 21 : La méthode `traceCourbe()` pourra être invoquée avec quatre arguments. Chacun d'entre eux pourra éventuellement être omis, puisque chacun des paramètres correspondants possède une valeur par défaut. Il sera également possible de fournir les arguments dans n'importe quel ordre.
- Lignes 23 à 31 : Pour le tracé de la courbe, la variable `t` prend successivement toutes les valeurs de 0 à 1000, et on calcule à chaque fois l'élongation `e` correspondante, à l'aide de la formule théorique (ligne 26). Les couples de valeurs `t` et `e` ainsi trouvées sont mises à l'échelle et transformées en coordonnées `x, y` aux lignes 27 & 28, puis accumulées dans la liste `curve`.
- Lignes 30 et 31 : La méthode `create_line()` trace alors la courbe correspondante en une seule opération, et elle renvoie le numéro d'ordre du nouvel objet ainsi instancié dans le canevas (ce numéro d'ordre nous permettra d'y accéder encore par après : pour l'effacer, par exemple). L'option `smooth =1` améliore l'aspect final, par lissage.

Exercices

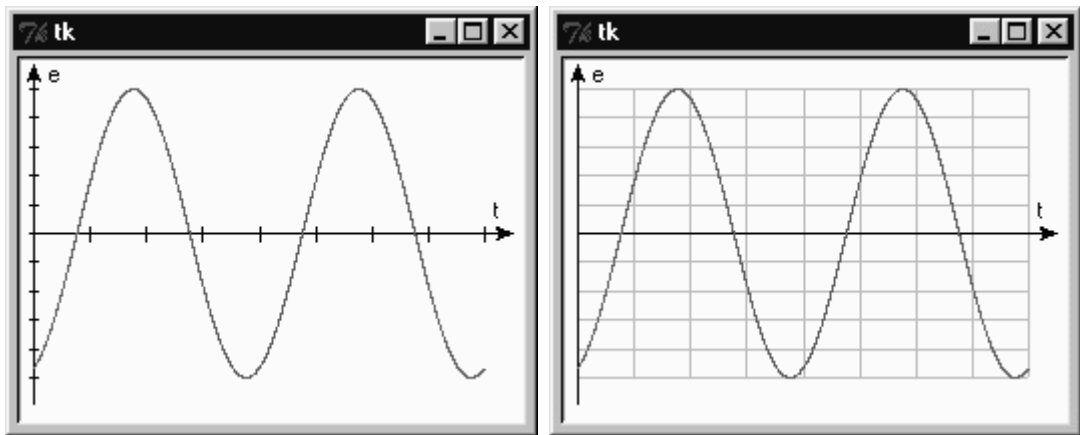
1. Modifiez le script de manière à ce que l'axe de référence vertical

comporte lui aussi une échelle, avec 5 tirets de part et d'autre de l'origine.

2. Comme les *widgets* de la classe `Canvas()` dont il dérive, votre *widget* peut intégrer des indications textuelles. Il suffit pour cela d'utiliser la méthode `create_text()`. Cette méthode attend au moins trois arguments : les coordonnées `x` et `y` de l'emplacement où vous voulez faire apparaître votre texte, et puis le texte lui-même, bien entendu. D'autres arguments peuvent être transmis sous forme d'options, pour préciser par exemple la police de caractères et sa taille. Afin de voir comment cela fonctionne, ajoutez provisoirement la ligne suivante dans le constructeur de la classe `OscilloGraphe()`, puis relancez le script :

```
self.create_text(130, 30, text = "Essai", anchor =CENTER)
```

Utilisez cette méthode pour ajouter au widget les indications suivantes aux extrémités des axes de référence : `e` (pour « élancement ») le long de l'axe vertical, et `t` (pour « temps ») le long de l'axe horizontal. Le résultat pourrait ressembler à ceci (figure de gauche) :



3. Vous pouvez compléter encore votre *widget*, en y faisant apparaître une grille de référence, plutôt que de simples tirets le long des axes. Pour éviter que cette grille ne soit trop visible, vous pouvez colorer ses traits en gris (option `fill = 'grey'`), comme dans la figure de droite.
4. Complétez encore votre widget en y faisant apparaître des repères numériques.

Solution

1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !

« Curseurs » : un widget composite

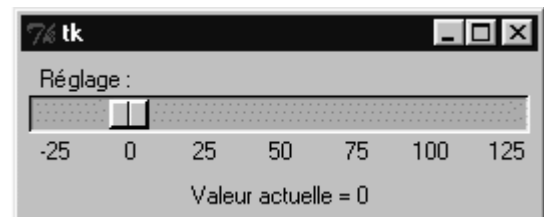
Dans l'exercice précédent, vous avez construit un nouveau type de widget que

vous avez sauvegardé dans le module `oscillo.py`. Conservez soigneusement ce module, car vous l'intégrerez bientôt dans un projet plus complexe.

Pour l'instant, vous allez construire encore un autre *widget*, plus interactif cette fois. Il s'agira d'une sorte de panneau de contrôle comportant trois curseurs de réglage et une case à cocher. Comme le précédent, ce *widget* est destiné à être réutilisé dans une application de synthèse.

Présentation du widget « Scale »

Commençons d'abord par découvrir un widget de base, que nous n'avons pas encore utilisé jusqu'ici : Le *widget* `Scale` se présente comme un curseur qui coulisse devant une échelle. Il permet à l'utilisateur de choisir rapidement la valeur d'un paramètre quelconque, d'une manière très attrayante.



Le petit script ci-dessous vous montre comment le paramétrer et l'utiliser dans une fenêtre :

```
1 from Tkinter import *
2
3 def updateLabel(x):
4     lab.configure(text='Valeur actuelle = ' + str(x))
5
6 root = Tk()
7 Scale(root, length=250, orient=HORIZONTAL, label='Réglage :',
8     troughcolor='dark grey', sliderlength=20,
9     showvalue=0, from_=-25, to=125, tickinterval=25,
10    command=updateLabel).pack()
11 lab = Label(root)
12 lab.pack()
13
14 root.mainloop()
```

Ces lignes ne nécessitent guère de commentaires.

Vous pouvez créer des *widgets* `Scale` de n'importe quelle taille (option `length`), en orientation horizontale (comme dans notre exemple) ou verticale (option `orient = VERTICAL`).

Les options `from_` (attention : n'oubliez pas le caractère 'souligné' !) et `to` définissent la plage de réglage. L'intervalle entre les repères numériques est défini dans l'option `tickinterval`, etc.

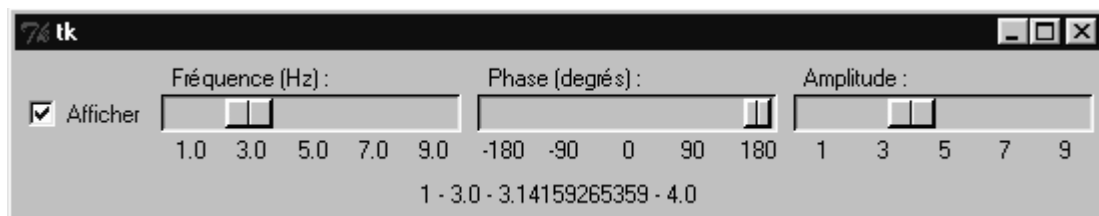
La fonction désignée dans l'option `command` est appelée automatiquement chaque fois que le curseur est déplacé, et la position actuelle du curseur par rapport à l'échelle lui est transmise en argument. Il est donc très facile d'utiliser cette valeur pour effectuer un traitement quelconque. Considérez par exemple le paramètre `x` de la fonction `updateLabel()`, dans notre exemple.

Le *widget* *scale* constitue une interface très intuitive et attrayante pour proposer différents réglages aux utilisateurs de vos programmes. Nous allons à présent l'incorporer en plusieurs exemplaires dans une nouvelle classe de *widget* : un panneau de contrôle destiné à choisir la fréquence, la phase et l'amplitude pour un mouvement vibratoire, dont nous afficherons ensuite le graphique élongation/temps à l'aide du *widget* *oscilloGraphe* construit dans les pages précédentes.

Construction d'un panneau de contrôle à trois curseurs

Comme le précédent, le script que nous décrivons ci-dessous est destiné à être sauvegardé dans un module, que vous nommerez cette fois *curseurs.py*. Les classes que vous sauvegardez ainsi seront réutilisées (par importation) dans une application de synthèse. Nous attirons votre attention sur le fait que le code ci-dessous peut être raccourci de différentes manières. Nous ne l'avons pas optimisé d'emblée, parce que cela nécessiterait d'y incorporer un concept supplémentaire (les expressions *lambda*), ce que nous préférons éviter pour l'instant.

Vous savez déjà que les lignes de code placées à la fin du script permettent de tester son fonctionnement. Vous devriez obtenir une fenêtre semblable à celle-ci :



```

1 from Tkinter import *
2 from math import pi
3
4 class ChoixVibra(Frame):
5     """Curseurs pour choisir fréquence, phase & amplitude d'une vibration"""
6     def __init__(self, boss=None, coul='red'):
7         Frame.__init__(self) # constructeur de la classe parente
8         # Initialisation de quelques attributs d'instance :
9         self.freq, self.phase, self.ampl, self.coul = 0, 0, 0, coul
10        # Variable d'état de la case à cocher :
11        self.chk = IntVar() # 'objet-variable' Tkinter
12        Checkbutton(self, text='Afficher', variable=self.chk,
13                   fg=self.coul, command=self.setCurve).pack(side=LEFT)
14        # Définition des 3 widgets curseurs :
15        Scale(self, length=150, orient=HORIZONTAL, sliderlength=25,
16              label='Fréquence (Hz) :', from_=1., to=9., tickinterval=2,
17              resolution=0.25,
18              showvalue=0, command=self.setFrequency).pack(side=LEFT)
19        Scale(self, length=150, orient=HORIZONTAL, sliderlength=15,
20              label='Phase (degrés) :', from_=-180, to=180, tickinterval=90,
21              showvalue=0, command=self.setPhase).pack(side=LEFT)
22        Scale(self, length=150, orient=HORIZONTAL, sliderlength=25,
23              label='Amplitude :', from_=1, to=9, tickinterval=2,
24              showvalue=0, command=self.setAmplitude).pack(side=LEFT)

```

```

25
26     def setCurve(self):
27         self.event_generate('<Control-Z>')
28
29     def setFrequency(self, f):
30         self.freq = float(f)
31         self.event_generate('<Control-Z>')
32
33     def setPhase(self, p):
34         pp =float(p)
35         self.phase = pp*2*pi/360           # conversion degrés -> radians
36         self.event_generate('<Control-Z>')
37
38     def setAmplitude(self, a):
39         self.ampl = float(a)
40         self.event_generate('<Control-Z>')
41
42     ##### Code pour tester la classe : #####
43
44     if __name__ == '__main__':
45         def afficherTout(event=None):
46             lab.configure(text = '%s - %s - %s - %s' %
47                             (fra.chk.get(), fra.freq, fra.phase, fra.ampl))
48             root = Tk()
49             fra = ChoixVibra(root, 'navy')
50             fra.pack(side =TOP)
51             lab = Label(root, text = 'test')
52             lab.pack()
53             root.bind('<Control-Z>', afficherTout)
54             root.mainloop()

```

Ce panneau de contrôle permettra à vos utilisateurs de régler aisément la valeur des paramètres indiqués (fréquence, phase et amplitude), lesquels pourront alors servir à commander l'affichage de graphiques élongation/temps dans un *widget* de la classe `OscilloGraphe()` construite précédemment, comme nous le montrerons dans l'application de synthèse.

Commentaires

- Ligne 6 : La méthode constructeur utilise un paramètre optionnel `coul`. Ce paramètre permettra de choisir une couleur pour le graphique soumis au contrôle du *widget*. Le paramètre `boss` sert à réceptionner la référence d'une fenêtre maîtresse éventuelle.
- Ligne 7 : Activation du constructeur de la classe parente (pour hériter sa fonctionnalité).
- Ligne 9 : Déclaration de quelques variables d'instance. Leurs vraies valeurs seront déterminées par les méthodes des lignes 29 à 40 (gestionnaires d'événements).
- Ligne 11 : Cette instruction instancie un objet de la classe `IntVar()`, laquelle fait partie du module `Tkinter` au même titre que les classes similaires `DoubleVar()`, `StringVar()` et `BooleanVar()`. Toutes ces classes permettent de définir des *variables Tkinter*, lesquels sont en fait des objets, mais qui se comportent comme des variables à l'intérieur des *widgets Tkinter*. Ainsi l'objet référencé dans `self.chk` contient l'équivalent d'une variable de type entier, dans un format utilisable par `Tkinter`. Pour accéder à sa valeur depuis Python, il faut utiliser des méthodes spécifiques de cette classe d'objets : la

méthode `set()` permet de lui assigner une valeur, et la méthode `get()` permet de la récupérer (ce que l'on met en pratique à la ligne 47).

- Ligne 12 : L'option variable de l'objet `checkboxbutton` est associée à la variable `Tkinter` définie à la ligne précédente. (Nous ne pouvons pas référencer directement une variable ordinaire dans la définition d'un *widget Tkinter*, parce que `Tkinter` lui-même est écrit dans un langage qui n'utilise pas les mêmes conventions que Python pour formater ses variables. Les objets construits à partir des classes de variables Tkinter sont donc nécessaires pour assurer l'interface).
- Ligne 13 : L'option `command` désigne la méthode que le système doit invoquer lorsque l'utilisateur effectue un clic de souris dans la case à cocher.
- Lignes 14 à 24 : Ces lignes définissent les trois *widgets* curseurs, en trois instructions similaires. Il serait plus élégant de programmer tout ceci en une seule instruction, répétée trois fois à l'aide d'une boucle. Cela nécessiterait cependant de faire appel à un concept que nous n'avons pas encore expliqué (les fonctions/expressions *lamdba*), et la définition du gestionnaire d'événements associé à ces widgets deviendrait elle aussi plus complexe. Conservons donc pour cette fois des instructions séparées : nous nous efforcerons d'améliorer tout cela plus tard.
- Lignes 26 à 40 : Les 4 *widgets* définis dans les lignes précédentes possèdent chacun une option `command`. Pour chacun d'eux, la méthode invoquée dans cette option `command` est différente : la case à cocher active la méthode `setCurve()`, le premier curseur active la méthode `setFrequency()`, le second curseur active la méthode `setPhase()`, et le troisième curseur active la méthode `setAmplitude()`. Remarquez bien au passage que l'option `command` des *widgets* `scale` transmet un argument à la méthode associée (la position actuelle du curseur), alors que la même option `command` ne transmet rien dans le cas du *widget* `checkboxbutton`. Ces 4 méthodes (qui sont donc les gestionnaires des événements produits par la case à cocher et les trois curseurs) provoquent elles-mêmes chacune l'émission d'un nouvel événement^[3], en faisant appel à la méthode `event_generate()`. Lorsque cette méthode est invoquée, Python envoie au système d'exploitation exactement le même message-événement que celui qui se produirait si l'utilisateur enfonçait simultanément les touches `<Ctrl>`, `<Maj>` et `<Z>` de son clavier. Nous produisons ainsi un message-événement bien particulier, qui peut être détecté et traité par un gestionnaire d'événement associé à un autre widget (voir page suivante). De cette manière, nous mettons en place *un véritable système de communication entre widgets* : chaque fois que l'utilisateur exerce une action sur notre panneau de contrôle, celui-ci génère un événement spécifique, qui signale cette action à l'attention des autres *widgets* présents.

Remarque : nous aurions pu choisir une autre combinaison de touches (ou même carrément un autre type d'événement). Notre choix s'est porté sur celle-ci parce qu'il y a vraiment très peu de chances que l'utilisateur s'en serve alors qu'il examine notre programme. Nous pourrions cependant produire nous-mêmes un tel événement au clavier à titre de test, lorsque le moment sera venu de vérifier le gestionnaire de cet événement, que nous mettrons en place par ailleurs.

- Lignes 42 à 54 : Comme nous l'avons déjà fait pour `oscillo.py`, nous

complétons ce nouveau module par quelques lignes de code au niveau principal. Ces lignes permettent de tester le bon fonctionnement de la classe : elles ne s'exécutent que si on lance le module directement, comme une application à part entière. Veillez à utiliser vous-même cette technique dans vos propres modules, car elle constitue une bonne pratique de programmation : l'utilisateur de modules construits ainsi peut en effet (re)découvrir très aisément leur fonctionnalité (en les exécutant) et la manière de s'en servir (en analysant ces quelques lignes de code). Dans ces lignes de test, nous construisons une fenêtre principale `root` qui contient deux *widgets* : un *widget* de la nouvelle classe `ChoixVibra()` et un *widget* de la classe `Label()`. À la ligne 53, nous associons à la fenêtre principale un gestionnaire d'événement : tout événement du type spécifié déclenche désormais un appel de la fonction `afficherTout()`. Cette fonction est donc notre gestionnaire d'événement spécialisé, qui est sollicité chaque fois qu'un événement de type `<Maj-Ctrl-Z>` est détecté par le système d'exploitation. Comme nous l'avons déjà expliqué plus haut, nous avons fait en sorte que de tels événements soient produits par les objets de la classe `ChoixVibra()`, chaque fois que l'utilisateur modifie l'état de l'un ou l'autre des trois curseurs, ou celui de la case à cocher. Conçue seulement pour effectuer un test, la fonction `afficherTout()` ne fait rien d'autre que provoquer l'affichage des valeurs des variables associées à chacun de nos quatre *widgets*, en (re)configurant l'option `text` d'un *widget* de classe `Label()`.

- Ligne 47, expression `fra.chk.get()` : nous avons vu plus haut que la variable mémorisant l'état de la case à cocher est un *objet-variable Tkinter*. Python ne peut pas lire directement le contenu d'une telle variable, qui est en réalité un objet-interface. Pour en extraire la valeur, il faut donc faire usage d'une méthode spécifique de cette classe d'objets : la méthode `get()`.

Propagation des événements

Le mécanisme de communication décrit ci-dessus respecte la hiérarchie de classes des *widgets*. Vous aurez noté que la méthode qui déclenche l'événement est associée au *widget* dont nous sommes en train de définir la classe, par l'intermédiaire de `self`. En général, un message-événement est en effet associé à un *widget* particulier (par exemple, un clic de souris sur un bouton est associé à ce bouton), ce qui signifie que le système d'exploitation va d'abord examiner s'il existe un gestionnaire pour ce type d'événement, qui soit lui aussi associé à ce *widget*. S'il en existe un, c'est celui-là qui est activé, et la propagation du message s'arrête. Sinon, le message-événement est « présenté » successivement aux *widgets* maîtres, dans l'ordre hiérarchique, jusqu'à ce qu'un gestionnaire d'événement soit trouvé, ou bien jusqu'à ce que la fenêtre principale soit atteinte.

Les événements correspondant à des frappes sur le clavier (telle la combinaison de touches `<Maj-Ctrl-Z>` utilisée dans notre exercice) sont cependant toujours expédiés directement à la fenêtre principale de l'application. Dans notre exemple, le gestionnaire de cet événement doit donc être associé à la fenêtre `root`.

Exercices

1. Votre nouveau *widget* hérite des propriétés de la classe `Frame()`. Vous pouvez donc modifier son aspect en modifiant les options par défaut de cette classe, à l'aide de la méthode `configure()`. Essayez par exemple de faire en sorte que le panneau de contrôle soit entouré d'une bordure de 4 pixels ayant l'aspect d'un sillon (`bd = 4`, `relief = GROOVE`). Si vous ne comprenez pas bien ce qu'il faut faire, inspirez-vous du script `oscillo.py` (ligne 10).
2. Si l'on assigne la valeur 1 à l'option `showvalue` des *widgets* `Scale()`, la position précise du curseur par rapport à l'échelle est affichée en permanence. Activez donc cette fonctionnalité pour le curseur qui contrôle le paramètre `phase`.
3. L'option `troughcolor` des *widgets* `Scale()` permet de définir la couleur de leur glissière. Utilisez cette option pour faire en sorte que la couleur des glissières des 3 curseurs soit celle qui est utilisée comme paramètre lors de l'instanciation de votre nouveau *widget*.
4. Modifiez le script de telle manière que les *widgets* curseurs soient écartés davantage les uns des autres (options `padx` et `pady` de la méthode `pack()`).

Solution

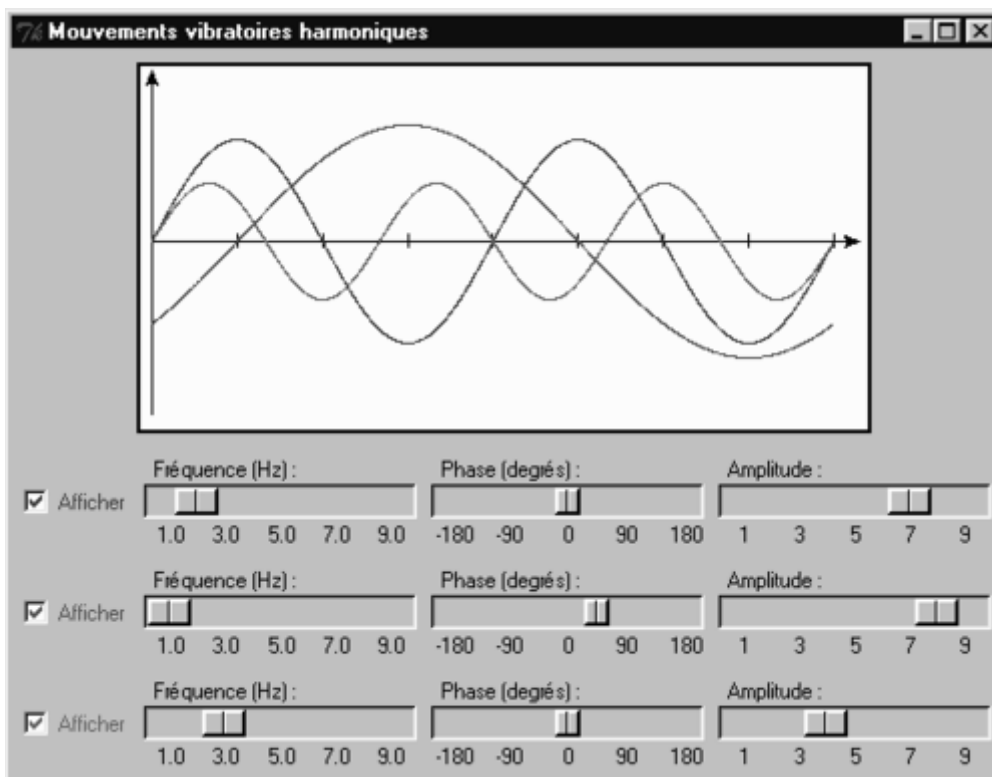
1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !

Intégration de widgets composites dans une application synthèse

Dans les exercices précédents, nous avons construit deux nouvelles classes de *widgets* : le *widget* `OscilloGraphe()`, canevas spécialisé pour le dessin de sinusoides, et le *widget* `ChoixVibra()`, panneau de contrôle à trois curseurs permettant de choisir les paramètres d'une vibration.

Ces *widgets* sont désormais disponibles dans les modules `oscillo.py` et `curseurs.py`^[4].

Nous allons à présent les utiliser dans une application synthèse, qui pourrait illustrer votre cours de physique : un *widget* `OscilloGraphe()` y affiche un, deux, ou trois graphiques superposés, de couleurs différentes, chacun d'entre eux étant soumis au contrôle d'un *widget* `ChoixVibra()` :



Le script correspondant est reproduit ci-après.

Nous attirons votre attention sur la technique mise en œuvre pour provoquer un rafraîchissement de l'affichage dans le canevas par l'intermédiaire d'un événement, chaque fois que l'utilisateur effectue une action quelconque au niveau de l'un des panneaux de contrôle.

Rappelez-vous que les applications destinées à fonctionner dans une interface graphique doivent être conçues comme des « programmes pilotés par les événements ».

En préparant cet exemple, nous avons arbitrairement décidé que l'affichage des graphiques serait déclenché par un événement particulier, tout à fait similaire à ceux que génère le système d'exploitation lorsque l'utilisateur accomplit une action quelconque. Dans la gamme (très étendue) d'événements possibles, nous en avons choisi un qui ne risque guère d'être utilisé pour d'autres raisons, pendant que notre application fonctionne : la combinaison de touches <Maj-Ctrl-Z>.

Lorsque nous avons construit la classe de *widgets* `ChoixVibra()`, nous y avons donc incorporé les instructions nécessaires pour que de tels événements soient générés, chaque fois que l'utilisateur actionne l'un des curseurs ou modifie l'état de la case à cocher. Nous allons à présent définir le gestionnaire de cet événement et l'inclure dans notre nouvelle classe : nous l'appellerons `montreCourbes()` et il se chargera de rafraîchir l'affichage. Étant donné que l'événement concerné est du type <enfoncement d'une touche>, nous devons cependant le détecter au niveau de la fenêtre principale de l'application.

```

1 from oscillo import *
2 from curseurs import *
3
4 class ShowVibra(Frame):
5     """Démonstration de mouvements vibratoires harmoniques"""
6     def __init__(self, boss=None):
7         Frame.__init__(self) # constructeur de la classe parente
8         self.couleur = ['dark green', 'red', 'purple']
9         self.trace = [0]*3 # liste des tracés (courbes à dessiner)
10        self.controle = [0]*3 # liste des panneaux de contrôle
11
12        # Instanciation du canevas avec axes X et Y :
13        self.gra = OscilloGraphe(self, larg=400, haut=200)
14        self.gra.configure(bg='white', bd=2, relief=SOLID)
15        self.gra.pack(side=TOP, pady=5)
16
17        # Instanciation de 3 panneaux de contrôle (curseurs) :
18        for i in range(3):
19            self.controle[i] = ChoixVibra(self, self.couleur[i])
20            self.controle[i].pack()
21
22        # Désignation de l'événement qui déclenche l'affichage des tracés :
23        self.master.bind('<Control-Z>', self.montrCourbes)
24        self.master.title('Mouvements vibratoires harmoniques')
25        self.pack()
26
27    def montrCourbes(self, event):
28        """(Ré)Affichage des trois graphiques élongation/temps"""
29        for i in range(3):
30
31            # D'abord, effacer le tracé précédent (éventuel) :
32            self.gra.delete(self.trace[i])
33
34            # Ensuite, dessiner le nouveau tracé :
35            if self.controle[i].chk.get():
36                self.trace[i] = self.gra.traceCourbe(
37                    coul = self.couleur[i],
38                    freq = self.controle[i].freq,
39                    phase = self.controle[i].phase,
40                    ampl = self.controle[i].ampl)
41
42    ##### Code pour tester la classe : ###
43
44    if __name__ == '__main__':
45        ShowVibra().mainloop()

```

Commentaires

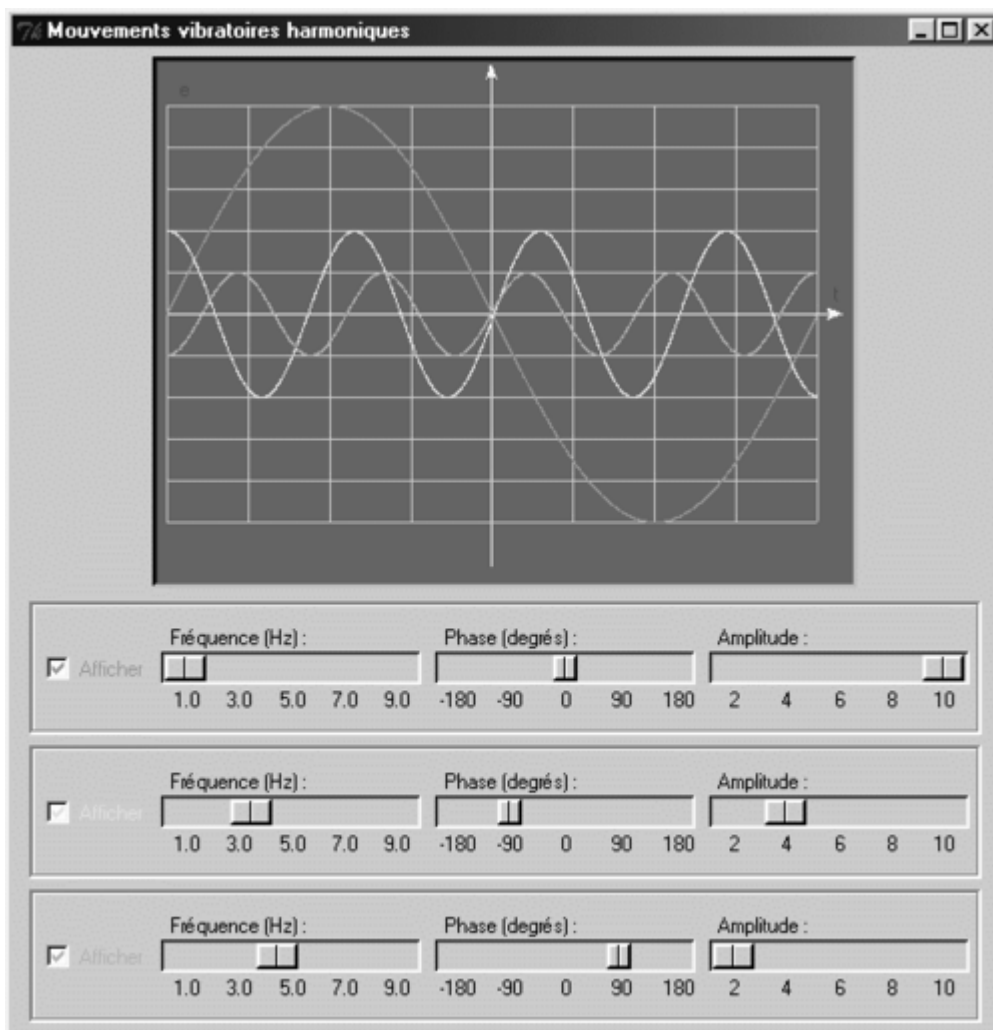
- Lignes 1-2 : Nous pouvons nous passer d'importer le module *Tkinter* : chacun de ces deux modules s'en charge déjà.
- Ligne 4 : Puisque nous commençons à connaître les bonnes techniques, nous décidons de construire l'application elle-même sous la forme d'une classe, dérivée de la classe `Frame()` : ainsi nous pourrons plus tard l'intégrer toute entière dans d'autres projets, si le cœur nous en dit.
- Lignes 8-10 : Définition de quelques variables d'instance (3 listes) : les trois courbes tracées seront des objets graphiques, dont les couleurs sont pré-définies dans la liste `self.couleur` ; nous devons préparer également une liste `self.trace` pour mémoriser les références de ces objets graphiques, et enfin

une liste `self.controle` pour mémoriser les références des trois panneaux de contrôle.

- Lignes 13 à 15 : Instanciation du *widget* d'affichage. Étant donné que la classe `OscilloGraphe()` a été obtenue par dérivation de la classe `Canvas()`, il est toujours possible de configurer ce *widget* en redéfinissant les options spécifiques de cette classe (ligne 13).
- Lignes 18 à 20 : Pour instancier les trois *widgets* « panneau de contrôle », on utilise une boucle. Leurs références sont mémorisées dans la liste `self.controle` préparée à la ligne 10. Ces panneaux de contrôle sont instanciés comme esclaves du présent *widget*, par l'intermédiaire du paramètre `self`. Un second paramètre leur transmet la couleur du tracé à contrôler.
- Lignes 23-24 : Au moment de son instanciation, chaque *widget Tkinter* reçoit automatiquement un attribut `master` qui contient la référence de la fenêtre principale de l'application. Cet attribut se révèle particulièrement utile si la fenêtre principale a été instanciée implicitement par *Tkinter*, comme c'est le cas ici. Rappelons en effet que lorsque nous démarrons une application en instanciant directement un *widget* tel que `Frame`, par exemple (c'est ce que nous avons fait à la ligne 4), *Tkinter* instancie automatiquement une fenêtre maîtresse pour ce *widget* (un objet de la classe `Tk()`). Comme cet objet a été créé automatiquement, nous ne disposons d'aucune référence dans notre code pour y accéder, si ce n'est par l'intermédiaire de l'attribut `master` que *Tkinter* associe automatiquement à chaque *widget*. Nous nous servons de cette référence pour redéfinir le bandeau-titre de la fenêtre principale (à la ligne 24), et pour y attacher un gestionnaire d'événement (à la ligne 23).
- Lignes 27 à 40 : La méthode décrite ici est le gestionnaire des événements <Maj-Ctrl-Z> spécifiquement déclenchés par nos *widgets* `ChoixVibra()` (ou « panneaux de contrôle »), chaque fois que l'utilisateur exerce une action sur un curseur ou une case à cocher. Dans tous les cas, les graphiques éventuellement présents sont d'abord effacés (ligne 28) à l'aide de la méthode `delete()` : le *widget* `OscilloGraphe()` a hérité cette méthode de sa classe parente `Canvas()`. Ensuite, de nouvelles courbes sont retracées, pour chacun des panneaux de contrôle dont on a coché la case « Afficher ». Chacun des objets ainsi dessinés dans le canevas possède un numéro de référence, renvoyé par la méthode `traceCourbe()` de notre *widget* `OscilloGraphe()`. Les numéros de référence de nos dessins sont mémorisés dans la liste `self.trace`. Ils permettent d'effacer individuellement chacun d'entre eux (cfr. instruction de la ligne 28).
- Lignes 38-40 : Les valeurs de fréquence, phase & amplitude que l'on transmet à la méthode `traceCourbe()` sont les attributs d'instance correspondants de chacun des trois panneaux de contrôle, eux-mêmes mémorisés dans la liste `self.controle`. Nous pouvons récupérer ces attributs en utilisant la qualification des noms par points.

Exercices

1. Modifiez le script, de manière à obtenir l'aspect ci-dessous (écran d'affichage avec grille de référence, panneaux de contrôle entourés d'un sillon) :



2. Modifiez le script, de manière à faire apparaître et contrôler 4 graphiques au lieu de trois. Pour la couleur du quatrième graphique, choisissez par exemple : 'blue', 'navy', 'maroon', ...
3. Aux lignes 33-35, nous récupérons les valeurs des fréquence, phase & amplitude choisies par l'utilisateur sur chacun des trois panneaux de contrôle, en accédant directement aux attributs d'instance correspondants. Python autorise ce raccourci - et c'est bien pratique - mais cette technique est dangereuse. Elle enfreint l'une des recommandations de la théorie générale de la « programmation orientée objet », qui préconise que l'accès aux propriétés des objets soit toujours pris en charge par des méthodes spécifiques. Pour respecter cette recommandation, ajoutez à la classe `ChoixVibra()` une méthode supplémentaire que vous appellerez `valeurs()`, et qui renverra un tuple contenant les valeurs de la fréquence, la phase et l'amplitude choisies. Les lignes 33 à 35 du présent script pourront alors être remplacées par quelque chose comme :

```
freq, phase, ampl = self.control[i].valeurs()
```

4. Écrivez une petite application qui fait apparaître une fenêtre avec un canevas et un *widget* curseur (*scale*). Dans le canevas, dessinez un cercle, dont l'utilisateur pourra faire varier la taille à l'aide du curseur.

5. Écrivez un script qui créera deux classes : une classe « Application », dérivée de `Frame()`, dont le constructeurinstanciera un canevas de 400x400 pixels, ainsi que deux boutons. Dans le canevas, vousinstancierez un objet de la classe « Visage » décrite ci-après. La classe « Visage » servira à définir des objets graphiques censés représenter des visages humains simplifiés. Ces visages seront constitués d'un cercle principal dans lequel trois ovales plus petits représenteront deux yeux et une bouche (ouverte). Une méthode "fermer" permettra de remplacer l'ovale de la bouche par une ligne horizontale. Une méthode « ouvrir » permettra de restituer la bouche de forme ovale. Les deux boutons définis dans la classe « Application » serviront respectivement à fermer et à ouvrir la bouche de l'objet « Visage » installé dans le canevas.
6. *Exercice de synthèse : élaboration d'un dictionnaire de couleurs. But :* réaliser un petit programme utilitaire, qui puisse vous aider à construire facilement et rapidement un nouveau dictionnaire de couleurs, lequel permettrait l'accès technique à une couleur quelconque par l'intermédiaire de son nom usuel en français. *Contexte :* En manipulant divers objets colorés avec *Tkinter*, vous avez constaté que cette bibliothèque graphique accepte qu'on lui désigne les couleurs les plus fondamentales sous la forme de chaînes de caractères contenant leur nom en anglais : 'red', 'blue', etc. Vous savez cependant qu'un ordinateur ne peut traiter que des informations numérisées. Cela implique que la désignation d'une couleur quelconque doit nécessairement tôt ou tard être encodée sous la forme d'un nombre. Il faut bien entendu adopter pour cela une convention, et celle-ci peut varier d'un système à un autre. L'une de ces conventions, parmi les plus courantes, consiste à représenter une couleur à l'aide de trois octets, qui indiqueront respectivement les intensités des trois composantes rouge, verte et bleue de cette couleur. Cette convention peut être utilisée avec *Tkinter* pour accéder à n'importe quelle nuance colorée. Vous pouvez en effet lui indiquer la couleur d'un élément graphique quelconque, à l'aide d'une chaîne de 7 caractères telle que '#00FA4E'. Dans cette chaîne, le premier caractère (#) signifie que ce qui suit est une valeur hexadécimale. Les six caractères suivants représentent les 3 valeurs hexadécimales des 3 composantes R, V et B. Pour visualiser concrètement la correspondance entre une couleur quelconque et son code, vous pouvez essayer le petit programme utilitaire *tkColorChooser.py* (qui se trouve généralement dans le sous-répertoire */lib-tk* de votre installation de Python). Étant donné qu'il n'est pas facile pour les humains que nous sommes de mémoriser de tels codes hexadécimaux, *Tkinter* est également doté d'un dictionnaire de conversion, qui autorise l'utilisation de noms communs pour un certain nombre de couleurs parmi les plus courantes, mais cela ne marche que pour des noms de couleurs exprimés en anglais. Le but du présent exercice est de réaliser un logiciel qui facilitera la construction d'un dictionnaire équivalent en français, lequel pourrait ensuite être incorporé à l'un ou l'autre de vos propres programmes. Une fois construit, ce dictionnaire serait donc de la forme : {'vert': '#00FF00', 'bleu': '#0000FF', ... etc ...}.

Cahier des charges

L'application à réaliser sera une application graphique, construite autour d'une classe. Elle comportera une fenêtre avec un certain nombre de champs d'entrée et de boutons, afin que l'utilisateur puisse aisément encoder de nouvelles couleurs en indiquant à chaque fois son nom français dans un champ, et son code hexadécimal dans un autre. Lorsque le dictionnaire contiendra déjà un certain nombre de données, il devra être possible de le tester, c'est-à-dire d'entrer un nom de couleur en français et de retrouver le code hexadécimal correspondant à l'aide d'un bouton (avec affichage éventuel d'une zone colorée). Un bouton provoquera l'enregistrement du dictionnaire dans un fichier texte. Un autre permettra de reconstruire le dictionnaire à partir du fichier.

7. Le script ci-dessous correspond à une ébauche de projet dessinant des ensembles de dés à jouer disposés à l'écran de plusieurs manières différentes (cette ébauche pourrait être une première étape dans la réalisation d'un logiciel de jeu). L'exercice consistera à analyser ce script et à le compléter. Vous vous placerez ainsi dans la situation d'un programmeur chargé de continuer le travail commencé par quelqu'un d'autre, ou encore dans celle de l'informaticien prié de participer à un travail d'équipe. Commencez par analyser ce script, et ajoutez-y des commentaires, en particulier aux lignes marquées : `***`, afin de montrer que vous comprenez ce que doit faire le programme à ces emplacements :

```

from Tkinter import *

class FaceDom:
    def __init__(self, can, val, pos, taille =70):
        self.can =can
        # ***
        x, y, c = pos[0], pos[1], taille/2
        can.create_rectangle(x -c, y-c, x+c, y+c, fill = 'ivory', width =2)
        d = taille/3
        # ***
        self.pList =[]
        # ***
        pDispo = [(0,0), ((-d,d),(d,-d)), ((-d,-d), (0,0), (d,d))]
        disp = pDispo[val -1]
        # ***
        for p in disp:
            self.cercle(x +p[0], y +p[1], 5, 'red')

    def cercle(self, x, y, r, coul):
        # ***
        self.pList.append(self.can.create_oval(x-r, y-r, x+r, y+r, fill=coul))

    def effacer(self):
        # ***
        for p in self.pList:
            self.can.delete(p)

class Projet(Frame):
    def __init__(self, larg, haut):
        Frame.__init__(self)
        self.larg, self.haut = larg, haut
    
```

```

self.can = Canvas(self, bg='dark green', width =larg, height =haut)
self.can.pack(padx =5, pady =5)
# ***
bList = [("A", self.boutA), ("B", self.boutB),
         ("C", self.boutC), ("D", self.boutD),
         ("Quitter", self.boutQuit)]
for b in bList:
    Button(self, text =b[0], command =b[1]).pack(side =LEFT)
self.pack()

def boutA(self):
    self.d3 = FaceDom(self.can, 3, (100,100), 50)

def boutB(self):
    self.d2 = FaceDom(self.can, 2, (200,100), 80)

def boutC(self):
    self.d1 = FaceDom(self.can, 1, (350,100), 110)

def boutD(self):
    # ***
    self.d3.effacer()

def boutQuit(self):
    self.master.destroy()

Projet(500, 300).mainloop()

```

Modifiez ensuite ce script, afin qu'il corresponde au cahier des charges suivant :

- Le canevas devra être plus grand : 600 x 600 pixels.
- Les boutons de commande devront être déplacés à droite et espacés davantage.
- La taille des points sur une face de dé devra varier proportionnellement à la taille de cette face

Variante 1 : Ne conservez que les 2 boutons A et B. Chaque utilisation du bouton A fera apparaître 3 nouveaux dés (de même taille, plutôt petits) disposés sur une colonne (verticale), les valeurs de ces dés étant tirées au hasard entre 1 et 6. Chaque nouvelle colonne sera disposée à la droite de la précédente. Si l'un des tirages de 3 dés correspond à 4, 2, 1 (dans n'importe quel ordre), un message « gagné » sera affiché dans la fenêtre (ou dans le canevas). Le bouton B provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 2 : Ne conservez que les 2 boutons A et B. Le bouton A fera apparaître 5 dés disposés en quinconce (c.à.d. comme les points d'une face de valeur 5). Les valeurs de ces dés seront tirées au hasard entre 1 et 6, mais il ne pourra pas y avoir de doublons. Le bouton B provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 3 : Ne conservez que les 3 boutons A, B et C. Le bouton A fera apparaître 13 dés de même taille disposés en cercle. Chaque utilisation du bouton B provoquera un changement de valeur du premier dé, puis

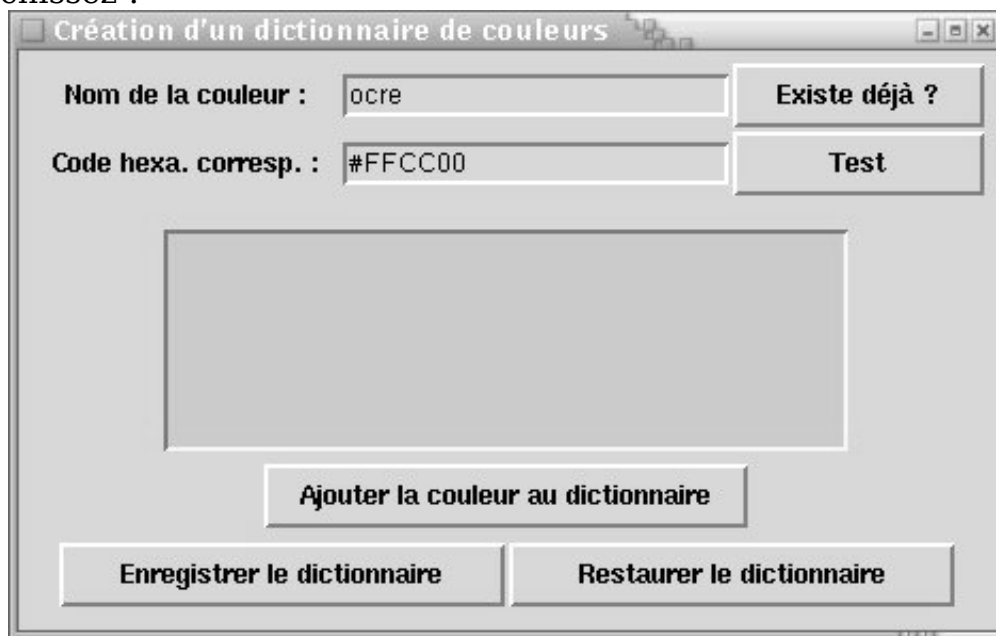
du deuxième, du troisième, etc. La nouvelle valeur d'un dé sera à chaque fois égale à sa valeur précédente augmentée d'une unité, sauf dans le cas où la valeur précédente était 6 : dans ce cas la nouvelle valeur est 1, et ainsi de suite. Le bouton C provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 4 : Ne conservez que les 3 boutons A, B et C. Le bouton A fera apparaître 12 dés de même taille disposés sur deux lignes de 6. Les valeurs des dés de la première ligne seront dans l'ordre 1, 2, 3, 4, 5, 6. Les valeurs des dés de la seconde ligne seront tirées au hasard entre 1 et 6. Chaque utilisation du bouton B provoquera un changement de valeur aléatoire du premier dé de la seconde ligne, tant que cette valeur restera différente de celle du dé correspondant dans la première ligne. Lorsque le 1er dé de la 2e ligne aura acquis la valeur de son correspondant, c'est la valeur du 2e dé de la seconde ligne qui sera changée au hasard, et ainsi de suite, jusqu'à ce que les 6 faces du bas soient identiques à celles du haut. Le bouton C provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Solution

1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !
5. Réfléchissez !

6.



```
# Dictionnaire de couleurs
from Tkinter import *
# Module donnant accès aux boîtes de dialogue standard pour
# la recherche de fichiers sur disque :
from tkFileDialog import asksaveasfile, askopenfile
```

```

class Application(Frame):
    '''Fenêtre d'application'''
    def __init__(self):
        Frame.__init__(self)
        self.master.title("Création d'un dictionnaire de couleurs")

        self.dico = {}          # création du dictionnaire

        # Les widgets sont regroupés dans deux cadres (Frames) :
        frSup = Frame(self)     # cadre supérieur contenant 6 widgets
        Label(frSup, text = "Nom de la couleur :",
              width = 20).grid(row = 1, column = 1)
        self.enNom = Entry(frSup, width = 25)          # champ d'entrée pour
        self.enNom.grid(row = 1, column = 2)          # le nom de la couleur
        Button(frSup, text = "Existe déjà ?", width = 12,
              command = self.chercheCoul).grid(row = 1, column = 3)
        Label(frSup, text = "Code hexa. corresp. :",
              width = 20).grid(row = 2, column = 1)
        self.enCode = Entry(frSup, width = 25)        # champ d'entrée pour
        self.enCode.grid(row = 2, column = 2)        # le code hexa.
        Button(frSup, text = "Test", width = 12,
              command = self.testeCoul).grid(row = 2, column = 3)
        frSup.pack(padx = 5, pady = 5)

        frInf = Frame(self)    # cadre inférieur contenant le reste
        self.test = Label(frInf, bg = "white", width = 45,    # zone de test
                          height = 7, relief = SUNKEN)
        self.test.pack(pady = 5)
        Button(frInf, text = "Ajouter la couleur au dictionnaire",
              command = self.ajouteCoul).pack()
        Button(frInf, text = "Enregistrer le dictionnaire", width = 25,
              command = self.enregistre).pack(side = LEFT, pady = 5)
        Button(frInf, text = "Restaurer le dictionnaire", width = 25,
              command = self.restaure).pack(side = RIGHT, pady = 5)
        frInf.pack(padx = 5, pady = 5)
        self.pack()

    def ajouteCoul(self):
        "ajouter la couleur présente au dictionnaire"
        if self.testeCoul() == 0:          # une couleur a-t-elle été définie ?
            return
        nom = self.enNom.get()
        if len(nom) > 1:                   # refuser les noms trop petits
            self.dico[nom] = self.cHexa
        else:
            self.test.config(text = "%s : nom incorrect" % nom, bg = 'white')

    def chercheCoul(self):
        "rechercher une couleur déjà inscrite au dictionnaire"
        nom = self.enNom.get()
        if self.dico.has_key(nom):
            self.test.config(bg = self.dico[nom], text = "")
        else:
            self.test.config(text = "%s : couleur inconnue" % nom, bg = 'white')

    def testeCoul(self):
        "vérifier la validité d'un code hexa. - afficher la couleur corresp."
        try:
            self.cHexa = self.enCode.get()
            self.test.config(bg = self.cHexa, text = "")
            return 1
        except:
            self.test.config(text = "Codage de couleur incorrect", bg = 'white')

```

```

        return 0

    def enregistre(self):
        "enregistrer le dictionnaire dans un fichier texte"
        # Cette méthode utilise une boîte de dialogue standard pour la
        # sélection d'un fichier sur disque. Tkinter fournit toute une série
        # de fonctions associées à ces boîtes, dans le module tkinterFileDialog.
        # La fonction ci-dessous renvoie un objet-fichier ouvert en écriture :
        ofi = asksaveasfile(filetypes=[("Texte", ".txt"), ("Tous", "*")])
        for clef, valeur in self.dico.items():
            ofi.write("%s %s\n" % (clef, valeur))
        ofi.close()

    def restaure(self):
        "restaurer le dictionnaire à partir d'un fichier de mémorisation"
        # La fonction ci-dessous renvoie un objet-fichier ouvert en lecture :
        ofi = askopenfile(filetypes=[("Texte", ".txt"), ("Tous", "*")])
        lignes = ofi.readlines()
        for li in lignes:
            cv = li.split()    # extraction de la clé et la valeur corresp.
            self.dico[cv[0]] = cv[1]
        ofi.close()

if __name__ == '__main__':
    Application().mainloop()

```

7. (variante 3) :

```

from Tkinter import *
from random import randrange
from math import sin, cos, pi

class FaceDom:
    def __init__(self, can, val, pos, taille =70):
        self.can = can
        x, y, c = pos[0], pos[1], taille/2
        self.carre = can.create_rectangle(x -c, y-c, x+c, y+c,
                                         fill='ivory', width =2)

        d = taille/3
        # disposition des points sur la face, pour chacun des 6 cas :
        self.pDispo = [((0,0),),
                       ((-d,d), (d,-d)),
                       ((-d,-d), (0,0), (d,d)),
                       ((-d,-d), (-d,d), (d,-d), (d,d)),
                       ((-d,-d), (-d,d), (d,-d), (d,d), (0,0)),
                       ((-d,-d), (-d,d), (d,-d), (d,d), (d,0), (-d,0))]

        self.x, self.y, self.dim = x, y, taille/15
        self.pList = []    # liste contenant les points de cette face
        self.tracer_points(val)

    def tracer_points(self, val):
        # créer les dessins de points correspondant à la valeur val :
        disp = self.pDispo[val -1]
        for p in disp:
            self.cercle(self.x +p[0], self.y +p[1], self.dim, 'red')
        self.val = val

    def cercle(self, x, y, r, coul):
        self.pList.append(self.can.create_oval(x-r, y-r, x+r, y+r, fill=coul))

```

```

def effacer(self, flag =0):
    for p in self.pList:
        self.can.delete(p)
    if flag:
        self.can.delete(self.carre)

class Projet(Frame):
    def __init__(self, larg, haut):
        Frame.__init__(self)
        self.larg, self.haut = larg, haut
        self.can = Canvas(self, bg='dark green', width =larg, height =haut)
        self.can.pack(padx =5, pady =5)
        # liste des boutons à installer, avec leur gestionnaire :
        bList = [("A", self.boutA), ("B", self.boutB),
                ("C", self.boutC), ("Quitter", self.boutQuit)]
        bList.reverse() # inverser l'ordre de la liste
        for b in bList:
            Button(self, text =b[0], command =b[1]).pack(side =RIGHT, padx=3)
        self.pack()
        self.des =[] # liste qui contiendra les faces de dés
        self.actu =0 # réf. du dé actuellement sélectionné

    def boutA(self):
        if len(self.des):
            return # car les dessins existent déjà !
        a, da = 0, 2*pi/13
        for i in range(13):
            cx, cy = self.larg/2, self.haut/2
            x = cx + cx*0.75*sin(a) # pour disposer en cercle,
            y = cy + cy*0.75*cos(a) # on utilise la trigono !
            self.des.append(FaceDom(self.can, randrange(1,7) , (x,y), 65))
            a += da

    def boutB(self):
        # incrémenter la valeur du dé sélectionné. Passer au suivant :
        v = self.des[self.actu].val
        v = v % 6
        v += 1
        self.des[self.actu].effacer()
        self.des[self.actu].tracer_points(v)
        self.actu += 1
        self.actu = self.actu % 13

    def boutC(self):
        for i in range(len(self.des)):
            self.des[i].effacer(1)
        self.des =[]
        self.actu =0

    def boutQuit(self):
        self.master.destroy()

Projet(600, 600).mainloop()

```

Notes

1. Comme nous l'avons déjà signalé précédemment, Python vous permet d'accéder aux attributs d'instance en utilisant la qualification des noms par points. D'autres langages de programmation l'interdisent, ou bien ne l'autorisent que moyennant une déclaration particulière de ces attributs (distinction entre attributs privés et

publics). Sachez en tous cas que ce n'est pas recommandé : le bon usage de la programmation orientée objet stipule en effet que vous ne devez pouvoir accéder aux attributs des objets que par l'intermédiaire de méthodes spécifiques.

2. *Tkinter* autorise également de construire la fenêtre principale d'une application par dérivation d'une classe de *widget* (le plus souvent, il s'agira d'un *widget* `Frame()`). La fenêtre englobant ce *widget* sera automatiquement ajoutée.
3. En fait, on devrait plutôt appeler cela un message (qui est lui-même la notification d'un événement). Programmes pilotés par des événements.
4. Il va de soi que nous pourrions rassembler toutes les classes que nous construisons dans un seul module.

Fichiers

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'information plus importante.

Utilité des fichiers

Imaginons par exemple que nous voulons écrire un petit programme exerciceur qui fasse apparaître à l'écran des questions à choix multiple, avec traitement automatique des réponses de l'utilisateur. Comment allons-nous mémoriser le texte des questions elles-mêmes ?

L'idée la plus simple consiste à placer chacun de ces textes dans une variable, en début de programme, avec des instructions d'affectation du genre :

```
a = "Quelle est la capitale du Guatemala ?"  
b = "Qui a succédé à Henri IV ?"  
c = "Combien font 26 × 43 ?"  
etc.
```

Cette idée est malheureusement beaucoup trop simpliste. Tout va se compliquer en effet lorsque nous essayerons d'élaborer la suite du programme, c'est-à-dire les instructions qui devront servir à sélectionner au hasard l'une ou l'autre de ces questions pour les présenter à l'utilisateur. Employer par exemple une longue suite d'instructions `if ... elif ... elif ...` comme dans l'exemple ci-dessous n'est certainement pas la bonne solution (ce serait d'ailleurs bien pénible à écrire : n'oubliez pas que nous souhaitons traiter un grand nombre de questions !)

```
if choix == 1:  
    selection = a  
elif choix == 2:  
    selection = b  
elif choix == 3:  
    selection = c  
... etc.
```

La situation se présente déjà beaucoup mieux si nous faisons appel à une liste :

```
liste = ["Qui a vaincu Napoléon à Waterloo ?",  
        "Comment traduit-on 'informatique' en anglais ?",  
        "Quelle est la formule chimique du méthane ?", ... etc ...]
```

On peut en effet extraire n'importe quel élément de cette liste à l'aide de son

indice. Exemple :

```
print liste[2]      ==> "Quelle est la formule chimique du méthane ?"
```

(rappel : l'indiciage commence à partir de zéro)

Même si cette façon de procéder est déjà nettement meilleure que la précédente, nous sommes toujours confrontés à plusieurs problèmes gênants :

- La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d'insérer l'une ou l'autre erreur de syntaxe dans la définition de cette longue liste. De telles erreurs seront bien difficiles à débusquer.
- L'ajout de nouvelles questions, ou la modification de certaines d'entre elles, imposeront à chaque fois de rouvrir le code source du programme. En corollaire, il deviendra malaisé de retravailler ce même code source, puisqu'il comportera de nombreuses lignes de données encombrantes.
- L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

Cette dernière remarque nous suggère la direction à prendre : il est temps que nous apprenions à *séparer les données, et les programmes qui les traitent, dans des fichiers différents.*

Pour que cela devienne possible, nous devons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par après.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches. Lorsque les quantités de données deviennent très importantes, il devient d'ailleurs rapidement nécessaire de structurer les relations entre ces données, et l'on doit alors élaborer des systèmes appelés *bases de données relationnelles*, dont la gestion peut s'avérer très complexe. Ce sera là l'affaire de logiciels très spécialisés tels que *Oracle*, *IBM DB*, *Adabas*, *PostgreSQL*, *MySQL*, etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard (voir chapitre Gestion d'une base de données).

Nos ambitions présentes sont plus modestes. Nos données ne se comptent pas encore par centaines de milliers, aussi nous pouvons nous contenter de mécanismes simples pour les enregistrer dans un fichier de taille moyenne, et les en extraire ensuite.

Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour

utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique aussi aux fichiers informatiques. Un fichier se compose de données enregistrées sur votre disque dur, sur une disquette ou sur un CD-ROM. Vous y accédez grâce à son nom (lequel peut inclure aussi un nom de répertoire). Vous pouvez toujours considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

Noms de fichiers - Répertoire courant

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers en question seront créés et/ou recherchés par Python *dans le répertoire courant*. Celui-ci est habituellement le répertoire où se trouve le script lui-même, sauf si vous lancez ce script depuis la fenêtre d'un shell IDLE, auquel cas le répertoire courant est défini au lancement de IDLE lui-même (Sous Windows, la définition de ce répertoire fait partie des propriétés de l'icône de lancement).

Si vous travaillez avec IDLE, vous souhaitez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Pour ce faire, utilisez les commandes suivantes en début de session. (Nous supposons ici que le répertoire visé est le répertoire `/home/jules/exercices`. Vous pouvez franchement utiliser cette syntaxe (c'est-à-dire des caractères `/` et non `\` en guise de séparateurs : c'est la convention en vigueur dans le monde Unix). Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous MacOS, Linux, ou Windows.^[1]

```
>>> from os import chdir
>>> chdir("/home/jules/exercices")
```

La première commande importe la fonction `chdir()` du module `os`. Le module `os` contient toute une série de fonctions permettant de dialoguer avec le système d'exploitation (`os` = *operating system*), quel que soit celui-ci.

La seconde commande provoque le changement de répertoire (`chdir` = *change directory*)

Notes

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux.

Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des *modules*. Ainsi vous connaissez déjà fort bien le module `math` et le module `Tkinter`.

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

Voici un exemple de la première méthode :

```
>>>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
C:\Python22\essais
```

La première ligne de cet exemple importe l'intégralité du module `os`, lequel contient de nombreuses fonctions intéressantes pour l'accès au système d'exploitation. La seconde ligne utilise la fonction `getcwd()` du module `os`^[2]. Comme vous pouvez le constater, la fonction `getcwd()` renvoie le nom du répertoire courant (`getcwd` = *get current working directory*).

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

```
>>> from os import getcwd
>>> rep_cour = getcwd()
>>> print rep_cour
C:\Python22\essais
```

Dans ce nouvel exemple, nous n'avons importé du module `os` que la fonction `getcwd()` seule. Importée de cette manière, la fonction s'intègre à notre propre code comme si nous l'avions écrite nous-mêmes. Dans les lignes où nous l'utilisons, il n'est pas nécessaire de rappeler qu'elle fait partie du module `os`.

Nous pouvons de la même manière importer plusieurs fonctions du même

module :

```
>>> from math import sqrt, pi, sin, cos
>>> print pi
3.14159265359
>>> print sqrt(5)      # racine carrée de 5
2.2360679775
>>> print sin(pi/6)   # sinus d'un angle de 30°
0.5
```

Nous pouvons même importer toutes les fonctions d'un module, comme dans :

```
from Tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. (Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible).

Dans les programmes d'une certaine importance, qui font appel à un grand nombre de modules d'origines diverses, il sera donc toujours préférable de privilégier plutôt la première méthode, c'est-à-dire celle qui utilise des noms pleinement qualifiés.

On fait généralement exception à cette règle dans le cas particulier du module `Tkinter`, parce que les fonctions qu'il contient sont très sollicitées (dès lors que l'on décide d'utiliser ce module).

Écriture séquentielle dans un fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un *objet-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 utilisant les méthodes spécifiques de cet objet-fichier.

L'exemple ci-dessous vous montre comment ouvrir un fichier « en écriture », y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> obFichier = open('Monfichier','a')
>>> obFichier.write('Bonjour, fichier !')
```

```
>>> obFichier.write("Quel beau temps, aujourd'hui !")
>>> obFichier.close()
>>>
```

Notes

- La première ligne crée l'objet-fichier « obFichier », lequel fait référence à un fichier véritable (sur disque ou disquette) dont le nom sera `Monfichier`. *Ne confondez pas le nom de fichier avec le nom de l'objet-fichier qui y donne accès*. A la suite de cet exercice, vous pouvez vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est `Monfichier` (et vous pouvez en visualiser le contenu à l'aide d'un éditeur quelconque).
- La fonction `open()` attend deux arguments, qui doivent être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture. `a` indique qu'il faut ouvrir ce fichier en mode « ajout » (*append*), ce qui signifie que les données à enregistrer doivent être ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent éventuellement déjà. Nous aurions pu utiliser aussi le mode `w` (pour *write*), mais lorsqu'on utilise ce mode, Python crée toujours un nouveau fichier (vide), et l'écriture des données commence à partir du début de ce nouveau fichier. S'il existe déjà un fichier de même nom, celui-ci est effacé au préalable.
- La méthode `write()` réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès *séquentiel*). Chaque nouvel appel de `write()` continue l'écriture à la suite de ce qui est déjà enregistré.
- La méthode `close()` referme le fichier. Celui-ci est désormais disponible pour tout usage.

Lecture séquentielle d'un fichier

Vous allez maintenant rouvrir le fichier, mais cette fois « en lecture », de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente :

```
>>> ofi = open('Monfichier', 'r')
>>> t = ofi.read()
>>> print t
Bonjour, fichier ! Quel beau temps, aujourd'hui !
>>> ofi.close()
```

Comme on pouvait s'y attendre, la méthode `read()` lit les données présentes dans le fichier et les transfère dans une variable de type « chaîne » (*string*) . Si on utilise cette méthode sans argument, la totalité du fichier est transférée.

Notes

- Le fichier que nous voulons lire s'appelle `Monfichier`. L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom-là. Si le fichier n'existe pas, nous obtenons un message d'erreur. Exemple :

```
>>> ofi = open('Monfichier', 'r')
IOError: [Errno 2] No such file or directory: 'Monfichier'
```

Par contre, nous ne sommes tenus à aucune obligation concernant le nom à choisir pour l'objet-fichier. C'est un nom de variable quelconque. Ainsi donc, dans notre première instruction, nous avons choisi de créer un objet-fichier `ofi`, faisant référence au fichier réel `Monfichier`, lequel est ouvert en lecture (argument `r`).

- Les deux chaînes de caractères que nous avons entrées dans le fichier sont à présent accolées en une seule. C'est normal, puisque nous n'avons fourni aucun caractère de séparation lorsque nous les avons enregistrées.
- La méthode `read()` peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier :

```
>>> ofi = open('Monfichier', 'r')
>>> t = ofi.read(7)
>>> print t
Bonjour
>>> t = ofi.read(15)
>>> print t
, fichier !Quel
```

S'il ne reste pas assez de caractères au fichier pour satisfaire la demande, la lecture s'arrête tout simplement à la fin du fichier :

```
>>> t = ofi.read(1000)
>>> print t
beau temps, aujourd'hui !
```

Si la fin du fichier est déjà atteinte, `read()` renvoie une chaîne vide :

```
>>> t = ofi.read()
>>> print t

>>> ofi.close()
```

L'instruction `break` pour sortir d'une boucle

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

La fonction ci-dessous illustre cette idée. Elle copie l'intégralité d'un fichier, quelle que soit sa taille, en transférant des portions de 50 caractères à la fois :

```
def copieFichier(source, destination):
    "copie intégrale d'un fichier"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt == "":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return
```

Si vous voulez tester cette fonction, vous devez lui fournir deux arguments : le premier est le nom du fichier original, le second est le nom à donner au fichier qui accueillera la copie. Exemple :

```
copieFichier('Monfichier', 'Tonfichier')
```

Vous aurez remarqué que la boucle `while` utilisée dans cette fonction est construite d'une manière différente de ce que vous avez rencontré précédemment. Vous savez en effet que l'instruction `while` doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également^[3] que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro.

Une boucle `while` construite comme nous l'avons fait ci-dessus devrait donc boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction `break`, laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle :

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
        break
    --- instructions diverses ---
    if <condition 3>:
        break
    etc.
```

Dans notre fonction `copieFichier()`, il est facile de voir que l'instruction `break` s'exécutera seulement lorsque la fin du fichier aura été atteinte.

Fichiers texte

Un fichier texte est un fichier qui contient des caractères imprimables et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non-imprimable appelé « marqueur de fin de ligne »^[4].

Il est très facile de traiter ce genre de fichiers sous Python. Les instructions suivantes créent un fichier texte de quatre lignes :

```
>>> f = open("Fichier texte", "w")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
>>> f.close()
```

Notez bien le marqueur de fin de ligne `\n` inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement. Sans ce marqueur, les caractères seraient enregistrés les uns à la suite des autres, comme dans les exemples précédents.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres. La méthode `readline()`, par exemple, ne lit qu'une seule ligne à la fois (en incluant le caractère de fin de ligne) :

```
>>> f = open('Fichier texte', 'r')
>>> t = f.readline()
>>> print t
Ceci est la ligne un
>>> print f.readline()
Voici la ligne deux
```

La méthode `readlines()` transfère toutes les lignes restantes dans une liste de chaînes :

```
>>> t = f.readlines()
>>> print t
['Voici la ligne trois\n', 'Voici la ligne quatre\n']
>>> f.close()
```

Remarques

- La liste apparaît ci-dessus en format brut, avec des apostrophes pour délimiter les chaînes, et les caractères spéciaux sous forme de codes numériques. Vous pourrez bien évidemment parcourir cette liste (à l'aide d'une boucle `while`, par exemple) pour en extraire les chaînes individuelles.
- La méthode `readlines()` permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros (Puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante). Si vous devez traiter de gros fichiers, utilisez plutôt la méthode

`readline()` dans une boucle, comme le montrera l'exemple de la page suivante.

- Notez bien que `readline()` est une méthode qui renvoie une chaîne de caractères, alors que la méthode `readlines()` renvoie une liste. A la fin du fichier, `readline()` renvoie une chaîne vide, tandis que `readlines()` renvoie une liste vide.

Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte en omettant toutes les lignes qui commencent par un caractère `#` :

```
def filtre(source,destination):
    "recopier un fichier en éliminant les lignes de remarques"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.readline()
        if txt == '':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

Pour appeler cette fonction, vous devez utiliser deux arguments : le nom du fichier original, et le nom du fichier destiné à recevoir la copie filtrée.
Exemple :

```
filtre('test.txt', 'test_f.txt')
```

Enregistrement et restitution de variables diverses

L'argument de la méthode `write()` doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères.

Nous pouvons réaliser cela à l'aide de la fonction intégrée `str()` :

```
>>> x = 52
>>> f.write(str(x))
```

Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les re-transformer correctement en valeurs numériques lorsque nous allons relire le fichier.

Exemple :

```

>>> a = 5
>>> b = 2.83
>>> c = 67
>>> f = open('Monfichier', 'w')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> print(f.read())
52.8367
>>> f.close()

```

Nous avons enregistré trois valeurs numériques. Mais comment pouvons-nous les distinguer dans la chaîne de caractères résultante, lorsque nous effectuons la lecture du fichier ? C'est impossible ! Rien ne nous indique d'ailleurs qu'il y a là trois valeurs plutôt qu'une seule, ou 2, ou 4,...

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module `pickle`^[5]. Voici comment il s'utilise :

```

>>> import pickle
>>> f = open('Monfichier', 'wb')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('Monfichier', 'rb')
>>> t = pickle.load(f)
>>> print(t, type(t))
5 <type 'int'>
>>> t = pickle.load(f)
>>> print(t, type(t))
2.83 <type 'float'>
>>> t = pickle.load(f)
>>> print(t, type(t))
67 <type 'int'>
>>> f.close()

```

Pour cet exemple, on considère que les variables `a`, `b` et `c` contiennent les mêmes valeurs que dans l'exemple précédent. La fonction `dump()` du module `pickle` attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction `pickle.load()` effectue le travail inverse, c'est-à-dire la restitution de chaque variable avec son type.

Vous pouvez aisément comprendre ce que font exactement les fonctions du module `pickle` en effectuant une lecture « classique » du fichier résultant, à l'aide de la méthode `read()` par exemple.

Exercices

Exercices

1. Écrivez un script qui compte dans un fichier texte quelconque le nombre de lignes contenant des caractères numériques.
2. Écrivez un script qui compte le nombre de mots contenus dans un fichier texte.
3. Écrivez un script qui recopie un fichier texte en veillant à ce que chaque ligne commence par une majuscule.
4. Écrivez un script qui recopie un fichier texte en fusionnant (avec la précédente) les lignes qui ne commencent pas par une majuscule.
5. Vous disposez d'un fichier contenant des valeurs numériques. Considérez que ces valeurs sont les diamètres d'une série de sphères. Écrivez un script qui utilise les données de ce fichier pour en créer un autre, organisé en lignes de texte qui exprimeront « en clair » les autres caractéristiques de ces sphères (surface de section, surface extérieure et volume), dans des phrases telles que :
 Diam. 46.20 cm Section = 1676.39 cm² Surf. = 6705.54 cm². Vol. = 51632.67 cm³
 Diam. 120.00 cm Section = 11309.73 cm² Surf. = 45238.93 cm². Vol. = 904778.68 cm³
 Diam. 0.03 cm Section = 0.00 cm² Surf. = 0.00 cm². Vol. = 0.00 cm³
 Diam. 13.90 cm Section = 151.75 cm² Surf. = 606.99 cm². Vol. = 1406.19 cm³
 Diam. 88.80 cm Section = 6193.21 cm² Surf. = 24772.84 cm². Vol. = 366638.04 cm³
 etc.
6. Vous avez à votre disposition un fichier texte dont les lignes représentent des valeurs numériques de type réel, sans exposant (et encodées sous forme de chaînes de caractères). Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant de telle sorte que leur partie décimale ne comporte plus qu'un seul chiffre après la virgule, celui-ci ne pouvant être que 0 ou 5 (l'arrondi doit être correct).

Solution

1. Réfléchissez !

2. *# Comptage du nombre de mots dans un texte*

```

fiSource = raw_input("Nom du fichier à traiter : ")
fs = open(fiSource, 'r')

n = 0          # variable compteur
while 1:
    ch = fs.readline()
    if ch == "":
        break
    # conversion de la chaîne lue en une liste de mots :
    li = ch.split()
    # totalisation des mots :
    n = n + len(li)

```

```

fs.close()
print("Ce fichier texte contient un total de %s mots" % (n))

```

3. *# Conversion en majuscule du premier caractère de chaque ligne*

```

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ch = fs.readline()
    if ch == "":
        break
    if ch[0] >= "A" and ch[0] <= "Z":
        # le premier car. est une majuscule. On passe.
        pass
    else:
        # Reconstruction de la chaîne:
        pc = ch[0].upper()      # Premier caractère converti
        rc = ch[1:]            # toute le reste de la chaîne
        ch = pc + rc           # fusion
        # variante utilisant une méthode encore plus intégrée :
        # ch = ch.capitalize()
    # Transcription :
    fd.write(ch)

fd.close()
fs.close()

```

4. *# Fusion de lignes pour former des phrases*

```

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

# On lit d'abord la première ligne :
ch1 = fs.readline()
# On lit ensuite les suivantes, en les fusionnant si nécessaire :
while 1:
    ch2 = fs.readline()
    if ch2 == "":
        break
    # Si la chaîne lue commence par une majuscule, on transcrit
    # la précédente dans le fichier destinataire, et on la
    # remplace par celle que l'on vient de lire :
    if ch2[0] >= "A" and ch2[0] <= "Z":
        fd.write(ch1)
        ch1 = ch2
    # Sinon, on la fusionne avec la précédente :
    else:
        ch1 = ch1[:-1] + " " + ch2
        # (veiller à enlever de ch1 le caractère de fin de ligne)

fd.write(ch1)      # ne pas oublier de transcrire la dernière !
fd.close()
fs.close()

```

```

5. # Caractéristiques de sphères :
# Le fichier de départ est un fichier <texte> dont chaque ligne contient
# un nombre réel (encodé sous la forme d'une chaîne de caractères)

from math import pi

def caractSphere(d):
    "renvoie les caractéristiques d'une sphère de diamètre d"
    d = float(d)      # conversion de l'argument (=chaîne) en réel
    r = d/2           # rayon
    ss = pi*r**2      # surface de section
    se = 4*pi*r**2    # surface extérieure
    v = 4./3*pi*r**3  # volume (! la 1e division doit être réelle !)
    # Le marqueur de conversion %8.2f utilisé ci-dessous formate le nombre
    # affiché de manière à occuper 8 caractères au total, en arrondissant
    # de manière à conserver deux chiffres après la virgule :
    ch = "Diam. %6.2f cm Section = %8.2f cm² " % (d, ss)
    ch = ch + "Surf. = %8.2f cm². Vol. = %9.2f cm³" % (se, v)
    return ch

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    diam = fs.readline()
    if diam == "" or diam == "\n":
        break
    fd.write(caractSphere(diam) + "\n")      # enregistrement
fd.close()
fs.close()

```

```

6. # Mise en forme de données numériques
# Le fichier traité est un fichier <texte> dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def arrondir(reel):
    "représentation arrondie à .0 ou .5 d'un nombre réel"
    ent = int(reel)      # partie entière du nombre
    fra = reel - ent     # partie fractionnaire
    if fra < .25 :
        fra = 0
    elif fra < .75 :
        fra = .5
    else:
        fra = 1
    return ent + fra

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    ligne = fs.readline()
    if ligne == "" or ligne == "\n":
        break
    n = arrondir(float(ligne))      # conversion en <float>, puis arrondi
    fd.write(str(n) + "\n")        # enregistrement
fd.close()
fs.close()

```

Exercices

1. Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier. L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c'est-à-dire utiliser la touche <Enter> seule). L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).
2. Considérons que vous avez à votre disposition un fichier texte contenant des phrases de différentes longueurs. Écrivez un script qui recherche et affiche la phrase la plus longue.
3. Écrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 30 (chacune d'entre elles incluant 20 termes seulement).
4. Écrivez un script qui recopie un fichier texte en triplant tous les espaces entre les mots.
5. Vous avez à votre disposition un fichier texte dont chaque ligne est la représentation d'une valeur numérique de type réel (mais sans exposants). Par exemple :
14.896
7894.6
123.278
etc.
Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant en nombres entiers (l'arrondi doit être correct).
6. Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.
7. A partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A, ... et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux. Complétez ensuite C avec les éléments restant sur l'autre.
8. Écrivez un script qui permette d'encoder un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et n° de téléphone de différentes personnes (considérez par exemple qu'il s'agit des membres d'un club)
9. Écrivez un script qui recopie le fichier utilisé dans l'exercice précédent, en y ajoutant la date de naissance et le sexe des personnes (l'ordinateur devra afficher les lignes une par une, et demander à l'utilisateur d'entrer pour chacune les données complémentaires).
10. Considérons que vous avez fait les exercices précédents et que vous disposez à présent d'un fichier contenant les coordonnées d'un certain nombre de personnes. Écrivez un script qui permette d'extraire de ce fichier les lignes qui correspondent à un code postal bien déterminé.
11. Modifiez le script de l'exercice précédent, de manière à retrouver les

lignes correspondant à des prénoms dont la première lettre est située entre F et M (inclus) dans l'alphabet.

- Écrivez des fonctions qui effectuent le même travail que celles du module "pickle". Ces fonctions doivent permettre l'enregistrement de variables diverses dans un fichier texte, en les accompagnant systématiquement d'informations concernant leur format exact.

Solution

```

1. 1. #(éditeur simple, pour lire et écrire dans un fichier 'texte') :
def sansDC(ch):
    "cette fonction renvoie la chaîne ch amputée de son dernier caractère"
    nouv = ""
    i, j = 0, len(ch) - 1
    while i < j:
        nouv = nouv + ch[i]
        i = i + 1
    return nouv

def ecrireDansFichier():
    of = open(nomF, 'a')
    while 1:
        ligne = raw_input("entrez une ligne de texte (ou <Enter>) : ")
        if ligne == '':
            break
        else:
            of.write(ligne + '\n')
    of.close()

def lireDansFichier():
    of = open(nomF, 'r')
    while 1:
        ligne = of.readline()
        if ligne == "":
            break
        # afficher en omettant le dernier caractère (= fin de ligne) :
        print sansDC(ligne)
    of.close()

nomF = raw_input('Nom du fichier à traiter : ')
choix = raw_input('Entrez "e" pour écrire, "c" pour consulter les données : ')

if choix == 'e':
    ecrireDansFichier()
else:
    lireDansFichier()

```

2. Réfléchissez !

```

3. 3. #(génération des tables de multiplication de 2 à 30) :
def tableMulti(n):
    # Fonction générant la table de multiplication par n (20 termes)
    # La table sera renvoyée sous forme d'une chaîne de caractères :
    i, ch = 0, ""
    while i < 20:
        i = i + 1
        ch = ch + str(i * n) + " "

```

```

    return ch

NomF = raw_input("Nom du fichier à créer : ")
fichier = open(NomF, 'w')

# Génération des tables de 2 à 30 :
table = 2
while table < 31:
    fichier.write(tableMulti(table) + '\n')
    table = table + 1
fichier.close()
Exercice 9.4 :
# Triplement des espaces dans un fichier texte.
# Ce script montre également comment modifier le contenu d'un fichier
# en le transférant d'abord tout entier dans une liste, puis en
# réenregistrant celle-ci après modifications

def triplerEspaces(ch):
    "fonction qui triple les espaces entre mots dans la chaîne ch"
    i, nouv = 0, ""
    while i < len(ch):
        if ch[i] == " ":
            nouv = nouv + "  "
        else:
            nouv = nouv + ch[i]
        i = i + 1
    return nouv

NomF = raw_input("Nom du fichier : ")
fichier = open(NomF, 'r+') # 'r+' = mode read/write
lignes = fichier.readlines() # lire toutes les lignes

n=0
while n < len(lignes):
    lignes[n] = triplerEspaces(lignes[n])
    n = n+1

fichier.seek(0) # retour au début du fichier
fichier.writelines(lignes) # réenregistrement
fichier.close()

```

4. Réfléchissez !

```

5. # Mise en forme de données numériques.
# Le fichier traité est un fichier texte dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def valArrondie(ch):
    "représentation arrondie du nombre présenté dans la chaîne ch"
    f = float(ch) # conversion de la chaîne en un nombre réel
    e = int(f + .5) # conversion en entier (On ajoute d'abord
    # 0.5 au réel pour l'arrondir correctement)
    return str(e) # reconversion en chaîne de caractères

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ligne = fs.readline() # lecture d'une ligne du fichier

```

```

    if ligne == "" or ligne == "\n":
        break
    ligne = valArrondie(ligne)
    fd.write(ligne + "\n")

fd.close()
fs.close()
Exercice 9.6 :
# Comparaison de deux fichiers, caractère par caractère :

fich1 = raw_input("Nom du premier fichier : ")
fich2 = raw_input("Nom du second fichier : ")
fi1 = open(fich1, 'r')
fi2 = open(fich2, 'r')

c, f = 0, 0          # compteur de caractères et "drapeau"
while 1:
    c = c + 1
    car1 = fi1.read(1)    # lecture d'un caractère dans chacun
    car2 = fi2.read(1)    # des deux fichiers
    if car1 == "" or car2 == "":
        break
    if car1 != car2 :
        f = 1
        break          # différence trouvée

fi1.close()
fi2.close()

print "Ces 2 fichiers",
if f ==1:
    print "diffèrent à partir du caractère n°", c
else:
    print "sont identiques."

```

6. Réfléchissez !

```

7. # Combinaison de deux fichiers texte pour en faire un nouveau

fichA = raw_input("Nom du premier fichier : ")
fichB = raw_input("Nom du second fichier : ")
fichC = raw_input("Nom du fichier destinataire : ")
fiA = open(fichA, 'r')
fiB = open(fichB, 'r')
fiC = open(fichC, 'w')

while 1:
    ligneA = fiA.readline()
    ligneB = fiB.readline()
    if ligneA == "" and ligneB == "":
        break          # On est arrivé à la fin des 2 fichiers
    if ligneA != "":
        fiC.write(ligneA)
    if ligneB != "":
        fiC.write(ligneB)

fiA.close()
fiB.close()
fiC.close()

```

```

8. # Enregistrer les coordonnées des membres d'un club

```

```

def encodage():
    "renvoie la liste des valeurs entrées, ou une liste vide"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    while 1:
        nom = raw_input("Nom : ")
        if nom == "":
            return []
        prenom = raw_input("Prénom : ")
        rueNum = raw_input("Adresse (N° et rue) : ")
        cPost = raw_input("Code postal : ")
        local = raw_input("Localité : ")
        tel = raw_input("N° de téléphone : ")
        print nom, prenom, rueNum, cPost, local, tel
        ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
        if ver == "":
            break
    return [nom, prenom, rueNum, cPost, local, tel]

def enregistrer(liste):
    "enregistre les données de la liste en les séparant par des <#>"
    i = 0
    while i < len(liste):
        of.write(liste[i] + "#")
        i = i + 1
    of.write("\n")           # caractère de fin de ligne

nomF = raw_input('Nom du fichier destinataire : ')
of = open(nomF, 'a')
while 1:
    tt = encodage()
    if tt == []:
        break
    enregistrer(tt)

of.close()

```

9. # Ajouter des informations dans le fichier du club

```

def traduire(ch):
    "convertir une ligne du fichier source en liste de données"
    dn = ""           # chaîne temporaire pour extraire les données
    tt = []          # la liste à produire
    i = 0
    while i < len(ch):
        if ch[i] == "#":
            tt.append(dn)   # on ajoute la donnée à la liste, et
            dn = ""        # on réinitialise la chaîne temporaire
        else:
            dn = dn + ch[i]
        i = i + 1
    return tt

def encodage(tt):
    "renvoyer la liste tt, complétée avec la date de naissance et le sexe"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    # Affichage des données déjà présentes dans la liste :
    i = 0
    while i < len(tt):
        print tt[i],
        i = i + 1
    print

```

```

while 1:
    daNai = raw_input("Date de naissance : ")
    sexe = raw_input("Sexe (m ou f) : ")
    print daNai, sexe
    ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
    if ver == "":
        break
tt.append(daNai)
tt.append(sexe)
return tt

def enregistrer(tt):
    "enregistrer les données de la liste tt en les séparant par des <#>"
    i = 0
    while i < len(tt):
        fd.write(tt[i] + "#")
        i = i + 1
    fd.write("\n")           # caractère de fin de ligne

fSource = raw_input('Nom du fichier source : ')
fDest = raw_input('Nom du fichier destinataire : ')
fs = open(fSource, 'r')
fd = open(fDest, 'w')
while 1:
    ligne = fs.readline()           # lire une ligne du fichier source
    if ligne == "" or ligne == "\n":
        break
    liste = traduire(ligne)         # la convertir en une liste
    liste = encodage(liste)         # y ajouter les données supplémentaires
    enregistrer(liste)              # sauvegarder dans fichier dest.

fd.close()
fs.close()

```

10. # Recherche de lignes particulières dans un fichier texte :

```

def chercheCP(ch):
    "recherche dans ch la portion de chaîne contenant le code postal"
    i, f, ns = 0, 0, 0           # ns est un compteur de codes #
    cc = ""                      # chaîne à construire
    while i < len(ch):
        if ch[i] == "#":
            ns = ns + 1
            if ns == 3:           # le CP se trouve après le 3e code #
                f = 1             # variable "drapeau" (flag)
            elif ns == 4:        # inutile de lire après le 4e code #
                break
        elif f == 1:             # le caractère lu fait partie du
            cc = cc + ch[i]       # CP recherché -> on mémorise
        i = i + 1
    return cc

nomF = raw_input("Nom du fichier à traiter : ")
codeP = raw_input("Code postal à rechercher : ")
fi = open(nomF, 'r')
while 1:
    ligne = fi.readline()
    if ligne == "":
        break
    if chercheCP(ligne) == codeP:
        print ligne
fi.close()

```

11. Réfléchissez !
12. Réfléchissez !

Exercices

1. Complétez l'exercice (mini-système de base de données) en lui ajoutant deux fonctions : l'une pour enregistrer le dictionnaire résultant dans un fichier texte, et l'autre pour reconstituer ce dictionnaire à partir du fichier correspondant. Chaque ligne de votre fichier texte correspondra à un élément du dictionnaire. Elle sera formatée de manière à bien séparer :
 - la clé et la valeur (c'est-à-dire le nom de la personne, d'une part, et l'ensemble : « âge + taille », d'autre part.
 - dans l'ensemble « âge + taille », ces deux données numériques.
 Vous utiliserez donc deux caractères séparateurs différents, par exemple « @ » pour séparer la clé et la valeur, et « # » pour séparer les données constituant cette valeur :

```
Juliette@18#1.67
Jean-Pierre@17#1.78
Delphine@19#1.71
Anne-Marie@17#1.63
etc.
```

2. Améliorez encore le script de l'exercice précédent, en utilisant un dictionnaire pour diriger le flux d'exécution du programme au niveau du menu principal. Votre programme affichera par exemple :

```
Choisissez :
(R)écupérer un dictionnaire préexistant sauvegardé dans un fichier
(A)jouter des données au dictionnaire courant
(C)onsulter le dictionnaire courant
(S)auvegarder le dictionnaire courant dans un fichier
(T)erminer :
```

Suivant le choix opéré par l'utilisateur, vous effectuerez alors l'appel de la fonction correspondante en la sélectionnant dans un dictionnaire de fonctions.

Solution

1. Sauvegarde d'un dictionnaire :

```
def enregistrement():
    fich = raw_input("Entrez le nom du fichier de sauvegarde : ")
    ofi = open(fich, "w")
    # parcours du dictionnaire entier, converti au préalable en une liste :
    for cle, valeur in dico.items():
        # utilisation du formatage des chaînes pour créer l'enregistrement :
        ofi.write("%s@%s#%s\n" % (cle, valeur[0], valeur[1]))
    ofi.close()
```

```
def lectureFichier():
    fich = raw_input("Entrez le nom du fichier de sauvegarde : ")
    try:
        ofi = open(fich, "r")
    except:
        print "*** fichier inexistant ***"
        return

    while 1:
        ligne = ofi.readline()
        if ligne == '':
            # détection de la fin de fichier
            break
        enreg = ligne.split("@") # restitution d'une liste [clé,valeur]
        cle = enreg[0]
        valeur = enreg[1][:-1] # élimination du caractère de fin de ligne
        data = valeur.split("#") # restitution d'une liste [âge, taille]
        age, taille = int(data[0]), float(data[1])
        dico[cle] = (age, taille) # reconstitution du dictionnaire
    ofi.close()
```

Ces deux fonctions peuvent être appelées respectivement à la fin et au début du programme principal, comme dans l'exemple ci-dessous :

```
dico = {}
lectureFichier()
while 1:
    choix = raw_input("Choisissez : (R)emplir - (C)onsulter - (T)erminer : ")
    if choix.upper() == 'T':
        break
    elif choix.upper() == 'R':
        remplissage()
    elif choix.upper() == 'C':
        consultation()
enregistrement()
```

2. Cet exercice complète le précédent. On ajoute encore deux petites fonctions, et on réécrit le corps principal du programme pour diriger le flux d'exécution en se servant d'un dictionnaire :

```
# Contrôle du flux d'exécution à l'aide d'un dictionnaire
def sortie():
    print "*** Job terminé ***"
    return 1 # afin de provoquer la sortie de la boucle

def autre():
    print "Veuillez frapper R, A, C, S ou T, svp."

dico = {}
fonc = {"R":lectureFichier, "A":remplissage, "C":consultation,
        "S":enregistrement, "T":sortie}
while 1:
    choix = raw_input("Choisissez :\n" +\
        "(R)écupérer un dictionnaire préexistant sauvegardé dans un fichier\n" +\
        "(A)jouter des données au dictionnaire courant\n" +\
        "(C)onsulter le dictionnaire courant\n" +\
        "(S)auvegarder le dictionnaire courant dans un fichier\n" +\
```

```
"(T)erminer : ")
# l'instruction ci-dessous appelle une fonction différente pour
# chaque choix, par l'intermédiaire du dictionnaire <fonc> :
if fonc.get(choix, autre()):
    break
# Rem : toutes les fonctions appelées ici renvoient <None> par défaut,
#       sauf la fonction sortie() qui renvoie 1 => sortie de la boucle
```

Notes

1. Dans le cas de Windows, vous pouvez également inclure dans ce chemin la lettre qui désigne le périphérique de stockage où se trouve le fichier. Par exemple : D:/home/jules/exercices.
2. Le point séparateur exprime donc ici une relation d'appartenance. Il s'agit d'un exemple de la *qualification des noms* qui sera de plus en plus largement exploitée dans la suite de ce cours. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-mêmes faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette `systeme.machin.truc` désigne l'élément `truc`, qui fait partie de l'ensemble `machin`, lequel fait lui-même partie de l'ensemble `systeme`. Nous verrons de nombreux exemples de cette technique de désignation, notamment lors de notre étude des classes d'objets.
3. Voir page `{{todo}}` : Véracité/fausseté d'une expression
4. Suivant le système d'exploitation utilisé, le codage correspondant au marqueur de fin de ligne peut être différent. Sous Windows, par exemple, il s'agit d'une séquence de deux caractères (Retour chariot et Saut de ligne), alors que dans les systèmes de type Unix (comme Linux) il s'agit d'un seul saut de ligne, MacOS pour sa part utilisant un seul retour chariot. En principe, vous n'avez pas à vous préoccuper de ces différences. Lors des opérations d'écriture, Python utilise la convention en vigueur sur votre système d'exploitation. Pour la lecture, Python interprète correctement chacune des trois conventions (qui sont donc considérées comme équivalentes).
5. En anglais, le terme *pickle* signifie "conserver". Le module a été nommé ainsi parce qu'il sert effectivement à enregistrer des données en conservant leur type.

Sources

- Apprendre à programmer avec Python 3

Exceptions

Principe

Les *exceptions* sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché. Exemple :

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

Remarque : d'autres informations complémentaires sont affichées, qui indiquent notamment à quel endroit du script l'erreur a été détectée, mais nous ne les reproduisons pas ici.

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme, et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à tout *un ensemble d'instructions*, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Syntaxe

Un mécanisme de ce type s'appelle en général *mécanisme de traitement des exceptions*. Celui de Python utilise l'ensemble d'instructions `try - except - else`, qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit :

Le bloc d'instructions qui suit directement une instruction `try` est exécuté par Python *sous réserve*. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction `except`. Si aucune erreur ne s'est produite dans les instructions qui suivent `try`, alors c'est le bloc qui suit l'instruction `else` qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom

de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se « plante ». Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
try:
    # Code pouvant générer une exception
except MonException:
    # Code en cas d'exception
else:
    # Code en cas de non exception
#code dans tous les cas
```

Exemple : Exemple de gestion d'exception

Exemple

Exemple de gestion d'exception : intercepter une division par zéro.

```
a = int(input("Veuillez saisir le numérateur : "))
b = int(input("Veuillez saisir le dénominateur : "))

try:
    c = a / b
    print(" Le resultat est : " + c)
except ZeroDivision:
    print(" Pas de division par zéro SVP")
finally:
    print("Fin du script")
```

```
filename = raw_input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r")
except:
    print "Le fichier", filename, "est introuvable"
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(fname):
    try:
        f = open(fname, 'r')
        f.close()
        return 1
    except:
        return 0
```

```
filename = raw_input("Veuillez entrer le nom du fichier : ")
if existe(filename):
    print "Ce fichier existe bel et bien."
else:
    print "Le fichier", filename, "est introuvable."
```

Il est également possible de faire suivre l'instruction "try" de plusieurs blocs "except", chacun d'entre eux traitant un type d'erreur spécifique.

Bases de données

Python contient des modules pour bases de données, comme MySQL, PostgreSQL, SQLite, IBM Informix Dynamic Server et Berkeley DB.

Par exemple pour installer le premier :

1. Sur Unix, taper : *apt-get install python-mysqldb*.
2. Sur Windows, télécharger en cliquant ici ou là.

MySQL

L'exemple ci-dessous permet d'exécuter une requête SQL de sélection :

```
1 import MySQLdb
2 db = MySQLdb.connect("machine", "dbuser", "password", "dbname")
3 cursor = db.cursor()
4 query = """SELECT * FROM sampletable"""
5 lines = cursor.execute(query)
6 data = cursor.fetchall()
7 db.close()
```

1. On ouvre l'accès aux fonctions du Module MySQLdb.
2. On tente d'établir une connexion à la base de données nommée (si cela ne fonctionne pas, essayer de pinger le nom de machine mentionné pour diagnostiquer un problème réseau).
3. La ligne trois définit l'objet "cursor" qui va servir d'interface avec la base de données.
4. On prépare ensuite la commande en langage SQL (ce qui pourrait aussi être défini en tout début du programme).
5. On exécute cette requête dans la base.
6. On formate les données brutes du résultat
7. On clos la connexion.

Remarque : quand il y a beaucoup de lignes, il est préférable d'utiliser `row = cursor.fetchone()` pour une meilleure visibilité :

```
1 import MySQLdb
2 db = MySQLdb.connect("machine", "dbuser", "password", "dbname")
3 cursor = db.cursor()
4 query = """SELECT * FROM sampletable"""
5 lines = cursor.execute(query)
6 while True:
7     row = cursor.fetchone()
8     if row == None: break
9 db.close()
```

Le résultat du `fetchone()` est de type *tuple*.

Par ailleurs, la connexion à la base (en ligne 2) peut être enregistrée dans un

fichier de configuration, celle-ci devient alors :

```
import MySQLdb
db = MySQLdb.connect(read_default_file=~/.my.cnf")
...
```

Postgres

```
import psycopg2
conn = psycopg2.connect("dbname=test")
cursor = conn.cursor()
cursor.execute("select * from test");
for i in cursor.next():
    print i
conn.close()
```

Installation de Gadfly

Gadfly est un système de bases de données écrit en Python.

Depuis le site <http://sourceforge.net/projects/gadfly>, télécharger le paquetage *gadfly-1.0.0.tar.gz*. Il s'agit d'un fichier archive comprimé. Copiez ce fichier dans un répertoire temporaire.

Sous Windows

Dans un répertoire temporaire quelconque, décompressez le fichier archive à l'aide d'un logiciel tel que *Winzip*.

Ouvrez une fenêtre DOS et entrez dans le sous-répertoire qui s'est créé automatiquement.

Lancez la commande : `python setup.py install`

C'est tout. Vous pouvez éventuellement améliorer les performances, en ajoutant l'opération suivante :

Dans le sous-répertoire qui s'est créé, ouvrez le sous-répertoire `kjbuckets`, puis le sous-répertoire qui correspond à votre version de Python. Recopiez le fichier `*.pyd` qui s'y trouve dans le répertoire racine de votre installation de Python.

Lorsque tout est terminé, effacez le contenu de votre répertoire temporaire.

Sous Linux

En tant qu'administrateur (root), choisissez un répertoire temporaire quelconque et décompressez-y le fichier archive à l'aide de l'utilitaire *tar*, qui fait certainement partie de votre distribution. Entrez simplement la

commande : `tar -xvzf gadfly-1.0.0.tar.gz`

Entrez dans le sous-répertoire qui s'est créé automatiquement : `cd gadfly-1.0.0`

Lancez la commande : `python setup.py install`

C'est tout. Si votre système Linux comporte un compilateur C, vous pouvez améliorer les performances de Gadfly en recompilant la bibliothèque *kjbuckets*. Pour ce faire, entrez encore les deux commandes suivantes :

- `cd kjbuckets`
- `python setup.py install`

Lorsque tout est terminé, effacez tout le contenu du répertoire temporaire.

Références



Tout ou partie de cette leçon est issu de la traduction d'une page sous licence GFDL « anglais *Python Programming/Databases* » .



Consultez l'historique de la page originale pour connaître la liste de ses auteurs.

Liens externes

- (anglais) documentation SQLite
- (anglais) Psycopg2 (module PostgreSQL)
- (anglais) module MySQL
- (anglais) module IBM Informix Dynamic Server

Gestion d'une base de données

Les bases de données sont des outils de plus en plus fréquemment utilisés. Elles permettent de stocker des données nombreuses dans un seul ensemble bien structuré. Lorsqu'il s'agit de bases de données relationnelles, il devient en outre tout à fait possible d'éviter l'« enfer des doublons ». Vous avez sûrement été déjà confrontés à ce problème :

Des données identiques ont été enregistrées dans plusieurs fichiers différents. Lorsque vous souhaitez modifier ou supprimer l'une de ces données, vous devez ouvrir et modifier tous les fichiers qui la contiennent ! Le risque d'erreur est très réel, qui conduit inévitablement à des incohérences, sans compter la perte de temps que cela représente.

Les bases de données constituent *la* solution à ce type de problème. Python vous permet d'en utiliser de nombreux systèmes, mais nous n'en examinerons que deux dans nos exemples : Gadfly et MySQL.

Les bases de données

Il existe de nombreux types de bases de données. On peut par exemple déjà considérer comme une base de données élémentaire, un fichier qui contient une liste de noms et d'adresses.

Si la liste n'est pas trop longue, et si l'on ne souhaite pas pouvoir y effectuer des recherches en fonction de critères complexes, il va de soi que l'on peut accéder à ce type de données en utilisant des instructions simples, telles celles que nous avons abordées dans Programmation Python/Bases de données.

La situation se complique cependant très vite si l'on souhaite pouvoir effectuer des sélections et des tris parmi les données, surtout si celles-ci deviennent très nombreuses. La difficulté augmente encore si les données sont répertoriées dans différents ensembles reliés par un certain nombre de relations hiérarchiques, et si plusieurs utilisateurs doivent pouvoir y accéder en parallèle.

Imaginez par exemple que la direction de votre école vous confie la charge de mettre au point un système de bulletins informatisé. En y réfléchissant quelque peu, vous vous rendrez compte rapidement que cela suppose la mise en œuvre de toute une série de tables différentes : une table des noms d'élèves (laquelle pourra bien entendu contenir aussi d'autres informations spécifiques à ces élèves : adresse, date de naissance, etc.) ; une table contenant la liste des cours (avec le nom du professeur titulaire, le nombre d'heures enseignées par semaine, etc.) ; une table mémorisant les travaux pris en compte pour l'évaluation (avec leur importance, leur date, leur contenu, etc.) ; une table décrivant la manière dont les élèves sont groupés par classes ou par options, les cours suivis par chacun, etc., etc.

Vous comprenez bien que ces différentes tables ne sont pas indépendantes. Les travaux effectués par un même élève sont liés à des cours différents. Pour établir le bulletin de cet élève, il faut donc extraire des données de la table des travaux, bien sûr, mais en relation avec des informations trouvées dans d'autres tables (celles des cours, des classes, des options, etc.)

SGBDR - Le modèle client/serveur

Les programmes informatiques capables de gérer efficacement de tels ensembles de données complexes sont forcément complexes, eux aussi. On appelle ces programmes des SGBDR (Systèmes de Gestion de Bases de Données Relationnelles). Il s'agit d'applications informatiques de première importance pour les entreprises. Certaines sont les fleurons de sociétés spécialisées (*IBM, Oracle, Microsoft, Informix, Sybase...*) et sont en général vendues à des prix fort élevés. D'autres ont été développées dans des centres de recherche et d'enseignement universitaires (*PostgreSQL, MySQL...*); elles sont alors en général tout à fait gratuites.

Ces systèmes ont chacun leurs spécificités et leurs performances, mais la plupart fonctionnant sur le modèle *client/serveur* : cela signifie que la plus grosse partie de l'application (ainsi que la base de données prise en charge) est installée en un seul endroit, en principe sur une machine puissante (cet ensemble constituant donc le serveur), alors que l'autre partie, beaucoup plus simple, est installée sur un nombre indéterminé de postes de travail, et on appelle celles-ci des *clients*.

Les clients sont reliés au serveur, en permanence ou non, par divers procédés et protocoles (éventuellement par l'intermédiaire de l'internet). Chacun d'entre eux peut accéder à une partie plus ou moins importante des données, avec autorisation ou non de modifier certaines d'entre elles, d'en ajouter ou d'en supprimer, en fonction de règles d'accès bien déterminées. (Ces règles sont définies par un *administrateur* de la base de données).

Le serveur et ses clients sont en fait des applications distinctes qui s'échangent des informations. Imaginez par exemple que vous êtes l'un des utilisateurs du système. Pour accéder aux données, vous devez lancer l'exécution d'une application cliente sur un poste de travail quelconque. Dans son processus de démarrage, l'application cliente commence par établir la connexion avec le serveur et la base de données^[1]. Lorsque la connexion est établie, l'application cliente peut interroger le serveur en lui envoyant une *requête* sous une forme convenue. Il s'agit par exemple de retrouver une information précise. Le serveur exécute alors la requête en recherchant les données correspondantes dans la base, puis il expédie en retour une certaine *réponse* au client.

Cette réponse peut être l'information demandée, ou encore un message d'erreur en cas d'insuccès.

La communication entre le client et le serveur est donc faite de requêtes et de réponses. Les requêtes sont de véritables instructions expédiées du client au

serveur, non seulement pour extraire des données de la base, mais aussi pour en ajouter, en supprimer, en modifier, etc.

Le langage SQL - Gadfly

Étant donnée la diversité des SGBDR existants, on pourrait craindre que chacun d'eux nécessite l'utilisation d'un langage particulier pour les requêtes qu'on lui adresse. En fait, de grands efforts ont été accomplis un peu partout pour la mise au point d'un langage commun, et il existe à présent un standard bien établi : *SQL (Structured Query Language, ou langage de requêtes structuré)*^[2].

Vous aurez probablement l'occasion de rencontrer SQL dans d'autres domaines (bureautique, par exemple). Dans le cadre de cette introduction à l'apprentissage de la programmation avec Python, nous allons nous limiter à la présentation de deux exemples : la mise en oeuvre d'un petit SGBDR réalisé exclusivement à l'aide de Python, et l'ébauche d'un logiciel client plus ambitieux destiné à communiquer avec un serveur de bases de données MySQL.

Notre première réalisation utilisera un module nommé *Gadfly*. Entièrement écrit en Python, ce module ne fait pas partie de la distribution standard et doit donc être installé séparément. Il intègre un large sous-ensemble de commandes SQL. Ses performances ne sont évidemment pas comparables à celles d'un gros SGBDR spécialisé^[3], mais elles sont tout à fait excellentes pour la gestion de bases de données modestes. Absolument portable comme Python lui-même, *Gadfly* fonctionnera indifféremment sous Windows, Linux ou MacOS. De même, les répertoires contenant des bases de données produites sous *Gadfly* pourront être utilisées sans modification depuis l'un ou l'autre de ces systèmes.

Si vous souhaitez développer une application qui doit gérer des relations relativement complexes dans une petite base de données, le module *Gadfly* peut vous faciliter grandement la tâche.

Mise en oeuvre d'une base de données simple avec Gadfly

Nous allons ci-après examiner comment mettre en place une application simple, qui fasse office à la fois de serveur et de client sur la même machine.

Création de la base de données

Comme vous vous y attendez certainement, il suffit d'importer le module *Gadfly* pour accéder aux fonctionnalités correspondantes^[4].

Vous devez ensuite créer une instance (un objet) de la classe *Gadfly* :

```
import gadfly
baseDonn = gadfly.gadfly()
```

L'objet `baseDonn` ainsi créé est votre moteur de base de données local, lequel effectuera la plupart de ses opérations en mémoire vive. Ceci permet une exécution très rapide des requêtes.

Pour créer la base de données proprement dite, il faut employer la méthode "startup" de cet objet :

```
baseDonn.startup("mydata", "E:/Python/essais/gadfly")
```

Le premier paramètre transmis, `mydata`, est le nom choisi pour la base de données (vous pouvez évidemment choisir un autre nom !). Le second paramètre est le répertoire où l'on souhaite installer cette base de données. (Ce répertoire doit avoir été créé au préalable, et toute base de données de même nom qui préexisterait dans ce répertoire est écrasée sans avertissement).

Les trois lignes de code que vous venez d'entrer sont suffisantes : vous disposez dès à présent d'une base de données fonctionnelle, dans laquelle vous pouvez créer différentes tables, puis ajouter, supprimer ou modifier des données dans ces tables.

Pour toutes ces opérations, vous allez utiliser le langage *SQL*.

Afin de pouvoir transmettre vos requêtes *SQL* à l'objet `baseDonn`, vous devez cependant mettre en œuvre un *curseur*. Il s'agit d'une sorte de tampon mémoire intermédiaire, destiné à mémoriser temporairement les données en cours de traitement, ainsi que les opérations que vous effectuez sur elles, avant leur transfert définitif dans de vrais fichiers. Cette technique permet donc d'annuler si nécessaire une ou plusieurs opérations qui se seraient révélées inadéquates (Vous pouvez en apprendre davantage sur ce concept en consultant l'un des nombreux manuels qui traitent du langage *SQL*). Veuillez à présent examiner le petit script ci-dessous, et noter que les requêtes *SQL* sont des chaînes de caractères, prises en charge par la méthode "execute" de l'objet *curseur* :

```
cur = baseDonn.cursor()
cur.execute("create table membres (age integer, nom varchar, taille float)")
cur.execute("insert into membres(age, nom, taille) values (21,'Dupont',1.83)")
cur.execute("INSERT INTO MEMBRES(AGE, NOM, TAILLE) VALUES (15,'Suleau',1.57)")
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (18,'Forcas',1.69)")
baseDonn.commit()
```

La première des lignes ci-dessus crée l'objet curseur `cur`. Les chaînes de caractères comprises entre guillemets dans les 4 lignes suivantes contiennent des requêtes *SQL* très classiques. *Notez bien que le langage SQL ne tient*

aucun compte de la casse des caractères : vous pouvez encoder vos requêtes SQL indifféremment en majuscules ou en minuscules (ce qui n'est pas le cas pour les instructions Python environnantes, bien entendu !)

La seconde ligne crée une table nommée `membres`, laquelle contiendra des enregistrements de 3 champs : le champ `age` de type « nombre entier », le champ `nom` de type « chaîne de caractères » (de longueur variable^[5]) et le champ `taille`, de type « nombre réel » (à virgule flottante). Le langage SQL autorise en principe d'autres types, mais ils ne sont pas implémentés dans Gadfly.

Les trois lignes qui suivent sont similaires. Nous y avons mélangé majuscules et minuscules pour bien montrer que la casse n'est pas significative en SQL. Ces lignes servent à insérer trois enregistrements dans la table `membres`.

À ce stade des opérations, les enregistrement n'ont pas encore été transférés dans de véritables fichiers sur disque. Il est donc possible de revenir en arrière. Le transfert sur disque est activé par la méthode `commit()` de la dernière ligne d'instructions.

Connexion à une base de données existante

Supposons qu'à la suite des opérations ci-dessus, nous décidions de terminer le script, ou même d'éteindre l'ordinateur. Comment devons-nous procéder par la suite pour accéder à nouveau à notre base de données ?

L'accès à une base de données existante ne nécessite que deux lignes de code :

```
import gadfly
baseDonn = gadfly.gadfly("mydata", "E:/Python/essais/gadfly")
```

Ces deux lignes suffisent en effet pour transférer en mémoire vive les tables contenues dans les fichiers enregistrés sur disque. La base de données peut désormais être interrogée et modifiée :

```
cur = baseDonn.cursor()
cur.execute("select * from membres")
print cur.pp()
```

La première de ces trois lignes ouvre un curseur. La requête émise dans la seconde ligne demande la sélection d'un ensemble d'enregistrements, qui seront transférés de la base de données au curseur. Dans le cas présent, la sélection n'en n'est pas vraiment une : on y demande en effet d'extraire tous les enregistrements de la table `membres` (le symbole `*` est fréquemment utilisé en informatique avec la signification « tout » ou « tous »).

La méthode `pp()` utilisée sur le curseur, dans la troisième ligne, provoque un affichage de tout ce qui est contenu dans le curseur sous une forme pré-

formatée (les données présentes sont automatiquement disposées en colonnes). « pp » doit en effet être compris comme « *pretty print* ».

Si vous préférez contrôler vous-même la mise en page des informations, il vous suffit d'utiliser à sa place la méthode `fetchall()`, laquelle renvoie une liste de tuples. Essayez par exemple :

```
for x in cur.fetchall():
    print x, x[0], x[1], x[2]
```

Vous pouvez bien entendu ajouter des enregistrements supplémentaires :

```
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (19,'Ricard',1.75)")
```

Pour modifier un ou plusieurs enregistrements, exécutez une requête du type :

```
cur.execute("update membres set nom ='Gerart' where nom='Ricard'")
```

Pour supprimer un ou plusieurs enregistrements, utilisez une requête telle que :

```
cur.execute("delete from membres where nom='Gerart'")
```

Si vous effectuez toutes ces opérations à la ligne de commande de Python, vous pouvez en observer le résultat à tout moment en effectuant un *pretty print* comme expliqué plus haut. Étant donné que toutes les modifications apportées au curseur se passent en mémoire vive, rien n'est enregistré définitivement tant que vous n'exécutez pas l'instruction `baseDonn.commit()`.

Vous pouvez donc annuler toutes les modifications apportées depuis le `commit()` précédent, en refermant la connexion à l'aide de l'instruction :

```
baseDonn.close()
```

Recherches dans une base de données

Exercices

1. Avant d'aller plus loin, et à titre d'exercice de synthèse, nous allons vous demander de créer entièrement vous-même une base de données *Musique* qui contiendra les deux tables suivantes (Cela représente un certain travail, mais il faut que vous puissiez disposer d'un certain nombre de données pour pouvoir expérimenter les fonctions de recherche et de tri) :

oeuvres	compositeurs
comp (chaîne)	comp (chaîne)
titre (chaîne)	a_naiss (entier)
duree (entier)	a_mort (entier)
interpr (chaîne)	

Solution

1. (Création de la base de données "musique") :

```
import gadfly

connex = gadfly.gadfly()
connex.startup("musique", "E:/Python/essais/gadfly")
cur = connex.cursor()
requete = "create table compositeurs (comp varchar, a_naiss integer,\
      a_mort integer)"
cur.execute(requete)
requete = "create table oeuvres (comp varchar, titre varchar,\
      duree integer, interpr varchar)"
cur.execute(requete)

print "Entrée des enregistrements, table des compositeurs :"
while 1:
    nm = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nm == '':
        break
    an = raw_input("Année de naissance : ")
    am = raw_input("Année de mort : ")
    requete = "insert into compositeurs(comp, a_naiss, a_mort) values \
        ('%s', %s, %s)" % (nm, an, am)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from compositeurs")
print cur.pp()

print "Entrée des enregistrements, table des oeuvres musicales :"
while 1:
    nom = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nom == '':
        break
    tit = raw_input("Titre de l'oeuvre : ")
    dur = raw_input("durée (minutes) : ")
    int = raw_input("interprète principal : ")
    requete = "insert into oeuvres(comp, titre, duree, interpr) values \
        ('%s', '%s', %s, '%s')" % (nom, tit, dur, int)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from oeuvres")
print cur.pp()

connex.commit()
```

Commencez à remplir la table compositeurs avec les données qui suivent (... et

profitez de cette occasion pour faire la preuve des compétences que vous maîtrisez déjà, en écrivant un petit script pour vous faciliter l'entrée des informations : une boucle s'impose !)

comp	a_naiss	a_mort
Mozart	1756	1791
Beethoven	1770	1827
Handel	1685	1759
Schubert	1797	1828
Vivaldi	1678	1741
Monteverdi	1567	1643
Chopin	1810	1849
Bach	1685	1750

Dans la table œuvres, entrez les données suivantes :

comp	titre	duree	interpr
Vivaldi	Les quatre saisons	20	T. Pinnock
Mozart	Concerto piano N°12	25	M. Perahia
Brahms	Concerto violon N°2	40	A. Grumiaux
Beethoven	Sonate "au clair de lune"	14	W. Kempf
Beethoven	Sonate "pathétique"	17	W. Kempf
Schubert	Quintette "la truite"	39	SE of London
Haydn	La création	109	H. Von Karajan
Chopin	Concerto piano N°1	42	M.J. Pires
Bach	Toccatà & fugue	9	P. Burmester
Beethoven	Concerto piano N°4	33	M. Pollini
Mozart	Symphonie N°40	29	F. Bruggen
Mozart	Concerto piano N°22	35	S. Richter
Beethoven	Concerto piano N°3	37	S. Richter

Les champs a_naiss et a_mort contiennent respectivement l'année de naissance et l'année de la mort des compositeurs. La durée des œuvres est fournie en minutes. Vous pouvez évidemment ajouter autant d'enregistrements d'œuvres et de compositeurs que vous le voulez, mais ceux qui précèdent devraient suffire pour la suite de la démonstration.

Pour ce qui va suivre, nous supposons donc que vous avez effectivement encodé les données des deux tables décrites ci-dessus. (Si vous éprouvez des difficultés à écrire le script nécessaire, nous en donnons un exemple voir la solution de l'exercice précédent).

Le petit script ci-dessous est fourni à titre purement indicatif. Il s'agit d'un client SQL rudimentaire, qui vous permet de vous connecter à la base de données « musique » qui devrait à présent exister dans l'un de vos répertoires, d'y ouvrir un curseur et d'utiliser celui-ci pour effectuer des requêtes. Notez encore une fois que rien n'est transcrit sur le disque tant que la méthode commit() n'a pas été invoquée.

```
# Utilisation d'une petite base de données acceptant les requêtes SQL
import gadfly

baseDonn = gadfly.gadfly("musique","E:/Python/essais/gadfly")
cur = baseDonn.cursor()
while 1:
    print "Veuillez entrer votre requête SQL (ou <Enter> pour terminer) :"
    requete = raw_input()
    if requete == "":
        break
    try:
        cur.execute(requete)          # tentative d'exécution de la requête SQL
    except:
        print '*** Requête incorrecte ***'
    else:
        print cur.pp()                # affichage du résultat de la requête
    print

choix = raw_input("Confirmez-vous l'enregistrement (o/n) ? ")
if choix[0] == "o" or choix[0] == "O":
    baseDonn.commit()
else:
    baseDonn.close()
```

Cette application très simple n'est évidemment qu'un exemple. Il faudrait y ajouter la possibilité de choisir la base de données ainsi que le répertoire de travail. Pour éviter que le script ne se « plante » lorsque l'utilisateur encode une requête incorrecte, nous avons utilisé ici le traitement des *exceptions* déjà décrit à la page

La requête select

L'une des instructions les plus puissantes du langage SQL est l'instruction `select`, dont nous allons à présent explorer quelques fonctionnalités. Rappelons encore une fois que nous n'abordons ici qu'une très petite partie du sujet : la description détaillée de SQL peut occuper plusieurs livres.

Lancez donc le script ci-dessus, et analysez attentivement ce qui se passe lorsque vous proposez les requêtes suivantes :

```
select * from oeuvres
select * from oeuvres where comp = 'Mozart'
select comp, titre, duree from oeuvres order by comp
select titre, comp from oeuvres where comp='Beethoven' or comp='Mozart' order by comp
select count(*) from oeuvres
select sum(duree) from oeuvres
select avg(duree) from oeuvres
select sum(duree) from oeuvres where comp='Beethoven'
select * from oeuvres where duree >35 order by duree desc
```

Pour chacune de ces requêtes, tâchez d'exprimer le mieux possible ce qui se passe. Fondamentalement, vous activez sur la base de données des filtres de

sélection et des tris. Les requêtes suivantes sont plus élaborées, car elles concernent les deux tables à la fois.

```
select o.titre, c.nom, c.a_naiss from oeuvres o, compositeurs c where o.comp = c.comp
select comp from oeuvres intersect select comp from compositeurs
select comp from oeuvres except select comp from compositeurs
select comp from compositeurs except select comp from oeuvres
select distinct comp from oeuvres union select comp from compositeurs
```

Il ne nous est pas possible de développer davantage le langage de requêtes dans le cadre restreint de ces notes. Nous allons cependant examiner encore un exemple de réalisation Python faisant appel à un système de bases de données, mais en supposant cette fois qu'il s'agisse de dialoguer avec un système serveur indépendant (lequel pourrait être par exemple un gros serveur de bases de données d'entreprise, un serveur de documentation dans une école, etc.).

Ébauche d'un logiciel client pour MySQL

Pour terminer ce chapitre, nous allons vous proposer dans les pages qui suivent un exemple de réalisation concrète. Il ne s'agira pas d'un véritable logiciel (le sujet exigerait qu'on lui consacre un ouvrage spécifique), mais plutôt d'une ébauche d'analyse, destinée à vous montrer comment vous pouvez « penser comme un programmeur » lorsque vous abordez un problème complexe.

Les techniques que nous allons mettre en œuvre ici sont de simples suggestions, dans lesquelles nous essayerons d'utiliser au mieux les outils que vous avez découverts au cours de votre apprentissage dans les chapitres précédents, à savoir : les structures de données de haut niveau (listes et dictionnaires), et la programmation par objets. Il va de soi que les options retenues dans cet exercice restent largement critiquables : vous pouvez bien évidemment traiter les mêmes problèmes en utilisant des approches différentes.

Notre objectif concret est d'arriver à réaliser rapidement un client rudimentaire, capable de dialoguer avec un « vrai » serveur de bases de données tel que MySQL. Nous voudrions que notre client reste un petit utilitaire très généraliste : qu'il soit capable de mettre en place une petite base de données comportant plusieurs tables, qu'il puisse servir à produire des enregistrements pour chacune d'elles, qu'il permette de tester le résultat de requêtes SQL basiques.

Dans les lignes qui suivent, nous supposerons que vous avez déjà accès à un serveur MySQL, sur lequel une base de données « discotheque » aura été créée pour l'utilisateur « jules », lequel s'identifie à l'aide du mot de passe « abcde ». Ce serveur peut être situé sur une machine distante accessible via un réseau, ou localement sur votre ordinateur personnel.

L'installation et la configuration d'un serveur MySQL sortent du cadre de cet ouvrage, mais ce n'est pas une tâche bien compliquée. C'est même fort simple si vous travaillez sous Linux, installé depuis une distribution « classique » telle que Debian, RedHat, SuSE ou Mandrake. Il vous suffit d'installer les paquetages MySQL-server et Python-MySQL, de démarrer le service MySQL, puis d'entrer les commandes :

```
mysqladmin -u root password xxxx
```

Cette première commande définit le mot de passe de l'administrateur principal de MySQL. Elle doit être exécutée par l'administrateur du système Linux (*root*), avec un mot de passe de votre choix. On se connecte ensuite au serveur sous le compte administrateur ainsi défini (le mot de passe sera demandé) :

```
mysql -u root mysql -p
grant all privileges on *.* to jules@localhost identified by 'abcde';
grant all privileges on *.* to jules@"%" identified by 'abcde';
\q
```

Ces commandes définissent un nouvel utilisateur « jules » pour le système MySQL, et cet utilisateur devra se connecter le mot de passe « abcde » (Les deux lignes autorisent respectivement l'accès local et l'accès via réseau).

Le nom d'utilisateur est quelconque : il ne doit pas nécessairement correspondre à un utilisateur système.

L'utilisateur « jules » peut à présent se connecter et créer des bases de données :

```
mysql -u jules -p
create database discotheque;
\q
```

etc.

À ce stade, le serveur MySQL est prêt à dialoguer avec le client Python décrit dans ces pages.

Décrire la base de données dans un dictionnaire d'application

Une application dialoguant avec une base de données est presque toujours une application complexe. Elle comporte donc de nombreuses lignes de code, qu'il s'agit de structurer le mieux possible en les regroupant dans des classes (ou au moins des fonctions) bien encapsulées.

En de nombreux endroits du code, souvent fort éloignés les uns des autres, des blocs d'instructions doivent prendre en compte la structure de la base de données, c'est-à-dire son découpage en un certain nombre de tables et de

champs, ainsi que les relations qui établissent une hiérarchie dans les enregistrements.

Or, l'expérience montre que la structure d'une base de données est rarement définitive. Au cours d'un développement, on réalise souvent qu'il est nécessaire de lui ajouter ou de lui retirer des champs, parfois même de remplacer une table mal conçue par deux autres, etc. Il n'est donc pas prudent de programmer des portions de code trop spécifiques d'une structure particulière, « en dur ».

Au contraire, il est hautement recommandable *de décrire plutôt la structure complète de la base de données en un seul endroit du programme*, et d'utiliser ensuite cette description comme référence pour la génération semi-automatique des instructions particulières concernant telle table ou tel champ. On évite ainsi, dans une large mesure, le cauchemar de devoir traquer et modifier un grand nombre d'instructions un peu partout dans le code, chaque fois que la structure de la base de données change un tant soit peu. Au lieu de cela, il suffit de changer seulement la description de référence, et la plus grosse partie du code reste correcte sans nécessiter de modification.

Remarque : nous tenons là une idée maîtresse pour réaliser des applications robustes : un logiciel destiné au traitement de données devrait toujours être construit sur la base d'un dictionnaire d'application.

Ce que nous entendons ici par « dictionnaire d'application » ne doit pas nécessairement revêtir la forme d'un dictionnaire Python. N'importe quelle structure de données peut convenir, l'essentiel étant de se construire une *référence centrale décrivant les données* que l'on se propose de manipuler, avec peut-être aussi un certain nombre d'informations concernant leur mise en forme.

Du fait de leur capacité à rassembler en une même entité des données de n'importe quel type, les listes, tuples et dictionnaires de Python conviennent parfaitement pour ce travail. Dans l'exemple des pages suivantes, nous avons utilisé nous-mêmes un dictionnaire, dont les valeurs sont des listes de tuples, mais vous pourriez tout aussi bien opter pour une organisation différente des mêmes informations.

Tout cela étant bien établi, il nous reste encore à régler une question d'importance : où allons-nous installer concrètement ce dictionnaire d'application ?

Ses informations devront pouvoir être consultées depuis n'importe quel endroit du programme. Il semble donc obligatoire de l'installer dans une variable globale, de même d'ailleurs que d'autres données nécessaires au fonctionnement de l'ensemble de notre logiciel. Or vous savez que l'utilisation de variables globales n'est pas recommandée : elle comporte des risques, qui augmentent avec la taille du programme. De toute façon, les variables dites globales ne sont en fait globales qu'à l'intérieur d'un même module. Si nous souhaitons organiser notre logiciel comme un ensemble de modules (ce qui

constitue par ailleurs une excellente pratique), nous n'aurons accès à nos variables globales que dans un seul d'entre eux.

Pour résoudre ce petit problème, il existe cependant une solution simple et élégante : *regrouper dans une classe particulière toutes les variables qui nécessitent un statut global pour l'ensemble de l'application*. Ainsi encapsulées dans l'espace de noms d'une classe, ces variables peuvent être utilisées sans problème dans n'importe quel module : il suffit en effet que celui-ci importe la classe en question. De plus, l'utilisation de cette technique entraîne une conséquence intéressante : le caractère « global » des variables définies de cette manière apparaît très clairement dans leur nom qualifié, puisque ce nom commence par celui de la classe contenante.

Si vous choisissez, par exemple, un nom explicite tel que `glob` pour la classe destinée à accueillir vos variables « globales », vous vous assurez de devoir faire référence à ces variables partout dans votre code avec des noms tout aussi explicites tels que `Glob.ceci` , `Glob.cela` , etc^[6].

C'est cette technique que vous allez découvrir à présent dans les premières lignes de notre script. Nous y définissons effectivement une classe `glob()`, qui n'est donc rien d'autre qu'un simple conteneur. Aucun objet ne sera instancié à partir de cette classe, laquelle ne comporte d'ailleurs aucune méthode. Nos variables « globales » y sont définies comme de simples variables de classe, et nous pourrons donc y faire référence dans tout le reste du programme en tant qu'attributs de `Glob`. Le nom de la base de données, par exemple, pourra être retrouvé partout dans la variable `Glob.dbName` ; le nom ou l'adresse IP du serveur dans la variable `Glob.host`, etc. :

```
class Glob:
    """Espace de noms pour les variables et fonctions <pseudo-globales>"""

    dbName = "discotheque"      # nom de la base de données
    user = "jules"              # propriétaire ou utilisateur
    passwd = "abcde"            # mot de passe d'accès
    host = "192.168.0.235"      # nom ou adresse IP du serveur

    # Structure de la base de données. Dictionnaire des tables & champs :
    dicoT = {"compositeurs": [('id_comp', "k", "clé primaire"),
                              ('nom', 25, "nom"),
                              ('prenom', 25, "prénom"),
                              ('a_naiss', "i", "année de naissance"),
                              ('a_mort', "i", "année de mort")],
            "oeuvres": [('id_oeuv', "k", "clé primaire"),
                        ('id_comp', "i", "clé compositeur"),
                        ('titre', 50, "titre de l'oeuvre"),
                        ('duree', "i", "durée (en minutes)",
                        ('interpr', 30, "interprète principal")}]}
```

Le dictionnaire d'application décrivant la structure de la base de données est contenu dans la variable `glob.dicoT`.

Il s'agit d'un dictionnaire Python, dont les clés sont les noms des tables. Quant aux valeurs, chacune d'elles est une liste contenant la description de tous les champs de la table, sous la forme d'autant de tuples.

Chaque tuple décrit donc un champ particulier de la table. Pour ne pas encombrer notre exercice, nous avons limité cette description à trois informations seulement : le nom du champ, son type et un bref commentaire. Dans une véritable application, il serait judicieux d'ajouter encore d'autres informations ici, concernant par exemple des valeurs limites éventuelles pour les données de ce champ, le formatage à leur appliquer lorsqu'il s'agit de les afficher à l'écran ou de les imprimer, le texte qu'il faut placer en haut de colonne lorsque l'on veut les présenter dans un tableau, etc.

Il peut vous paraître assez fastidieux de décrire ainsi très en détail la structure de vos données, alors que vous voudriez probablement commencer tout de suite une réflexion sur les divers algorithmes à mettre en œuvre afin de les traiter. Sachez cependant que si elle est bien faite, une telle description structurée vous fera certainement gagner beaucoup de temps par la suite, parce qu'elle vous permettra d'automatiser pas mal de choses. En outre, vous devez vous convaincre que cette tâche un peu ingrate vous prépare à bien structurer aussi le reste de votre travail : organisation des formulaires, tests à effectuer, etc.

Définir une classe d'objets-interfaces

La classe `Glob()` décrite à la rubrique précédente sera donc installée en début de script, ou bien dans un module séparé importé en début de script. Pour la suite de l'exposé, nous supposons que c'est cette dernière formule qui est retenue : nous avons sauvegardé la classe `Glob()` dans un module nommé `dict_app.py`, d'où nous pouvons à présent l'importer dans le script suivant.

Ce nouveau script définit une classe d'objets-interfaces. Nous voulons en effet essayer de mettre à profit ce que nous avons appris dans les chapitres précédents, et donc privilégier la programmation par objets, afin de créer des portions de code bien encapsulées et largement réutilisables.

Les objets-interfaces que nous voulons construire seront similaires aux objets-fichiers que nous avons abondamment utilisés pour la gestion des fichiers au chapitre 9. Vous vous rappelez par exemple que nous ouvrons un fichier en créant un objet-fichier, à l'aide de la fonction-fabrique `open()`. D'une manière similaire, nous ouvrirons la communication avec la base de données en commençant par créer un objet-interface à l'aide de la classe `GestionBD()`, ce qui établira la connexion. Pour lire ou écrire dans un fichier ouvert, nous utilisons diverses méthodes de l'objet-fichier. D'une manière analogue, nous effectuerons nos opérations sur la base de données par l'intermédiaire des diverses méthodes de l'objet-interface.

```
1 import MySQLdb, sys
2 from dict_app import *
```

```

3
4 class GestionBD:
5     """Mise en place et interfaçage d'une base de données MySQL"""
6     def __init__(self, dbName, user, passwd, host, port =3306):
7         "Établissement de la connexion - Création du curseur"
8         try:
9             self.baseDonn = MySQLdb.connect(db =dbName,
10                user =user, passwd =passwd, host =host, port =port)
11         except Exception, err:
12             print 'La connexion avec la base de données a échoué :\n'\
13                 'Erreur détectée :\n%s' % err
14             self.echec =1
15         else:
16             self.cursor = self.baseDonn.cursor() # création du curseur
17             self.echec =0
18
19     def creerTables(self, dicTables):
20         "Création des tables décrites dans le dictionnaire <dicTables>."
21         for table in dicTables: # parcours des clés du dict.
22             req = "CREATE TABLE %s (" % table
23             pk =''
24             for descr in dicTables[table]:
25                 nomChamp = descr[0] # libellé du champ à créer
26                 tch = descr[1] # type de champ à créer
27                 if tch =='i':
28                     typeChamp ='INTEGER'
29                 elif tch =='k':
30                     # champ 'clé primaire' (incrémenté automatiquement)
31                     typeChamp ='INTEGER AUTO_INCREMENT'
32                     pk = nomChamp
33                 else:
34                     typeChamp ='VARCHAR(%s)' % tch
35             req = req + "%s %s, " % (nomChamp, typeChamp)
36             if pk == '':
37                 req = req[:-2] + ")"
38             else:
39                 req = req + "CONSTRAINT %s_pk PRIMARY KEY(%s)" % (pk, pk)
40             self.executerReq(req)
41
42     def supprimerTables(self, dicTables):
43         "Suppression de toutes les tables décrites dans <dicTables>"
44         for table in dicTables.keys():
45             req ="DROP TABLE %s" % table
46             self.executerReq(req)
47         self.commit() # transfert -> disque
48
49     def executerReq(self, req):
50         "Exécution de la requête <req>, avec détection d'erreur éventuelle"
51         try:
52             self.cursor.execute(req)
53         except Exception, err:
54             # afficher la requête et le message d'erreur système :
55             print "Requête SQL incorrecte :\n%s\nErreur détectée :\n%s"\
56                 % (req, err)
57             return 0
58         else:
59             return 1
60
61     def resultatReq(self):
62         "renvoie le résultat de la requête précédente (un tuple de tuples)"
63         return self.cursor.fetchall()
64
65     def commit(self):

```

```
66         if self.baseDonn:
67             self.baseDonn.commit()           # transfert curseur -> disque
68
69     def close(self):
70         if self.baseDonn:
71             self.baseDonn.close()
72     '-----'
```

Commentaires

- Lignes 1-2 : Outre notre propre module `dict_app` qui contient les variables « globales », nous importons le module `sys` qui contient quelques fonctions système, et le module `MySQLdb` qui contient tout ce qui est nécessaire pour communiquer avec MySQL. Rappelons que ce module ne fait pas partie de la distribution standard de Python, et qu'il doit donc être installé séparément.
- Ligne 5 : Lors de la création des objets-interfaces, nous devons fournir les paramètres de la connexion : nom de la base de données, nom de son utilisateur, nom ou adresse IP de la machine où est situé le serveur. Le numéro du port de communication est habituellement celui que nous avons prévu par défaut. Toutes ces informations sont supposées être en votre possession.
- Lignes 8 à 17 : Il est hautement recommandable de placer le code servant à établir la connexion à l'intérieur d'un gestionnaire d'exceptions `try-except-else`, car nous ne pouvons pas présumer que le serveur sera nécessairement accessible. Remarquons au passage que la méthode `__init__()` ne peut pas renvoyer de valeur (à l'aide de l'instruction `return`), du fait qu'elle est invoquée automatiquement par Python lors de l'instanciation d'un objet. En effet : ce qui est renvoyé dans ce cas au programme appelant est l'objet nouvellement construit. Nous ne pouvons donc pas signaler la réussite ou l'échec de la connexion au programme appelant à l'aide d'une valeur de retour. Une solution simple à ce petit problème consiste à mémoriser le résultat de la tentative de connexion dans un attribut d'instance (variable `self.echec`), que le programme appelant peut ensuite tester quand bon lui semble.
- Lignes 19 à 40 : Cette méthode automatise la création de toutes les tables de la base de données, en tirant profit de la description du dictionnaire d'application, lequel doit lui être transmis en argument. Une telle automatisation sera évidemment d'autant plus appréciable, que la structure de la base de données sera plus complexe (Imaginez par exemple une base de données contenant 35 tables !). Afin de ne pas alourdir la démonstration, nous avons restreint les capacités de cette méthode à la création de champs des types *integer* et *varchar*. Libre à vous d'ajouter les instructions nécessaires pour créer des champs d'autres types. Si vous détaillez le code, vous constaterez qu'il consiste simplement à construire une requête SQL pour chaque table, morceau par morceau, dans la chaîne de caractères `req`. Celle-ci est ensuite transmise à la méthode `executerReq()` pour exécution. Si vous souhaitez visualiser la requête ainsi construite, vous pouvez évidemment ajouter une instruction `print req` juste après la ligne 40. Vous pouvez également ajouter à cette méthode la capacité de mettre en place *les contraintes d'intégrité référentielle*, sur la base d'un complément au dictionnaire d'application qui décrirait ces contraintes. Nous ne développons

pas cette question ici, mais cela ne devrait pas vous poser de problème si vous savez de quoi il retourne.

- Lignes 42 à 47 : Beaucoup plus simple que la précédente, cette méthode utilise le même principe pour supprimer toutes les tables décrites dans le dictionnaire d'application.
- Lignes 49 à 59 : Cette méthode transmet simplement la requête à l'objet curseur. Son utilité est de simplifier l'accès à celui-ci et de produire un message d'erreur si nécessaire.
- Lignes 61 à 71 : Ces méthodes ne sont que de simples relais vers les objets produits par le module *MySQLdb* : l'objet-connecteur produit par la fonction-fabrique `MySQLdb.connect()`, et l'objet curseur correspondant. Elles permettent de simplifier légèrement le code du programme appelant.

Construire un générateur de formulaires

Nous avons ajouté cette classe à notre exercice pour vous expliquer comment vous pouvez utiliser le même dictionnaire d'application afin d'élaborer du code généraliste. L'idée développée ici est de réaliser une classe d'objets-formulaires capables de prendre en charge l'encodage des enregistrements de n'importe quelle table, en construisant automatiquement les instructions d'entrée adéquates grâce aux informations tirées du dictionnaire d'application.

Dans une application véritable, ce formulaire trop simpliste devrait certainement être fortement remanié, et il prendrait vraisemblablement la forme d'une fenêtre spécialisée, dans laquelle les champs d'entrée et leurs libellés pourraient encore une fois être générés de manière automatique. Nous ne prétendons donc pas qu'il constitue un bon exemple, mais nous voulons simplement vous montrer comment vous pouvez automatiser sa construction dans une large mesure. Tâchez de réaliser vos propres formulaires en vous servant de principes semblables.

```

1 class Enregistreur:
2     """classe pour gérer l'entrée d'enregistrements divers"""
3     def __init__(self, bd, table):
4         self.bd =bd
5         self.table =table
6         self.descriptif =Glob.dicoT[table] # descriptif des champs
7
8     def entrer(self):
9         "procédure d'entrée d'un enregistrement entier"
10        champs = "(" # ébauche de chaîne pour les noms de champs
11        valeurs = "(" # ébauche de chaîne pour les valeurs
12        # Demander successivement une valeur pour chaque champ :
13        for cha, type, nom in self.descriptif:
14            if type == "k": # on ne demandera pas le n° d'enregistrement
15                continue # à l'utilisateur (numérotation auto.)
16            champs = champs + cha + ","
17            val = raw_input("Entrez le champ %s : " % nom)
18            if type == "i":
19                valeurs = valeurs + val + ","

```



```

20         else:
21             valeurs = valeurs + "'%s'," % (val)
22
23         champs = champs[:-1] + ")" # supprimer la dernière virgule,
24         valeurs = valeurs[:-1] + ")" # ajouter une parenthèse
25         req = "INSERT INTO %s %s VALUES %s" % (self.table, champs, valeurs)
26         self.bd.executerReq(req)
27
28         ch = raw_input("Continuer (O/N) ? ")
29         if ch.upper() == "O":
30             return 0
31         else:
32             return 1

```

Commentaires

- Lignes 1 à 6 : Au moment de leur instanciation, les objets de cette classe reçoivent la référence de l'une des tables du dictionnaire. C'est ce qui leur donne accès au descriptif des champs.
- Ligne 8 : Cette méthode `entrer()` génère le formulaire proprement dit. Elle prend en charge l'entrée des enregistrements dans la table, en s'adaptant à leur structure propre grâce au descriptif trouvé dans le dictionnaire. Sa fonctionnalité concrète consiste encore une fois à construire morceau par morceau une chaîne de caractères qui deviendra une requête SQL, comme dans la méthode `creerTables()` de la classe `GestionBD()` décrite à la rubrique précédente. Vous pourriez bien entendu ajouter à la présente classe encore d'autres méthodes, pour gérer par exemple la suppression et/ou la modification d'enregistrements.
- Lignes 12 à 21 : L'attribut d'instance `self.descriptif` contient une liste de tuples, et chacun de ceux-ci est fait de trois éléments, à savoir le nom d'un champ, le type de données qu'il est censé recevoir, et sa description « en clair ». La boucle `for` de la ligne 13 parcourt cette liste et affiche pour chaque champ un message d'invite construit sur la base de la description qui accompagne ce champ. Lorsque l'utilisateur a entré la valeur demandée, celle-ci et formatée dans une chaîne en construction. Le formatage s'adapte aux conventions du langage SQL, conformément au type requis pour le champ.
- Lignes 23 à 26 : Lorsque tous les champs ont été parcourus, la requête proprement dite est assemblée et exécutée. Si vous souhaitez visualiser cette requête, vous pouvez bien évidemment ajouter une instruction `print req` juste après la ligne 25.

Le corps de l'application

Il ne nous paraît pas utile de développer davantage encore cet exercice dans le cadre d'un manuel d'initiation. Si le sujet vous intéresse, vous devriez maintenant en savoir assez pour commencer déjà quelques expériences personnelles. Veuillez alors consulter les bons ouvrages de référence, comme par exemple *Python : How to program* de Deitel & coll., ou encore les sites web consacrés aux extensions de Python.

Le script qui suit est celui d'une petite application destinée à tester les classes

décrites dans les pages qui précèdent. Libre à vous de la perfectionner, ou alors d'en écrire une autre tout à fait différente !

```

1 ##### Programme principal : #####
2
3 # Création de l'objet-interface avec la base de données :
4 bd = GestionBD(Glob.dbName, Glob.user, Glob.passwd, Glob.host)
5 if bd.echec:
6     sys.exit()
7
8 while 1:
9     print "\nQue voulez-vous faire :\n"
10         "1) Créer les tables de la base de données\n"
11         "2) Supprimer les tables de la base de données ?\n"
12         "3) Entrer des compositeurs\n"
13         "4) Entrer des oeuvres\n"
14         "5) Lister les compositeurs\n"
15         "6) Lister les oeuvres\n"
16         "7) Exécuter une requête SQL quelconque\n"
17         "9) terminer ?                               Votre choix :",
18 ch = int(raw_input())
19 if ch ==1:
20     # création de toutes les tables décrites dans le dictionnaire :
21     bd.creerTables(Glob.dicoT)
22 elif ch ==2:
23     # suppression de toutes les tables décrites dans le dic. :
24     bd.supprimerTables(Glob.dicoT)
25 elif ch ==3 or ch ==4:
26     # création d'un <enregistrement> de compositeurs ou d'oeuvres :
27     table = {3:'compositeurs', 4:'oeuvres'}[ch]
28     enreg =Enregistreur(bd, table)
29     while 1:
30         if enreg.entrer():
31             break
32 elif ch ==5 or ch ==6:
33     # listage de tous les compositeurs, ou toutes les oeuvres :
34     table = {5:'compositeurs', 6:'oeuvres'}[ch]
35     if bd.executerReq("SELECT * FROM %s" % table):
36         # analyser le résultat de la requête ci-dessus :
37         records = bd.resultatReq() # ce sera un tuple de tuples
38         for rec in records: # => chaque enregistrement
39             for item in rec: # => chaque champ dans l'enreg.
40                 print item,
41             print
42 elif ch ==7:
43     req =raw_input("Entrez la requête SQL : ")
44     if bd.executerReq(req):
45         print bd.resultatReq() # ce sera un tuple de tuples
46 else:
47     bd.commit()
48     bd.close()
49     break

```

Commentaires

On supposera bien évidemment que les classes décrites plus haut soient présentes dans le même script, ou qu'elles aient été importées.

- Lignes 3 à 6 : L'objet-interface est créé ici. Si la création échoue, l'attribut d'instance `bd.echec` contient la valeur 1. Le test des lignes 5 et 6 permet alors

d'arrêter l'application immédiatement (la fonction `exit()` du module `sys` sert spécifiquement à cela).

- Ligne 8 : Le reste de l'application consiste à proposer sans cesse le même menu, jusqu'à ce que l'utilisateur choisisse l'option n° 9.
- Lignes 27 et 28 : La classe `Enregistreur()` accepte de gérer les enregistrements de n'importe quelle table. Afin de déterminer laquelle doit être utilisée lors de l'instanciation, on utilise un petit dictionnaire qui indique quel nom retenir, en fonction du choix opéré par l'utilisateur (option n° 3 ou n° 4).
- Lignes 29 à 31 : La méthode `entrer()` de l'objet-enregistreur renvoie une valeur 0 ou 1 suivant que l'utilisateur ait choisi de continuer à entrer des enregistrements, ou bien d'arrêter. Le test de cette valeur permet d'interrompre la boucle de répétition en conséquence.
- Lignes 35 et 44 : La méthode `executerReq()` renvoie une valeur 0 ou 1 suivant que la requête ait été acceptée ou non par le serveur. On peut donc tester cette valeur pour décider si le résultat doit être affiché ou non.

Exercices

1. Modifiez le script décrit dans ces pages de manière à ajouter une table supplémentaire à la base de données. Ce pourrait être par exemple une table « orchestres », dont chaque enregistrement contiendrait le nom de l'orchestre, le nom de son chef, et le nombre total d'instruments.
2. Ajoutez d'autres types de champ à l'une des tables (par exemple un champ de type `float` (réel) ou de type `date`), et modifiez le script en conséquence.

Solution

1. Réfléchissez !
2. Réfléchissez !

Notes

1. Il vous faudra certainement entrer quelques informations pour obtenir l'accès : adresse du serveur sur le réseau, nom de la base de données, nom d'utilisateur, mot de passe, ...
2. Quelques variantes subsistent entre différentes implémentations du SQL, pour des requêtes très spécifiques, mais la base reste cependant la même.
3. Gadfly se révèle relativement efficace pour la gestion de bases de données de taille moyenne, en mode mono-utilisateur. Pour gérer de grosses bases de données en mode multi-utilisateur, il faut faire appel à des SGDBR plus ambitieux tels que *PostgreSQL*, pour lesquels des modules clients Python existent aussi (*Pygresql*, par ex.).
4. Le module Gadfly est disponible gratuitement sur l'internet. Voir <http://sourceforge.net/projects/gadfly>
5. Veuillez noter qu'en SQL, les chaînes de caractères doivent être délimitées par

des apostrophes. Si vous souhaitez que la chaîne contienne elle-même une ou plusieurs apostrophes, il vous suffit de doubler celles-ci.

6. Vous pourriez également placer vos variables « globales » dans un module nommé *Glob.py*, puis importer celui-ci. Utiliser un module ou une classe comme espace de noms pour stocker des variables sont donc des techniques assez similaires. L'utilisation d'une classe est peut-être un peu plus souple et plus lisible, puisque la classe peut accompagner le reste du script, alors qu'un module est nécessairement un fichier distinct.

L'interface CGI

Description

L'interface CGI (pour *Common Gateway Interface*) est un composant de la plupart des logiciels serveurs de pages web. Il s'agit d'une passerelle qui leur permet de communiquer avec d'autres logiciels tournant sur le même ordinateur. Avec CGI, vous pouvez écrire des scripts dans différents langages (*Perl, C, Tcl, PHP, Python ...*).

Plutôt que de limiter le web à des documents écrits à l'avance, CGI permet de générer des pages web sur le champ, en fonction des données que fournit l'internaute par l'intermédiaire de son logiciel de navigation. Vous pouvez utiliser les scripts CGI pour créer une large palette d'applications : des services d'inscription en ligne, des outils de recherche dans des bases de données, des instruments de sondage d'opinions, des jeux, etc.

L'apprentissage de la programmation CGI peut faire l'objet de manuels entiers. Dans cet ouvrage d'initiation, nous vous expliquerons seulement quelques principes de base, afin de vous faire comprendre, par comparaison, l'énorme avantage que présentent les modules serveurs d'applications spécialisés tels que *Karrigell, CherryPy* ou *Zope*, pour le programmeur désireux de développer un site web interactif.

Installation

Pour plus de détails voir : **Apache/CGI**.

Par défaut, lire un fichier .py en HTTP renvoie son contenu. Pour que le serveur compile et exécute de le code source, il faut que la configuration suivante figure dans sa configuration, ou bien que les scripts soient placés dans un répertoire contenant un fichier nommé *.htaccess*, avec les lignes^[1] :

```
AddHandler cgi-script .py
Options +ExecCGI
```



Attention !

Sur les serveurs Unix les fichiers ne sont pas exécutables par défaut, il faut donc le préciser pour chacun avec la commande : *chmod +x *.py*.

Exemples

Le module `cgitb` sert aux éventuels débogages :

```
#!C:\Program Files (x86)\Python\python.exe
# -*- coding: UTF-8 -*-
print "Content-type: text/html; charset=utf-8\n\n"
print "<html><head><title>Répertoire local</title></head><body>"
import cgitb
cgitb.enable()
import os
print "Le fichier CGI se trouve dans :"
print os.path.dirname(__file__)
print "</body></html>"
```

L'utilisation d'un formulaire nécessite un `import cgi`^[2]. Pour une base de données MySQL c'est `import MySQLdb`^[3]. Soit le fichier suivant `CGI_MySQL.py` utilisant les deux :

```
#!C:\Program Files (x86)\Python\python.exe
# -*- coding: UTF-8 -*-
print "Content-type: text/html; charset=utf-8\n\n"
print "<html><head><title>CGI BDD</title></head><body>"
print "<h1>Extraction MySQL</h1>"
print "<ul>"
import cgitb
cgitb.enable()
import cgi, MySQLdb
formulaire = cgi.FieldStorage()
if formulaire.getvalue('nom') == None:
    print "<h2>Rechercher un nom</h2>"
    print '''
    <form action="CGI_MySQL.py" method="post">
    <input type="text" name="nom" />
    <input type="submit"></form>
    '''
else:
    print "<h2>Résultat</h2>"
    print "Liste des correspondances pour " + formulaire.getvalue('nom') + " :"
    connection = MySQLdb.connect(user='login1', passwd='mdp1', db='base1')
    cursor = connection.cursor()
    cursor.execute("SELECT page_title FROM page WHERE nom ='"+formulaire.getvalue('nom')+")")
    for row in cursor.fetchall():
        print "<li>%s</li>" % row[0]
    connection.close()
print "</ul>"
print "</body></html>"
```

Une interaction CGI rudimentaire

Veillez donc encoder le document HTML ci-dessous à l'aide d'un éditeur quelconque :

```
1 <HTML>
2 <HEAD><TITLE>Exercice avec Python</TITLE></HEAD>
3 <BODY>
4
```

```
5 <DIV ALIGN="center">
6 <IMG SRC="penguin.gif">
7 <H2>Page Web interactive</H2>
8 <P>Cette page est associée à un script Python</P>
9
10 <FORM ACTION="http://localhost/cgi-bin/input_query.py" METHOD="post">
11 <INPUT TYPE="submit" NAME="send" VALUE="Exécuter le script">
12 </FORM>
13
14 </DIV></BODY></HTML>
```

Vous savez certainement déjà que les balises initiales `<HTML>`, `<HEAD>`, `<TITLE>`, `<BODY>`, ainsi que les balises finales correspondantes, sont communes à tous les documents HTML. Nous ne détaillerons donc pas leur rôle ici.

La balise `<DIV>` utilisée à la ligne 5 sert habituellement à diviser un document HTML en sections distinctes. Nous l'utilisons ici pour définir une section dans laquelle tous les éléments seront centrés (horizontalement) sur la page.

À la ligne 6, nous insérons une petite image.

La ligne 7 définit une ligne de texte comme étant un titre de 2e importance.

La ligne 8 est un paragraphe ordinaire.

Les lignes 10 à 12 contiennent le code important (pour ce qui nous occupe ici). Les balises `<FORM>` et `</FORM>` définissent en effet un formulaire, c'est-à-dire une portion de page Web susceptible de contenir divers *widgets* à l'aide desquels l'internaute pourra exercer une certaine activité : champs d'entrée, boutons, cases à cocher, boutons radio, etc.

La balise `FORM` doit contenir deux indications essentielles : l'*action* à accomplir lorsque le formulaire sera expédié (il s'agit en fait de fournir ici l'adresse URL du logiciel à invoquer pour traiter les données transmises), et la *méthode* à utiliser pour transmettre l'information (en ce qui nous concerne, ce sera toujours la méthode `post`).

Dans notre exemple, le logiciel que nous voulons invoquer est un script Python nommé `input_query.py` qui est situé dans un répertoire particulier du serveur d'intranet. Sur de nombreux serveurs, ce répertoire s'appelle souvent *cgi-bin*, par pure convention. Nous supposons ici que l'administrateur de votre intranet scolaire vous autorise à installer vos scripts Python dans le même répertoire que celui où vous placez vos pages web personnelles.

Vous devrez donc modifier la ligne 10 de notre exemple, en remplaçant le nom de domaine dans l'adresse `http://localhost/cgi-bin/input_query.py`.

La ligne 11 contient la balise qui définit un *widget* de type « bouton d'envoi » (balise `<INPUT TYPE="submit">`). Le texte qui doit apparaître sur le bouton est précisé par l'attribut `VALUE="texte"`. L'indication `NAME` est facultative dans le cas présent. Elle mentionne le nom du widget lui-même (au cas où le logiciel destinataire en aurait besoin).

Lorsque vous aurez terminé l'encodage de ce document, sauvegardez-le dans le répertoire que l'on vous a attribué spécifiquement pour y placer vos pages, sous un nom quelconque, mais de préférence avec l'extension `.html` ou `.htm` (par exemple : `essai.html`).

Le script Python `input_query.py` est détaillé ci-dessous. Comme déjà signalé plus haut, vous pouvez installer ce script dans le même répertoire que votre document HTML initial :

```
1 #! /usr/bin/python
2 # -*- coding: utf-8 -*-
3 # Affichage d'un formulaire HTML simplifié :
4 print "Content-Type: text/html\n"
5 print """
6 <H3><FONT COLOR="Royal blue">
7 Page web produite par un script Python
8 </FONT></H3>
9
10 <FORM ACTION="print_result.py" METHOD="post">
11 <P>Veuillez entrer votre nom dans le champ ci-dessous, s.v.p. :</P>
12 <P><INPUT NAME="visiteur" SIZE=20 MAXLENGTH=20 TYPE="text"></P>
13 <P>Veuillez également me fournir une phrase quelconque :</P>
14 <TEXTAREA NAME="phrase" ROWS=2 COLS=50>Mississippi</TEXTAREA>
15 <P>J'utiliserai cette phrase pour établir un histogramme.</P>
16 <INPUT TYPE="submit" NAME="send" VALUE="Action">
17 </FORM>
18 """
```

Ce script ne fait rien d'autre que d'afficher une nouvelle page web, laquelle contient encore une fois un formulaire, mais celui-ci nettement plus élaboré que le précédent.

La première ligne est absolument nécessaire : elle indique à l'interface CGI qu'il faut lancer l'interpréteur Python pour pouvoir exécuter le script. La seconde ligne spécifie l'encodage du code source.

La ligne 4 est indispensable. Elle permet à l'interpréteur Python d'initialiser un véritable document HTML qui sera transmis au serveur web. Celui-ci pourra à son tour le réexpédier au logiciel navigateur de l'internaute, et celui-ci le verra donc s'afficher dans la fenêtre de navigation.

La suite est du pur code HTML, traité par Python comme une simple chaîne de caractères que l'on affiche à l'aide de l'instruction `print`. Pour pouvoir y insérer tout ce que nous voulons, y compris les sauts à la ligne, les apostrophes, les guillemets, etc., nous délimitons cette chaîne de caractères à l'aide de « triples guillemets » (Rappelons également ici que les sauts à la ligne sont complètement ignorés en HTML : nous pouvons donc en utiliser autant que nous voulons pour « aérer » notre code et le rendre plus lisible).

Un formulaire HTML pour l'acquisition des données

Analysons à présent le code HTML lui-même. Nous y trouvons essentiellement un nouveau formulaire, qui comporte plusieurs paragraphes, parmi lesquels on

peut reconnaître quelques *widgets*. La ligne 10 indique le nom du script CGI auquel les données du formulaire seront transmises : il s'agira bien évidemment d'un autre script Python.

À la ligne 12, on trouve la définition d'un *widget* de type « champ d'entrée » (Balise `INPUT`, avec `TYPE="text"`). L'utilisateur est invité à y encoder son nom. Le paramètre `MAXLENGTH` définit une longueur maximale pour la chaîne de caractères qui sera entrée ici (20 caractères, en l'occurrence). Le paramètre `SIZE` définit la taille du champ tel qu'il doit apparaître à l'écran, et le paramètre `NAME` est le nom que nous choisissons pour la variable destinée à mémoriser la chaîne de caractères attendue.

Un second champ d'entrée un peu différent est défini à la ligne 14 (balise `TEXTAREA`). Il s'agit d'un réceptacle plus vaste, destiné à accueillir des textes de plusieurs lignes. (Ce champ est automatiquement pourvu d'ascenseurs si le texte à insérer se révèle trop volumineux). Ses paramètres `ROWS` et `COLS` sont assez explicites. Entre les balises initiale et finale, on peut insérer un texte par défaut (Mississippi dans notre exemple).

Comme dans l'exemple précédent, la ligne 16 contient la définition du bouton qu'il faudra actionner pour transmettre les données au script CGI destinataire, lequel est décrit ci-après.

Un script CGI pour le traitement des données

Le mécanisme utilisé à l'intérieur d'un script CGI pour réceptionner les données transmises par un formulaire HTML est fort simple, comme vous pouvez l'analyser dans l'exemple ci-dessous :

```

1  #!/usr/bin/python
2  # Traitement des données transmises par un formulaire HTML
3
4  import cgi                # Module d'interface avec le serveur web
5  form = cgi.FieldStorage()  # Réception de la requête utilisateur :
6                               # il s'agit d'une sorte de dictionnaire
7  if form.has_key("phrase"): # La clé n'existera pas si le champ
8      text = form["phrase"].value # correspondant est resté vide
9  else:
10     text = "*** le champ phrase était vide ! ***"
11
12 if form.has_key("visiteur"): # La clé n'existera pas si le champ
13     nomv = form["visiteur"].value # correspondant est resté vide
14 else:
15     nomv = "mais vous ne m'avez pas indiqué votre nom"
16
17 print "Content-Type: text/html\n"
18 print ""
19 <H3>Merci, %s !</H3>
20 <H4>La phrase que vous m'avez fournie était : </H4>
21 <H3><FONT Color="red"> %s </FONT></H3>"" % (nomv, text)
22
23 histogr = {}
24 for c in text:
25     histogr[c] = histogr.get(c, 0) + 1

```



```
26
27 liste = histogr.items()           # conversion en une liste de tuples
28 liste.sort()                     # tri de la liste
29 print "<H4>Fréquence de chaque caractère dans la phrase :</H4>"
30 for c, f in liste:
31     print 'le caractère <B>"%s"</B> apparaît %s fois <BR>' % (c, f)
```

Les lignes 4 et 5 sont les plus importantes :

Le module `cgi` importé à la ligne 4 assure la connexion du script Python avec l'interface CGI, laquelle permet de dialoguer avec le serveur web.

À la ligne 5, la fonction `FieldStorage()` de ce module renvoie un objet qui contient l'ensemble des données transmises par le formulaire HTML. Nous plaçons cet objet, lequel est assez semblable à un dictionnaire classique, dans la variable `form`.

Par rapport à un véritable dictionnaire, l'objet placé dans `form` présente la différence essentielle qu'il faudra lui appliquer la méthode `value()` pour en extraire les données. Les autres méthodes applicables aux dictionnaires, telles la méthode `has_key()`, par exemple, peuvent être utilisées de la manière habituelle.

Une caractéristique importante de l'objet dictionnaire retourné par `FieldStorage()` est qu'il ne possédera aucune clé pour les champs laissés vides dans le formulaire HTML correspondant.

Dans notre exemple, le formulaire comporte deux champs d'entrée, auxquels nous avons associé les noms `visiteur` et `phrase`. Si ces champs ont effectivement été complétés par l'utilisateur, nous trouverons leurs contenus dans l'objet dictionnaire, aux index « `visiteur` » et « `phrase` ». Par contre, si l'un ou l'autre de ces champs n'a pas été complété, l'index correspondant n'existera tout simplement pas. Avant toute forme de traitement de valeurs, il est donc indispensable de s'assurer de la présence de chacun des index attendus, et c'est ce que nous faisons aux lignes 7 à 15.

Exercices

1. Pour vérifier ce qui précède, vous pouvez par exemple désactiver (en les transformant en commentaires) les lignes 7, 9, 10, 12, 14 & 15 du script. Si vous testez le fonctionnement de l'ensemble, vous constaterez que tout se passe bien si l'utilisateur complète effectivement les champs qui lui sont proposés. Si l'un des champs est laissé vide, par contre, une erreur se produit.

Solution

1. Réfléchissez !

Remarque : le script étant lancé par l'intermédiaire d'une page web, les messages d'erreur de Python ne seront pas affichés dans cette page, mais

plutôt enregistrés dans le journal des événements du serveur web. Veuillez consulter l'administrateur de ce serveur pour savoir comment vous pouvez accéder à ce journal. De toute manière, attendez-vous à ce que la recherche des erreurs dans un script CGI soit plus ardue que dans une application ordinaire.

Le reste du script est assez classique.

- Aux lignes 17 à 21, nous ne faisons qu'afficher les données transmises par le formulaire. Veuillez noter que les variables `nomv` et `text` doivent exister au préalable, ce qui rend indispensables les lignes 9, 10, 14 et 15.
- Aux lignes 23, 24 et 25, nous nous servons d'un dictionnaire pour construire un histogramme simple.
- À la ligne 27, nous convertissons le dictionnaire résultant en une liste de tuples, pour pouvoir trier celle-ci dans l'ordre alphabétique à la ligne 28.
- La boucle `for` des lignes 30 et 31 se passe de commentaires.

Références

1. *(en)* HOWTO Use Python in the web
2. <http://fr.openclassrooms.com/informatique/cours/aperçu-de-la-cgi-avec-python>
3. <https://pypi.python.org/pypi/MySQL-python/1.2.5>

Applications web

Vous avez certainement déjà appris par ailleurs un grand nombre de choses concernant la rédaction de pages web. Vous savez que ces pages sont des documents au format HTML, que l'on peut consulter *via* un réseau (intranet ou internet) à l'aide d'un logiciel appelé browser web ou *navigateur* (ex : *Firefox*, *Google Chrome*, *Konqueror*, *Internet Explorer*, ...).

Les pages HTML sont installées dans les répertoires publics d'un autre ordinateur où fonctionne en permanence un logiciel appelé *serveur Web* (*Apache*, *IIS*, *Zope*, ...). Lorsqu'une connexion a été établie entre cet ordinateur et le vôtre, votre logiciel navigateur peut dialoguer avec le logiciel serveur (par l'intermédiaire de toute une série de dispositifs matériels et logiciels dont nous ne traiterons pas ici : lignes téléphoniques, routeurs, caches, protocoles de communication ...).

Le *protocole HTTP* qui gère la transmission des pages web autorise l'échange de données dans les deux sens. Mais dans la grande majorité des cas, le transfert d'informations n'a pratiquement lieu que dans un seul, à savoir du serveur vers le navigateur : des textes, des images, des fichiers divers lui sont expédiés en grand nombre (ce sont les pages consultées) ; en revanche, le navigateur n'envoie guère au serveur que de toutes petites quantités d'information : essentiellement les adresses URL des pages que l'internaute désire consulter.

Pages web interactives

Vous savez cependant qu'il existe des sites web où vous êtes invité à fournir vous-même des quantités d'information plus importantes : vos références personnelles pour l'inscription à un club ou la réservation d'une chambre d'hôtel, votre numéro de carte de crédit pour la commande d'un article sur un site de commerce électronique, votre avis ou vos suggestions, etc.

Dans un cas comme ceux-là, vous vous doutez bien que l'information transmise doit être prise en charge, du côté du serveur, par un programme spécifique. Il faut donc que les pages web destinées à accueillir cette information soient dotées d'un mécanisme assurant son transfert vers le logiciel destiné à le traiter. Il faudra également que ce logiciel puisse lui-même transmettre en retour une information au serveur, afin que celui-ci puisse présenter le résultat de l'opération à l'internaute, sous la forme d'une nouvelle page web.

Le but du présent chapitre est de vous expliquer comment vous pouvez vous servir de vos compétences de programmeur Python pour ajouter une telle interactivité à un site web, en y intégrant de véritables applications.

Remarque importante : Ce que nous allons expliquer dans les paragraphes qui suivent sera directement fonctionnel sur l'intranet de votre établissement scolaire ou de votre entreprise (à la condition toutefois que l'administrateur de

cet intranet ait configuré son serveur de manière appropriée). En ce qui concerne l'internet, par contre, les choses sont un peu plus compliquées. Il va de soi que l'installation de logiciels sur un ordinateur serveur relié à l'internet ne peut se faire qu'avec l'accord de son propriétaire. Si un fournisseur d'accès à l'internet a mis à votre disposition un certain espace où vous êtes autorisé à installer des pages web « statiques » (c'est-à-dire de simples documents à consulter), cela ne signifie pas pour autant que vous pourrez y faire fonctionner des scripts Python. Pour que cela puisse marcher, vous devrez demander une autorisation et un certain nombre de renseignements à votre fournisseur d'accès. Il faudra en particulier lui demander si vous pouvez activer des scripts CGI écrits en Python à partir de vos pages, et dans quel(s) répertoire(s) vous pouvez les installer.

Un serveur web en pur Python !

Dans les pages précédentes, nous vous avons expliqué quelques rudiments de programmation CGI afin que vous puissiez mieux comprendre comment fonctionne une application web. Mais si vous voulez véritablement développer une telle application (par exemple un site web personnel doté d'une certaine interactivité), vous constaterez rapidement que l'interface CGI est un outil trop sommaire. Son utilisation telle quelle dans des scripts se révèle fort lourde, et il est donc préférable de faire appel à des outils plus élaborés.

L'intérêt pour le développement web est devenu très important, et il existe donc une forte demande pour des interfaces et des environnements de programmation bien adaptés à cette tâche. Or, même s'il ne peut pas prétendre à l'universalité de langages tels que C/C++, Python est déjà largement utilisé un peu partout dans le monde pour écrire des programmes très ambitieux, y compris dans le domaine des serveurs d'applications web. La robustesse et la facilité de mise en œuvre du langage ont séduit de nombreux développeurs de talent, qui ont réalisé des outils de développement web de très haut niveau. Plusieurs de ces applications peuvent vous intéresser si vous souhaitez réaliser vous-même des sites web interactifs de différents types.

Les produits existants sont pour la plupart des logiciels libres. Ils permettent de couvrir une large gamme de besoins, depuis le petit site personnel de quelques pages, jusqu'au gros site commercial collaboratif, capable de répondre à des milliers de requêtes journalières, et dont les différents secteurs sont gérés sans interférence par des personnes de compétences variées (infographistes, programmeurs, spécialistes de bases de données, etc.).

Le plus célèbre de ces produits est le logiciel *Zope*, déjà adopté par de grands organismes privés et publics pour le développement d'intranets et d'extranets collaboratifs. Il s'agit en fait d'un système serveur d'applications, très performant, sécurisé, presque entièrement écrit en Python, et que l'on peut administrer à distance à l'aide d'une simple interface web. Il ne nous est pas possible de décrire l'utilisation de Zope dans ces pages : le sujet est trop vaste, et un livre entier n'y suffirait pas. Sachez cependant que ce produit est parfaitement capable de gérer de très gros sites d'entreprise en offrant

d'énormes avantages par rapport à des solutions classiques telles que *PHP* ou *Java*.

D'autres outils moins ambitieux mais tout aussi intéressants sont disponibles. Tout comme Zope, la plupart d'entre eux peuvent être téléchargés librement depuis l'internet. Le fait qu'ils soient écrits en Python assure en outre leur portabilité : vous pourrez donc les employer aussi bien sous Windows que sous Linux ou MacOS. Chacun d'eux peut être utilisé en conjonction avec un serveur web « classique » tel que Apache ou Xitami (c'est préférable si le site à réaliser est destiné à supporter une charge de connexions très importante), mais certains d'entre eux intègrent en outre leur propre serveur web, ce qui leur permet de fonctionner également de manière tout à fait autonome. Cette possibilité se révèle particulièrement intéressante au cours de la mise au point d'un site, car elle facilite la recherche des erreurs.

Cette totale autonomie alliée à la grande facilité de leur mise en oeuvre fait de ces produits de fort bonnes solutions pour la réalisation de sites web d'intranet spécialisés, notamment dans des petites et moyennes entreprises, des administrations, ou dans des écoles. Si vous souhaitez développer une application Python qui soit accessible par l'intermédiaire d'un simple navigateur web, via un intranet d'entreprise (ou même via l'internet, si la charge prévisible n'est pas trop importante), ces applications sont faites pour vous.

Il en existe une grande variété : *Poor man's Zope*, *Spyce*, *Karrigell*, *Webware*, *CherryPy*, *Quixote*, *Twisted*, etc. Choisissez en fonction de vos besoins : vous n'aurez que l'embarras du choix.

Dans les lignes qui suivent, nous allons décrire une petite application web fonctionnant à l'aide de *Karrigell*. Vous pouvez trouver ce système à l'adresse : <http://karrigell.sourceforge.net>. Il s'agit d'une solution de développement web simple, bien documentée en anglais et en français (son auteur, Pierre Quentel, est en effet originaire de Bretagne, tout comme le mot *karrigell*, d'ailleurs, lequel signifie « charrette »).

Installation de Karrigell

L'installation de *Karrigell* est un jeu d'enfant : il vous suffit d'extraire dans un répertoire quelconque le fichier archive que vous aurez téléchargé depuis l'internet. L'opération de désarchivage crée automatiquement un sous-répertoire nommé *Karrigell-numéro de version*. C'est ce répertoire que nous considérerons comme répertoire racine dans les lignes qui suivent.

Si vous ne comptez pas utiliser le serveur de bases de données *Gadfly*^[1] qui vous est fourni en complément de *Karrigell* lui-même, c'est tout ! Sinon, entrez dans le sous-répertoire *gadfly-1.0.0* et lancez la commande : `python setup.py install` (Sous Linux, il faut être root). Vous devez effectuer cette opération si vous souhaitez visualiser la totalité de la démonstration intégrée.

Démarrage du serveur :

Il s'agit donc bel et bien de mettre en route un *serveur* web, auquel vous pourrez accéder ensuite à l'aide d'un navigateur quelconque, localement ou par l'intermédiaire d'un réseau. Avant de le faire démarrer, il est cependant conseillé de jeter un petit coup d'œil dans son fichier de configuration, lequel se nomme *Karrigell.ini* et se trouve dans le répertoire-racine.

Par défaut, Karrigell attend les requêtes *http* sur le port n° 80. Et c'est bien ce numéro de port que la plupart des logiciels navigateurs utilisent eux-mêmes par défaut. Cependant, si vous installez Karrigell sur une machine Linux dont vous n'êtes pas l'administrateur, vous n'avez pas le droit d'utiliser les numéros de port inférieurs à 1024 (pour des raisons de sécurité). Si vous êtes dans ce cas, vous devez donc modifier le fichier de configuration afin que Karrigell utilise un numéro de port plus élevé. En général, vous choisirez d'enlever simplement le caractère # au début de la ligne 39, ce qui activera l'utilisation du n° de port 8080. Plus tard, vous souhaiterez peut-être encore modifier le fichier de configuration afin de modifier l'emplacement du répertoire racine pour votre site web (par défaut, c'est le répertoire du serveur lui-même).

Une fois le fichier de configuration modifié, entrez dans le répertoire racine du serveur, si vous n'y êtes pas déjà, et lancez simplement la commande :

```
python Karrigell.py
```

C'est tout. Votre serveur Karrigell se met en route, et vous pouvez en vérifier le fonctionnement tout de suite à l'aide de votre navigateur web préféré. Si vous lancez celui-ci sur la même machine que le serveur, vous le dirigerez vers une adresse telle que : *http://localhost:8080/index.html*, « localhost » étant le terme consacré pour désigner la machine locale, « 8080 » le numéro de port choisi dans le fichier de configuration, et « index.html » le nom du fichier qui contient la page d'accueil du site. Par contre, si vous voulez accéder à cette même page d'accueil depuis une autre machine, vous devrez (dans le navigateur de celle-ci) indiquer le nom ou l'adresse IP du serveur, en lieu et place de *localhost*.

Avec l'adresse indiquée au paragraphe précédent^[2], vous atteindrez la page d'accueil d'un site de démonstration de Karrigell, qui est déjà pré-installé dans le répertoire racine. Vous y retrouverez la documentation de base, ainsi que toute une série d'exemples.

Dans ce qui précède, il est sous-entendu que vous avez lancé le serveur depuis une console texte, ou depuis une fenêtre de terminal. Dans un cas comme dans l'autre, les messages de contrôle émis par le serveur apparaîtront dans cette console ou cette fenêtre. C'est là que vous pourrez rechercher des messages d'erreur éventuels. C'est là aussi que vous devrez intervenir si vous voulez arrêter le serveur (avec la combinaison de touches CTRL-C).

Ébauche de site web

Essayons à présent de réaliser notre propre ébauche de site web. À la différence d'un serveur web classique, Karrigell peut gérer non seulement des pages HTML statiques (fichiers *.htm*, *.html*, *.gif*, *.jpg*, *.css*) mais également :

- des scripts Python (fichiers *.py*) ;
- des scripts hybrides Python Inside HTML (fichiers *.pih*) ;
- des scripts hybrides HTML Inside Python (fichiers *.hip*).

Laissons de côté les scripts hybrides, dont vous pourrez étudier vous-même la syntaxe (par ailleurs très simple) si vous vous lancez dans une réalisation d'une certaine importance (ils pourront vous faciliter la vie). Dans le contexte limité de ces pages, nous nous contenterons de quelques expériences de base avec des scripts Python ordinaires.

Comme tous les autres éléments du site (fichiers *.html*, *.gif*, *.jpeg*, etc.), ces scripts Python devront être placés dans le répertoire racine^[3]. Vous pouvez tout de suite effectuer un test élémentaire en rédigeant un petit script d'une seule ligne, tel que :

```
print "Bienvenue sur mon site web !"
```

Sauvegardez ce script sous le nom *hello.py* dans le répertoire racine, puis entrez l'adresse : *http://localhost/hello.py* (ou même : *http://localhost/hello* - l'extension *.py* peut être omise) dans votre navigateur. Vous devriez y voir apparaître le message. Cela signifie donc que dans l'environnement Karrigell, la sortie de l'instruction `print` est redirigée vers la fenêtre du navigateur client, plutôt que la console (ou la fenêtre de terminal) du serveur.

Étant donné que l'affichage a lieu dans une fenêtre de navigateur web, vous pouvez utiliser toutes les ressources de la syntaxe HTML afin d'obtenir un formatage déterminé. Vous pouvez par exemple afficher un petit tableau de 2 lignes et 3 colonnes, avec les instructions suivantes :

```
print """
<TABLE BORDER="1" CELLPADDING="5">
<TR> <TD> Rouge </TD> <TD> Vert </TD> <TD> Bleu </TD> </TR>
<TR> <TD> 15 % </TD> <TD> 62 % </TD> <TD> 23 % </TD> </TR>
</TABLE>
"""
```

Rappelons que la balise `TABLE` définit un tableau. Son option `BORDER` spécifie la largeur des bordures de séparation, et `CELLPADDING` l'écart à réserver autour du contenu des cellules. Les Balises `TR` et `TD` (*Table Row* et *Table Data*) définissent les lignes et les cellules du tableau.

Vous pouvez bien entendu utiliser également toutes les ressources de Python, comme dans l'exemple ci-dessous où nous construisons une table des sinus,

cosinus et tangentes des angles compris entre 0° et 90°, à l'aide d'une boucle classique.

The screenshot shows a Mozilla Firefox browser window with a table titled 'Getting Started'. The table has four columns: 'Angle', 'Sinus', 'Cosinus', and 'Tangente'. The rows contain numerical values for angles from 0 to 60 degrees in increments of 10. The status bar at the bottom of the browser window says 'Done'.

Angle	Sinus	Cosinus	Tangente
0	0.0000000	1.0000000	0
10	0.1736482	0.9848078	0.176327
20	0.3420201	0.9396926	0.3639702
30	0.5000000	0.8660254	0.5773503
40	0.6427876	0.7660444	0.8390996
50	0.7660444	0.6427876	1.191754
60	0.8660254	0.5000000	1.732051

```

1 from math import sin, cos, tan, pi
2
3 # Construction de l'en-tête du tableau avec les titres de colonnes :
4 print """<TABLE BORDER="1" CELLPADDING="5">
5 <TR><TD>Angle</TD><TD>Sinus</TD><TD>Cosinus</TD><TD>Tangente</TD></TR>"""
6
7 for angle in range(0,62,10):
8     # conversion des degrés en radians :
9     aRad = angle * pi / 180
10    # construction d'une ligne de tableau, en exploitant le formatage des
11    # chaînes de caractères pour figoler l'affichage :
12    print "<TR><TD>%s</TD><TD>%8.7f</TD><TD>%8.7f</TD><TD>%8.7g</TD></TR>"\
13          % (angle, sin(aRad), cos(aRad), tan(aRad))
14
15 print "</TABLE>"

```

Commentaires

Ligne 7 : Nous nous servons de la fonction `range()` pour définir la gamme d'angles à couvrir (de zéro à 60 degrés par pas de 10).

Ligne 9 : Les fonctions trigonométriques de Python nécessitent que les angles soient exprimés en radians. Il faut donc effectuer une conversion.

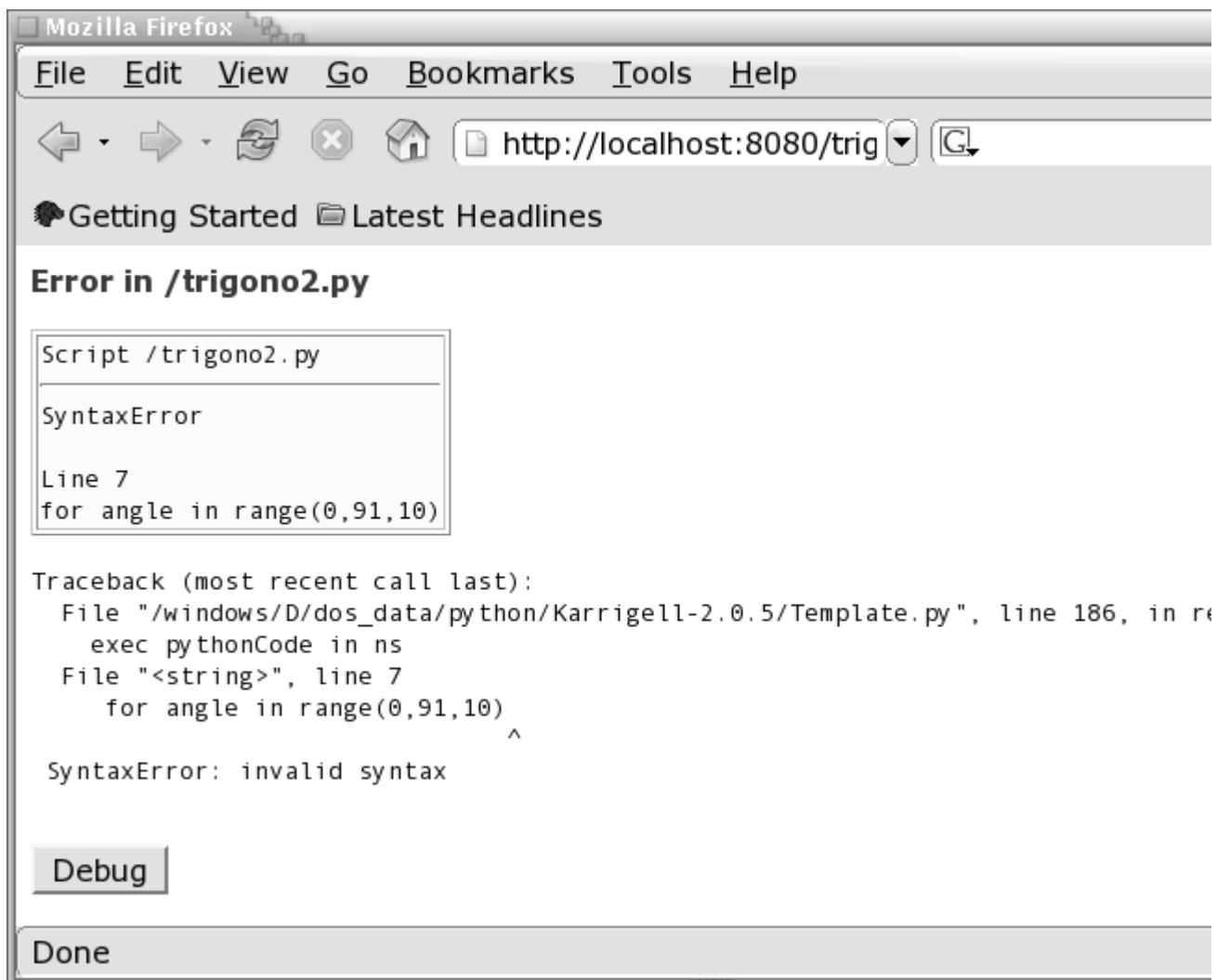
Ligne 12 : Chaque ligne du tableau comporte quatre valeurs, lesquelles sont

prises en forme à l'aide du système de formatage des chaînes de caractères : le marqueur de conversion `%8.7f` force un affichage à 8 chiffres, dont 7 après la « virgule » décimale. Le marqueur `%8.7g` fait à peu près la même chose, mais passe à la notation scientifique lorsque c'est nécessaire.

À ce stade, vous vous demandez peut-être où se situe la différence entre ce que nous venons d'expérimenter ici et un script CGI classique.

L'intérêt de travailler dans un environnement plus spécifique tel que Karrigell apparaît cependant très vite si vous faites des erreurs. En programmation CGI classique, les messages d'erreur émis par l'interpréteur Python ne s'affichent pas dans la fenêtre du navigateur. Ils sont enregistrés dans un fichier journal du serveur (Apache, par exemple), ce qui ne facilite pas leur consultation.

Avec un outil comme Karrigell, par contre, vous disposez d'une signalisation très efficace, ainsi que d'un outil de débogage complet. Faites l'expérience d'introduire une petite erreur dans le script ci-dessus, et relancez votre navigateur sur la page modifiée. Par exemple, en supprimant le double point à la fin de la ligne 7, nous avons obtenu nous-mêmes l'affichage suivant :



En cliquant sur le bouton « Debug », on obtient encore une foule

d'informations complémentaires (affichage du script complet, variables d'environnement, etc.).

Prise en charge des sessions

Lorsque l'on élabore un site web interactif, on souhaite fréquemment que la personne visitant le site puisse s'identifier et fournir un certain nombre de renseignements tout au long de sa visite dans différentes pages (l'exemple type étant le remplissage d'un « caddy » au cours de la consultation d'un site commercial), toutes ces informations étant conservées quelque part jusqu'à la fin de sa visite. Et il faut bien entendu réaliser cela indépendamment pour chaque client connecté.

Il serait possible de transmettre les informations de page en page à l'aide de champs de formulaires cachés, mais ce serait compliqué et très contraignant. Il est préférable que le système serveur soit doté d'un mécanisme spécifique, qui attribue à chaque client une *session* particulière. Karrigell réalise cet objectif par l'intermédiaire de *cookies*. Lorsqu'un nouveau visiteur du site s'identifie, le serveur génère un *cookie* appelé *sessionId* et l'envoie au navigateur web, qui l'enregistre. Ce cookie contient un « identifiant de session » unique, auquel correspond un objet-session sur le serveur. Lorsque le visiteur parcourt les autres pages du site, son navigateur renvoie à chaque fois le contenu du cookie au serveur, et celui-ci peut donc retrouver l'objet-session correspondant, à l'aide de son identifiant. L'objet-session reste donc disponible tout au long de la visite de l'internaute : il s'agit d'un objet Python ordinaire, dans lequel on mémorise un nombre quelconque d'informations sous forme d'attributs.

Au niveau de la programmation, voici comment cela se passe :

Pour chaque page dans laquelle vous voulez consulter ou modifier une information de session, vous commencez par créer un objet de la classe `Session()` :

```
objet_session = Session()
```

Si vous êtes au début de la session, Karrigell génère un identifiant unique, le place dans un cookie et envoie celui-ci au navigateur web. Vous pouvez alors ajouter un nombre quelconque d'attributs à l'objet-session :

```
objet_session.nom = "Jean Dupont"
```

Dans les autres pages, vous procédez de la même manière, mais l'objet produit dans ce cas par la classe `Session()` n'est pas nouveau : c'est l'objet créé en début de session, retrouvé en interne par le serveur grâce à son identifiant relu dans le cookie. Vous pouvez accéder aux valeurs de ses attributs, et aussi en ajouter de nouveaux :

```
obj_sess = Session()           # récupérer l'objet indiqué par le cookie
nom = obj_sess.nom             # retrouver la valeur d'un attribut existant
obj_sess.article = 49137      # ajouter un nouvel attribut
```

Les objets-sessions prennent aussi en charge une méthode `close()`, qui a pour effet d'effacer l'information de session. Vous n'êtes cependant pas obligé de clore explicitement les sessions : Karrigell s'assure de toute façon qu'il n'y ait jamais plus de 1000 sessions simultanées : il efface les plus anciennes quand on arrive à la 1000ème.

Exemple de mise en œuvre

Sauvegardez les trois petits scripts ci-dessous dans le répertoire-racine. Le premier génère un formulaire HTML similaire à ceux qui ont été décrits plus haut. Nommez-le *sessionTest1.py* :

```
1 # Affichage d'un formulaire d'inscription :
2
3 print """
4 <H3>Veuillez vous identifier, SVP :</H3>
5
6 <FORM ACTION = "sessionTest2.py">
7 Votre nom : <INPUT NAME = "nomClient"> <BR>
8 Votre prénom : <INPUT NAME = "prenomClient"> <BR>
9 Votre sexe (m/f) : <INPUT NAME = "sexeClient" SIZE = "1"> <BR>
10 <INPUT TYPE = "submit" VALUE = "OK">
11 </FORM>"""
```

Le suivant sera nommé *sessionTest2.py*. C'est le script mentionné dans la balise d'ouverture du formulaire ci-dessus à la ligne 6, et qui sera invoqué lorsque l'utilisateur actionnera le bouton mis en place à la ligne 10. Ce script recevra les valeurs entrées par l'utilisateur dans les différents champs du formulaire, par l'intermédiaire d'un dictionnaire de requête situé dans la variable d'environnement `QUERY` de Karrigell^[4] :

```
1 obSess = Session()
2
3 obSess.nom = QUERY["nomClient"]
4 obSess.prenom = QUERY["prenomClient"]
5 obSess.sexe = QUERY["sexeClient"]
6
7 if obSess.sexe.upper() == "M":
8     vedette = "Monsieur"
9 else:
10    vedette = "Madame"
11 print "<H3> Bienvenue, %s %s </H3>" % (vedette, obSess.nom)
12 print "<HR>"
13 print """
14 <a href = "sessionTest3.py"> Suite...</a>"""
```

La première ligne de ce script crée l'objet-session, génère pour lui un identifiant unique, et expédie celui-ci au navigateur sous la forme d'un cookie.

Dans les lignes 3, 4, 5, on récupère les valeurs entrées dans les champs du formulaire précédent, en utilisant leurs noms comme clés d'accès au dictionnaire de requêtes.

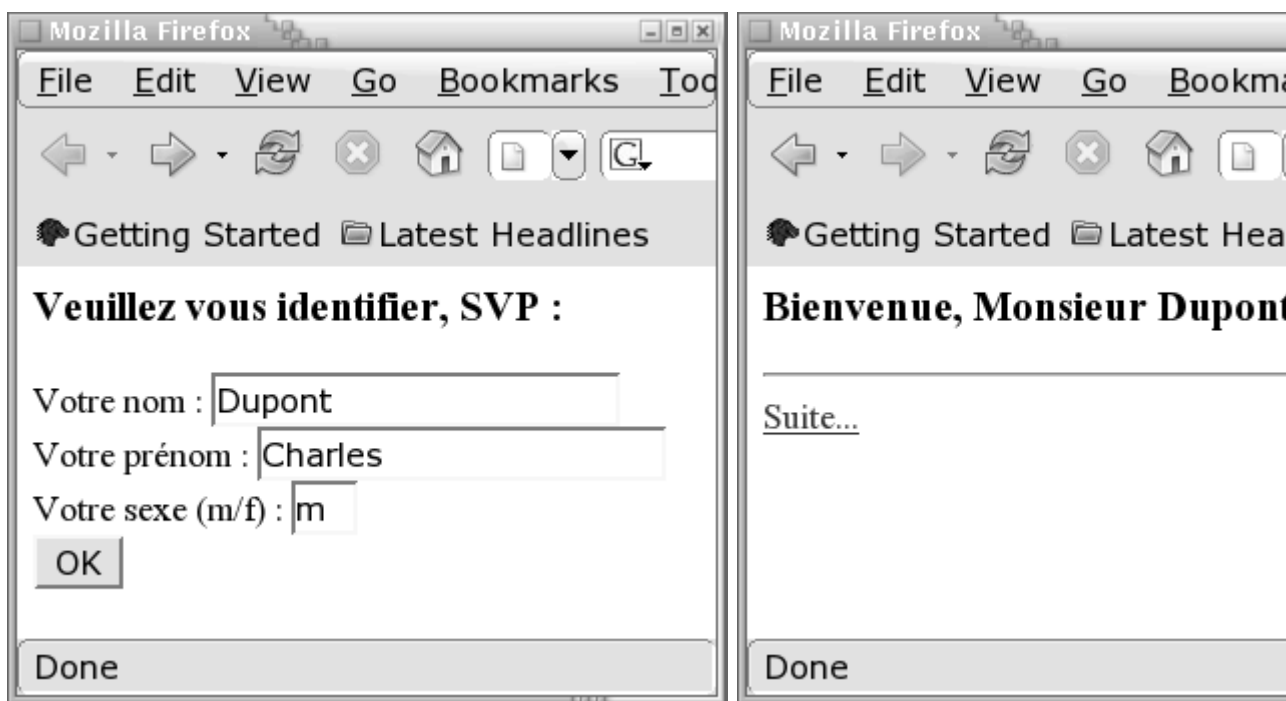
La ligne 14 définit un lien http pointant vers le troisième script, nommé *sessionTest3.py* :

```

1 suiviSess = Session()           # retrouver l'objet-session
2 suiviSess.article = 12345       # lui ajouter des attributs
3 suiviSess.prix = 43.67
4
5 print """
6 <H3> Page suivante </H3> <HR>
7 Suivi de la commande du client : <BR> %s %s <BR>
8 Article n° %s, Prix : %s <HR>
9 """ % (suiviSess.prenom, suiviSess.nom,
10        suiviSess.article, suiviSess.prix)

```

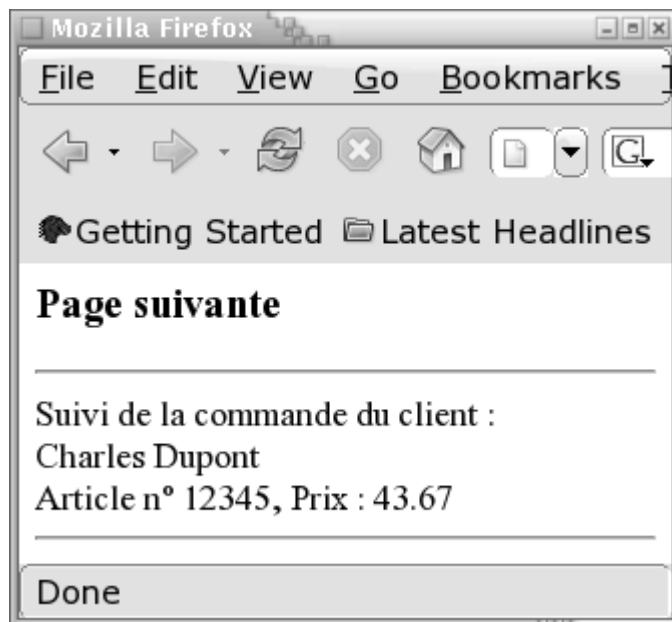
Dirigez votre navigateur web vers l'adresse : <http://localhost:8080/sessionTest1>. Entrez des valeurs de votre choix dans les champs du formulaire, et cliquez sur le bouton OK :



Comme attendu, les informations entrées dans le formulaire sont transmises à la deuxième page. À présent, si vous cliquez sur le lien : « Suite... » dans celle-ci, vous dirigez encore une fois votre navigateur vers une nouvelle page, mais celle-ci n'aura fait l'objet d'aucune transmission de données (puisque l'on n'y accède pas par l'intermédiaire d'un formulaire). Dans le script *sessionTest3.py* qui génère cette page, vous ne pouvez donc pas utiliser la variable `QUERY` pour retrouver les informations entrées par le visiteur.

C'est ici qu'intervient le mécanisme des objets-sessions. Lors du lancement de

ce troisième script, le cookie mémorisé par le navigateur est relu par le serveur, ce qui lui permet de régénérer l'objet-session créé dans le script précédent.



Analysez les trois premières lignes du script *sessionTest3.py* : l'objet `suiviSess` instancié à partir de la classe `Session()` est l'objet-session régénéré. Il contient les informations sauvegardées à la page précédente, et on peut lui en ajouter d'autres dans des attributs supplémentaires.

Vous aurez compris que vous pouvez désormais récupérer toutes ces informations de la même manière dans n'importe quelle autre page, car elles persisteront jusqu'à ce que l'utilisateur termine sa visite du site, à moins que vous ne fermiez vous-même cette session par programme, à l'aide de la méthode `close()` évoquée plus haut.

Exercices

1. Ajoutez au script précédent un lien vers une quatrième page, et écrivez le script qui générera celle-ci. Les informations devront cette fois être affichées dans un tableau :

Nom	Prénom	Sexe	Article	Prix
-----	--------	------	---------	------

Solution

1. Réfléchissez !

Autres développements

Nous terminons ici cette brève étude de Karrigell, car il nous semble vous avoir expliqué l'essentiel de ce qu'il vous faut connaître pour démarrer. Si vous désirez en savoir davantage, il vous suffira de consulter la documentation et les

exemples fournis avec le produit. Comme nous l'avons déjà signalé plus haut, l'installation de Karrigell inclut l'installation du système de bases de données Gadfly. Vous pouvez donc très rapidement et très aisément réaliser un site interactif permettant la consultation à distance d'un ensemble de données quelconques, en admettant bien entendu que la charge de requêtes de votre site reste modérée, et que la taille de la base de données elle-même ne devienne pas gigantesque. N'espérez pas gérer à l'aide de Karrigell un site commercial susceptible de traiter plusieurs millions de requêtes journalières !

Si vous ambitionnez de réaliser ce genre de choses, il vous faudra étudier d'autres offres logicielles, comme par exemple *CherryPy* ou *Zope* associés à *Apache* pour le système serveur, et *SQLite*, *MySQL* ou *PostgreSQL* pour le gestionnaire de bases de données.

Notes

1. Voyez le chapitre précédent : Gadfly est un serveur de bases de données écrit en Python.
2. Si vous avez laissé en place le n° de port par défaut (80), il est inutile de le rappeler dans les adresses, puisque c'est ce n° de port qui est utilisé par défaut par la plupart des navigateurs. Une autre convention consiste à considérer que la page d'accueil d'un site Web se trouve presque toujours dans un fichier nommé *index.htm* ou *index.html*. Lorsque l'on souhaite visiter un site Web en commençant par sa page d'accueil, on peut donc en général omettre ce nom dans l'adresse. Karrigell respecte cette convention, et vous pouvez donc vous connecter en utilisant une adresse simplifiée telle que : *http://localhost:8080* ou même : *http://localhost* (si le n° de port est 80).
3. ...ou bien dans des sous-répertoires du répertoire racine, comme il est d'usage de le faire lorsque l'on cherche à structurer convenablement le site en construction. Il vous suffira dans ce cas d'inclure le nom de ces sous-répertoires dans les adresses correspondantes.
4. Karrigell met en place un certain nombre de variables globales dont les noms sont en majuscules pour éviter un conflit éventuel avec les vôtres. Celle-ci joue le même rôle que la fonction `FieldStorage()` du module *cgi*. Veuillez consulter la documentation de Karrigell si vous souhaitez obtenir des explications plus détaillées.

Réseau

Communication entre programmes

Le développement extraordinaire de l'internet a amplement démontré que les ordinateurs peuvent être des outils de communication très efficaces. Dans ce chapitre, nous allons expérimenter la plus simple des techniques d'interconnexion de deux programmes, qui leur permette de s'échanger des informations par l'intermédiaire d'un réseau.

Pour ce qui va suivre, nous supposerons donc que vous collaborez avec un ou plusieurs de vos condisciples, et que vos postes de travail Python sont connectés à un réseau local dont les communications utilisent le protocole TCP/IP. Le système d'exploitation n'a pas d'importance : vous pouvez par exemple installer l'un des scripts Python décrits ci-après sur un poste de travail fonctionnant sous Linux, et le faire dialoguer avec un autre script mis en œuvre sur un poste de travail confié aux bons soins d'un système d'exploitation différent, tel que MacOS ou Windows.

Vous pouvez également expérimenter ce qui suit sur une seule et même machine, en mettant les différents scripts en œuvre dans des fenêtres indépendantes.

Les sockets

Le premier exercice qui va vous être proposé consistera à établir une communication entre deux machines seulement. L'une et l'autre pourront s'échanger des messages à tour de rôle, mais vous constaterez cependant que leurs configurations ne sont pas symétriques. Le script installé sur l'une de ces machines jouera en effet le rôle d'un logiciel serveur, alors que l'autre se comportera comme un logiciel *client*.

Le logiciel serveur fonctionne en continu, sur une machine dont l'identité est bien définie sur le réseau grâce à une *adresse* IP spécifique^[1]. Il guette en permanence l'arrivée de requêtes expédiées par les clients potentiels en direction de cette adresse, par l'intermédiaire d'un *port de communication* bien déterminé. Pour ce faire, le script correspondant doit mettre en œuvre un objet logiciel associé à ce port, que l'on appelle un socket.

Au départ d'une autre machine, le logiciel client tente d'établir la connexion en émettant une requête appropriée. Cette requête est un message qui est confié au réseau, un peu comme on confie une lettre à la Poste. Le réseau pourrait en effet acheminer la requête vers n'importe quelle autre machine, mais une seule est visée : pour que la destination visée puisse être atteinte, la requête contient dans son en-tête l'indication de l'adresse IP et du port de communication destinataires.

Lorsque la connexion est établie avec le serveur, le client lui assigne lui-même l'un de ses propres ports de communication. A partir de ce moment, on peut considérer qu'un canal privilégié relie les deux machines, comme si on les avait connectées l'une à l'autre par l'intermédiaire d'un fil (les deux ports de communication respectifs jouant le rôle des deux extrémités de ce fil). L'échange d'informations proprement dit peut commencer.

Pour pouvoir utiliser les ports de communication réseau, les programmes font appel à un ensemble de procédures et de fonctions du système d'exploitation, par l'intermédiaire d'objets interfaces que l'on appelle des sockets. Ceux-ci peuvent mettre en œuvre deux techniques de communication différentes et complémentaires : celle des paquets (que l'on appelle aussi des datagrammes), très largement utilisée sur l'internet, et celle de la connexion continue, ou *stream socket*, qui est un peu plus simple.

Construction d'un serveur élémentaire

Pour nos premières expériences, nous allons utiliser la technique des stream sockets. Celle-ci est en effet parfaitement appropriée lorsqu'il s'agit de faire communiquer des ordinateurs interconnectés par l'intermédiaire d'un réseau local. C'est une technique particulièrement aisée à mettre en œuvre, et elle permet un débit élevé pour l'échange de données.

L'autre technologie (celle des paquets) serait préférable pour les communications expédiées via l'internet, en raison de sa plus grande fiabilité (les mêmes paquets peuvent atteindre leur destination par différents chemins, être émis ou ré-émis en plusieurs exemplaires si cela se révèle nécessaire pour corriger les erreurs de transmission), mais sa mise en œuvre est un peu plus complexe. Nous ne l'étudierons pas dans ce cours.

Le script ci-dessous met en place un serveur capable de communiquer avec un seul client :

```
1 # Définition d'un serveur réseau rudimentaire
2 # Ce serveur attend la connexion d'un client, pour entamer un dialogue avec lui
3
4 import socket, sys
5
6 HOST = '192.168.14.152'
7 PORT = 50000
8
9 # 1) création du socket :
10 mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11
12 # 2) liaison du socket à une adresse précise :
13 try:
14     mySocket.bind((HOST, PORT))
15 except socket.error:
16     print "La liaison du socket à l'adresse choisie a échoué."
17     sys.exit()
18
19 while 1:
```



```

i20  # 3) Attente de la requête de connexion d'un client :
i21  print "Serveur prêt, en attente de requêtes ..."
i22  mySocket.listen(5)
i23
i24  # 4) Etablissement de la connexion :
i25  connexion, adresse = mySocket.accept()
i26  print "Client connecté, adresse IP %s, port %s" % (adresse[0], adresse[1])
i27
i28  # 5) Dialogue avec le client :
i29  connexion.send("Vous êtes connecté au serveur Marcel. Envoyez vos messages.")
i30  msgClient = connexion.recv(1024)
i31  while 1:
i32      print "C>", msgClient
i33      if msgClient.upper() == "FIN" or msgClient == "":
i34          break
i35      msgServeur = raw_input("S> ")
i36      connexion.send(msgServeur)
i37      msgClient = connexion.recv(1024)
i38
i39  # 6) Fermeture de la connexion :
i40  connexion.send("Au revoir !")
i41  print "Connexion interrompue."
i42  connexion.close()
i43
i44  ch = raw_input("<R>ecommencer <T>erminer ? ")
i45  if ch.upper() == 'T':
i46      break

```

Commentaires

- Ligne 4 : Le module *socket* contient toutes les fonctions et les classes nécessaires pour construire des programmes communicants. Comme nous allons le voir dans les lignes suivantes, l'établissement de la communication comporte six étapes.
- Lignes 6 et 7 : Ces deux variables définissent l'identité du serveur, telle qu'on l'intégrera au socket. `HOST` doit contenir une chaîne de caractères indiquant l'adresse IP du serveur sous la forme décimale habituelle, ou encore le nom DNS de ce même serveur (mais à la condition qu'un mécanisme de résolution des noms ait été mis en place sur le réseau). `PORT` doit contenir un entier, à savoir le numéro d'un port qui ne soit pas déjà utilisé pour un autre usage, et de préférence une valeur supérieure à 1024 (Cfr. votre cours sur les services réseau).
- Lignes 9 et 10 : Première étape du mécanisme d'interconnexion. On instancie un objet de la classe `socket()`, en précisant deux options qui indiquent le type d'adresses choisi (nous utiliserons des adresses de type « internet ») ainsi que la technologie de transmission (datagrammes ou connexion continue (*stream*) : nous avons décidé d'utiliser cette dernière).
- Lignes 12 à 17 : Seconde étape. On tente d'établir la liaison entre le socket et le port de communication. Si cette liaison ne peut être établie (port de communication occupé, par exemple, ou nom de machine incorrect), le programme se termine sur un message d'erreur.
Remarque : la méthode `bind()` du socket attend un argument du type tuple, raison pour laquelle nous devons enfermer nos deux variables dans une

double paire de parenthèses.

- Ligne 19 : Notre programme serveur étant destiné à fonctionner en permanence dans l'attente des requêtes de clients potentiels, nous le lançons dans une boucle sans fin.
- Lignes 20 à 22 : Troisième étape. Le socket étant relié à un port de communication, il peut à présent se préparer à recevoir les requêtes envoyées par les clients. C'est le rôle de la méthode `listen()`. L'argument qu'on lui transmet indique le nombre maximum de connexions à accepter en parallèle.
- Lignes 24 à 26 : Quatrième étape. Lorsqu'on fait appel à sa méthode `accept()`, le socket attend indéfiniment qu'une requête se présente. Le script est donc interrompu à cet endroit, un peu comme il le serait si nous faisons appel à une fonction `input()` pour attendre une entrée clavier. Si une requête est réceptionnée, la méthode `accept()` renvoie un tuple de deux éléments : le premier est la référence d'un nouvel objet de la classe `socket()`^[2], qui sera la véritable interface de communication entre le client et le serveur, et le second un autre tuple contenant les coordonnées de ce client (son adresse IP et le n° de port qu'il utilise lui-même).
- Lignes 28 à 30 : Cinquième étape. La communication proprement dite est établie. Les méthodes `send()` et `recv()` du socket servent évidemment à l'émission et à la réception des messages, qui doivent être de simples chaînes de caractères.
Remarques : la méthode `send()` renvoie le nombre d'octets expédiés. L'appel de la méthode `recv()` doit comporter un argument entier indiquant le nombre maximum d'octets à réceptionner en une fois (Les octets surnuméraires sont mis en attente dans un tampon. Ils sont transmis lorsque la même méthode `recv()` est appelée à nouveau).
- Lignes 31 à 37 : Cette nouvelle boucle sans fin maintient le dialogue jusqu'à ce que le client décide d'envoyer le mot « fin » ou une simple chaîne vide. Les écrans des deux machines afficheront chacune l'évolution de ce dialogue.
- Lignes 39 à 42 : Sixième étape. Fermeture de la connexion.

Construction d'un client rudimentaire

Le script ci-dessous définit un logiciel client complémentaire du serveur décrit dans les pages précédentes. On notera sa grande simplicité.

```
1 # Définition d'un client réseau rudimentaire
2 # Ce client dialogue avec un serveur ad hoc
3
4 import socket, sys
5
6 HOST = '192.168.14.152'
7 PORT = 50000
8
```

```

9 # 1) création du socket :
10 mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11
12 # 2) envoi d'une requête de connexion au serveur :
13 try:
14     mySocket.connect((HOST, PORT))
15 except socket.error:
16     print "La connexion a échoué."
17     sys.exit()
18 print "Connexion établie avec le serveur."
19
20 # 3) Dialogue avec le serveur :
21 msgServeur = mySocket.recv(1024)
22
23 while 1:
24     if msgServeur.upper() == "FIN" or msgServeur == "":
25         break
26     print "S>", msgServeur
27     msgClient = raw_input("C> ")
28     mySocket.send(msgClient)
29     msgServeur = mySocket.recv(1024)
30
31 # 4) Fermeture de la connexion :
32 print "Connexion interrompue."
33 mySocket.close()

```

Commentaires

- Le début du script est similaire à celui du serveur. L'adresse IP et le port de communication doivent être ceux du serveur.
- Lignes 12 à 18 : On ne crée cette fois qu'un seul objet socket, dont on utilise la méthode `connect()` pour envoyer la requête de connexion.
- Lignes 20 à 33 : Une fois la connexion établie, on peut dialoguer avec le serveur en utilisant les méthodes `send()` et `recv()` déjà décrites plus haut pour celui-ci.

Récupérer une page Web en python

Python intègre le module `httplib`^[3] qui permet d'émettre et de recevoir des requêtes HTTP.

Afficher une page Web

Le code suivant (src) permet de récupérer, à l'aide d'une requête HTTP, une page Web et affiche son code source à l'écran.

Exemple d'appel HTTP GET

```

# On utilise le module httplib
import httplib

# Connexion au proxy

```

```
# (si vous n'ête pas derrière un proxy, alors mettre directement 'fr.wikibooks.org')
conn = urllib.HTTP('proxy:3128')

# Requête GET
# (si vous n'ête pas derrière un proxy, alors mettre directement
# '/w/index.php?title=Programmation_Python_Le_r%C3%A9seau&action=edit'
conn.putrequest('GET', 'http://fr.wikibooks.org/w/index.php?title=Programmation_Python_Le

conn.putheader('Accept', 'text/html')
conn.putheader('Accept', 'text/plain')

# Décommenter les 2 lignes suivantes si votre proxy nécessite une authentification
# auth = "Basic " + "username:password".encode('base64')
# h1.putheader('Proxy-Authorization', auth)

conn.endheaders()

# Récupération de la réponse
errcode, errmsg, headers = conn.getreply()

# Affichage d'éventuelles erreurs
print errcode
print errmsg
print headers

# Affichage de la réponse ligne après ligne
f=conn.getfile()
for line in f:
    print line

# fin de la connexion
conn.close()
```

Références

1. Une machine particulière peut également être désignée par un nom plus explicite, mais à la condition qu'un mécanisme ait été mis en place sur le réseau (DNS) pour traduire automatiquement ce nom en adresse IP. Veuillez consulter votre cours sur les systèmes d'exploitation et les réseaux pour en savoir davantage.
2. . En bref, si nous voulons que notre serveur puisse prendre en charge simultanément les connexions de plusieurs clients, il nous faudra disposer d'un socket distinct pour chacun d'eux, indépendamment du premier que l'on laissera fonctionner en permanence pour réceptionner les requêtes qui continuent à arriver en provenance de nouveaux clients.
3. <http://docs.python.org/lib/module-urllib.html>

Threads

Gestion de plusieurs tâches en parallèle à l'aide des threads

Le système de communication que nous avons élaboré dans les pages précédentes est vraiment très rudimentaire : d'une part il ne met en relation que deux machines seulement, et d'autre part il limite la liberté d'expression des deux interlocuteurs. Ceux-ci ne peuvent en effet envoyer des messages que chacun à leur tour. Par exemple, lorsque l'un d'eux vient d'émettre un message, son système reste bloqué tant que son partenaire ne lui a pas envoyé une réponse. Lorsqu'il vient de recevoir une telle réponse, son système reste incapable d'en réceptionner une autre, tant qu'il n'a pas entré lui-même un nouveau message, ... et ainsi de suite.

Tous ces problèmes proviennent du fait que nos scripts habituels ne peuvent s'occuper que d'une seule chose à la fois. Lorsque le flux d'instructions rencontre une fonction `input()`, par exemple, il ne se passe plus rien tant que l'utilisateur n'a pas introduit la donnée attendue. Et même si cette attente dure très longtemps, il n'est habituellement pas possible que le programme effectue d'autres tâches pendant ce temps. Ceci n'est toutefois vrai qu'au sein d'un seul et même programme : vous savez certainement que vous pouvez exécuter d'autres applications entretemps sur votre ordinateur, car les systèmes d'exploitation modernes sont *multi-tâches*.

Les pages qui suivent sont destinées à vous expliquer comment vous pouvez introduire cette fonctionnalité multi-tâches dans vos programmes, afin que vous puissiez développer de véritables applications réseau, capables de communiquer simultanément avec plusieurs partenaires.

Veillez à présent considérer le script de la page précédente. Sa fonctionnalité essentielle réside dans la boucle `while` des lignes 23 à 29. Or, cette boucle s'interrompt à deux endroits :

- À la ligne 27, pour attendre les entrées clavier de l'utilisateur (fonction `raw_input()`) ;
- À la ligne 29, pour attendre l'arrivée d'un message réseau.

Ces deux attentes sont donc *successives*, alors qu'il serait bien plus intéressant qu'elles soient *simultanées*. Si c'était le cas, l'utilisateur pourrait expédier des messages à tout moment, sans devoir attendre à chaque fois la réaction de son partenaire. Il pourrait également recevoir n'importe quel nombre de messages, sans l'obligation d'avoir à répondre à chacun d'eux pour recevoir les autres.

Nous pouvons arriver à ce résultat si nous apprenons à gérer plusieurs séquences d'instructions *en parallèle* au sein d'un même programme. Mais comment cela est-il possible ?

Au cours de l'histoire de l'informatique, plusieurs techniques ont été mises au point pour partager le temps de travail d'un processeur entre différentes tâches, de telle manière que celles-ci paraissent être effectuées en même temps (alors qu'en réalité le processeur s'occupe d'un petit bout de chacune d'elles à tour de rôle). Ces techniques sont implémentées dans le système d'exploitation, et il n'est pas nécessaire de les détailler ici, même s'il est possible d'accéder à chacune d'elles avec Python.

Dans les pages suivantes, nous allons apprendre à utiliser celle de ces techniques qui est à la fois la plus facile à mettre en oeuvre, et la seule qui soit véritablement portable (elle est en effet supportée par tous les grands systèmes d'exploitation) : on l'appelle la technique des processus légers ou threads^[1].

Dans un programme d'ordinateur, les threads sont des flux d'instructions qui sont menés en parallèle (quasi-simultanément), tout en partageant le même espace de noms global.

En fait, le flux d'instructions de n'importe quel programme Python suit toujours au moins un thread : le *thread principal*.

À partir de celui-ci, d'autres threads « enfants » peuvent être amorcés, qui seront exécutés en parallèle. Chaque thread enfant se termine et disparaît sans autre forme de procès lorsque toutes les instructions qu'il contient ont été exécutées. Par contre, lorsque le thread principal se termine, il faut parfois s'assurer que tous ses threads enfants « meurent » avec lui.

Client gérant l'émission et la réception simultanées

Nous allons maintenant mettre en pratique la technique des threads pour construire un système de « chat »^[2] simplifié. Ce système sera constitué d'un seul serveur et d'un nombre quelconque de clients. Contrairement à ce qui se passait dans notre premier exercice, personne n'utilisera le serveur lui-même pour communiquer, mais lorsque celui-ci aura été mis en route, plusieurs clients pourront s'y connecter et commencer à s'échanger des messages.

Chaque client enverra tous ses messages au serveur, mais celui-ci les ré-expédiera immédiatement à tous les autres clients connectés, de telle sorte que chacun puisse voir l'ensemble du trafic. Chacun pourra à tout moment envoyer ses messages, et recevoir ceux des autres, dans n'importe quel ordre, la réception et l'émission étant gérées simultanément, dans des threads séparés.

Le script ci-après définit le programme client. Vous constaterez que la partie principale du script (ligne 38 et suivantes) est similaire à celle de l'exemple précédent. Seule la partie « Dialogue avec le serveur » a été remplacée. Au lieu d'une boucle `while`, vous y trouvez à présent les instructions de création de deux objets threads (aux lignes 49 et 50), dont on démarre la fonctionnalité aux deux

lignes suivantes. Ces objets threads sont créés par dérivation, à partir de la classe `Thread()` du module *threading*. Ils s'occuperont indépendamment de la réception et de l'émission des messages. Les deux threads « enfants » sont ainsi parfaitement encapsulés dans des objets distincts, ce qui facilite la compréhension du mécanisme.

```

1 # Définition d'un client réseau gérant en parallèle l'émission
2 # et la réception des messages (utilisation de 2 THREADS).
3
4 host = '192.168.0.235'
5 port = 40000
6
7 import socket, sys, threading
8
9 class ThreadReception(threading.Thread):
10     """objet thread gérant la réception des messages"""
11     def __init__(self, conn):
12         threading.Thread.__init__(self)
13         self.connexion = conn # réf. du socket de connexion
14
15     def run(self):
16         while 1:
17             message_recu = self.connexion.recv(1024)
18             print "*" + message_recu + "*"
19             if message_recu == '' or message_recu.upper() == "FIN":
20                 break
21             # Le thread <réception> se termine ici.
22             # On force la fermeture du thread <émission> :
23             th_E_Thread.stop()
24             print "Client arrêté. Connexion interrompue."
25             self.connexion.close()
26
27 class ThreadEmission(threading.Thread):
28     """objet thread gérant l'émission des messages"""
29     def __init__(self, conn):
30         threading.Thread.__init__(self)
31         self.connexion = conn # réf. du socket de connexion
32
33     def run(self):
34         while 1:
35             message_emis = raw_input()
36             self.connexion.send(message_emis)
37
38 # Programme principal - Établissement de la connexion :
39 connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40 try:
41     connexion.connect((host, port))
42 except socket.error:
43     print "La connexion a échoué."
44     sys.exit()
45 print "Connexion établie avec le serveur."
46
47 # Dialogue avec le serveur : on lance deux threads pour gérer
48 # indépendamment l'émission et la réception des messages :
49 th_E = ThreadEmission(connexion)
50 th_R = ThreadReception(connexion)
51 th_E.start()
52 th_R.start()

```

Commentaires

Remarque générale : Dans cet exemple, nous avons décidé de créer deux objets threads indépendants du thread principal, afin de bien mettre en évidence les mécanismes. Notre programme utilise donc trois threads en tout, alors que le lecteur attentif aura remarqué que deux pourraient suffire. En effet : le thread principal ne sert en définitive qu'à lancer les deux autres ! Il n'y a cependant aucun intérêt à limiter le nombre de threads. Au contraire : à partir du moment où l'on décide d'utiliser cette technique, il faut en profiter pour compartimenter l'application en unités bien distinctes.

Ligne 7 : Le module *threading* contient la définition de toute une série de classes intéressantes pour gérer les threads. Nous n'utiliserons ici que la seule classe `Thread()`, lorsque nous devrons nous préoccuper de problèmes de synchronisation entre différents threads concurrents.

Lignes 9 à 25 : Les classes dérivées de la classe `Thread()` contiendront essentiellement une méthode `run()`. C'est dans celle-ci que l'on placera la portion de programme spécifiquement confiée au thread. Il s'agira souvent d'une boucle répétitive, comme ici. Vous pouvez parfaitement considérer le contenu de cette méthode comme un script indépendant, qui s'exécute en parallèle avec les autres composants de votre application. Lorsque ce code a été complètement exécuté, le thread se referme.

Lignes 16 à 20 : Cette boucle gère la réception des messages. À chaque itération, le flux d'instructions s'interrompt à la ligne 17 dans l'attente d'un nouveau message, mais le reste du programme n'est pas figé pour autant : les autres threads continuent leur travail indépendamment.

Ligne 19 : La sortie de boucle est provoquée par la réception d'un message 'fin' (en majuscules ou en minuscules), ou encore d'un message vide (c'est notamment le cas si la connexion est coupée par le partenaire). Quelques instructions de « nettoyage » sont alors exécutées, et puis le thread se termine.

Ligne 23 : Lorsque la réception des messages est terminée, nous souhaitons que le reste du programme se termine lui aussi. Il nous faut donc forcer la fermeture de l'autre objet thread, celui que nous avons mis en place pour gérer l'émission des messages. Cette fermeture forcée peut être obtenue à l'aide de la méthode `_Thread__stop()`^[3].

Lignes 27 à 36 : Cette classe définit donc un autre objet thread, qui contient cette fois une boucle de répétition perpétuelle. Il ne pourra donc se terminer que contraint et forcé par méthode décrite au paragraphe précédent. À chaque itération de cette boucle, le flux d'instructions s'interrompt à la ligne 35 dans l'attente d'une entrée clavier, mais cela n'empêche en aucune manière les autres threads de faire leur travail.

Lignes 38 à 45 : Ces lignes sont reprises à l'identique des scripts précédents.

Lignes 47 à 52 : Instanciation et démarrage des deux objets threads « enfants ». Veuillez noter qu'il est recommandé de provoquer ce démarrage en invoquant la méthode intégrée `start()`, plutôt qu'en faisant appel directement à

la méthode `run()` que vous aurez définie vous-même. Sachez également que vous ne pouvez invoquer `start()` qu'une seule fois (une fois arrêté, un objet `thread` ne peut pas être redémarré).

Serveur gérant les connexions de plusieurs clients en parallèle

Le script ci-après crée un serveur capable de prendre en charge les connexions d'un certain nombre de clients du même type que ce que nous avons décrit dans les pages précédentes.

Ce serveur n'est pas utilisé lui-même pour communiquer : ce sont les clients qui communiquent les uns avec les autres, par l'intermédiaire du serveur. Celui-ci joue donc le rôle d'un relais : il accepte les connexions des clients, puis attend l'arrivée de leurs messages. Lorsqu'un message arrive en provenance d'un client particulier, le serveur le ré-expédie à tous les autres, en lui ajoutant au passage une chaîne d'identification spécifique du client émetteur, afin que chacun puisse voir tous les messages, et savoir de qui ils proviennent.

```
1 # Définition d'un serveur réseau gérant un système de CHAT simplifié.
2 # Utilise les threads pour gérer les connexions clientes en parallèle.
3
4 HOST = '192.168.0.235'
5 PORT = 40000
6
7 import socket, sys, threading
8
9 class ThreadClient(threading.Thread):
10     '''dérivation d'un objet thread pour gérer la connexion avec un client'''
11     def __init__(self, conn):
12         threading.Thread.__init__(self)
13         self.connexion = conn
14
15     def run(self):
16         # Dialogue avec le client :
17         nom = self.getName()          # Chaque thread possède un nom
18         while 1:
19             msgClient = self.connexion.recv(1024)
20             if msgClient.upper() == "FIN" or msgClient == "":
21                 break
22             message = "%s> %s" % (nom, msgClient)
23             print message
24             # Faire suivre le message à tous les autres clients :
25             for cle in conn_client:
26                 if cle != nom:        # ne pas le renvoyer à l'émetteur
27                     conn_client[cle].send(message)
28
29             # Fermeture de la connexion :
30             self.connexion.close()    # couper la connexion côté serveur
31             del conn_client[nom]      # supprimer son entrée dans le dictionnaire
32             print "Client %s déconnecté." % nom
33             # Le thread se termine ici
34
35 # Initialisation du serveur - Mise en place du socket :
36 mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
37 try:
38     mySocket.bind((HOST, PORT))
39 except socket.error:
40     print "La liaison du socket à l'adresse choisie a échoué."
41     sys.exit()
42 print "Serveur prêt, en attente de requêtes ..."
43 mySocket.listen(5)
44
45 # Attente et prise en charge des connexions demandées par les clients :
46 conn_client = {} # dictionnaire des connexions clients
47 while 1:
48     connexion, adresse = mySocket.accept()
49     # Créer un nouvel objet thread pour gérer la connexion :
50     th = ThreadClient(connexion)
51     th.start()
52     # Mémoriser la connexion dans le dictionnaire :
53     it = th.getName() # identifiant du thread
54     conn_client[it] = connexion
55     print "Client %s connecté, adresse IP %s, port %s." %\
56           (it, adresse[0], adresse[1])
57     # Dialogue avec le client :
58     connexion.send("Vous êtes connecté. Envoyez vos messages.")
```

Commentaires

Lignes 35 à 43 : L'initialisation de ce serveur est identique à celle du serveur rudimentaire décrit au début du présent chapitre.

Ligne 46 : Les références des différentes connexions doivent être mémorisées. Nous pourrions les placer dans une liste, mais il est plus judicieux de les placer dans un dictionnaire, pour deux raisons : La première est que nous devons pouvoir ajouter ou enlever ces références dans n'importe quel ordre, puisque les clients se connecteront et se déconnecteront à leur guise. La seconde est que nous pouvons disposer aisément d'un identifiant unique pour chaque connexion, lequel pourra servir de clé d'accès dans un dictionnaire. Cet identifiant nous sera en effet fourni automatiquement par La classe `Thread()`.

Lignes 47 à 51 : Le programme commence ici une boucle de répétition perpétuelle, qui va constamment attendre l'arrivée de nouvelles connexions. Pour chacune de celles-ci, un nouvel objet `ThreadClient()` est créé, lequel pourra s'occuper d'elle indépendamment de toutes les autres.

Lignes 52 à 54 : Obtention d'un identifiant unique à l'aide de la méthode `getName()`. Nous pouvons profiter ici du fait que Python attribue automatiquement un nom unique à chaque nouveau thread : ce nom convient bien comme identifiant (ou clé) pour retrouver la connexion correspondante dans notre dictionnaire. Vous pourrez constater qu'il s'agit d'une chaîne de caractères, de la forme : « Thread-N » (N étant le numéro d'ordre du thread).

Lignes 15 à 17 : Gardez bien à l'esprit qu'il se créera autant d'objets `ThreadClient()` que de connexions, et que tous ces objets fonctionneront en parallèle. La méthode `getName()` peut alors être utilisée au sein de l'un quelconque de ces objets pour retrouver son identité particulière. Nous

utiliserons cette information pour distinguer la connexion courante de toutes les autres (voir ligne 26).

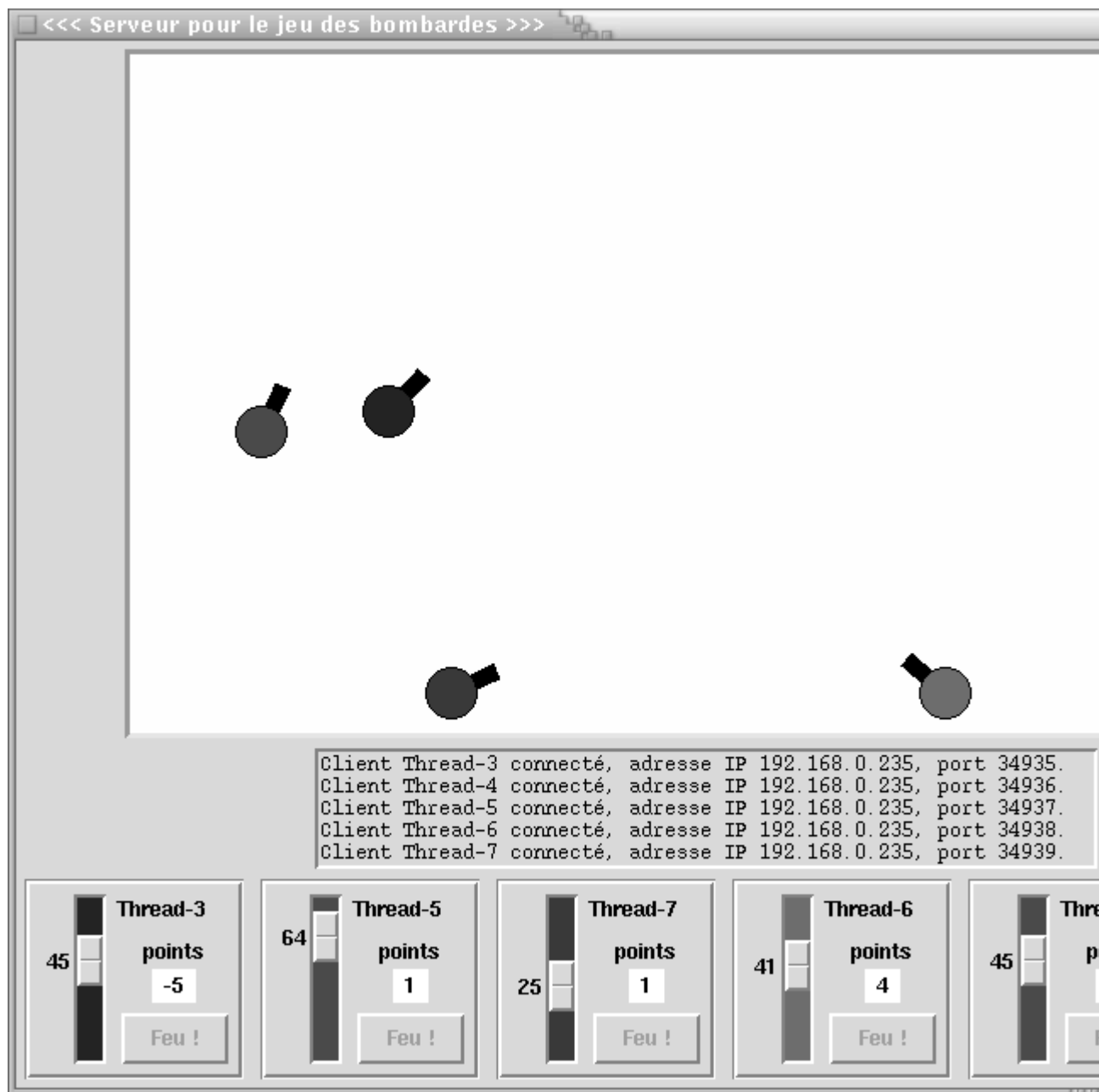
Lignes 18 à 23 : L'utilité du thread est de réceptionner tous les messages provenant d'un client particulier. Il faut donc pour cela une boucle de répétition perpétuelle, qui ne s'interrompra qu'à la réception du message spécifique : « fin », ou encore à la réception d'un message vide (cas où la connexion est coupée par le partenaire).

Lignes 24 à 27 : Chaque message reçu d'un client doit être ré-expédié à tous les autres. Nous utilisons ici une boucle `for` pour parcourir l'ensemble des clés du dictionnaire des connexions, lesquelles nous permettent ensuite de retrouver les connexions elles-mêmes. Un simple test (à la ligne 26) nous évite de ré-expédier le message au client dont il provient.

Ligne 31 : Lorsque nous fermons un socket de connexion, il est préférable de supprimer sa référence dans le dictionnaire, puisque cette référence ne peut plus servir. Et nous pouvons faire cela sans précaution particulière, car les éléments d'un dictionnaire ne sont pas ordonnés (nous pouvons en ajouter ou en enlever dans n'importe quel ordre).

Jeu des bombardes, version réseau

Au chapitre 15, nous avons commenté le développement d'un petit jeu de combat dans lequel des joueurs s'affrontaient à l'aide de bombardes. L'intérêt de ce jeu reste toutefois fort limité, tant qu'il se pratique sur un seul et même ordinateur. Nous allons donc le perfectionner, en y intégrant les techniques que nous venons d'apprendre. Comme le système de « chat » décrit dans les pages précédentes, l'application complète se composera désormais de deux programmes distincts : un logiciel serveur qui ne sera mis en fonctionnement que sur une seule machine, et un logiciel client qui pourra être lancé sur toute une série d'autres. Du fait du caractère portable de Python, il vous sera même possible d'organiser des combats de bombardes entre ordinateurs gérés par des systèmes d'exploitation différents (MacOS <> Linux <> Windows !).



Programme serveur : vue d'ensemble

Les programmes serveur et client exploitent la même base logicielle, elle-même largement récupérée de ce qui avait déjà été mis au point tout au long du chapitre 15. Nous admettrons donc pour la suite de cet exposé que les deux versions précédentes du jeu ont été sauvegardées dans les fichiers-modules `canon03.py` et `canon04.py`, installés dans le répertoire courant. Nous pouvons en effet réutiliser une bonne partie du code qu'ils contiennent, en nous servant judicieusement de l'importation et de l'héritage de classes.

Du module `canon04`, nous allons réutiliser la classe `Canon()` telle quelle, aussi bien pour le logiciel serveur que pour le logiciel client. De ce même module, nous importerons également la classe `AppBombardes()`, dont nous ferons dériver la classe maîtresse de notre application serveur : `AppServeur()`.

Du module *canon03*, nous récupérerons la classe `Pupitre()` dont nous tirerons une version plus adaptée au « contrôle à distance ».

Enfin, deux nouvelles classes viendront s'ajouter aux précédentes, chacune spécialisée dans la création d'un objet thread : la classe `ThreadClients()`, dont une instance surveillera en permanence le socket destiné à réceptionner les demandes de connexion de nouveaux clients, et la classe `ThreadConnexion()`, qui servira à créer autant d'objets sockets que nécessaire pour assurer le dialogue avec chacun des clients déjà connectés.

Ces nouvelles classes seront inspirées de celles que nous avons développées pour notre serveur de *chat* dans les pages précédentes. La principale différence par rapport à celui-ci est que nous devons activer un thread spécifique pour le code qui gère l'attente et la prise en charge des connexions clientes, afin que l'application principale puisse faire autre chose pendant ce temps.

À partir de là, notre plus gros travail consistera à *développer un protocole de communication* pour le dialogue entre le serveur et ses clients. De quoi est-il question ? Tout simplement de définir la teneur des messages que vont s'échanger les machines connectées. Rassurez-vous : la mise au point de ce « langage » peut être progressive. On commence par établir un dialogue de base, puis on y ajoute petit à petit un « vocabulaire » plus étendu.

L'essentiel de ce travail peut être accompli en s'aidant du logiciel client développé précédemment pour le système de *chat*. On se sert de celui-ci pour envoyer des « ordres » au serveur en cours de développement, et on corrige celui-ci jusqu'à ce qu'il « obéisse » : en clair, les procédures que l'on met en place progressivement sur le serveur sont testées au fur et à mesure, en réponse aux messages correspondants émis « à la main » à partir du client.

Protocole de communication

Il va de soi que le protocole décrit ci-après est tout à fait arbitraire. Il serait parfaitement possible de choisir d'autres conventions complètement différentes. Vous pouvez bien évidemment critiquer les choix effectués, et vous souhaitez peut-être même les remplacer par d'autres, plus efficaces ou plus simples.

Vous savez déjà que les messages échangés sont de simples chaînes de caractères. Prévoyant que certains de ces messages devront transmettre plusieurs informations à la fois, nous avons décidé que chacun d'eux pourrait comporter plusieurs champs, que nous séparerons à l'aide de virgules. Lors de la réception de l'un quelconque de ces messages, nous pourrons alors aisément récupérer tous ses composants dans une liste, à l'aide de la méthode intégrée `split()`.

Voici un exemple de dialogue type, tel qu'il peut être suivi du côté d'un client. Les messages entre astérisques sont ceux qui sont reçus du serveur ; les autres sont ceux qui sont émis par le client lui-même :

```
1 *serveur OK*
2 client OK
3 *canons,Thread-3;104;228;1;dark red,Thread-2;454;166;-1;dark blue,*
4 OK
5 *nouveau_canon,Thread-4,481,245,-1,dark green,le_vôtre*
6 orienter,25,
7 feu
8 *mouvement_de,Thread-4,549,280,*
9 feu
10 *mouvement_de,Thread-4,504,278,*
11 *scores,Thread-4;1,Thread-3;-1,Thread-2;0,*
12 *angle,Thread-2,23,*
13 *angle,Thread-2,20,*
14 *tir_de,Thread-2,*
15 *mouvement_de,Thread-2,407,191,*
16 *départ_de,Thread-2*
17 *nouveau_canon,Thread-5,502,276,-1,dark green*
```

Lorsqu'un nouveau client démarre, il envoie une requête de connexion au serveur, lequel lui expédie en retour le message : « serveur OK ». À la réception de ce dernier, le client répond alors en envoyant lui-même : « client OK ». Ce premier échange de politesses n'est pas absolument indispensable, mais il permet de vérifier que la communication passe bien dans les deux sens. Étant donc averti que le client est prêt à travailler, le serveur lui expédie alors une description des canons déjà présents dans le jeu (éventuellement aucun) : identifiant, emplacement sur le canevas, orientation et couleur (ligne 3).

En réponse à l'accusé de réception du client (ligne 4), le serveur installe un nouveau canon dans l'espace de jeu, puis il signale les caractéristiques de cette installation non seulement au client qui l'a provoquée, mais également à tous les autres clients connectés. Le message expédié au nouveau client comporte cependant une différence (car c'est lui le propriétaire de ce nouveau canon) : en plus des caractéristiques du canon, qui sont fournies à tout le monde, il comporte un champ supplémentaire contenant simplement « le_vôtre » (comparez par exemple la ligne 5 avec la ligne 17, laquelle signale la connexion d'un autre joueur). Cette indication supplémentaire permet au client propriétaire du canon de distinguer parmi plusieurs messages similaires éventuels, celui qui contient l'identifiant unique que lui a attribué le serveur.

Les messages des lignes 6 et 7 sont des commandes envoyées par le client (réglage de la hausse et commande de tir). Dans la version précédente du jeu, nous avons déjà convenu que les canons se déplaceraient quelque peu (et au hasard) après chaque tir. Le serveur effectue donc cette opération, et s'empresse ensuite d'en faire connaître le résultat à tous les clients connectés. Le message reçu du serveur à la ligne 8 est donc l'indication d'un tel déplacement (les coordonnées fournies sont les coordonnées résultantes pour le canon concerné).

La ligne 11 reproduit le type de message expédié par le serveur lorsqu'une cible a été touchée. Les nouveaux scores de tous les joueurs sont ainsi communiqués à tous les clients.

Les messages serveur des lignes 12, 13 et 14 indiquent les actions entreprises par un autre joueur (réglage de hausse suivi d'un tir). Cette fois encore, le canon concerné est déplacé au hasard après qu'il ait tiré (ligne 15).

Lignes 16 et 17 : lorsque l'un des clients coupe sa connexion, le serveur en avertit tous les autres, afin que le canon correspondant disparaisse de l'espace de jeu sur tous les postes. À l'inverse, de nouveaux clients peuvent se connecter à tout moment pour participer au jeu.

Remarques complémentaires

Le premier champ de chaque message indique sa teneur. Les messages envoyés par le client sont très simples : ils correspondent aux différentes actions entreprises par le joueur (modifications de l'angle de tir et commandes de feu). Ceux qui sont envoyés par le serveur sont un peu plus complexes. La plupart d'entre eux sont expédiés à tous les clients connectés, afin de les tenir informés du déroulement du jeu. En conséquence, ces messages doivent mentionner l'identifiant du joueur qui a commandé une action ou qui est concerné par un changement quelconque. Nous avons vu plus haut que ces identifiants sont des noms générés automatiquement par le gestionnaire de threads du serveur, chaque fois qu'un nouveau client se connecte.

Certains messages concernant l'ensemble du jeu contiennent plusieurs informations par champ. Dans ce cas, les différents « sous-champs » sont séparés par des points-virgules (lignes 3 et 11).

Programme serveur : première partie

Vous trouverez dans les pages qui suivent le script complet du programme serveur. Nous vous le présentons en trois morceaux successifs afin de rapprocher les commentaires du code correspondant, mais la numérotation de ses lignes est continue. Bien qu'il soit déjà relativement long et complexe, vous estimerez probablement qu'il mérite d'être encore perfectionné, notamment au niveau de la présentation générale. Nous vous laisserons le soin d'y ajouter vous-même tous les compléments qui vous sembleront utiles (par exemple, une proposition de choisir les coordonnées de la machine hôte au démarrage, une barre de menus, etc.) :

```

1 #####
2 # Jeu des bombardes - partie serveur #
3 # (C) Gérard Swinnen, Liège (Belgique)- Juillet 2004 #
4 # Licence : GPL #
5 # Avant d'exécuter ce script, vérifiez que l'adresse #
6 # IP ci-dessous soit bien celle de la machine hôte. #
7 # Vous pouvez choisir un numéro de port différent, ou #
8 # changer les dimensions de l'espace de jeu. #
9 # Dans tous les cas, vérifiez que les mêmes choix ont #
10 # été effectués pour chacun des scripts clients. #
11 #####
12
13 host, port = '192.168.0.235', 35000
14 largeur, hauteur = 700, 400 # dimensions de l'espace de jeu

```

```

15
16 from Tkinter import *
17 import socket, sys, threading, time
18 import canon03
19 from canon04 import Canon, AppBombardes
20
21 class Pupitre(canon03.Pupitre):
22     """Pupitre de pointage amélioré"""
23     def __init__(self, boss, canon):
24         canon03.Pupitre.__init__(self, boss, canon)
25
26     def tirer(self):
27         "déclencher le tir du canon associé"
28         self.appli.tir_canon(self.canon.id)
29
30     def orienter(self, angle):
31         "ajuster la hausse du canon associé"
32         self.appli.orienter_canon(self.canon.id, angle)
33
34     def valeur_score(self, sc =None):
35         "imposer un nouveau score <sc>, ou lire le score existant"
36         if sc == None:
37             return self.score
38         else:
39             self.score =sc
40             self.points.config(text = ' %s ' % self.score)
41
42     def inactiver(self):
43         "désactiver le bouton de tir et le système de réglage d'angle"
44         self.bTir.config(state =DISABLED)
45         self.regl.config(state =DISABLED)
46
47     def activer(self):
48         "activer le bouton de tir et le système de réglage d'angle"
49         self.bTir.config(state =NORMAL)
50         self.regl.config(state =NORMAL)
51
52     def réglage(self, angle):
53         "changer la position du curseur de réglage"
54         self.regl.config(state =NORMAL)
55         self.regl.set(angle)
56         self.regl.config(state =DISABLED)

```

La classe `Pupitre()` est construite par dérivation de la classe de même nom importée du module `canon03`. Elle hérite donc toutes les caractéristiques de celle-ci, mais nous devons surcharger^[4] ses méthodes `tirer()` et `orienter()` :

Dans la version monoposte du logiciel, en effet, chacun des pupitres pouvait commander directement l'objet canon correspondant. Dans cette version réseau, par contre, ce sont les clients qui contrôlent à distance le fonctionnement des canons. Par conséquent, les pupitres qui apparaissent dans la fenêtre du serveur ne peuvent être que de simples répéteurs des manœuvres effectuées par les joueurs sur chaque client. Le bouton de tir et le curseur de réglage de la hausse sont donc désactivés, mais les indications fournies obéissent aux injonctions qui leur sont adressées par l'application principale.

Cette nouvelle classe `Pupitre()` sera également utilisée telle quelle dans chaque

exemplaire du programme client. Dans la fenêtre de celui-ci comme dans celle du serveur, tous les pupitres seront affichés comme des répétiteurs, mais l'un d'entre eux cependant sera complètement fonctionnel : celui qui correspond au canon du joueur.

Toutes ces raisons expliquent également l'apparition des nouvelles méthodes : `activer()`, `desactiver()`, `reglage()` et `valeur_score()`, qui seront elles aussi invoquées par l'application principale, en réponse aux messages-instructions échangés entre le serveur et ses clients.

La classe `ThreadConnexion()` ci-dessous sert à instancier la série d'objets threads qui s'occuperont en parallèle de toutes les connexions lancées par les clients. Sa méthode `run()` contient la fonctionnalité centrale du serveur, à savoir la boucle d'instructions qui gère la réception des messages provenant d'un client particulier, lesquels entraînent chacun toute une cascade de réactions. Vous y trouverez la mise en œuvre concrète du protocole de communication décrit dans les pages précédentes.

```

58 class ThreadConnexion(threading.Thread):
59     """objet thread gestionnaire d'une connexion client"""
60     def __init__(self, boss, conn):
61         threading.Thread.__init__(self)
62         self.connexion = conn           # réf. du socket de connexion
63         self.app = boss                 # réf. de la fenêtre application
64
65     def run(self):
66         "actions entreprises en réponse aux messages reçus du client"
67         nom = self.getName()           # id. du client = nom du thread
68         while 1:
69             msgClient = self.connexion.recv(1024)
70             print ***%s** de %s % (msgClient, nom)
71             deb = msgClient.split(',')[0]
72             if deb == "fin" or deb == "":
73                 self.app.enlever_canon(nom)
74                 # signaler le départ de ce canon aux autres clients :
75                 self.app.verrou.acquire()
76                 for cli in self.app.conn_client:
77                     if cli != nom:
78                         message = "départ_de,%s" % nom
79                         self.app.conn_client[cli].send(message)
80                 self.app.verrou.release()
81                 # fermer le présent thread :
82                 break
83             elif deb == "client OK":
84                 # signaler au nouveau client les canons déjà enregistrés :
85                 msg = "canons,"
86                 for g in self.app.guns:
87                     gun = self.app.guns[g]
88                     msg = msg + "%s;%s;%s;%s;%s," % \
89                             (gun.id, gun.x1, gun.y1, gun.sens, gun.coul)
90                 self.app.verrou.acquire()
91                 self.connexion.send(msg)
92                 # attendre un accusé de réception ('OK') :
93                 self.connexion.recv(100)
94                 self.app.verrou.release()
95                 # ajouter un canon dans l'espace de jeu serveur.
96                 # la méthode invoquée renvoie les caract. du canon créé :

```

```

107         x, y, sens, coul = self.app.ajouter_canon(nom)
108         # signaler les caract. de ce nouveau canon à tous les
109         # clients déjà connectés :
110         self.app.verrou.acquire()
111         for cli in self.app.conn_client:
112             msg = "nouveau_canon,%s,%s,%s,%s,%s" % \
113                 (nom, x, y, sens, coul)
114             # pour le nouveau client, ajouter un champ indiquant
115             # que le message concerne son propre canon :
116             if cli == nom:
117                 msg = msg + ",le_vôtre"
118             self.app.conn_client[cli].send(msg)
119         self.app.verrou.release()
120     elif deb == 'feu':
121         self.app.tir_canon(nom)
122         # Signaler ce tir à tous les autres clients :
123         self.app.verrou.acquire()
124         for cli in self.app.conn_client:
125             if cli != nom:
126                 message = "tir_de,%s," % nom
127                 self.app.conn_client[cli].send(message)
128         self.app.verrou.release()
129     elif deb == "orienter":
130         t = msgClient.split(',')
131         # on peut avoir reçu plusieurs angles. utiliser le dernier:
132         self.app.orienter_canon(nom, t[-2])
133         # Signaler ce changement à tous les autres clients :
134         self.app.verrou.acquire()
135         for cli in self.app.conn_client:
136             if cli != nom:
137                 # virgule terminale, car messages parfois groupés :
138                 message = "angle,%s,%s," % (nom, t[-2])
139                 self.app.conn_client[cli].send(message)
140         self.app.verrou.release()
141
142     # Fermeture de la connexion :
143     self.connexion.close() # couper la connexion
144     del self.app.conn_client[nom] # suppr. sa réf. dans le dictionn.
145     self.app.afficher("Client %s déconnecté.\n" % nom)
146     # Le thread se termine ici

```

Synchronisation de threads concurrents à l'aide de « verrous » (thread locks)

Au cours de votre examen du code ci-dessus, vous aurez certainement remarqué la structure particulière des blocs d'instructions par lesquelles le serveur expédie un même message à tous ses clients. Considérez par exemple les lignes 74 à 80 :

La ligne 75 active la méthode `acquire()` d'un objet « verrou » qui a été créé par le constructeur de l'application principale. Cet objet est une instance de la classe `Lock()`, laquelle fait partie du module *threading* que nous avons importé en début de script. Les lignes suivantes (76 à 79) provoquent l'envoi d'un message à tous les clients connectés (sauf un). Ensuite, l'objet « verrou » est à nouveau sollicité, cette fois pour sa méthode `release()`.

A quoi cet objet « verrou » peut-il donc bien servir ? Puisqu'il est produit par une classe du module *threading*, vous pouvez deviner que son utilité concerne

les threads. En fait, de tels objets « verrous » servent à *synchroniser les threads concurrents*. De quoi s'agit-il ?

Vous savez que le serveur démarre un thread différent pour chacun des clients qui se connecte. Ensuite, tous ces threads fonctionnent en parallèle. Il existe donc un risque que de temps à autre, deux ou plusieurs de ces threads essaient d'utiliser une ressource commune en même temps.

Dans les lignes de code que nous venons de discuter, par exemple, nous avons affaire à un thread qui souhaite exploiter quasiment toutes les connexions présentes pour poster un message. Il est donc parfaitement possible que pendant ce temps, un autre thread tente d'exploiter lui aussi l'une ou l'autre de ces connexions, ce qui risque de provoquer un dysfonctionnement (en l'occurrence, la superposition chaotique de plusieurs messages).

Un tel problème de *concurrence entre threads* peut être résolu par l'utilisation d'un objet-verrou (thread lock). Un tel objet n'est créé qu'en un seul exemplaire, dans un espace de noms accessible à tous les threads concurrents. Il se caractérise essentiellement par le fait qu'il se trouve toujours dans l'un ou l'autre de deux états : soit verrouillé, soit déverrouillé. Son état initial est l'état déverrouillé.

Utilisation

Lorsqu'un thread quelconque s'apprête à accéder à une ressource commune, il active d'abord la méthode `acquire()` du verrou. Si celui-ci était dans l'état déverrouillé, il se verrouille, et le thread demandeur peut alors utiliser la ressource commune, en toute tranquillité. Lorsqu'il aura fini d'utiliser la ressource, il s'empressera cependant d'activer la méthode `release()` du verrou, ce qui le fera repasser dans l'état déverrouillé.

En effet : Si un autre thread concurrent active lui aussi la méthode `acquire()` du verrou, alors que celui-ci est dans l'état verrouillé, la méthode « ne rend pas la main », provoquant le blocage de ce thread, lequel suspend donc son activité jusqu'à ce que le verrou repasse dans l'état déverrouillé. Ceci l'empêche donc d'accéder à la ressource commune durant tout le temps où un autre thread s'en sert. Lorsque le verrou est déverrouillé, l'un des threads en attente (il peut en effet y en avoir plusieurs) reprend alors son activité, et ainsi de suite.

L'objet verrou mémorise les références des threads bloqués, de manière à n'en débloquent qu'un seul à la fois lorsque sa méthode `release()` est invoquée. Il faut donc toujours veiller à ce que chaque thread qui active la méthode `acquire()` du verrou avant d'accéder à une ressource, active également sa méthode `release()` peu après.

Pour autant que tous les threads concurrents respectent la même procédure, cette technique simple empêche donc qu'une ressource commune soit exploitée en même temps par plusieurs d'entre eux. On dira dans ce cas que les threads ont été *synchronisés*.

Programme serveur : suite et fin

Les deux classes ci-dessous complètent le script serveur. Le code implémenté dans la classe `ThreadClients()` est assez similaire à celui que nous avons développé précédemment pour le corps d'application du logiciel de « Chat ». Dans le cas présent, toutefois, nous le plaçons dans une classe dérivée de `Thread()`, parce que devons faire fonctionner ce code dans un thread indépendant de celui de l'application principale. Celui-ci est en effet déjà complètement accaparé par la boucle `mainloop()` de l'interface graphique.

La classe `AppServeur()` dérive de la classe `AppBombardes()` du module `canon04`. Nous lui avons ajouté un ensemble de méthodes complémentaires destinées à exécuter toutes les opérations qui résulteront du dialogue entamé avec les clients. Nous avons déjà signalé plus haut que les clients instancieront chacun une version dérivée de cette classe (afin de profiter des mêmes définitions de base pour la fenêtre, le canevas, etc.).

```

138 class ThreadClients(threading.Thread):
139     """objet thread gérant la connexion de nouveaux clients"""
140     def __init__(self, boss, connex):
141         threading.Thread.__init__(self)
142         self.boss = boss           # réf. de la fenêtre application
143         self.connex = connex      # réf. du socket initial
144
145     def run(self):
146         "attente et prise en charge de nouvelles connexions clientes"
147         txt = "Serveur prêt, en attente de requêtes ...\n"
148         self.boss.afficher(txt)
149         self.connex.listen(5)
150         # Gestion des connexions demandées par les clients :
151         while 1:
152             nouv_conn, adresse = self.connex.accept()
153             # Créer un nouvel objet thread pour gérer la connexion :
154             th = ThreadConnexion(self.boss, nouv_conn)
155             th.start()
156             it = th.getName()      # identifiant unique du thread
157             # Mémoriser la connexion dans le dictionnaire :
158             self.boss.enregistrer_connexion(nouv_conn, it)
159             # Afficher :
160             txt = "Client %s connecté, adresse IP %s, port %s.\n" % \
161                 (it, adresse[0], adresse[1])
162             self.boss.afficher(txt)
163             # Commencer le dialogue avec le client :
164             nouv_conn.send("serveur OK")
165
166 class AppServeur(AppBombardes):
167     """fenêtre principale de l'application (serveur ou client)"""
168     def __init__(self, host, port, larg_c, haut_c):
169         self.host, self.port = host, port
170         AppBombardes.__init__(self, larg_c, haut_c)
171         self.active = 1           # témoin d'activité
172         # veiller à quitter proprement si l'on referme la fenêtre :
173         self.bind('<Destroy>', self.fermer_threads)
174
175     def specificites(self):
176         "préparer les objets spécifiques de la partie serveur"
177         self.master.title('<<< Serveur pour le jeu des bombardes >>>')

```

```

178
179     # widget Text, associé à une barre de défilement :
180     st =Frame(self)
181     self.avis =Text(st, width =65, height =5)
182     self.avis.pack(side =LEFT)
183     scroll =Scrollbar(st, command =self.avis.yview)
184     self.avis.configure(yscrollcommand =scroll.set)
185     scroll.pack(side =RIGHT, fill =Y)
186     st.pack()
187
188     # partie serveur réseau :
189     self.conn_client = {}           # dictionn. des connexions clients
190     self.verrou =threading.Lock()   # verrou pour synchroniser threads
191     # Initialisation du serveur - Mise en place du socket :
192     connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
193     try:
194         connexion.bind((self.host, self.port))
195     except socket.error:
196         txt ="La liaison du socket à l'hôte %s, port %s a échoué.\n" %\
197             (self.host, self.port)
198         self.avis.insert(END, txt)
199         self.accueil =None
200     else:
201         # démarrage du thread guettant la connexion des clients :
202         self.accueil = ThreadClients(self, connexion)
203         self.accueil.start()
204
205     def depl_aleat_canon(self, id):
206         "déplacer aléatoirement le canon <id>"
207         x, y = AppBombardes.depl_aleat_canon(self, id)
208         # signaler ces nouvelles coord. à tous les clients :
209         self.verrou.acquire()
210         for cli in self.conn_client:
211             message = "mouvement_de,%s,%s,%s," % (id, x, y)
212             self.conn_client[cli].send(message)
213         self.verrou.release()
214
215     def goal(self, i, j):
216         "le canon <i> signale qu'il a atteint l'adversaire <j>"
217         AppBombardes.goal(self, i, j)
218         # Signaler les nouveaux scores à tous les clients :
219         self.verrou.acquire()
220         for cli in self.conn_client:
221             msg ='scores,'
222             for id in self.pupi:
223                 sc = self.pupi[id].valeur_score()
224                 msg =msg +"%s;%s," % (id, sc)
225             self.conn_client[cli].send(msg)
226         time.sleep(.5)           # pour mieux séparer les messages
227         self.verrou.release()
228
229     def ajouter_canon(self, id):
230         "instancier un canon et un pupitre de nom <id> dans 2 dictionn."
231         # on alternera ceux des 2 camps :
232         n = len(self.guns)
233         if n %2 ==0:
234             sens = -1
235         else:
236             sens = 1
237         x, y = self.coord_aleat(sens)
238         coul =( 'dark blue', 'dark red', 'dark green', 'purple',
239               'dark cyan', 'red', 'cyan', 'orange', 'blue', 'violet')[n]
240         self.guns[id] = Canon(self.jeu, id, x, y, sens, coul)

```

```

241     self.pupi[id] = Pupitre(self, self.guns[id])
242     self.pupi[id].inactiver()
243     return (x, y, sens, coul)
244
245     def enlever_canon(self, id):
246         "retirer le canon et le pupitre dont l'identifiant est <id>"
247         if self.active == 0:           # la fenêtre a été refermée
248             return
249         self.guns[id].effacer()
250         del self.guns[id]
251         self.pupi[id].destroy()
252         del self.pupi[id]
253
254     def orienter_canon(self, id, angle):
255         "régler la hausse du canon <id> à la valeur <angle>"
256         self.guns[id].orienter(angle)
257         self.pupi[id].reglage(angle)
258
259     def tir_canon(self, id):
260         "déclencher le tir du canon <id>"
261         self.guns[id].feu()
262
263     def enregistrer_connexion(self, conn, it):
264         "Mémoriser la connexion dans un dictionnaire"
265         self.conn_client[it] = conn
266
267     def afficher(self, txt):
268         "afficher un message dans la zone de texte"
269         self.avis.insert(END, txt)
270
271     def fermer_threads(self, evt):
272         "couper les connexions existantes et fermer les threads"
273         # couper les connexions établies avec tous les clients :
274         for id in self.conn_client:
275             self.conn_client[id].send('fin')
276         # forcer la terminaison du thread serveur qui attend les requêtes :
277         if self.accueil != None:
278             self.accueil._Thread__stop()
279         self.active =0           # empêcher accès ultérieurs à Tk
280
281 if __name__ == '__main__':
282     AppServeur(host, port, largeur, hauteur).mainloop()

```

Commentaires

- Ligne 173 : Il vous arrivera de temps à autre de vouloir « intercepter » l'ordre de fermeture de l'application que l'utilisateur déclenche en quittant votre programme, par exemple parce que vous voulez forcer la sauvegarde de données importantes dans un fichier, ou fermer aussi d'autres fenêtres, etc. Il suffit pour ce faire de détecter l'événement <Destroy>, comme nous le faisons ici pour forcer la terminaison de tous les threads actifs.
- Lignes 179 à 186 : Au passage, voici comment vous pouvez associer une barre de défilement (widget *Scrollbar*) à un widget *Text* (vous pouvez faire de même avec un widget *Canvas*), sans faire appel à la bibliothèque *Pmw*^[5].
- Ligne 190 : Instanciation de l'objet « verrou » permettant de synchroniser les threads.

- Lignes 202, 203 : Instanciation de l'objet thread qui attendra en permanence les demandes de connexion des clients potentiels.
- Lignes 205 à 213, 215 à 227 : Ces méthodes surchargent les méthodes de même nom héritées de leur classe parente. Elles commencent par invoquer celles-ci pour effectuer le même travail (lignes 207, 217), puis ajoutent leur fonctionnalité propre, laquelle consiste à signaler à tout le monde ce qui vient de se passer.
- Lignes 229 à 243 : Cette méthode instancie un nouveau poste de tir, chaque fois qu'un nouveau client se connecte. Les canons sont placés alternativement dans le camp de droite et dans celui de gauche, procédure qui pourrait bien évidemment être améliorée. La liste des couleurs prévues limite le nombre de clients à 10, ce qui devrait suffire.

Programme client

Le script correspondant au logiciel client est reproduit ci-après. Comme celui qui correspond au serveur, il est relativement court, parce qu'il utilise lui aussi l'importation de modules et l'héritage de classes. Le script serveur doit avoir été sauvegardé dans un fichier-module nommé *canon_serveur.py*. Ce fichier doit être placé dans le répertoire courant, de même que les fichiers-modules *canon03.py* et *canon04.py* qu'il utilise lui-même.

De ces modules ainsi importés, le présent script utilise les classes `Canon()` et `Pupitre()` à l'identique, ainsi qu'une forme dérivée de la classe `AppServeur()`. Dans cette dernière, de nombreuses méthodes ont été surchargées, afin d'adapter leur fonctionnalité. Considérez par exemple les méthodes `goal()` et `depl_aleat_canon()`, dont la variante surchargée ne fait plus rien du tout (instruction `pass`), parce que le calcul des scores et le repositionnement des canons après chaque tir ne peuvent être effectués que sur le serveur seulement.

C'est dans la méthode `run()` de la classe `ThreadSocket()` (lignes 86 à 126) que se trouve le code traitant les messages échangés avec le serveur. Nous y avons d'ailleurs laissé une instruction `print` (à la ligne 88) afin que les messages reçus du serveur apparaissent sur la sortie standard. Si vous réalisez vous-même une forme plus définitive de ce jeu, vous pourrez bien évidemment supprimer cette instruction.

```

1 #####
2 # Jeu des bombardes - partie cliente #
3 # (C) Gérard Swinnen, Liège (Belgique) - Juillet 2004 #
4 # Licence : GPL #
5 # Avant d'exécuter ce script, vérifiez que l'adresse, #
6 # le numéro de port et les dimensions de l'espace de #
7 # jeu indiquées ci-dessous correspondent exactement #
8 # à ce qui a été défini pour le serveur. #
9 #####
10
11 from Tkinter import *
```



```

12 import socket, sys, threading, time
13 from canon_serveur import Canon, Pupitre, AppServeur
14
15 host, port = '192.168.0.235', 35000
16 largeur, hauteur = 700, 400          # dimensions de l'espace de jeu
17
18 class AppClient(AppServeur):
19     def __init__(self, host, port, larg_c, haut_c):
20         AppServeur.__init__(self, host, port, larg_c, haut_c)
21
22     def specificites(self):
23         "préparer les objets spécifiques de la partie client"
24         self.master.title('<<< Jeu des bombardes >>>')
25         self.connex = ThreadSocket(self, self.host, self.port)
26         self.connex.start()
27         self.id = None
28
29     def ajouter_canon(self, id, x, y, sens, coul):
30         "instancier 1 canon et 1 pupitre de nom <id> dans 2 dictionnaires"
31         self.guns[id] = Canon(self.jeu, id, int(x), int(y), int(sens), coul)
32         self.pupi[id] = Pupitre(self, self.guns[id])
33         self.pupi[id].inactiver()
34
35     def activer_pupitre_personnel(self, id):
36         self.id = id          # identifiant reçu du serveur
37         self.pupi[id].activer()
38
39     def tir_canon(self, id):
40         r = self.guns[id].feu()          # renvoie False si enrayé
41         if r and id == self.id:
42             self.connex.signaler_tir()
43
44     def imposer_score(self, id, sc):
45         self.pupi[id].valeur_score(int(sc))
46
47     def deplacer_canon(self, id, x, y):
48         "note: les valeurs de x et y sont reçues en tant que chaînes"
49         self.guns[id].deplacer(int(x), int(y))
50
51     def orienter_canon(self, id, angle):
52         "régler la hausse du canon <id> à la valeur <angle>"
53         self.guns[id].orienter(angle)
54         if id == self.id:
55             self.connex.signaler_angle(angle)
56         else:
57             self.pupi[id].reglage(angle)
58
59     def fermer_threads(self, evt):
60         "couper les connexions existantes et refermer les threads"
61         self.connex.terminer()
62         self.active = 0          # empêcher accès ultérieurs à Tk
63
64     def depl_aleat_canon(self, id):
65         pass          # => méthode inopérante
66
67     def goal(self, a, b):
68         pass          # => méthode inopérante
69
70
71 class ThreadSocket(threading.Thread):
72     """objet thread gérant l'échange de messages avec le serveur"""
73     def __init__(self, boss, host, port):
74         threading.Thread.__init__(self)

```



```

75     self.app = boss                # réf. de la fenêtre application
76     # Mise en place du socket - connexion avec le serveur :
77     self.connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
78     try:
79         self.connexion.connect((host, port))
80     except socket.error:
81         print "La connexion a échoué."
82         sys.exit()
83     print "Connexion établie avec le serveur."
84
85     def run(self):
86         while 1:
87             msg_recu = self.connexion.recv(1024)
88             print "%s*" % msg_recu
89             # le message reçu est d'abord converti en une liste :
90             t =msg_recu.split(',')
91             if t[0] =="" or t[0] == "fin":
92                 # fermer le présent thread :
93                 break
94             elif t[0] == "serveur OK":
95                 self.connexion.send("client OK")
96             elif t[0] == "canons":
97                 self.connexion.send("OK")           # accusé de réception
98                 # éliminons le 1er et le dernier élément de la liste.
99                 # ceux qui restent sont eux-mêmes des listes :
100                lc = t[1:-1]
101                # chacune est la description complète d'un canon :
102                for g in lc:
103                    s = g.split(';')
104                    self.app.ajouter_canon(s[0], s[1], s[2], s[3], s[4])
105             elif t[0] == "nouveau_canon":
106                 self.app.ajouter_canon(t[1], t[2], t[3], t[4], t[5])
107                 if len(t) >6:
108                     self.app.activer_pupitre_personnel(t[1])
109             elif t[0] == 'angle':
110                 # il se peut que l'on ait reçu plusieurs infos regroupées.
111                 # on ne considère alors que la première :
112                 self.app.orienter_canon(t[1], t[2])
113             elif t[0] == "tir_de":
114                 self.app.tir_canon(t[1])
115             elif t[0] == "scores":
116                 # éliminons le 1er et le dernier élément de la liste.
117                 # ceux qui restent sont eux-mêmes des listes :
118                 lc = t[1:-1]
119                 # chaque élément est la description d'un score :
120                 for g in lc:
121                     s = g.split(';')
122                     self.app.imposer_score(s[0], s[1])
123             elif t[0] == "mouvement_de":
124                 self.app.deplacer_canon(t[1],t[2],t[3])
125             elif t[0] == "départ_de":
126                 self.app.enlever_canon(t[1])
127
128             # Le thread <réception> se termine ici.
129             print "Client arrêté. Connexion interrompue."
130             self.connexion.close()
131
132     def signaler_tir(self):
133         self.connexion.send('feu')
134
135     def signaler_angle(self, angle):
136         self.connexion.send('orienter,%s,' % angle)
137

```

```
|138     def terminer(self):  
|139         self.connexion.send('fin')  
|140  
|141 # Programme principal :  
|142 if __name__ == '__main__':  
|143     AppClient(host, port, largeur, hauteur).mainloop()  
|-----|
```

Commentaires

- Lignes 15, 16 : Vous pouvez vous-même perfectionner ce script en lui ajoutant un formulaire qui demandera ces valeurs à l'utilisateur au cours du démarrage.
- Lignes 19 à 27 : Le constructeur de la classe parente se termine en invoquant la méthode `specificites()`. On peut donc placer dans celle-ci ce qui doit être construit différemment dans le serveur et dans les clients. (Le serveur instancie notamment un widget text qui n'est pas repris dans les clients ; l'un et l'autre démarrent des objets threads différents pour gérer les connexions).
- Lignes 39 à 42 : Cette méthode est invoquée chaque fois que l'utilisateur enfonce le bouton de tir. Le canon ne peut cependant pas effectuer des tirs en rafale. Par conséquent, aucun nouveau tir ne peut être accepté tant que l'obus précédent n'a pas terminé sa trajectoire. C'est la valeur « vraie » ou « fausse » renvoyée par la méthode `feu()` de l'objet canon qui indique si le tir a été accepté ou non. On utilise cette valeur pour ne signaler au serveur (et donc aux autres clients) que les tirs qui ont effectivement eu lieu.

Lignes 105 à 108 : Un nouveau canon doit être ajouté dans l'espace de jeu de chacun (c'est-à-dire dans le canevas du serveur, et dans le canevas de tous les clients connectés), chaque fois qu'un nouveau client se connecte. Le serveur envoie donc à ce moment un même message à tous les clients pour les informer de la présence de ce nouveau partenaire. Mais le message envoyé à celui-ci en particulier comporte un champ supplémentaire (lequel contient simplement la chaîne « `le_vôtre` »), afin que ce partenaire sache que ce message concerne son propre canon, et qu'il puisse donc activer le pupitre correspondant, tout en mémorisant l'identifiant qui lui a été attribué par le serveur (voir également les lignes 35 à 37).

Conclusions et perspectives

Cette application vous a été présentée dans un but didactique. Nous y avons délibérément simplifié un certain nombre de problèmes. Par exemple, si vous testez vous-même ces logiciels, vous constaterez que les messages échangés sont souvent rassemblés en « paquets », ce qui nécessiterait d'affiner les algorithmes mis en place pour les interpréter. De même, nous avons à peine esquissé le mécanisme fondamental du jeu : répartition des joueurs dans les deux camps, destruction des canons touchés, obstacles divers, etc. Il vous reste bien des pistes à explorer !

Exercices

1. Simplifiez le script correspondant au client de « chat » vu

précédemment, en supprimant l'un des deux objets threads. Arrangez-vous par exemple pour traiter l'émission de messages au niveau du thread principal.

2. Modifiez le jeu des bombardes vu précédemment (version monoposte), en ne gardant qu'un seul canon et un seul pupitre de pointage. Ajoutez-y une cible mobile, dont le mouvement sera géré par un objet thread indépendant (de manière à bien séparer les portions de code qui contrôlent l'animation de la cible et celle du boulet).

Solution

1. Réfléchissez !

```

2. #####
# Bombardement d'une cible mobile #
# (C) G. Swinnen - Avril 2004 - GPL #
#####

from Tkinter import *
from math import sin, cos, pi
from random import randrange
from threading import Thread

class Canon:
    """Petit canon graphique"""
    def __init__(self, boss, num, x, y, sens):
        self.boss = boss          # référence du canevas
        self.num = num            # n° du canon dans la liste
        self.x1, self.y1 = x, y   # axe de rotation du canon
        self.sens = sens          # sens de tir (-1:gauche, +1:droite)
        self.lbu = 30             # longueur de la buse
        # dessiner la buse du canon (horizontale) :
        self.x2, self.y2 = x + self.lbu * sens, y
        self.buse = boss.create_line(self.x1, self.y1,
                                     self.x2, self.y2, width =10)
        # dessiner le corps du canon (cercle de couleur) :
        self.rc = 15              # rayon du cercle
        self.corps = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
                                     y +self.rc, fill ='black')
        # pré-dessiner un obus (au départ c'est un simple point) :
        self.obus = boss.create_oval(x, y, x, y, fill='red')
        self.anim = 0
        # retrouver la largeur et la hauteur du canevas :
        self.xMax = int(boss.cget('width'))
        self.yMax = int(boss.cget('height'))

    def orienter(self, angle):
        "régler la hausse du canon"
        # rem : le paramètre <angle> est reçu en tant que chaîne.
        # il faut donc le traduire en réel, puis le convertir en radians :
        self.angle = float(angle)*2*pi/360
        self.x2 = self.x1 + self.lbu * cos(self.angle) * self.sens
        self.y2 = self.y1 - self.lbu * sin(self.angle)
        self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)

    def feu(self):
        "déclencher le tir d'un obus"
        # référence de l'objet cible :
```

```

self.cible = self.boss.master.cible
if self.anim ==0:
    self.anim =1
    # position de départ de l'obus (c'est la bouche du canon) :
    self.xo, self.yo = self.x2, self.y2
    v = 20 # vitesse initiale
    # composantes verticale et horizontale de cette vitesse :
    self.vy = -v *sin(self.angle)
    self.vx = v *cos(self.angle) *self.sens
    self.animer_obus()

def animer_obus(self):
    "animer l'obus (trajectoire balistique)"
    # positionner l'obus, en re-définissant ses coordonnées :
    self.boss.coords(self.obus, self.xo -3, self.yo -3,
                    self.xo +3, self.yo +3)

    if self.anim >0:
        # calculer la position suivante :
        self.xo += self.vx
        self.yo += self.vy
        self.vy += .5
        self.test_obstacle() # a-t-on atteint un obstacle ?
        self.boss.after(1, self.animer_obus)
    else:
        # fin de l'animation :
        self.boss.coords(self.obus, self.x1, self.y1, self.x1, self.y1)

def test_obstacle(self):
    "évaluer si l'obus a atteint une cible ou les limites du jeu"
    if self.yo >self.yMax or self.xo <0 or self.xo >self.xMax:
        self.anim =0
        return
    if self.yo > self.cible.y -3 and self.yo < self.cible.y +18 \
    and self.xo > self.cible.x -3 and self.xo < self.cible.x +43:
        # dessiner l'explosion de l'obus (cercle orange) :
        self.explo = self.boss.create_oval(self.xo -10,
            self.yo -10, self.xo +10, self.yo +10,
            fill ='orange', width =0)
        self.boss.after(150, self.fin_explosion)
        self.anim =0

def fin_explosion(self):
    "effacer le cercle d'explosion - gérer le score"
    self.boss.delete(self.explo)
    # signaler le succès à la fenêtre maîtresse :
    self.boss.master.goal()

class Pupitre(Frame):
    """Pupitre de pointage associé à un canon"""
    def __init__(self, boss, canon):
        Frame.__init__(self, bd =3, relief =GROOVE)
        self.score =0
        s =Scale(self, from_ =88, to =65,
            troughcolor ='dark grey',
            command =canon.orienter)
        s.set(45) # angle initial de tir
        s.pack(side =LEFT)
        Label(self, text ='Hausse').pack(side =TOP, anchor =W, pady =5)
        Button(self, text ='Feu !', command =canon.feue).\
            pack(side =BOTTOM, padx =5, pady =5)
        Label(self, text ="points").pack()
        self.points =Label(self, text=' 0 ', bg ='white')
        self.points.pack()

```

```

        # positionner à gauche ou à droite suivant le sens du canon :
        gd =(LEFT, RIGHT)[canon.sens == -1]
        self.pack(padx =3, pady =5, side =gd)

    def attribuerPoint(self, p):
        "incrémenter ou décrémenter le score"
        self.score += p
        self.points.config(text = ' %s ' % self.score)

class Cible:
    """objet graphique servant de cible"""
    def __init__(self, can, x, y):
        self.can = can          # référence du canevas
        self.x, self.y = x, y
        self.cible = can.create_oval(x, y, x+40, y+15, fill ='purple')

    def deplacer(self, dx, dy):
        "effectuer avec la cible un déplacement dx,dy"
        self.can.move(self.cible, dx, dy)
        self.x += dx
        self.y += dy
        return self.x, self.y

class Thread_cible(Thread):
    """objet thread gérant l'animation de la cible"""
    def __init__(self, app, cible):
        Thread.__init__(self)
        self.cible = cible      # objet à déplacer
        self.app = app         # réf. de la fenêtre d'application
        self.sx, self.sy = 6, 3 # incréments d'espace et de
        self.dt =300           # temps pour l'animation (ms)

    def run(self):
        "animation, tant que la fenêtre d'application existe"
        x, y = self.cible.deplacer(self.sx, self.sy)
        if x > self.app.xm -50 or x < self.app.xm /5:
            self.sx = -self.sx
        if y < self.app.ym /2 or y > self.app.ym -20:
            self.sy = -self.sy
        if self.app != None:
            self.app.after(int(self.dt), self.run)

    def stop(self):
        "fermer le thread si la fenêtre d'application est refermée"
        self.app =None

    def accelere(self):
        "accélérer le mouvement"
        self.dt /= 1.5

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.master.title('<<< Tir sur cible mobile >>>')
        self.pack()
        self.xm, self.ym = 600, 500
        self.jeu = Canvas(self, width =self.xm, height =self.ym,
                           bg ='ivory', bd =3, relief =SUNKEN)
        self.jeu.pack(padx =4, pady =4, side =TOP)

        # Instanciation d'un canon et d'un pupitre de pointage :
        x, y = 30, self.ym -20
        self.gun =Canon(self.jeu, 1, x, y, 1)

```

```
self.pup =Pupitre(self, self.gun)

# instantiation de la cible mobile :
self.cible = Cible(self.jeu, self.xm/2, self.ym -25)
# animation de la cible mobile, sur son propre thread :
self.tc = Thread_cible(self, self.cible)
self.tc.start()
# arrêter tous les threads lorsque l'on ferme la fenêtre :
self.bind('<Destroy>',self.fermer_threads)

def goal(self):
    "la cible a été touchée"
    self.pup.attribuerPoint(1)
    self.tc.accelere()

def fermer_threads(self, evt):
    "arrêter le thread d'animation de la cible"
    self.tc.stop()

if __name__ == '__main__':
    Application().mainloop()
```

Utilisation de threads pour optimiser les animations.

Le dernier exercice proposé à la fin de la section précédente nous suggère une méthodologie de développements d'applications qui peut se révéler particulièrement intéressante, dans le cas de jeux vidéo impliquant plusieurs animations simultanées.

En effet : si vous programmez les différents éléments animés d'un jeu comme des objets indépendants fonctionnant chacun sur son propre thread, alors non seulement vous vous simplifiez la tâche et vous améliorez la lisibilité de votre script, mais encore vous augmentez la vitesse d'exécution et donc la fluidité de ces animations. Pour arriver à ce résultat, vous devrez abandonner la technique de temporisation que vous avez exploitée jusqu'ici, mais celle que vous allez utiliser à sa place est finalement plus simple !

Temporisation des animations à l'aide de `after()`

Dans toutes les animations que nous avons décrites jusqu'à présent, le « moteur » était constitué à chaque fois par une fonction contenant la méthode `after()`, laquelle est associée d'office à tous les widgets *Tkinter*. Vous savez que cette méthode permet d'introduire une temporisation dans le déroulement de votre programme : un chronomètre interne est activé, de telle sorte qu'après un intervalle de temps convenu, le système invoque automatiquement une fonction quelconque. En général, c'est la fonction contenant `after()` qui est elle-même invoquée : on réalise ainsi une boucle récursive, dans laquelle il reste à programmer les déplacements des divers objets graphiques.

Vous devez bien comprendre que pendant l'écoulement de l'intervalle de temps programmé à l'aide de la méthode `after()`, votre application n'est pas du tout

« figée ». Vous pouvez par exemple pendant ce temps : cliquer sur un bouton, redimensionner la fenêtre, effectuer une entrée clavier, etc. Comment cela est-il rendu possible ?

Nous avons mentionné déjà à plusieurs reprises le fait que les applications graphiques modernes comportent toujours une sorte de moteur qui « tourne » continuellement en tâche de fond : ce dispositif se met en route lorsque vous activez la méthode `mainloop()` de votre fenêtre principale. Comme son nom l'indique fort bien, cette méthode met en oeuvre une boucle répétitive perpétuelle, du même type que les boucles `while` que vous connaissez bien. De nombreux mécanismes sont intégrés à ce « moteur ». L'un d'entre eux consiste à réceptionner tous les événements qui se produisent, et à les signaler ensuite à l'aide de messages appropriés aux programmes qui en font la demande (voir : Programmes pilotés par des événements^[6]), d'autres contrôlent les actions à effectuer au niveau de l'affichage, etc. Lorsque vous faites appel à la méthode `after()` d'un widget, vous utilisez en fait un mécanisme de chronométrage qui est intégré lui aussi à `mainloop()`, et c'est donc ce gestionnaire central qui déclenche l'appel de fonction que vous souhaitez, après un certain intervalle de temps.

La technique d'animation utilisant la méthode `after()` est la seule possible pour une application fonctionnant toute entière sur un seul thread, parce que c'est la boucle `mainloop()` qui dirige l'ensemble du comportement d'une telle application de manière absolue. C'est notamment elle qui se charge de redessiner tout ou partie de la fenêtre chaque fois que cela s'avère nécessaire. Pour cette raison, vous ne pouvez pas imaginer de construire un moteur d'animation qui redéfinirait les coordonnées d'un objet graphique à l'intérieur d'une simple boucle `while`, par exemple, parce que pendant tout ce temps l'exécution de `mainloop()` resterait suspendue, ce qui aurait pour conséquence que pendant tout ce temps aucun objet graphique ne serait redessiné (en particulier celui que vous souhaitez mettre en mouvement !). En fait, toute l'application apparaîtrait figée, aussi longtemps que la boucle `while` ne serait pas interrompue.

Puisqu'elle est la seule possible, c'est donc cette technique que nous avons utilisée jusqu'à présent dans tous nos exemples d'applications mono-thread. Elle comporte cependant un inconvénient gênant : du fait du grand nombre d'opérations prises en charge à chaque itération de la boucle `mainloop()`, la temporisation que l'on peut programmer à l'aide de `after()` ne peut pas être très courte. Par exemple, elle ne peut guère descendre en dessous de 15 ms sur un PC typique (processeur de type Pentium IV, $f = 1,5$ GHz). Vous devez tenir compte de cette limitation si vous souhaitez développer des animations rapides.

Un autre inconvénient lié à l'utilisation de la méthode `after()` réside dans la structure de la boucle d'animation (à savoir une fonction ou une méthode « récursive », c'est-à-dire qui s'appelle elle-même) : il n'est pas toujours simple en effet de bien maîtriser ce genre de construction logique, en particulier si l'on souhaite programmer l'animation de plusieurs objets graphiques

indépendants, dont le nombre ou les mouvements doivent varier au cours du temps.

Temporisation des animations à l'aide de `time.sleep()`

Vous pouvez ignorer les limitations de la méthode `after()` évoquées ci-dessus, si vous en confiez l'animation de vos objets graphiques à des threads indépendants. En procédant ainsi, vous vous libérez de la tutelle de `mainloop()`, et il vous est permis alors de construire des procédures d'animation sur la base de structures de boucles plus « classiques », utilisant l'instruction `while` ou l'instruction `for` par exemple.

Au cœur de chacune de ces boucles, vous devez cependant toujours veiller à insérer une temporisation pendant laquelle vous « rendez la main » au système d'exploitation (afin qu'il puisse s'occuper des autres threads). Pour ce faire, vous ferez appel à la fonction `sleep()` du module `time`. Cette fonction permet de suspendre l'exécution du thread courant pendant un certain intervalle de temps, pendant lequel les autres threads et applications continuent à fonctionner. La temporisation ainsi produite ne dépend pas de `mainloop()`, et par conséquent, elle peut être beaucoup plus courte que celle que vous autorise la méthode `after()`.

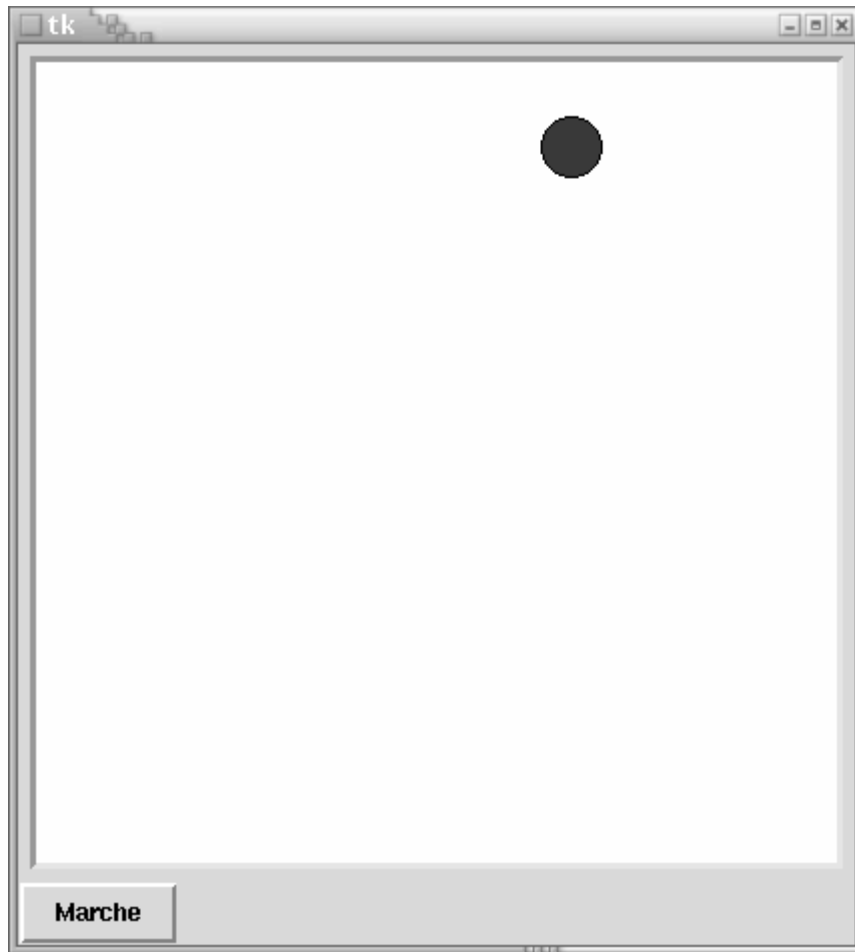
Attention : cela ne signifie pas que le rafraîchissement de l'écran sera lui-même plus rapide, car ce rafraîchissement continue à être assuré par `mainloop()`. Vous pourrez cependant accélérer fortement les différents mécanismes que vous installez vous-même dans vos procédures d'animation. Dans un logiciel de jeu, par exemple, il est fréquent d'avoir à comparer périodiquement les positions de deux mobiles (tels qu' un projectile et une cible), afin de pouvoir entreprendre une action lorsqu'ils se rejoignent (explosion, ajout de points à un score, etc.). Avec la technique d'animation décrite ici, vous pouvez effectuer beaucoup plus souvent ces comparaisons et donc espérer un résultat plus précis. De même, vous pouvez augmenter le nombre de points pris en considération pour le calcul d'une trajectoire en temps réel, et donc affiner celle-ci.

Remarque : Lorsque vous utilisez la méthode `after()`, vous devez lui indiquer la temporisation souhaitée en millisecondes, sous la forme d'un argument entier. Lorsque vous faites appel à la fonction `sleep()`, par contre, l'argument que vous transmettez doit être exprimé en secondes, sous la forme d'un réel (*float*). Vous pouvez cependant utiliser des très petites valeurs (0.0003 par ex.).

Exemple concret

Le petit script reproduit ci-dessous illustre la mise en oeuvre de cette technique, dans un exemple volontairement minimaliste. Il s'agit d'une petite application graphique dans laquelle une figure se déplace en cercle à l'intérieur d'un canevas. Son « moteur » `mainloop()` est lancé comme d'habitude sur le thread principal. Le constructeur de l'application instancie un canevas contenant le dessin d'un cercle, un bouton et un objet thread. C'est cet objet thread qui assure l'animation du dessin, mais sans faire appel à la méthode

after() d'un widget. Il utilise plutôt une simple boucle while très classique, installée dans sa méthode run().



```

1 from Tkinter import *
2 from math import sin, cos
3 import time, threading
4
5 class App(Frame):
6     def __init__(self):
7         Frame.__init__(self)
8         self.pack()
9         can =Canvas(self, width =400, height =400,
10                bg ='ivory', bd =3, relief =SUNKEN)
11         can.pack(padx =5, pady =5)
12         cercle = can.create_oval(185, 355, 215, 385, fill ='red')
13         tb = Thread_balle(can, cercle)
14         Button(self, text ='Marche', command =tb.start).pack(side =LEFT)
15         # Button(self, text ='Arrêt', command =tb.stop).pack(side =RIGHT)
16         # arrêter l'autre thread si l'on ferme la fenêtre :
17         self.bind('<Destroy>', tb.stop)
18
19 class Thread_balle(threading.Thread):
20     def __init__(self, canevas, dessin):
21         threading.Thread.__init__(self)
22         self.can, self.dessin = canevas, dessin
23         self.anim =1
24
25     def run(self):

```

```
126     a = 0.0
127     while self.anim == 1:
128         a += .01
129         x, y = 200 + 170*sin(a), 200 +170*cos(a)
130         self.can.coords(self.dessin, x-15, y-15, x+15, y+15)
131         time.sleep(0.010)
132
133     def stop(self, evt =0):
134         self.anim =0
135
136 App().mainloop()
```

Commentaires

- Lignes 13 & 14 : Afin de simplifier notre exemple au maximum, nous créons l'objet thread chargé de l'animation, directement dans le constructeur de l'application principale. Cet objet thread ne démarrera cependant que lorsque l'utilisateur aura cliqué sur le bouton « Marche », qui active sa méthode `start()` (rappelons ici que c'est cette méthode intégrée qui lancera elle-même la méthode `run()` où nous avons installé notre boucle d'animation).
- Ligne 15 : Vous ne pouvez pas redémarrer un thread qui s'est terminé. De ce fait, vous ne pouvez lancer cette animation qu'une seule fois (tout au moins sous la forme présentée ici). Pour vous en convaincre, activez la ligne n° 15 en enlevant le caractère `#` situé au début (et qui fait que Python considère qu'il s'agit d'un simple commentaire) : lorsque l'animation est lancée, un clic de souris sur le bouton ainsi mis en place provoque la sortie de la boucle `while` des lignes 27-31, ce qui termine la méthode `run()`. L'animation s'arrête, mais le thread qui la gérait s'est terminé lui aussi. Si vous essayez de le relancer à l'aide du bouton « Marche », vous n'obtenez rien d'autre qu'un message d'erreur.
- Lignes 26 à 31 : Pour simuler un mouvement circulaire uniforme, il suffit de faire varier continuellement la valeur d'un angle `a`. Le sinus et le cosinus de cet angle permettent alors de calculer les coordonnées `x` et `y` du point de la circonférence qui correspond à cet angle. À chaque itération, l'angle ne varie que d'un centième de radian seulement (environ $0,6^\circ$), et il faudra donc 628 itérations pour que le mobile effectue un tour complet. La temporisation choisie pour ces itérations se trouve à la ligne 31 : 10 millisecondes. Vous pouvez accélérer le mouvement en diminuant cette valeur, mais vous ne pourrez guère descendre en dessous de 1 milliseconde (0.001 s), ce qui n'est déjà pas si mal.

Notes

1. Dans un système d'exploitation de type Unix (comme Linux), les différents *threads* d'un même programme font partie d'un seul *processus*. Il est également possible de gérer différents processus à l'aide d'un même script Python (opération *fork*), mais l'explication de cette technique dépasse largement le cadre de ce cours.
2. Le *chat* est l'occupation qui consiste à « papoter » par l'intermédiaire d'ordinateurs. Les canadiens francophones ont proposé le terme de *clavardage*.

pour désigner ce « bavardage par claviers interposés ».

3. Que les puristes veillent bien me pardonner : j'admets volontiers que cette astuce pour forcer l'arrêt d'un thread n'est pas vraiment recommandable. Je me suis autorisé ce raccourci afin de ne pas trop alourdir ce texte, qui se veut seulement une initiation. Le lecteur exigeant pourra approfondir cette question en consultant l'un ou l'autre des ouvrages de référence mentionnés dans la bibliographie située en fin d'ouvrage.
4. Rappel : dans une classe dérivée, vous pouvez définir une nouvelle méthode avec le même nom qu'une méthode de la classe parente, afin de modifier sa fonctionnalité dans la classe dérivée. Cela s'appelle surcharger cette méthode
5. Voir : Python Mega Widgets
- 6.

Bibliothèques pour Python

Une des grandes forces du langage Python réside dans le nombre important de bibliothèques logicielles externes disponibles. Une bibliothèque est un ensemble de fonctions. Celles-ci sont regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

Celles-ci permettent de faire : du calcul numérique, du graphisme, de la programmation internet ou réseau, du formatage de texte, de la génération de documents...

Bibliothèques standards

La distribution standard de Python contient un certain nombre de bibliothèques qui ont été considérées comme suffisamment génériques pour intéresser la majorité des utilisateurs.

Leur utilisation est relativement bien expliquée dans la documentation de la distribution.

Les couches de présentation des applications (couche IHM avec wxPython, PyQt, PyKDE Tk, tkinter 3000, pyGTK, pybwidget, Pmw, TIX)

les couches controller des serveurs d'application Web (analyse HTML -htmlib, xmllib, urlParse, mimetools- Serveur d'application : Zope - Django, Turbogears, CherryPy, Plone, GGI)

les couches Modele d'accès aux données (MySQL -MySQLdb- , Oracle -dcoracle-, MS SQL Server, PostgreSQL -psycopg-, FireBird -kinterbasdb- , SybODBC, GadFly, PySqlite, Zodb- BDD objet -)

la couche de persistance XML (4Suite, PySimpleXML, XmlSerializer, Jaxml) ou spécifique à Python (Cpickle, Shelve)

les couches d'accès au middleware ICE, COM/CORBA/.NET (win32, OmniORB, Ironpython) : programmation orientée composant (pont vers des bibliothèques Fortran, C et C++)

les couches de communication standalone (port série : pySerial, port parallèle : pyParallel) , réseau (Twisted, urllib2, HTMLparser, ftplib], socket, poplib, rfc822, mailbox, mllib, imaplib, smtplib, telnet, etc.)

les couches de frameWork bas niveau (ajout de capacité de script. exemple : Boost.Python)

Les couches multimédia : traitement d'image (PIL)

Les couches utilitaires :

- de gestion de l'heure (datetime, time)
- de compression (gzip)
- de codage/décodage de données binaires (hashlib -md5, sha- , base64, binHex, binascii)
- de structure de données (decimal, deque, array, dico, list, queue, heapq)
- de parallélisme (thread)
- d'expressions régulières (re)
- de différences (difflib)
- d'accès au dll ou.so (ctypes)
- de manipulation de chaînes (string, str, StringIO)
- de parseur (standard - shlex, ConfigParser, email, parser, cmd - ou Tierce-pyBison, ples, pyparsing, ply, PyGgy, Yapps, pyLR)
- de calcul (math, numarray - tableaux multidimensionnaires - , cmath, random)
- de log (logging)
- de test unitaire (unittest)

Le déploiement se fait soit en utilisant des modules d'installation standardisés (distutils), soit en générant un exécutable qui ne nécessite plus l'existence de l'interpréteur sur les machines cibles (Windows : py2exe, Cx_freeze; Unix : freeze)

Bibliothèques alternatives

- Un site internet recense la liste des bibliothèques utilisables avec ce langage : <http://www.vex.net/parnassus>
- Les bibliothèques les plus populaires et les plus utilisées (XML, interfaces graphiques...) bénéficient de pages dédiées sur le site principal du langage : <http://cheeseshop.python.org/pypi?%3Aaction=index>

Il existe de nombreuses bibliothèques Python parmi lesquelles :

Nom	Description
CGAL ^[1]	<i>Computational Geometry Algorithms Library</i> : CGAL-Python bindings pour la CGAL library
CherryPy	Framework de développement web.
Cocos2d ^[2]	Frameworks multiplateformes pour construire des jeux 2d, démos ou des applications interactives graphiques en OpenGL (nécessite Pyglet)
CWM ^[3]	Modules de parseur pour le web sémantique
directpython ^[4]	Binding vers Direct X
Django	Framework de développement web.
epydoc ^[5]	Utilisé pour générer la documentation
FANN ^[6]	<i>Fast Artificial Neural Network Library</i> : binding Python pour FANN
Gadfly	Base de données
GENA ^[7]	Algorithme génétique en Python
GMPY ^[8]	<i>General Multiprecision PYthon</i> : interface vers la bibliothèque de précision arithmétique GNU GMP
gnuplot-py ^[9]	Bibliothèque pour s'interfacer avec gnuplot
guidata ^[10]	Librairie graphique basée sur Qt dédiée à l'affichage de données
guiqwt ^[11]	Librairie graphique basée sur Pyqwt dédiée à l'affichage de courbes
Karrigell ^[12]	framework de développement web.
Kinterbasdb	Base de données
Matplotlib ^[13]	Bibliothèque de dessin de courbe 2D (très riche)
Maximum Entropy Modeling Toolkit ^[14]	Framework qui met en œuvre le principe de l'entropie maximum
MayaVi2 ^[15]	Visualisation des données scientifiques en 2D/3D
MySQLdb	Base de données
Buzhug	Base de données
Orange ^[16]	Technique d'exploration de données, data mining
Panda 3D ^[17]	Moteur 3D
PIL ^[18]	Manipulation et traitement d'image
Pmw	Interface graphique
Pybwidget	Interface graphique
Pyscopg	Base de données

Nom ◆	Description ◆
Py2exe ^[19]	Créer un exécutable Windows pour vos scripts
pybel	Interface pour la bibliothèque Open source de CHIMIE Open Babel.
Pychinko	Implémentation de l'algorithme de Rete (pour le chaînage avant)
pyCLIPS ^[20]	Module pour scripter le système expert CLIPS
pydot ^[21]	Module pour scripter le générateur de graphique Graphviz
Pygame ^[22]	Module de création de jeu 2D
Pyglet ^[23]	module de création de jeu 2D utilisant l'openGL
pygsl ^[24]	Interface vers GNU scientific library (gsl): vecteur, matrice, transformation de fourrier, recuit simulé, algèbre linéaire...
PyGTK	Interface graphique
PyIMSL Studio ^[25]	Framework pour le calcul scientifique s'appuyant sur les bibliothèques mathématiques et statistiques IMSL
pyinstaller ^[26]	Création d'exécutable pour toute plateforme
Pylons ^[27]	Framework de développement web.
pymedia ^[28]	Module pour manipulations de fichiers wav, mp3, ogg, avi, divx, dvd, cdda, etc.
PyML ^[29]	<i>Python machine learning package</i> : framework pour l'apprentissage par les machines (data mining ...)
pyMPI ^[30]	Calcul parallèle
PyNGL ^[31]	Bibliothèque pour créer des graphes 2D
PyOgre ^[32]	Moteur 3D
PyPar ^[33]	Calcul parallèle
pyParallel ^[34]	Accès aux ports parallèles
pyro ^[35]	Middleware Python Remote Objects
pyrorobotics ^[36]	Environnement pour l'étude de la robotique et l'intelligence artificielle. Réseaux de neurones
pySerial ^[37]	Manipulation des ports séries, non intégrés par défaut, bien que très souvent utilisés dans le monde industriel
PySFML ^[38]	module de création de jeu 2D
PyUSB ^[39]	Manipulation du port USB

Nom ◆	Description ◆
PyVISA ^[40]	Contrôle des ports GPIB, RS232, and USB
pyX ^[41]	Python graphics package - Analyse de donnée
SAGE ^[42]	logiciel d'algèbre et de géométrie (alternative à Mathematica, Maple ...) géométrie, théorie des nombres, cryptographie, calcul numérique...
scipy et NumPy ^[43]	Framework pour le calcul scientifique. Interpolation, intégration (ODE integrators), algèbre linéaire (LAPACK), Interpolation, systèmes dynamiques (PyDSTool)
SCons ^[44]	Alternative puissante à make (next-generation build tool)
simPy ^[45]	Simulation de systèmes dynamiques à file d'attente
Soya ^[46]	Moteur 3D
SVGdraw ^[47]	Création d'image au format SVG (Scalable Vector Graphics)
Tkinter 3000	Interface graphique
TurboGears	framework de développement web.
Twisted ^[48]	Pluggable, asynchronous protocols (TCP, UDP, SSL/TLS, multicast, Unix sockets, HTTP, NNTP, IMAP, SSH, IRC, FTP)
VPython ^[49]	Simulation 3D basée sur OpenGL
Web2py ^[50]	framework de développement web.
wxPython ^[51]	Bibliothèque d'accès à un toolkit très puissant (en particulier pour les interfaces graphiques)
Zope ^[52]	Serveur d'application web orienté objet et base de données Objet

Références

1. <http://cgal-python.gforge.inria.fr/>
2. <http://cocos2d.org/>
3. <http://infomesh.net/2001/cwm/>
4. <http://directpython.sourceforge.net/>
5. <http://epydoc.sourceforge.net/>
6. <http://leenissen.dk/fann/index.php>
7. <http://www.madiku.org/ylrt3i0sfy/?p=1291>
8. <http://gmpy.sourceforge.net/>
9. <http://gnuplot-py.sourceforge.net/>
10. <http://pypi.python.org/pypi/guidata/>
11. <http://packages.python.org/guiqwt/>
12. <http://karrigell.sourceforge.net>

13. <http://matplotlib.sourceforge.net/>
 14. http://homepages.inf.ed.ac.uk/s0450736/maxent_toolkit.html
 15. <https://svn.enthought.com/enthought/wiki/MayaVi>
 16. <http://www.ailab.si/orange>
 17. <http://www.panda3d.org/>
 18. <http://www.pythonware.com/products/pil/>
 19. <http://www.py2exe.org/> (version pour Python 2.7)
 20. <http://pyclips.sourceforge.net/>
 21. <http://dkbza.org/pydot.html>
 22. <http://www.pygame.org/news.html>
 23. <http://www.pyglet.org/>
 24. <http://pygsl.sourceforge.net/>
 25. <http://sites.google.com/site/roguewavesoftwarefrance/produits/PyIMSL-Studio>
 26. <http://pyinstaller.python-hosting.com/>
 27. <http://www.pylonshq.com>
 28. <http://pymedia.org/>
 29. <http://pyml.sourceforge.net/>
 30. <http://pypmi.sourceforge.net/index.html>
 31. <http://www.pyngl.ucar.edu/index.shtml>
 32. <http://www.ogre3d.org/>
 33. <http://datamining.anu.edu.au/~ole/pypar/>
 34. <http://pyserial.sourceforge.net/pyparallel.html>
 35. <http://pyro.sourceforge.net/>
 36. <https://pypi.python.org/pypi/pyRobotics/1.5>
 37. <http://pyserial.sourceforge.net>
 38. <http://www.sfml-dev.org/tutorials/1.6/start-python-fr.php>
 39. <http://pyusb.berlios.de/>
 40. <http://pyvisa.sourceforge.net/>
 41. <http://pyx.sourceforge.net/>
 42. <http://modular.math.washington.edu/sage/>
 43. <http://www.scipy.org/>
 44. <http://www.scons.org/>
 45. <http://simpy.sourceforge.net/>
 46. <http://home.gna.org/oomadness/en/soya3d/index.html>
 47. <http://www2.sfk.nl/svg>
 48. <http://twistedmatrix.com/trac/>
 49. <http://www.vpython.org/>
 50. <http://www.web2py.com>
 51. <http://www.wxpython.org/>
 52. <http://www.zope.org/>
-

L'interface graphique

L'interface graphique pour Python

Comme dans tout langage de programmation, l'exécution du code ne se voit pas par l'utilisateur !

Jusqu'à maintenant, la seule relation entre le programme une fois lancé et l'utilisateur était le terminal :

Un *print* par-ci, pour donner des informations à l'utilisateur, uniquement sous forme d'une chaîne de caractères.

Un *input* par-là, afin que l'utilisateur puisse envoyer des données, toujours sous forme d'une chaîne de caractères, au programme.

Dorénavant, ayant des acquis sur la programmation dans ce langage, il va vous être enfin possible de donner du style à vos programmes en créant une véritable Interface Homme-Machine (IHM) !

Pour cela, python intègre déjà avec son interpréteur : Tkinter, qui est une bibliothèque graphique libre. Créer vos Interfaces Homme-Machine avec cette bibliothèque permettra à l'utilisateur de n'avoir aucune bibliothèque à télécharger en plus de votre code. Il sera donc très portable !

Sinon, dans les bibliothèques graphiques libres, les principaux modules sont :

- Tkinter pour Tk
- wxPython pour wxWidgets
- PyGTK pour GTK+
- PyQt pour Qt.

Inutile d'en dire davantage ; les liens Wikipédia sont très bien documentés. Sachez qu'il n'y a pas une solution unique ; ces bibliothèques ont toutes leurs caractéristiques ; tout avis ici ne serait que subjectif ; par conséquent, la seule chose qu'il reste à dire est : essayez-les, et choisissez celle que vous voudrez... L'esprit du libre, c'est aussi d'avoir le choix !

Utilisation de fenêtres et de graphismes

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

Interfaces graphiques (GUI)

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

Avec Python, la bibliothèque graphique la plus utilisée jusqu'à présent est la bibliothèque *Tkinter*, qui est une adaptation de la bibliothèque *Tk* développée à l'origine pour le langage *Tcl*. Plusieurs autres bibliothèques graphiques fort intéressantes ont été proposées pour Python : *wxPython*, *pyQT*, *pyGTK*, etc. Il existe également des possibilités d'utiliser les bibliothèques de *widgets Java* et les *MFC* de *Windows*.

Dans le cadre de ces notes, nous nous limiterons cependant à *Tkinter*, dont il existe fort heureusement des versions similaires (et gratuites) pour les plates-formes Linux, Windows et Mac.

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 :

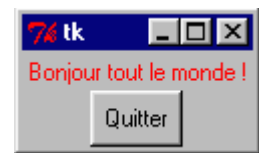
```
from Tkinter import *
```

Attention, suivant les versions de Python (a priori à partir de la version 3)

l'appel du module Tkinter se fait avec un t minuscule :

```
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*^[1] typiques : un bout de texte (ou *label*) et un bouton (ou *button*).



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

Note : Suivant la version de Python utilisée, vous verrez déjà apparaître la fenêtre d'application immédiatement après avoir entré la deuxième commande de cet exemple, ou bien seulement après la septième^[2].

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

1. Comme cela a déjà été expliqué précédemment, il est aisé de construire différents *modules* Python, qui contiendront des scripts, des définitions de fonctions, des classes d'objets, etc. On peut alors importer tout ou partie de ces modules dans n'importe quel programme, ou même dans l'interpréteur fonctionnant en mode interactif (c'est-à-dire directement à la ligne de commande). C'est ce que nous faisons à la première ligne de notre exemple : `from Tkinter import *` consiste à importer toutes les *classes* contenues dans le module *Tkinter*. Nous devrons de plus en plus souvent parler de ces *classes*. En programmation, on appelle ainsi des *générateurs d'objets*, lesquels sont eux-mêmes des morceaux de programmes réutilisables. Nous n'allons pas essayer de vous fournir dès à présent une définition définitive et précise de ce que sont les objets et les classes, mais plutôt vous proposer d'en utiliser directement quelques-un(e)s. Nous affinerons notre compréhension petit à petit par la suite.
2. À la deuxième ligne de notre exemple : `fen1 = Tk()`, nous utilisons l'une des classes du module *Tkinter*, la classe `Tk()`, et nous en créons une *instance* (autre terme désignant un *objet* spécifique), à savoir la fenêtre `fen1`. Ce processus d'*instanciation d'un objet à partir d'une classe* est une opération fondamentale dans les techniques actuelles de programmation. Celles-ci font en effet de plus en plus souvent appel à une méthodologie que l'on appelle *programmation orientée objet* (ou *OOP : Object Oriented Programming*). La *classe* est en quelque sorte un modèle général (ou un

moule) à partir duquel on demande à la machine de construire un *objet* informatique particulier. La classe contient toute une série de définitions et d'options diverses, dont nous n'utilisons qu'une partie dans l'objet que nous créons à partir d'elle. Ainsi la classe `Tk()`, qui est l'une des classes les plus fondamentales de la bibliothèque *Tkinter*, contient tout ce qu'il faut pour engendrer différents types de fenêtres d'application, de tailles ou de couleurs diverses, avec ou sans barre de menus, etc. Nous nous en servons ici pour créer notre objet graphique de base, à savoir la fenêtre qui contiendra tout le reste. Dans les parenthèses de `Tk()`, nous pourrions préciser différentes options, mais nous laisserons cela pour un peu plus tard. L'instruction d'instanciation ressemble à une simple affectation de variable. Comprenons bien cependant qu'il se passe ici deux choses à la fois :

- *la création d'un nouvel objet*, (lequel peut être complexe et donc occuper un espace mémoire considérable)
- *l'affectation d'une variable* qui va désormais servir de référence pour manipuler l'objet^[3].

3. A la troisième ligne : `tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')`, nous créons un autre objet (un *widget*), cette fois à partir de la classe `Label()`. Comme son nom l'indique, cette classe définit toutes sortes d'*étiquettes* (ou de *libellés*). En fait, il s'agit tout simplement de fragments de texte quelconques, utilisables pour afficher des informations et des messages divers à l'intérieur d'une fenêtre. Nous efforçant d'apprendre au passage la manière correcte d'exprimer les choses, nous dirons donc que nous créons ici l'objet `tex1` par *instanciation* de la classe `Label()`. Quels arguments avons-nous donc fournis pour cette instanciation ?

- Le premier argument transmis (`fen1`), indique que le nouveau *widget* que nous sommes en train de créer *sera contenu dans un autre widget préexistant*, que nous désignons donc ici comme son « maître » : l'objet `fen1` est le *widget maître* de l'objet `tex1`. (On pourra dire aussi que l'objet `tex1` est un *widget esclave* de l'objet `fen1`).
- Les deux arguments suivants servent à préciser la forme exacte que doit prendre notre *widget*. Ce sont en effet deux *options* de création, chacune fournie sous la forme d'une chaîne de caractères : d'abord le texte de l'étiquette, ensuite sa couleur d'avant-plan (ou *foreground*, en abrégé `fg`). Ainsi le texte que nous voulons afficher est bien défini, et il doit apparaître coloré en rouge.

Nous pourrions encore préciser bien d'autres caractéristiques : la police à utiliser, ou la couleur d'arrière-plan, par exemple. Toutes ces caractéristiques ont cependant une valeur par défaut dans les définitions internes de la classe `Label()`. Nous ne devons indiquer des options que pour les caractéristiques que nous souhaitons différentes du modèle standard.

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

- Un certain nombre de données (numériques ou autres), contenues dans

des variables de types divers : on les appelle les *attributs* (ou les propriétés) de l'objet.

- Un certain nombre de procédures ou de fonctions (qui sont donc des algorithmes) : on les appelle les *méthodes* de l'objet.

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

5. A la cinquième ligne : `boul = Button(fen1, text='Quitter', command = fen1.destroy)`, nous créons notre second *widget* « esclave » : un bouton. Comme nous l'avons fait pour le *widget* précédent, nous appelons la classe `Button()` en fournissant entre parenthèses un certain nombre d'arguments. Étant donné qu'il s'agit cette fois d'un objet interactif, nous devons préciser avec l'option `command` ce qui devra se passer lorsque l'utilisateur effectuera un clic sur le bouton. Dans ce cas précis, nous actionnerons la méthode `destroy` associée à l'objet `fen1`, ce qui devrait provoquer l'effacement de la fenêtre.
6. La sixième ligne utilise la méthode `pack()` pour adapter la géométrie de la fenêtre au nouvel objet que nous venons d'y intégrer.
7. La septième ligne : `fen1.mainloop()` est très importante, parce que c'est elle qui provoque le démarrage du *réceptionnaire d'événements* associé à la fenêtre. Cette instruction est nécessaire pour que votre application soit « à l'affût » des clics de souris, des pressions exercées sur les touches du clavier, etc. C'est donc cette instruction qui « la met en marche », en quelque sorte. Comme son nom l'indique (*mainloop*), il s'agit d'une méthode de l'objet `fen1`, qui active une boucle de programme, laquelle « tournera » en permanence en tâche de fond, dans l'attente de messages émis par le système d'exploitation de l'ordinateur. Celui-ci interroge en effet sans cesse son environnement, notamment au niveau des périphériques d'entrée (souris, clavier, etc.). Lorsqu'un événement quelconque est détecté, divers messages décrivant cet événement sont expédiés aux programmes qui souhaitent en être avertis. Voyons cela un peu plus en détail.

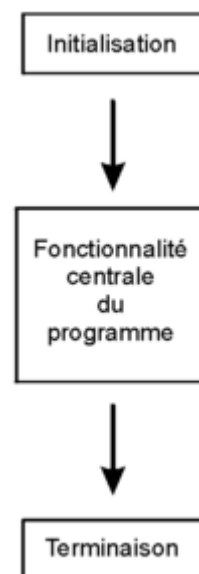
Programmes pilotés par des événements

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

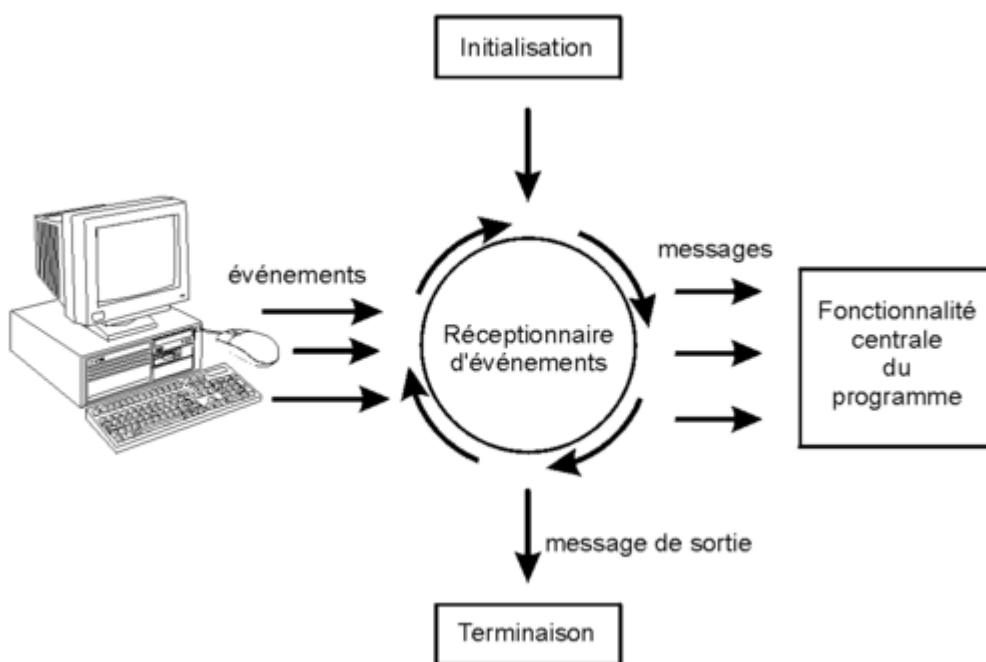
Tous les programmes d'ordinateur comportent grosso-modo trois phases principales : *une phase d'initialisation*, laquelle contient les instructions qui préparent le travail à effectuer (appel des modules externes nécessaires, ouverture de fichiers, connexion à un serveur de bases de données ou à l'internet, etc.), *une phase centrale* où l'on trouve la véritable fonctionnalité du

programme (c'est-à-dire tout ce qu'il est censé faire : afficher des données à l'écran, effectuer des calculs, modifier le contenu d'un fichier, imprimer, etc.), et enfin *une phase de terminaison* qui sert à clôturer « proprement » les opérations (c'est-à-dire fermer les fichiers restés ouverts, couper les connexions externes, etc.)

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



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



Ce *réceptionnaire d'événements* scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et réagit immédiatement lorsqu'un événement y est détecté.

Un tel événement peut être une action quelconque de l'utilisateur : déplacement de la souris, appui sur une touche, etc., mais aussi un événement externe ou un automatisme (top d'horloge, par ex.)

Lorsqu'il détecte un événement, le réceptionnaire envoie un message

spécifique au programme^[4], lequel doit être conçu pour réagir en conséquence.

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

Alors que dans un programme « textuel », la phase centrale est constituée d'une suite d'instructions qui décrivent à l'avance l'ordre dans lequel la machine devra exécuter ses différentes tâches (même s'il est prévu des cheminements différents en réponse à certaines conditions rencontrées en cours de route), on ne trouve dans la phase centrale d'un programme avec interface graphique qu'un ensemble de fonctions indépendantes. Chacune de ces fonctions est appelée spécifiquement lorsqu'un événement particulier est détecté par le système d'exploitation : elle effectue alors le travail que l'on attend du programme en réponse à cet événement, et rien d'autre^[5].

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

S'il arrive d'autres événements, il peut donc se faire qu'une seconde fonction (ou une 3e, une 4e, ...) soit activée et commence à effectuer son travail « en parallèle » avec la première qui n'a pas encore terminé le sien. **Erreur de référence : Balise fermante </ref> manquante pour la balise <ref>.**, pour modifier son attribut text. L'attribut en question reçoit donc ici une nouvelle valeur, déterminée par ce que nous avons écrit à la droite du signe égale : il s'agit en l'occurrence d'une chaîne de caractères construite dynamiquement, à l'aide de deux fonctions intégrées dans Python : `eval()` et `str()`, et d'une méthode associée à un widget *Tkinter* : la méthode `get()`.

`eval()` fait appel à l'interpréteur pour évaluer une expression Python qui lui est transmise dans une chaîne de caractères. Le résultat de l'évaluation est fourni en retour. Exemple :

```
-----  
'chaîne = "(25 + 8)/3" # chaîne contenant une expression mathématique  
'res = eval(chaîne)   # évaluation de l'expression contenue dans la chaîne  
print res +5         # => le contenu de la variable res est numérique  
-----
```

`str()` transforme une expression numérique en chaîne de caractères. Nous devons faire appel à cette fonction parce que la précédente renvoie une valeur numérique, que nous convertissons à nouveau en chaîne de caractères pour pouvoir l'incorporer au message « Résultat = ».

`get()` est une méthode associée aux widgets de la classe `Entry`. Dans notre petit programme exemple, nous utilisons un widget de ce type pour permettre à

l'utilisateur d'entrer une expression numérique quelconque à l'aide de son clavier. La méthode `get()` permet en quelque sorte « d'extraire » du widget « `entree` » la chaîne de caractères qui lui a été fournie par l'utilisateur.

Le corps du programme principal contient la phase d'initialisation, qui se termine par la mise en route de l'observateur d'événements (*mainloop*). On y trouve l'instanciation d'une fenêtre `Tk()`, contenant un widget « `chaine` » instancié à partir de la classe `Label()`, et un widget « `entree` » instancié à partir de la classe `Entry()`.

Attention, à présent : afin que ce dernier widget puisse vraiment faire son travail, c'est-à-dire transmettre au programme l'expression que l'utilisateur y aura encodée, *nous lui associons un événement* à l'aide de la méthode `bind()`^[6] :

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

Cette instruction signifie : « Lier l'événement “pression sur la touche Return” à l'objet `<entree>`, le gestionnaire de cet événement étant la fonction `<evaluer>` ».

L'événement à prendre en charge est décrit dans une chaîne de caractères spécifique (dans notre exemple, il s'agit de la chaîne `<Return>`). Il existe un grand nombre de ces événements (mouvements et clics de la souris, enfoncement des touches du clavier, positionnement et redimensionnement des fenêtres, passage au premier plan, etc.). Vous trouverez la liste des chaînes spécifiques de tous ces événements dans les ouvrages de référence traitant de *Tkinter*.

Profitons de l'occasion pour observer encore une fois la syntaxe des instructions destinées à mettre en œuvre une méthode associée à un objet :

```
objet.méthode(arguments)
```

On écrit d'abord le nom de l'objet sur lequel on désire intervenir, puis le point (qui fait office d'opérateur), puis le nom de la méthode à mettre en œuvre ; entre les parenthèses associées à cette méthode, on indique enfin les arguments qu'on souhaite lui transmettre.

Exemple graphique : détection et positionnement d'un clic de souris

Dans la définition de la fonction « `evaluer` » de l'exemple précédent, vous aurez remarqué que nous avons fourni un argument `event` (entre les parenthèses).

Cet argument est obligatoire. Lorsque vous définissez une fonction gestionnaire d'événement qui est associée à un widget quelconque à l'aide de sa méthode `bind()`, vous devez toujours l'utiliser comme premier argument. Il s'agit d'un objet Python standard, créé automatiquement, qui permet de

transmettre au gestionnaire d'événement un certain nombre d'attributs de cet événement :

- le type d'événement : déplacement de la souris, enfoncement ou relâchement de l'un de ses boutons, appui sur une touche du clavier, entrée du curseur dans une zone prédéfinie, ouverture ou fermeture d'une fenêtre, etc.
- une série de propriétés de l'événement : l'instant où il s'est produit, ses coordonnées, les caractéristiques du ou des widget(s) concerné(s), etc.

Nous n'allons pas entrer dans trop de détails. Si vous voulez bien encoder et expérimenter le petit script ci-dessous, vous aurez vite compris le principe.

```
# Détection et positionnement d'un clic de souris dans une fenêtre :
from Tkinter import *
def pointeur(event):
    chaine.configure(text = "Clic détecté en X =" + str(event.x) + \
        ", Y =" + str(event.y))
fen = Tk()
cadre = Frame(fen, width =200, height =150, bg="light yellow")
cadre.bind("<Button-1>", pointeur)
cadre.pack()
chaine = Label(fen)
chaine.pack()
fen.mainloop()
```

Le script fait apparaître une fenêtre contenant un cadre (*frame*) rectangulaire de couleur jaune pâle, dans lequel l'utilisateur est invité à effectuer des clics de souris.

La méthode `bind()` du widget cadre associe l'événement *<clic à l'aide du premier bouton de la souris>* au gestionnaire d'événement « *pointeur* ».

Ce gestionnaire d'événement peut utiliser les attributs `x` et `y` de l'objet `event` généré automatiquement par Python, pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.



Exercices

1. Modifiez le script ci-dessus de manière à faire apparaître un petit cercle rouge à l'endroit où l'utilisateur a effectué son clic (vous devrez d'abord remplacer le widget `Frame` par un widget `Canvas`).

Solution

Solution

```
def avance(event):
```

```
    can1.coords(oval, event.x-r, event.y-r, event.x+r, event.y+r)
```

```
x,y,r=10,10,10
```

```
def oval(x,y,r,coul):
```

```
    can1.create_oval(x-r,y-r,x+r,y+r, outline=coul)
```

```
fen1=Tk() fen1.title("Exercice d'animation avec Tkinter")
```

```
can1=Canvas(fen1,bg='dark grey',height=300,width=300)
```

```
oval=can1.create_oval(x,y,x+30,y+30,width=2,fill='red') can1.bind("<Button-1>",avance) can1.pack() can1.pack(side=LEFT)
```

```
Button(fen1,text='Quitter',command=fen1.quit).pack(side=BOTTOM)
```

```
fen1.mainloop() fen1.destroy()
```

Les classes de widgets Tkinter

Remarque : au long de ce cours, nous vous présenterons petit à petit le mode d'utilisation d'un certain nombre de widgets. Comprenez bien cependant qu'il n'entre pas dans nos intentions de fournir ici un manuel de référence complet sur Tkinter. Nous limiterons nos explications aux widgets qui nous semblent les plus intéressants d'un point de vue didactique, c'est-à-dire ceux qui pourront nous aider à mettre en évidence des concepts importants, tel le concept de classe.

Il existe 15 classes de base pour les widgets Tkinter :

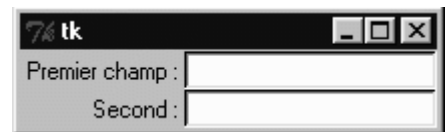
Widget	Description
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une « case à cocher » qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
Label	Un texte (ou libellé) quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher.
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « pop up » apparaissant n'importe où à la suite d'un clic.
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un « bouton radio » donne la valeur correspondante à la variable, et "vide" tous les autres boutons radio associés à la même variable.
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
Scrollbar	« ascenseur » ou « barre de défilement » que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formaté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
Toplevel	Une fenêtre affichée séparément, « par-dessus ».

Ces classes de widgets intègrent chacune un grand nombre de méthodes et on peut aussi leur associer (lier) des événements. En outre, tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode `grid()`, la méthode `pack()` et la méthode `place()`.

L'utilité de ces méthodes apparaît clairement lorsque l'on s'efforce de réaliser des programmes *portables* (c'est-à-dire susceptibles de fonctionner indifféremment sur des systèmes d'exploitation aussi différents que Unix, MacOS ou Windows), et dont les fenêtres soient redimensionnables.

Utilisation de la méthode `grid()` pour contrôler la disposition des widgets

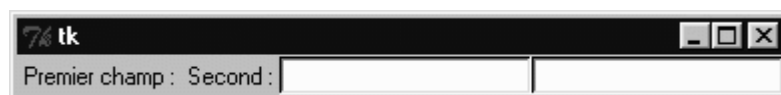
Jusqu'à présent, nous avons toujours disposé les widgets dans leur fenêtre, à l'aide de la méthode `pack()`. Cette méthode présentait l'avantage d'être extraordinairement simple, mais elle ne nous donnait pas beaucoup de liberté pour disposer les widgets à notre guise. Comment faire, par exemple, pour obtenir la fenêtre ci-contre ?



Nous pourrions effectuer un certain nombre de tentatives en fournissant à la méthode `pack()` des arguments de type « `side =` », comme nous l'avons déjà fait précédemment, mais cela ne nous mène pas très loin. Essayons par exemple :

```
from Tkinter import *
fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.pack(side =LEFT)
txt2.pack(side =LEFT)
entr1.pack(side =RIGHT)
entr2.pack(side =RIGHT)
fen1.mainloop()
```

... mais le résultat n'est pas vraiment celui que nous recherchions !!! :



Pour mieux comprendre comment fonctionne la méthode `pack()`, vous pouvez encore essayer différentes combinaisons d'options, telles que `side =TOP`, `side =BOTTOM`, pour chacun de ces quatre widgets. Mais vous n'arriverez certainement pas à obtenir ce qui vous a été demandé. Vous pourriez peut-être y parvenir en définissant deux widgets `Frame()` supplémentaires, et en y

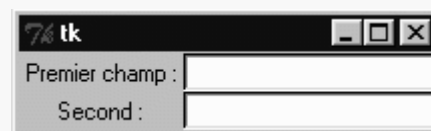
incorporant ensuite séparément les widgets `Label()` et `Entry()`. Cela devient fort compliqué.

Il est temps que nous apprenions à utiliser une autre approche du problème. Veuillez donc analyser le script ci-dessous : il contient en effet (presque) la solution :

```

from Tkinter import *
fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.grid(row =0)
txt2.grid(row =1)
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
fen1.mainloop()

```



Dans ce script, nous avons donc remplacé la méthode `pack()` par la méthode `grid()`. Comme vous pouvez le constater, l'utilisation de la méthode `grid()` est très simple. Cette méthode considère la fenêtre comme un tableau (ou une grille). Il suffit alors de lui indiquer dans quelle ligne (*row*) et dans quelle colonne (*column*) de ce tableau on souhaite placer les widgets. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre 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 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 souhaitons les aligner l'une et l'autre par la droite. Pour obtenir ce résultat, il nous suffit d'ajouter un argument dans l'appel de la méthode `grid()` utilisée pour ces widgets. L'option `sticky` peut prendre l'une des quatre valeurs `N`, `S`, `W`, `E` (les quatre points cardinaux en anglais). En fonction de cette valeur, on obtiendra un alignement des widgets par le haut, par le bas, par la gauche ou par la droite. Remplacez donc les deux premières instructions `grid()` du script par :

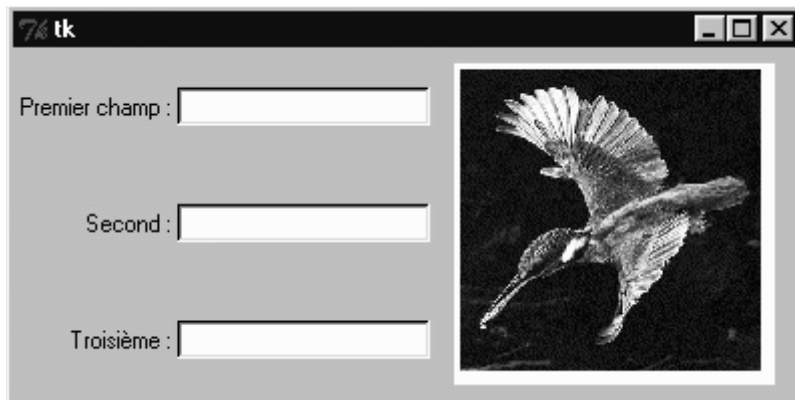
```

txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)

```

... et vous atteindrez enfin exactement le but recherché.

Analysons à présent la fenêtre suivante :



Cette fenêtre comporte 3 colonnes : une première avec les 3 chaînes de caractères, une seconde avec les 3 champs d'entrée, et une troisième avec l'image. Les deux premières colonnes comportent chacune 3 lignes, mais l'image située dans la dernière colonne s'étale en quelque sorte sur les trois.

Le code correspondant est le suivant :

```

from Tkinter import *

fen1 = Tk()

# création de widgets 'Label' et 'Entry' :
txt1 = Label(fen1, text='Premier champ :')
txt2 = Label(fen1, text='Second :')
txt3 = Label(fen1, text='Troisième :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
entr3 = Entry(fen1)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width=160, height=160, bg='white')
photo = PhotoImage(file='Martin_P.gif')
item = can1.create_image(80, 80, image=photo)

# Mise en page à l'aide de la méthode 'grid' :
txt1.grid(row=1, sticky=E)
txt2.grid(row=2, sticky=E)
txt3.grid(row=3, sticky=E)
entr1.grid(row=1, column=2)
entr2.grid(row=2, column=2)
entr3.grid(row=3, column=2)
can1.grid(row=1, column=3, rowspan=3, padx=10, pady=5)

# démarrage :
fen1.mainloop()

```

Pour pouvoir faire fonctionner ce script, il vous faudra probablement remplacer le nom du fichier image (*Martin_P.gif*) par le nom d'une image de votre choix. Attention : la bibliothèque *Tkinter* standard n'accepte qu'un petit nombre de formats pour cette image. Choisissez de préférence le format GIF.

Nous pouvons remarquer un certain nombre de choses dans ce script :

1. La technique utilisée pour incorporer une image :
Tkinter ne permet pas d'insérer directement une image dans une fenêtre. Il faut d'abord installer un canevas, et ensuite positionner l'image dans celui-ci. Nous avons opté pour un canevas de couleur blanche, afin de pouvoir le distinguer de la fenêtre. Vous pouvez remplacer le paramètre `bg = 'white'` par `bg = 'gray'` si vous souhaitez que le canevas devienne invisible. Étant donné qu'il existe de nombreux types d'images, nous devons en outre déclarer l'objet image comme étant un bitmap GIF, à partir de la classe `PhotoImage()` ^[7].
2. La ligne où nous installons l'image dans le canevas est la ligne :

```
item = can1.create_image(80, 80, image =photo)
```


Pour employer un vocabulaire correct, nous dirons que nous utilisons ici la *méthode* `create_image()` associée à l'*objet* `can1` (lequel objet est lui-même une *instance* de la *classe* `Canvas`). Les deux premiers arguments transmis (80, 80) indiquent les coordonnées x et y du canevas où il faut placer le centre de l'image. (Les dimensions du canevas étant de 160x160, notre choix aboutira donc à un centrage de l'image au milieu du canevas).
3. La numérotation des lignes et colonnes dans la méthode `grid()` : On peut constater que la numérotation des lignes et des colonnes dans la méthode `grid()` utilisée ici commence cette fois à partir de 1 (et non à partir de zéro comme dans le script précédent). Comme nous l'avons déjà signalé plus haut, ce choix de numérotation est tout à fait libre. On pourrait tout aussi bien numéroter : 5, 10, 15, 20... puisque *Tkinter* ignore les lignes et les colonnes vides. Numéroter à partir de 1 augmente probablement la lisibilité de notre code.
4. Les arguments utilisés avec `grid()` pour positionner le canevas :

```
can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)
```


Les deux premiers arguments indiquent que le canevas sera placé dans la première ligne de la troisième colonne. Le troisième (`rowspan =3`) indique qu'il pourra « s'étaler » sur trois lignes. Les deux derniers (`padx =10, pady =5`) indiquent la dimension de l'espace qu'il faut réserver autour de ce widget (en largeur et en hauteur).
5. Et tant que nous y sommes, profitons de cet exemple de script que nous avons déjà bien décortiqué, pour apprendre à simplifier quelque peu notre code...

Composition d'instructions pour écrire un code plus compact

Du fait que Python est un langage de programmation de haut niveau, il est souvent possible (et souhaitable) de retravailler un script afin de le rendre plus compact.

Vous pouvez par exemple assez fréquemment utiliser la composition d'instructions pour appliquer la méthode de mise en page des widgets (`grid()`, `pack()` ou `place()`) au moment même où vous créez ces widgets. Le code correspondant devient alors un peu plus simple, et parfois plus lisible. Vous pouvez par exemple remplacer les deux lignes :


```
txt1 = Label(fen1, text = 'Premier champ :')
txt1.grid(row =1, sticky =E)
```

du script précédent par une seule, telle que :

```
Label(fen1, text = 'Premier champ :').grid(row =1, sticky =E)
```

Dans cette nouvelle écriture, vous pouvez constater que nous faisons l'économie de la variable intermédiaire `txt1`. Nous avons utilisé cette variable pour bien dégager les étapes successives de notre démarche, mais elle n'est pas toujours indispensable. Le simple fait d'invoquer la classe `Label()` provoque en effet l'instanciation d'un objet de cette classe, même si l'on ne mémorise pas la référence de cet objet dans une variable (*Tkinter* la conserve de toute façon dans sa représentation interne de la fenêtre). Si l'on procède ainsi, la référence est perdue pour le restant du script, mais elle peut tout de même être transmise à une méthode de mise en page telle que `grid()` au moment même de l'instanciation, en une seule instruction composée. Voyons cela un peu plus en détail :

Jusqu'à présent, nous avons créé des objets divers (par instanciation à partir d'une classe quelconque), en les affectant à chaque fois à des variables. Par exemple, lorsque nous avons écrit :

```
txt1 = Label(fen1, text = 'Premier champ :')
```

Nous avons créé une instance de la classe `Label()`, que nous avons assignée à la variable `txt1`.

La variable `txt1` peut alors être utilisée pour faire référence à cette instance, partout ailleurs dans le script, mais dans les faits nous ne l'utilisons qu'une seule fois pour lui appliquer la méthode `grid()`, le widget dont il est question n'étant rien d'autre qu'une simple étiquette descriptive. Or, créer ainsi une nouvelle variable pour n'y faire référence ensuite qu'une seule fois (et directement après sa création) n'est pas une pratique très recommandable, puisqu'elle consiste à réserver inutilement un certain espace mémoire.

Lorsque ce genre de situation se présente, il est plus judicieux d'utiliser la composition d'instructions. Par exemple, on préférera le plus souvent remplacer les deux instructions :

```
somme = 45 + 72
print somme
```

par une seule instruction composée, telle que :

```
print 45 + 72
```

On fait ainsi l'économie d'une variable.

De la même manière, lorsque l'on met en place des widgets auxquels on ne souhaite plus revenir par après, comme c'est souvent le cas pour les widgets de la classe `Label()`, on peut en général appliquer la méthode de mise en page (`grid()`, `pack()` ou `place()`) directement au moment de la création du widget, en une seule instruction composée.

Cela s'applique seulement aux widgets qui ne sont plus référencés après qu'on les ait créés. *Tous les autres doivent impérativement être assignés à des variables*, afin que l'on puisse encore interagir avec eux ailleurs dans le script.

Et dans ce cas, il faut obligatoirement utiliser deux instructions distinctes, l'une pour instancier le widget et l'autre pour lui appliquer ensuite la méthode de mise en page. Vous ne pouvez pas, par exemple, construire une instruction composée telle que :

```
entree = Entry(fen1).pack() # faute de programmation !!!
```

En apparence, cette instruction devrait instancier un nouveau widget et l'assigner à la variable `entree`, la mise en page s'effectuant dans la même opération à l'aide de la méthode `pack()`.

Dans la réalité, cette instruction produit bel et bien un nouveau widget de la classe `Entry()`, et la méthode `pack()` effectue bel et bien sa mise en page dans la fenêtre, mais la valeur qui est mémorisée dans la variable `entree` est la valeur de retour de la méthode `pack()` : *ce n'est pas la référence du widget*. Et vous ne pouvez rien faire de cette valeur de retour : il s'agit d'un objet vide : `None`.

Pour obtenir une vraie référence du widget, vous devez utiliser deux instructions :

```
entree = Entry(fen1) # instantiation du widget
entree.pack()      # application de la mise en page
```

Note : Lorsque vous utilisez la méthode `grid()`, vous pouvez simplifier encore un peu votre code, en omettant l'indication de nombreux numéros de lignes et de colonnes. À partir du moment où c'est la la méthode `grid()` qui est utilisée pour positionner les widgets, *Tkinter* considère en effet qu'il existe forcément des lignes et des colonnes¹¹. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant est placé dans la première case vide disponible.

Le script ci-dessous intègre les simplifications que nous venons d'expliquer :

```

from Tkinter import *
fen1 = Tk()

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

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg = 'white')
photo = PhotoImage(file = 'Martin_P.gif')
can1.create_image(80,80, image =photo)
can1.grid(row =0, column =2, rowspan =4, padx =10, pady =5)

# démarrage :
fen1.mainloop()

```

Modification des propriétés d'un objet - Animation

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

Veillez donc écrire, tester, puis analyser le script ci-dessous :

```

from Tkinter import *

# procédure générale de déplacement :
def avance(gd, hb):
    global x1, y1
    x1, y1 = x1 +gd, y1 +hb
    can1.coords(ovall, x1, y1, x1+30, y1+30)

# gestionnaires d'événements :
def depl_gauche():
    avance(-10, 0)

def depl_droite():
    avance(10, 0)

def depl_haut():
    avance(0, -10)

def depl_bas():
    avance(0, 10)

```

```

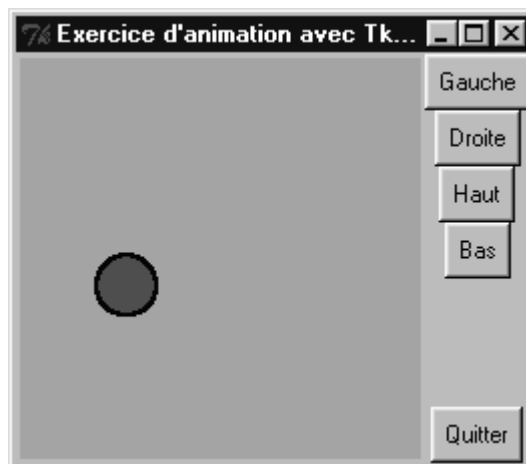
#----- Programme principal -----
# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10      # coordonnées initiales

# Création du widget principal ("maître") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")

# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=300,width=300)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
can1.pack(side=LEFT)
Button(fen1,text='Quitter',command=fen1.quit).pack(side=BOTTOM)
Button(fen1,text='Gauche',command=depl_gauche).pack()
Button(fen1,text='Droite',command=depl_droite).pack()
Button(fen1,text='Haut',command=depl_haut).pack()
Button(fen1,text='Bas',command=depl_bas).pack()

# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()

```



Le corps de ce programme reprend de nombreuses éléments connus : nous y créons une fenêtre `fen1`, dans laquelle nous installons un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle. Veuillez remarquer au passage que nous n'instancions pas les widgets boutons dans des variables (c'est inutile, puisque nous n'y faisons plus référence par après) : nous devons donc appliquer la méthode `pack()` directement au moment de la création de ces objets.

La vraie nouveauté de ce programme réside dans la fonction `avance()` définie au début du script. Chaque fois qu'elle sera appelée, cette fonction redéfinira les coordonnées de l'objet « cercle coloré » que nous avons installé dans le canevas, ce qui provoquera l'animation de cet objet.

Cette manière de procéder est tout à fait caractéristique de la programmation « orientée objet » :

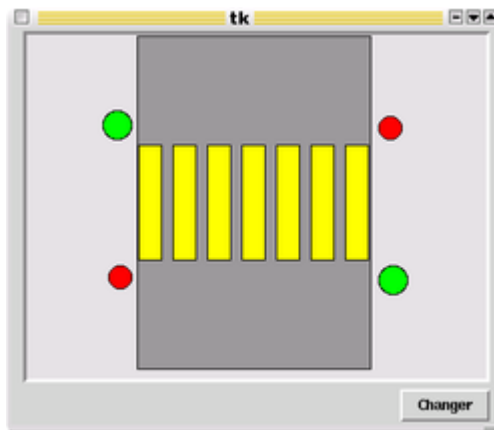
On commence par créer des objets, et puis *on agit sur ces objets en modifiant leurs propriétés*, par l'intermédiaire de méthodes.

En programmation procédurale « à l'ancienne » (c'est-à-dire sans utilisation d'objets), on anime des figures en les effaçant à un endroit pour les redessiner ensuite un petit peu plus loin. En programmation « orientée objet », par contre, ces tâches sont prises en charge automatiquement par les classes dont les objets dérivent, et il ne faut donc pas perdre son temps à les reprogrammer.

Exercices

1. Écrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas on verra deux cercles (de tailles et de couleurs différentes), qui sont censés représenter deux astres. Des boutons doivent permettre de les déplacer à volonté tous les deux dans toutes les directions. Sous le canevas, le programme doit afficher en permanence : a) la distance séparant les deux astres ; b) la force gravitationnelle qu'ils exercent l'un sur l'autre (Penser à afficher en haut de fenêtre les masses choisies pour chacun d'eux, ainsi que l'échelle des distances). Dans cet exercice, vous utiliserez évidemment la loi universelle de la gravitation de Newton.
2. 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.
3. 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 !).
4. Même exercice avec des charges électriques (loi de Coulomb). Donner cette fois une possibilité de choisir le signe des charges.
5. Écrivez un petit programme qui fait apparaître une fenêtre avec deux champs : l'un indique une température en degrés *Celsius*, et l'autre la même température exprimée en degrés *Fahrenheit*. Chaque fois que l'on change une quelconque des deux températures, l'autre est corrigée en conséquence. Pour convertir les degrés *Fahrenheit* en *Celsius* et vice-versa, on utilise la formule $T_F = T_C \times 1,80 + 32$. (cf. manuel de Physique). Revoyez aussi le petit programme concernant la calculatrice simplifiée.
6. 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.
7. Améliorez le programme ci-dessus pour que la balle décrive cette fois une trajectoire circulaire ou elliptique dans le canevas (lorsque l'on clique continuellement). Note : pour arriver au résultat escompté, vous devrez nécessairement définir une variable qui représentera l'angle décrit, et utiliser les fonctions sinus et cosinus pour

- positionner la balle en fonction de cet angle.
8. Modifiez le programme ci-dessus, de telle manière que la balle en se déplaçant laisse derrière elle une trace de la trajectoire décrite.
 9. Modifiez le programme ci-dessus de manière à tracer d'autres figures. Consultez votre professeur pour des suggestions (courbes de Lissajous).
 10. Écrivez un programme qui fasse apparaître une fenêtre avec un canevas et un bouton. Dans le canevas, tracez un rectangle gris foncé, lequel représentera une route, et par-dessus, une série de rectangles jaunes censés représenter un passage pour piétons. Ajoutez quatre cercles colorés pour figurer les feux de circulation concernant les piétons et les véhicules. Chaque utilisation du bouton devra provoquer le changement de couleur des feux :



11. Écrivez un programme qui montre un canevas dans lequel est dessiné un circuit électrique simple (générateur + interrupteur + résistance). La fenêtre doit être pourvue de champs d'entrée qui permettront de paramétrer chaque élément (c'est-à-dire choisir les valeurs des résistances et tensions). L'interrupteur doit être fonctionnel (Prévoyez un bouton « Marche/arrêt » pour cela). Des « étiquettes » doivent afficher en permanence les tensions et intensités résultant des choix opérés par l'utilisateur.

Solution

```

1. # Simulation du phénomène de gravitation universelle

from Tkinter import *
from math import sqrt

def distance(x1, y1, x2, y2):
    "distance séparant les points x1,y1 et x2,y2"
    d = sqrt((x2-x1)**2 + (y2-y1)**2) # théorème de Pythagore
    return d

def forceG(m1, m2, di):
    "force de gravitation s'exerçant entre m1 et m2 pour une distance di"
    return m1*m2*6.67e-11/di**2 # loi de Newton

def avance(n, gd, hb):
    "déplacement de l'astre n, de gauche à droite ou de haut en bas"

```

```

global x, y, step
# nouvelles coordonnées :
x[n], y[n] = x[n] +gd, y[n] +hb
# déplacement du dessin dans le canevas :
can.coords(astre[n], x[n]-10, y[n]-10, x[n]+10, y[n]+10)
# calcul de la nouvelle interdistance :
di = distance(x[0], y[0], x[1], y[1])
# conversion de la distance "écran" en distance "astronomique" :
diA = di*1e9 # (1 pixel => 1 million de km)
# calcul de la force de gravitation correspondante :
f = forceG(m1, m2, diA)
# affichage des nouvelles valeurs de distance et force :
valDis.configure(text="Distance = " +str(diA) + " m")
valFor.configure(text="Force = " +str(f) + " N")
# adaptation du "pas" de déplacement en fonction de la distance :
step = di/10

def gauche1():
    avance(0, -step, 0)

def droite1():
    avance(0, step, 0)

def haut1():
    avance(0, 0, -step)

def bas1():
    avance(0, 0, step)

def gauche2():
    avance(1, -step, 0)

def droite2():
    avance (1, step, 0)

def haut2():
    avance(1, 0, -step)

def bas2():
    avance(1, 0, step)

# Masses des deux astres :
m1 = 6e24 # (valeur de la masse de la terre, en kg)
m2 = 6e24 #
astre = [0]*2 # liste servant à mémoriser les références des dessins
x =[50., 350.] # liste des coord. X de chaque astre (à l'écran)
y =[100., 100.] # liste des coord. Y de chaque astre
step =10 # "pas" de déplacement initial

# Construction de la fenêtre :
fen = Tk()
fen.title(' Gravitation universelle suivant Newton')
# Libellés :
valM1 = Label(fen, text="M1 = " +str(m1) + " kg")
valM1.grid(row =1, column =0)
valM2 = Label(fen, text="M2 = " +str(m2) + " kg")
valM2.grid(row =1, column =1)
valDis = Label(fen, text="Distance")
valDis.grid(row =3, column =0)
valFor = Label(fen, text="Force")
valFor.grid(row =3, column =1)
# Canevas avec le dessin des 2 astres:
can = Canvas(fen, bg ="light yellow", width =400, height =200)

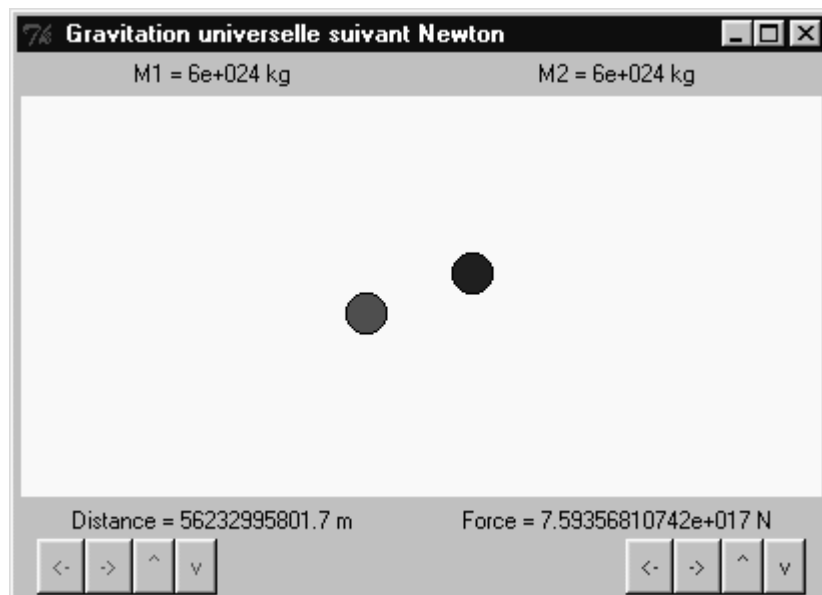
```

```

can.grid(row =2, column =0, columnspan =2)
astre[0] = can.create_oval(x[0]-10, y[0]-10, x[0]+10, y[0]+10,
                          fill = "red", width =1)
astre[1] = can.create_oval(x[1]-10, y[1]-10, x[1]+10, y[1]+10,
                          fill = "blue", width =1)
# 2 groupes de 4 boutons, chacun installé dans un cadre (frame) :
fra1 = Frame(fen)
fra1.grid(row =4, column =0, sticky =W, padx =10)
Button(fra1, text="<-", fg = 'red', command =gauche1).pack(side =LEFT)
Button(fra1, text="->", fg = 'red', command =droite1).pack(side =LEFT)
Button(fra1, text="^", fg = 'red', command =haut1).pack(side =LEFT)
Button(fra1, text="v", fg = 'red', command =bas1).pack(side =LEFT)
fra2 = Frame(fen)
fra2.grid(row =4, column =1, sticky =E, padx =10)
Button(fra2, text="<-", fg = 'blue', command =gauche2).pack(side =LEFT)
Button(fra2, text="->", fg = 'blue', command =droite2).pack(side =LEFT)
Button(fra2, text="^", fg = 'blue', command =haut2).pack(side =LEFT)
Button(fra2, text="v", fg = 'blue', command =bas2).pack(side =LEFT)

fen.mainloop()

```



2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !

5. *# Conversions de températures Fahrenheit <=> Celsius*

```

from Tkinter import *

def convFar(event):
    "valeur de cette température, exprimée en degrés Fahrenheit"
    tF = eval(champTC.get())
    varTF.set(str(tF*1.8 +32))

def convCel(event):
    "valeur de cette température, exprimée en degrés Celsius"
    tC = eval(champTF.get())
    varTC.set(str((tC-32)/1.8))

fen = Tk()
fen.title('Fahrenheit/Celsius')

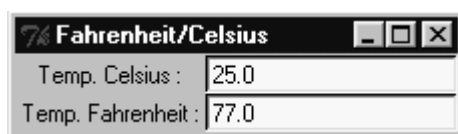
```



```
Label(fen, text='Temp. Celsius :').grid(row =0, column =0)
# "variable Tkinter" associée au champ d'entrée. Cet "objet-variable"
# assure l'interface entre TCL et Python (voir notes, page 165) :
varTC =StringVar()
champTC = Entry(fen, textvariable =varTC)
champTC.bind("<Return>", convFar)
champTC.grid(row =0, column =1)
# Initialisation du contenu de la variable Tkinter :
varTC.set("100.0")

Label(fen, text='Temp. Fahrenheit :').grid(row =1, column =0)
varTF =StringVar()
champTF = Entry(fen, textvariable =varTF)
champTF.bind("<Return>", convCel)
champTF.grid(row =1, column =1)
varTF.set("212.0")

fen.mainloop()
```



- 6. Réfléchissez !
- 7. Voir ci dessous.
- 8. Voir ci dessous.

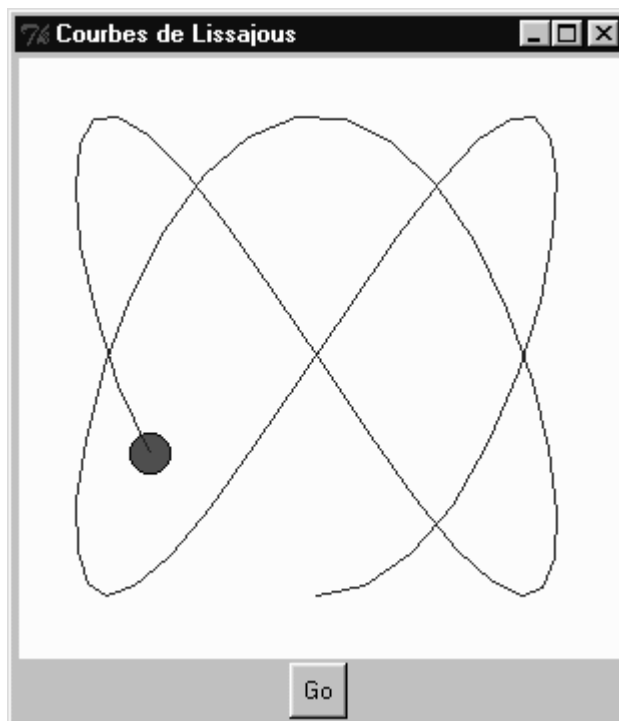
9. # Cercles et courbes de Lissajous

```
from Tkinter import *
from math import sin, cos

def move():
    global ang, x, y
    # on mémorise les coord. précédentes avant de calculer les nouvelles :
    xp, yp = x, y
    # rotation d'un angle de 0.1 radian :
    ang = ang +.1
    # sinus et cosinus de cet angle => coord. d'un point du cercle trigono.
    x, y = sin(ang), cos(ang)
    # Variante déterminant une courbe de Lissajous avec f1/f2 = 2/3 :
    # x, y = sin(2*ang), cos(3*ang)
    # mise à l'échelle (120 = rayon du cercle, (150,150) = centre du canevas)
    x, y = x*120 + 150, y*120 + 150
    can.coords(balle, x-10, y-10, x+10, y+10)
    can.create_line(xp, yp, x, y, fill ="blue")

ang, x, y = 0., 150., 270.
fen = Tk()
fen.title('Courbes de Lissajous')
can = Canvas(fen, width =300, height=300, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Go', command =move).pack()

fen.mainloop()
```



- 10. Réfléchissez !
- 11. Réfléchissez !

Animation automatique - Récursivité

Pour conclure cette première prise de contact avec l'interface graphique *Tkinter*, voici un dernier exemple d'animation, qui fonctionne cette fois de manière autonome dès qu'on l'a mise en marche.

```

from Tkinter import *

# définition des gestionnaires
# d'événements :

def move():
    "déplacement de la balle"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 + dx, y1 + dy
    if x1 > 210:
        x1, dx, dy = 210, 0, 15
    if y1 > 210:
        y1, dx, dy = 210, -15, 0
    if x1 < 10:
        x1, dx, dy = 10, 0, -15
    if y1 < 10:
        y1, dx, dy = 10, 15, 0
    can1.coords(oval1, x1, y1, x1+30, y1+30)
    if flag > 0:
        fen1.after(50, move)           # => boucler après 50 millisecondes

def stop_it():
    "arrêt de l'animation"
    global flag
    flag = 0
    
```

```

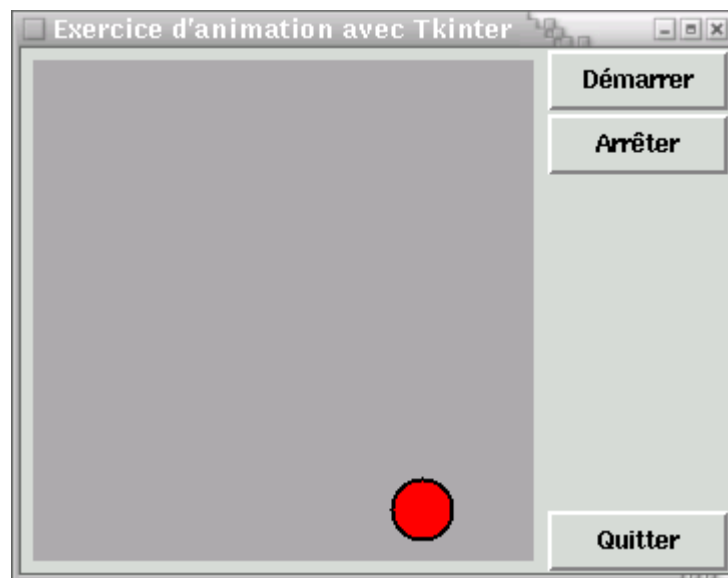
def start_it():
    "démarrage de l'animation"
    global flag
    if flag ==0:          # pour ne lancer qu'une seule boucle
        flag =1
        move()

#===== Programme principal =====

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10          # coordonnées initiales
dx, dy = 15, 0           # 'pas' du déplacement
flag =0                  # commutateur

# Création du widget principal ("parent") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")
# création des widgets "enfants" :
can1 = Canvas(fen1,bg='dark grey',height=250, width=250)
can1.pack(side=LEFT, padx =5, pady =5)
oval1 = can1.create_oval(x1, y1, x1+30, y1+30, width=2, fill='red')
bou1 = Button(fen1,text='Quitter', width =8, command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Démarrer', width =8, command=start_it)
bou2.pack()
bou3 = Button(fen1, text='Arrêter', width =8, command=stop_it)
bou3.pack()
# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()

```



La seule nouveauté mise en œuvre dans ce script se trouve tout à la fin de la définition de la fonction `move()` : vous y noterez l'utilisation de la méthode `after()`. Cette méthode peut s'appliquer à un widget quelconque. Elle déclenche l'appel d'une fonction *après qu'un certain laps de temps se soit écoulé*. Ainsi par exemple, `window.after(200,qqc)` déclenche pour le widget `window` un appel de la fonction `qqc()` après une pause de 200 millisecondes.

Dans notre script, la fonction qui est appelée par la méthode `after()` est la fonction `move()` elle-même. Nous utilisons donc ici pour la première fois une technique de programmation très puissante, que l'on appelle *récurtivité*.

Pour faire simple, nous dirons que *la récursivité est ce qui se passe lorsqu'une fonction s'appelle elle-même*. On obtient bien évidemment ainsi un bouclage, qui peut se perpétuer indéfiniment si l'on ne prévoit pas aussi un moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple :

La fonction `move()` est invoquée une première fois lorsque l'on clique sur le bouton « Démarrer ». Elle effectue son travail (c'est-à-dire positionner la balle), 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...

C'est du moins ce qui se passerait si nous n'avions pas pris la précaution de placer quelque part dans la boucle une instruction de sortie. En l'occurrence, il s'agit d'un simple test conditionnel : à chaque itération de la boucle, nous examinons le contenu de la variable `flag` à l'aide d'une instruction `if`. Si le contenu de la variable `flag` est zéro, alors le bouclage ne s'effectue plus et l'animation s'arrête. `flag` étant une variable globale, nous pouvons aisément changer sa valeur à l'aide d'autres fonctions, celles que nous avons associées aux boutons « Démarrer » et « Arrêter ».

Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter notre animation :

Un premier clic sur le bouton « Démarrer » assigne une valeur non-nulle à la variable `flag`, puis provoque immédiatement un premier appel de la fonction `move()`. Celle-ci s'exécute 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 fonction `move()` ne peut plus être appelée tant que la valeur de `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

1. Dans la fonction `start_it()`, supprimez l'instruction `if flag == 0:` (et l'indentation des deux lignes suivantes). Que se passe-t-il ? (Cliquez plusieurs fois sur le bouton « Démarrer »).
Tâchez d'exprimer le plus clairement possible votre explication des faits observés.
2. Modifiez le programme de telle façon que la balle change de couleur à chaque « virage ».
3. 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 »).
4. Modifiez le programme de manière à obtenir d'autres mouvements. Tâchez par exemple d'obtenir un mouvement circulaire.
5. 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 !

6. À partir des scripts précédents, vous pouvez à présent écrire un programme de jeu fonctionnant de la manière suivante : Une balle se déplace au hasard sur un canevas, à vitesse faible. Le joueur doit essayer de cliquer sur cette balle à l'aide de la souris. S'il y arrive, il gagne un point mais la balle se déplace désormais un peu plus vite, et ainsi de suite. Arrêter le jeu après un certain nombre de clics et afficher le score atteint.
7. Variante du jeu précédent : chaque fois que le joueur parvient à « l'attraper », la balle devient plus petite (elle peut également changer de couleur).
8. Écrivez un programme dans lequel évoluent plusieurs balles de couleurs différentes, qui rebondissent les unes sur les autres ainsi que sur les parois.
9. Perfectionnez le jeu des précédents exercices en y intégrant l'algorithme ci-dessus. Il s'agit à présent pour le joueur de cliquer seulement sur la balle rouge. Un clic erroné (sur une balle d'une autre couleur) lui fait perdre des points.
10. 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...).
11. Écrivez un programme pour le jeu du serpent : un « serpent » (constitué en fait d'une courte ligne de carrés) se déplace sur le canevas dans l'une des 4 directions : droite, gauche, haut, bas. Le joueur peut à tout moment changer la direction suivie par le serpent à l'aide des touches fléchées du clavier. Sur le canevas se trouvent également des « proies » (des petits cercles fixes disposés au hasard). Il faut diriger le serpent de manière à ce qu'il « mange » les proies sans arriver en contact avec les bords du canevas. A chaque fois qu'une proie est mangée, le serpent s'allonge d'un carré, le joueur gagne un point, et une nouvelle proie apparaît ailleurs. La partie s'arrête lorsque le serpent touche l'une des parois, ou lorsqu'il a atteint une certaine taille.
12. Perfectionnement du jeu précédent : la partie s'arrête également si le serpent « se recoupe ».

Solution

1. Réfléchissez !
2. Réfléchissez !
3. Réfléchissez !
4. Réfléchissez !

```

5. # Chutes et rebonds
   from Tkinter import *
   def move():
       global x, y, v, dx, dv, flag

```

```

xp, yp = x, y          # mémorisation des coord. précédentes
# déplacement horizontal :
if x > 385 or x < 15 : # rebond sur les parois latérales :
    dx = -dx          # on inverse le déplacement
x = x + dx
# variation de la vitesse verticale (toujours vers le bas):
v = v + dv
# déplacement vertical (proportionnel à la vitesse)
y = y + v
if y > 240:           # niveau du sol à 240 pixels :
    y = 240          # défense d'aller + loin !
    v = -v           # rebond : la vitesse s'inverse
# on repositionne la balle :
can.coords(balle, x-10, y-10, x+10, y+10)
# on trace un bout de trajectoire :
can.create_line(xp, yp, x, y, fill='light grey')
# ... et on remet ça jusqu'à plus soif :
if flag > 0:
    fen.after(50,move)

def start():
    global flag
    flag = flag +1
    if flag == 1:
        move()

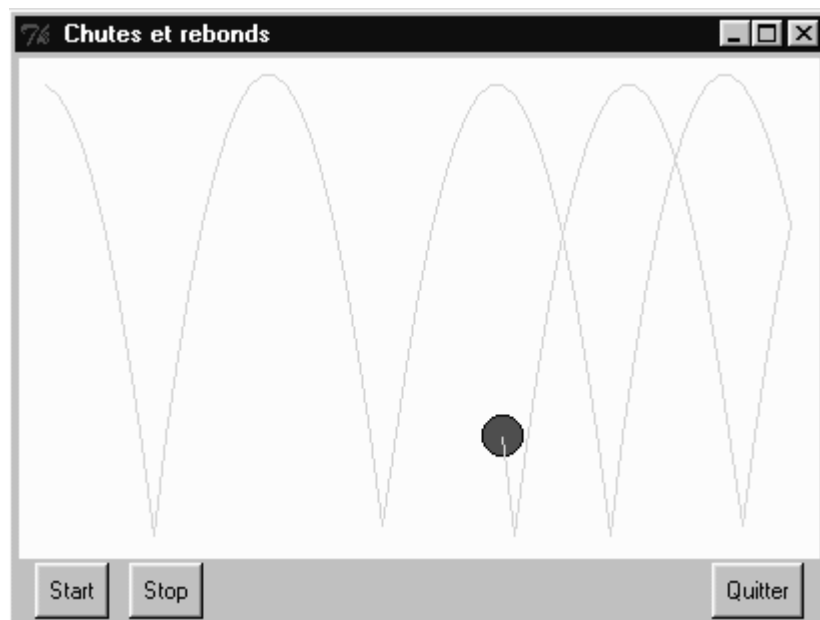
def stop():
    global flag
    flag =0

# initialisation des coordonnées, des vitesses et du témoin d'animation :
x, y, v, dx, dv, flag = 15, 15, 0, 6, 5, 0

fen = Tk()
fen.title(' Chutes et rebonds')
can = Canvas(fen, width =400, height=250, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Start', command =start).pack(side =LEFT, padx =10)
Button(fen, text='Stop', command =stop).pack(side =LEFT)
Button(fen, text='Quitter', command =fen.quit).pack(side =RIGHT, padx =10)

fen.mainloop()

```



6. Réfléchissez !
7. Réfléchissez !
8. Réfléchissez !
9. Réfléchissez !
10. Réfléchissez !
11. Réfléchissez !
12. Réfléchissez !

Notes

1. *widget* est le résultat de la contraction de l'expression *window gadget*. Dans certains environnements de programmation, on appellera cela plutôt un « contrôle » ou un « composant graphique ». Ce terme désigne en fait toute entité susceptible d'être placée dans une fenêtre d'application, comme par exemple un bouton, une case à cocher, une image, etc., et parfois aussi la fenêtre elle-même.
2. Si vous effectuez cet exercice sous *Windows*, nous vous conseillons d'utiliser de préférence une version standard de Python dans une fenêtre DOS ou dans IDLE plutôt que *PythonWin*. Vous pourrez mieux observer ce qui se passe après l'entrée de chaque commande.
3. Cette concision du langage est une conséquence du typage dynamique des variables en vigueur sous Python. D'autres langages utilisent une instruction particulière (telle que `new`) pour instancier un nouvel objet. Exemple :
`maVoiture = new Cadillac` (instanciation d'un objet de classe `Cadillac`, référencé dans la variable `maVoiture`)
4. Ces messages sont souvent notés WM (*Window messages*) dans un environnement graphique constitué de fenêtres (avec de nombreuses zones réactives : boutons, cases à cocher, menus déroulants, etc.). Dans la description des algorithmes, il arrive fréquemment aussi qu'on confonde ces messages avec les événements eux-mêmes.
5. Au sens strict, une telle fonction qui ne devra renvoyer aucune valeur est donc plutôt une *procédure*.
6. En anglais, le mot *bind* signifie « lier »
7. 1. Il existe d'autres classes d'images, mais pour les utiliser il faut importer dans

le script d'autres modules graphiques que la seule bibliothèque *Tkinter*. Vous pouvez par exemple expérimenter la bibliothèque PIL (*Python Imaging Library*).

Tkinter

Créer des interfaces python avec Tkinter

Que faut il posséder d'abord ?

Tkinter (pour Tool kit interface) est une boîte à outils d'interface graphique pour Python. L'interface Tkinter s'installe avec Python. Il suffit donc juste d'installer Python 2.3 ou 2.4 ou supérieur pour pouvoir utiliser Tkinter^(Référence nécessaire).

Ensuite il vous suffit d'importer la bibliothèque dans votre programme :

```
from Tkinter import *
```

Principes de base

Créez un fichier texte dans le dossier python 2.4 ou python 2.3.

Donnez lui le nom que vous voulez, mais pour que votre fichier puisse être interprété par python, il doit porter l'extension .py ou .pyw

Créez donc un fichier du type : monfichier.py (dans ce cas, la console s'affichera, ce qui peut être pratique pour suivre le fonctionnement du programme)

ou monfichier.pyw (dans ce cas la console ne s'ouvrira pas : c'est ce type d'extension qu'il faut utiliser pour la version définitive du programme).

Pour modifier le programme :

- clic droit sur le fichier
- ouvrir avec un logiciel d'édition de texte pur.

Créer une fenêtre pour l'application

Propriétés et méthodes de l'objet fenêtre

le programme est le suivant :

```
from Tkinter import * # le programme va aller chercher toutes les fonctions de la bibl  
Fenetre= Tk() # vous pouvez choisir le nom que vous voulez pour votre fenêtre  
Fenetre.mainloop() # lance la boucle principale
```

Qu'est-ce qu'un widget ?

widget : contraction de l'anglais *windows gadget* (gadget fenêtre). Les widgets sont tous les objets graphiques que l'on peut insérer dans une

interface (fenêtre). Les principaux sont :

- Les boutons : Button (pour commander une action)
- Les labels : Label (pour insérer un texte)
- Les zones de saisie : Entry (pour permettre l'entrée d'une donnée)
- Les canevas : Canvas (pour insérer des dessins)

Chaque widget a des propriétés et des méthodes qui permettent de régler son apparence et les interactions avec l'utilisateur.

Le widget Button

Chaque widget (objet d'interface) doit être créé puis placé dans la fenêtre

```
#!/usr/bin/python          # Emplacement de l'interpréteur Python (sous Linux)
# -*- coding: utf-8 -*-   # Définition l'encodage des caractères
from Tkinter import *     # le programme va aller chercher toutes les fonctions de la bibli
Fenetre= Tk()             # création de la fenêtre, avec un nom de votre choix
bouton=Button(Fenetre, text="quitter", command=Fenetre.destroy) # Bouton qui détruit la
bouton.pack()             # insère le bouton dans la fenêtre
Fenetre.mainloop()       # lance la boucle principale
```

- La commande Fenetre.destroy() est une méthode de destruction qui s'applique à l'objet fenêtre. La pression du bouton a donc pour conséquence la fermeture de la fenêtre Fenetre

propriétés et méthodes de l'objet "bouton"

- Ce qui sera affiché sur votre bouton est contenu dans la propriété "text", passée ici en paramètre de l'objet "bouton".
- La procédure "command" permet une action lorsqu'on clique sur ce bouton. Cette procédure peut également être choisie parmi les fonctions définies dans le programme.

Le widget Label

L'incontournable "Hello world"

```
#!/usr/bin/python          # Emplacement de l'interpréteur Python (sous Linux)
# -*- coding: utf-8 -*-   # Définition l'encodage des caractères
from Tkinter import *
Fenetre=Tk()
texte=Label(Fenetre, text="Hello World")
texte['fg']='black' # Création du texte "Hello World" de couleur noire
texte.pack() # Insère le texte dans la fenêtre
Fenetre.mainloop()
```

Propriétés et méthodes de l'objet label

- "fg" contient la couleur du texte (en anglais)
- "bg" contient la couleur de fond du texte (en anglais)

Le widget Entry

Propriétés et méthodes de l'objet Entry

L'objet Entry() est une zone de saisie de texte que l'on crée de la manière suivante:

```
#!/usr/bin/python          # Emplacement de l'interpréteur Python (sous Linux)
# -*- coding: utf-8 -*-   # Définition l'encodage des caractères
from tkinter import *    # On importe l'ensemble du module Tkinter
Fenetre = Tk()
Entree = Entry(Fenetre)   # On définit l'objet Entry qui porte le nom Entree
Entree.pack()            # On place "Entree"
Fenetre.mainloop()       # On lance la boucle du programme
```

Vous pouvez l'utiliser dans des situations plus complexes comme, par exemple, un formulaire que je vous laisserai examiner:

```
#!/usr/bin/python          # Emplacement de l'interpréteur Python (sous Linux)
# -*- coding: utf-8 -*-   # Définition l'encodage des caractères
from Tkinter import *

def repondre():
    affichage['text'] = reponse.get() # lecture du contenu du widget "reponse"

Fenetre = Tk()
Fenetre.title('Mon nom')

nom = Label(Fenetre, text = 'Votre nom :')
reponse = Entry(Fenetre)
valeur = Button(Fenetre, text = ' Valider', command=repondre)
affichage = Label(Fenetre, width=30)
votre_nom=Label(Fenetre, text='Votre nom est :')
nom.pack()
reponse.pack()
valeur.pack()
votre_nom.pack()
affichage.pack()

Fenetre.mainloop()
```

Le widget Canvas

Le widget Canvas (canevas, en français) est une zone de dessin rectangulaire.

Notons que l'angle haut gauche du canevas est l'origine des coordonnées (x,y)=(0,0).

Un exemple d'utilisation :

```
#!/usr/bin/python          # Emplacement de l'interpréteur Python (sous Linux)
# -*- coding: utf-8 -*-   # Définition l'encodage des caractères

from Tkinter import *

racine= Tk()

zone_dessin = Canvas(racine, width=500, height=500) #Définit les dimensions du canevas
zone_dessin.pack() #Affiche le canevas
zone_dessin.create_line(0,0,500,500) #Dessine une ligne en diagonale
zone_dessin.create_rectangle(100,100,200,200) #dessine un rectangle

bouton_sortir = Button(racine,text="Sortir",command=racine.destroy)
bouton_sortir.pack()

racine.mainloop()
```

Quelques propriétés de l'objet Canvas

Les propriétés sont définies en paramètre lors de la construction de l'objet

- height : Hauteur Y du canvas
- width : Largeur X du canvas
- bg : Couleur de fond du canvas
- bd : Taille en pixels du bord du canvas (2 par défaut)
- relief : Style de la bordure (flat (par défaut),raised,sunken,groove,ridge)
- ...

Quelques méthodes du widget Canvas

- .create_arc(): Pour créer un arc de cercle
- .create_bitmap(): Image bitmap
- .create_image(): Image graphique
- .create_line(): Pour créer une ligne
- .create_oval(): Pour créer un cercle ou une ellipse
- .create_polygon(): Pour créer un polygone
- .create_rectangle(): Pour créer un rectangle
- .create_text(): Texte
- .create_window(): Une fenêtre rectangulaire

Exemple supplémentaire

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from Tkinter import *

Fenetre=Tk() #La fonction Tk() du module Tkinter permet de créer une fenêtre qui se
Fenetre.title("Mon programme avec Tkinter") # Donne un titre à la fenêtre (par défaut c

# Dans Fenetre nous allons créer un objet type Canvas qui se nomme zone_dessin
# Nous donnons des valeurs aux propriétés "width", "height", "bg", "bd", "relief"
```

```

zone_dessin = Canvas(Fenetre,width=500,height=500,
                    bg='yellow',bd=8,relief="ridge")
zone_dessin.pack() #Affiche le Canvas

#Nous allons maintenant utiliser quelques méthodes du widget "zone_dessin"
zone_dessin.create_line(0,0,500,500,fill='red',width=4) # Dessine une ligne
zone_dessin.create_line(0,500,500,0,fill='red',width=4) # Dessine une ligne
zone_dessin.create_rectangle(150,150,350,350) # Dessine un rectangle
zone_dessin.create_oval(150,150,350,350,fill='white',width=4) # Dessine un cercle

# boutons_sortir est un widget de type "Button"
# dont nous définissons les propriétés "text" et "command")
bouton_sortir= Button(Fenetre,text="Sortir",command=Fenetre.destroy)
# la commande "destroy" appliquée à la fenêtre détruit l'objet "Fenetre" et clôture le
bouton_sortir.pack()

Fenetre.mainloop() # Lancement de la boucle du programme, en attente d'événements (clav

```

Images .gif

```

#!/usr/bin/python
# -*- coding: utf-8 -*-

from tkinter import *
Fenetre=Tk()
photo=PhotoImage(file="Wikibooks.gif")
labl = Label(Fenetre, image=photo)
labl.pack()
Fenetre.mainloop()

```

Installation des Python méga-widgets

Python megawidgets (Pmw) est un créateur de widget utilisant Tkinter.

Visitez le site web : <http://pmw.sourceforge.net> et cliquez sur le lien :
« Download Pmw12tar.gz » pour télécharger le fichier correspondant.

Décompressez ce fichier archive dans un répertoire temporaire, à l'aide d'un logiciel de décompression tel que *tar*, *Winzip*, *Info-Zip*, *unzip*...

Recopiez l'intégralité du sous-répertoire *Pmw* qui s'est créé automatiquement, dans le répertoire où se trouve déjà l'essentiel de votre installation de Python.

Sous *Windows*, il s'agira par exemple de *C:\Python23*.

Sous *Linux*, il s'agira vraisemblablement de */usr/lib/python*.

Voir aussi

- PMW (Python megawidgets) pour ajouter des menus déroulants, cases à cocher et autres boîtes de dialogues.

- PIL (Python Imaging Library) pour incruster des images.
- py2.exe : pour créer des exécutable (limité à Python 2.6).
- tkRAD: Tkinter XML widget builder : pour générer automatiquement des widgets Tkinter à partir d'un fichier source XML.
- (anglais) Site officiel

Et pour quelques widgets de plus...

Les pages qui suivent contiennent des indications et des exemples complémentaires qui pourront vous être utiles pour le développement de vos projets personnels. Il ne s'agit évidemment pas d'une documentation de référence complète sur *Tkinter*. Pour en savoir plus, vous devrez tôt ou tard consulter des ouvrages spécialisés, comme par exemple l'excellent *Python and Tkinter programming* de John E. Grayson, dont vous trouverez la référence complète à la page.

Les « boutons radio »

Les *widgets* « boutons radio » permettent de proposer à l'utilisateur un ensemble de choix mutuellement exclusifs. On les appelle ainsi par analogie avec les boutons de sélection que l'on trouvait jadis sur les postes de radio. Ces boutons étaient conçus de telle manière qu'un seul à la fois pouvait être enfoncé : tous les autres ressortaient automatiquement.

La caractéristique essentielle de ces *widgets* est qu'on les utilise toujours par groupes. Tous les boutons radio faisant partie d'un même groupe sont associés à une seule et même *variable Tkinter*, mais chacun d'entre eux se voit aussi attribuer une *valeur* particulière.



Lorsque l'utilisateur sélectionne l'un des boutons, la valeur correspondant à ce bouton est affectée à la *variable Tkinter* commune.

```
1 from Tkinter import *
2
3 class RadioDemo(Frame):
4     """Démo : utilisation de widgets 'boutons radio'"""
5     def __init__(self, boss =None):
6         """Création d'un champ d'entrée avec 4 boutons radio"""
7         Frame.__init__(self)
8         self.pack()
9         # Champ d'entrée contenant un petit texte :
10        self.texte = Entry(self, width =30, font ="Arial 14")
11        self.texte.insert(END, "La programmation, c'est génial")
12        self.texte.pack(padx =8, pady =8)
13        # Nom français et nom technique des quatre styles de police :
14        stylePoliceFr =["Normal", "Gras", "Italique", "Gras/Italique"]
15        stylePoliceTk =["normal", "bold", "italic", "bold italic"]
16        # Le style actuel est mémorisé dans un 'objet-variable' Tkinter ;
17        self.choixPolice = StringVar()
18        self.choixPolice.set(stylePoliceTk[0])
19        # Création des quatre 'boutons radio' :
20        for n in range(4):
21            bout = Radiobutton(self,
```

```

|22             text = stylePoliceFr[n],
|23             variable = self.choixPolice,
|24             value = stylePoliceTk[n],
|25             command = self.changePolice)
|26         bout.pack(side =LEFT, padx =5)
|27
|28     def changePolice(self):
|29         """Remplacement du style de la police actuelle"""
|30         police = "Arial 15 " + self.choixPolice.get()
|31         self.texte.configure(font =police)
|32
|33 if __name__ == '__main__':
|34     RadioDemo().mainloop()

```

Commentaires

- Ligne 3 : Cette fois encore, nous préférons construire notre petite application comme une classe dérivée de la classe `Frame()`, ce qui nous permettrait éventuellement de l'intégrer sans difficulté dans une application plus importante.
- Ligne 8 : En général, on applique les méthodes de positionnement des *widgets* (`pack()`, `grid()`, ou `place()`) après instanciation de ceux-ci, ce qui permet de choisir librement leur disposition à l'intérieur des fenêtres maîtresses. Comme nous le montrons ici, il est cependant tout à fait possible de déjà prévoir ce positionnement dans le constructeur du *widgset*.
- Ligne 11 : Les *widgsets* de la classe `Entry` disposent de plusieurs méthodes pour accéder à la chaîne de caractères affichée. La méthode `get()` permet de récupérer la chaîne entière. La méthode `delete()` permet d'en effacer tout ou partie. La méthode `insert()` permet d'insérer de nouveaux caractères à un emplacement quelconque (c'est-à-dire au début, à la fin, ou même à l'intérieur d'une chaîne préexistante éventuelle). Cette méthode s'utilise donc avec deux arguments, le premier indiquant l'emplacement de l'insertion (utilisez 0 pour insérer au début, `END` pour insérer à la fin, ou encore un indice numérique quelconque pour désigner un caractère dans la chaîne).
- Lignes 14-15 : Plutôt que de les instancier dans des instructions séparées, nous préférons créer nos quatre boutons à l'aide d'une boucle. Les options spécifiques à chacun d'eux sont d'abord préparées dans les deux listes `stylePoliceFr` et `stylePoliceTk` : la première contient les petits textes qui devront s'afficher en regard de chaque bouton, et la seconde les valeurs qui devront leur être associées.
- Lignes 17-18 : Comme expliqué précédemment, les quatre boutons forment un groupe autour d'une variable commune. Cette variable prendra la valeur associée au bouton radio que l'utilisateur décidera de choisir. Nous ne pouvons cependant pas utiliser une variable ordinaire pour remplir ce rôle, parce que les attributs internes des objets *Tkinter* ne sont accessibles qu'au travers de méthodes spécifiques. Une fois de plus, nous utilisons donc ici un objet-variable *Tkinter*, de type 'chaîne de caractères', que nousinstancions à partir de la classe `StringVar()`, et auquel nous donnons une valeur par défaut à la ligne 18.
- Lignes 20 à 26 : Instanciation des quatre boutons radio. Chacun d'entre eux se voit attribuer une étiquette et une valeur différentes, mais tous sont associés à la même *variable Tkinter* commune (`self.choixPolice`). Tous

invoquent également la même méthode `self.changePolice()`, chaque fois que l'utilisateur effectue un clic de souris sur l'un ou l'autre.

- Lignes 28 à 31 : Le changement de police s'obtient par re-configuration de l'option `font` du *widget* `Entry`. Cette option attend un tuple contenant le nom de la police, sa taille, et éventuellement son style. Si le nom de la police ne contient pas d'espaces, le tuple peut aussi être remplacé par une chaîne de caractères. Exemples :

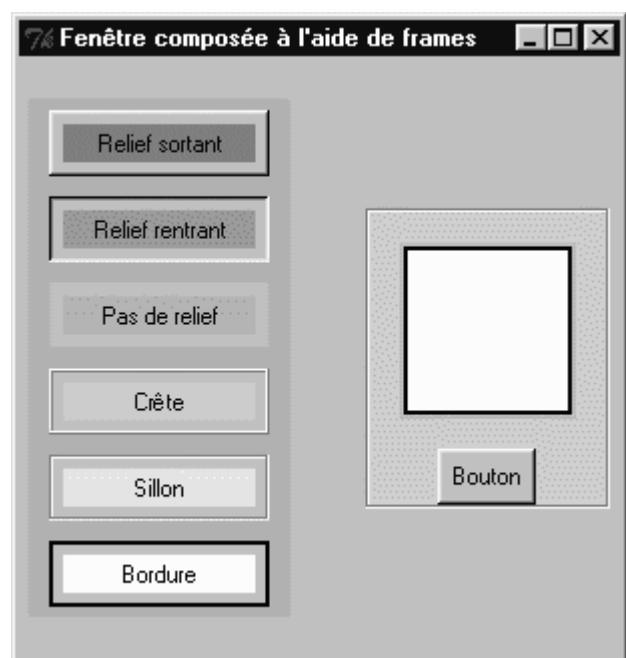
```
('Arial', 12, 'italic')
('Helvetica', 10)
('Times New Roman', 12, 'bold italic')
"Verdana 14 bold"
"President 18 italic"
```

Utilisation des cadres (frames) pour la composition d'une fenêtre

Vous avez déjà abondamment utilisé la classe de *widgets* `Frame()` (« cadre », en français), notamment pour créer de nouveaux *widgets* complexes par dérivation.

Le petit script ci-dessous vous montre l'utilité de cette même classe pour regrouper des ensembles de *widgets* et les disposer d'une manière déterminée dans une fenêtre. Il vous démontre également l'utilisation de certaines options décoratives (bordures, relief, etc.).

Pour composer la fenêtre ci-contre, nous avons utilisé deux cadres `f1` et `f2`, de manière à réaliser deux groupes de *widgets* bien distincts, l'un à gauche et l'autre à droite. Nous avons coloré ces deux cadres pour bien les mettre en évidence, mais ce n'est évidemment pas indispensable.



Le cadre `f1` contient lui-même 6 autres cadres, qui contiennent chacun un *widget* de la classe `Label()`. Le cadre `f2` contient un *widget* `Canvas()` et un *widget* `Button()`. Les couleurs et garnitures sont de simples options.

```
from Tkinter import * #1
fen = Tk() #2
fen.title("Fenêtre composée à l'aide de frames") #3
fen.geometry("300x300") #4
#5
#6
```

```

f1 = Frame(fen, bg = '#80c0c0') #7
f1.pack(side =LEFT, padx =5) #8
#9
fint = [0]*6 #10
for (n, col, rel, txt) in [(0, 'grey50', RAISED, 'Relief sortant'), #11
                          (1, 'grey60', SUNKEN, 'Relief rentrant'), #12
                          (2, 'grey70', FLAT, 'Pas de relief'), #13
                          (3, 'grey80', RIDGE, 'Crête'), #14
                          (4, 'grey90', GROOVE, 'Sillon'), #15
                          (5, 'grey100', SOLID, 'Bordure')]: #16
    fint[n] = Frame(f1, bd =2, relief =rel) #17
    e = Label(fint[n], text =txt, width =15, bg =col) #18
    e.pack(side =LEFT, padx =5, pady =5) #19
    fint[n].pack(side =TOP, padx =10, pady =5) #20
#21
f2 = Frame(fen, bg ='#d0d0b0', bd =2, relief =GROOVE) #22
f2.pack(side =RIGHT, padx =5) #23
#24
can = Canvas(f2, width =80, height =80, bg ='white', bd =2, relief =SOLID) #25
can.pack(padx =15, pady =15) #26
bou =Button(f2, text='Bouton') #27
bou.pack() #28
#29
fen.mainloop() #30

```

- Lignes 3 à 5 : Afin de simplifier au maximum la démonstration, nous ne programmons pas cet exemple comme une nouvelle classe. Remarquez à la ligne 5 l'utilité de la méthode `geometry()` pour fixer les dimensions de la fenêtre principale.
- Ligne 7 : Instanciation du cadre de gauche. La couleur de fond (une variété de bleu cyan) est déterminée par l'argument `bg` (*background*). Cette chaîne de caractères contient en notation hexadécimale la description des trois composantes rouge, verte et bleue de la teinte que l'on souhaite obtenir : Après le caractère `#` signalant que ce qui suit est une valeur numérique hexadécimale, on trouve trois groupes de deux symboles alphanumériques. Chacun de ces groupes représente un nombre compris entre 1 et 255. Ainsi 80 correspond à 128, et c0 correspond à 192 en notation décimale. Dans notre exemple, les composantes rouge, verte et bleue de la teinte à représenter valent donc respectivement 128, 192 & 192. En application de cette technique descriptive, le noir serait obtenu avec `#000000`, le blanc avec `#ffffff`, le rouge pur avec `#ff0000`, un bleu sombre avec `#000050`, etc.
- Ligne 8 : Puisque nous lui appliquons la méthode `pack()`, le cadre sera automatiquement dimensionné par son contenu. L'option `side =LEFT` le positionnera à gauche dans sa fenêtre maîtresse. L'option `padx =5` ménagera un espace de 5 pixels à sa gauche et à sa droite (nous pouvons traduire « `padx` » par « *espacement horizontal* »).
- Ligne 10 : Dans le cadre `f1` que nous venons de préparer, nous avons l'intention de regrouper 6 autres cadres similaires contenant chacun une étiquette. Le code correspondant sera plus simple et plus efficient si nous instancions ces *widgets* dans une liste plutôt que dans des variables indépendantes. Nous préparons donc cette liste avec 6 éléments que nous

remplacerons plus loin.

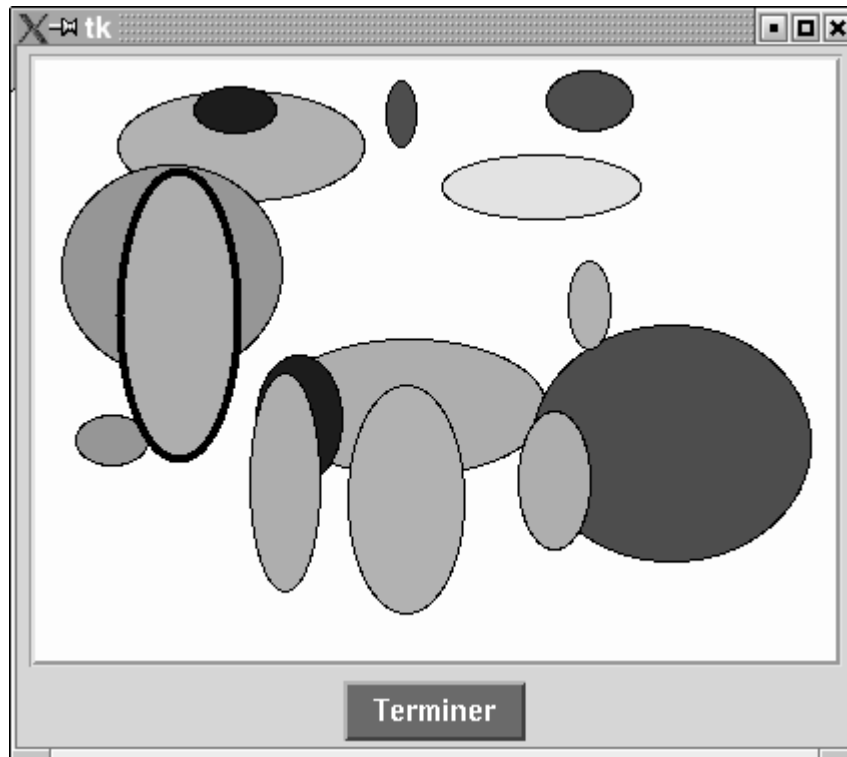
- Lignes 11 à 16 : Pour construire nos 6 cadres similaires, nous allons parcourir une liste de 6 tuples contenant les caractéristiques particulières de chaque cadre. Chacun de ces tuples est constitué de 4 éléments : un indice, une constante *Tkinter* définissant un type de relief, et deux chaînes de caractères décrivant respectivement la couleur et le texte de l'étiquette. La boucle `for` effectue 6 itérations pour parcourir les 6 éléments de la liste. A chaque itération, le contenu d'un des tuples est affecté aux variables `n`, `col`, `rel` et `txt` (et ensuite les instructions des lignes 17 à 20 sont exécutées). Le parcours d'une liste de tuples à l'aide d'une boucle `for` constitue donc une construction particulièrement compacte, qui permet de réaliser de nombreuses affectations avec un très petit nombre d'instructions.
- Ligne 17 : Les 6 cadres sont instanciés comme des éléments de la liste `fin`. Chacun d'entre eux est agrémenté d'une bordure décorative de 2 pixels de large, avec un certain effet de relief.
- Lignes 18-20 : Les étiquettes ont toutes la même taille, mais leurs textes et leurs couleurs de fond diffèrent. Du fait de l'utilisation de la méthode `pack()`, c'est la dimension des étiquettes qui détermine la taille des petits cadres. Ceux-ci à leur tour déterminent la taille du cadre qui les regroupe (le cadre `f1`). Les options `padx` et `pady` permettent de réserver un petit espace autour de chaque étiquette, et un autre autour de chaque petit cadre. L'option `side=TOP` positionne les 6 petits cadres les uns en dessous des autres dans le cadre conteneur `f1`.
- Lignes 22-23 : Préparation du cadre `f2` (cadre de droite). Sa couleur sera une variété de jaune, et nous l'entourerons d'une bordure décorative ayant l'aspect d'un sillon.
- Lignes 25 à 28 : Le cadre `f2` contiendra un canevas et un bouton. Notez encore une fois l'utilisation des options `padx` et `pady` pour ménager des espaces autour des *widgets* (Considérez par exemple le cas du bouton, pour lequel cette option n'a pas été utilisée : de ce fait, il entre en contact avec la bordure du cadre qui l'entoure). Comme nous l'avons fait pour les cadres, nous avons placé une bordure autour du canevas. Sachez que d'autres *widgets* acceptent également ce genre de décoration : boutons, champs d'entrée, etc.

Comment déplacer des dessins à l'aide de la souris

Le *widget* canevas est l'un des points forts de la bibliothèque graphique *Tkinter*. Il intègre en effet un grand nombre de dispositifs très efficaces pour manipuler des dessins. Le script ci-après est destiné à vous montrer quelques techniques de base. Si vous voulez en savoir plus, notamment en ce qui concerne la manipulation de dessins composés de plusieurs parties, veuillez consulter l'un ou l'autre ouvrage de référence traitant de *Tkinter*.

Au démarrage de notre petite application, une série de dessins sont tracés au hasard dans un canevas (il s'agit en l'occurrence de simples ellipses colorées). Vous pouvez déplacer n'importe lequel de ces dessins en les « saisissant » à l'aide de votre souris.

Lorsqu'un dessin est déplacé, il passe à l'avant-plan par rapport aux autres, et sa bordure apparaît plus épaisse pendant toute la durée de sa manipulation.



Pour bien comprendre la technique utilisée, vous devez vous rappeler qu'un logiciel utilisant une interface graphique est un logiciel « piloté par les événements » (revoyez au besoin les explications de la page). Dans cette application, nous allons mettre en place un mécanisme qui réagit aux événements : « enfoncement du bouton gauche de la souris », « déplacement de la souris, le bouton gauche restant enfoncé », « relâchement du bouton gauche ».

Ces événements sont générés par le système d'exploitation et pris en charge par l'interface *Tkinter*. Notre travail de programmation consistera donc simplement à les associer à des gestionnaires différents (fonctions ou méthodes).

```
# Exemple montrant comment faire en sorte que les objets dessinés dans un
# canevas puissent être manipulés à l'aide de la souris

from Tkinter import *
from random import randrange

class Draw(Frame):
    "classe définissant la fenêtre principale du programme"
    def __init__(self):
```

```

Frame.__init__(self)
# mise en place du canevas - dessin de 15 ellipses colorées :
self.c = Canvas(self, width =400, height =300, bg ='ivory')
self.c.pack(padx =5, pady =3)
for i in range(15):
    # tirage d'une couleur au hasard :
    coul =['brown','red','orange','yellow','green','cyan','blue',
          'violet','purple'][randrange(9)]
    # tracé d'une ellipse avec coordonnées aléatoires :
    x1, y1 = randrange(300), randrange(200)
    x2, y2 = x1 + randrange(10, 150), y1 + randrange(10, 150)
    self.c.create_oval(x1, y1, x2, y2, fill =coul)
# liaison d'événements <souris> au widget <canevas> :
self.c.bind("<Button-1>", self.mouseDown)
self.c.bind("<Button1-Motion>", self.mouseMove)
self.c.bind("<Button1-ButtonRelease>", self.mouseUp)
# mise en place d'un bouton de sortie :
b_fin = Button(self, text ='Terminer', bg ='royal blue', fg ='white',
               font =( 'Helvetica', 10, 'bold'), command =self.quit)
b_fin.pack(pady =2)
self.pack()

def mouseDown(self, event):
    "Op. à effectuer quand le bouton gauche de la souris est enfoncé"
    self.currObject =None
    # event.x et event.y contiennent les coordonnées du clic effectué :
    self.x1, self.y1 = event.x, event.y
    # <find_closest> renvoie la référence du dessin le plus proche :
    self.selObject = self.c.find_closest(self.x1, self.y1)
    # modification de l'épaisseur du contour du dessin :
    self.c.itemconfig(self.selObject, width =3)
    # <lift> fait passer le dessin à l'avant-plan :
    self.c.lift(self.selObject)

def mouseMove(self, event):
    "Op. à effectuer quand la souris se déplace, bouton gauche enfoncé"
    x2, y2 = event.x, event.y
    dx, dy = x2 -self.x1, y2 -self.y1
    if self.selObject:
        self.c.move(self.selObject, dx, dy)
        self.x1, self.y1 = x2, y2

def mouseUp(self, event):
    "Op. à effectuer quand le bouton gauche de la souris est relâché"
    if self.selObject:
        self.c.itemconfig(self.selObject, width =1)
        self.selObject =None

if __name__ == '__main__':
    Draw().mainloop()

```

Commentaires

Le script contient essentiellement la définition d'une classe graphique dérivée de `Frame()`.

Comme c'est souvent le cas pour les programmes exploitant les classes d'objets, le corps principal du script se résume à une seule instruction composée, dans laquelle on réalise deux opérations consécutives : instantiation d'un objet de la classe définie précédemment, et activation de

sa méthode `mainloop()` (laquelle démarre l'observateur d'événements).

Le constructeur de la classe `Draw()` présente une structure qui doit vous être devenue familière, à savoir : appel au constructeur de la classe parente, puis mise en place de divers *widgets*.

Dans le *widget* canevas, nous instancions 15 dessins sans nous préoccuper de conserver leurs références dans des variables. Nous pouvons procéder ainsi parce que *Tkinter* conserve lui-même une référence interne pour chacun de ces objets. (Si vous travaillez avec d'autres bibliothèques graphiques, vous devrez probablement prévoir une mémorisation de ces références).

Les dessins sont de simples ellipses colorées. Leur couleur est choisie au hasard dans une liste de 9 possibilités, l'indice de la couleur choisie étant déterminé par la fonction `randrange()` importée du module `random`.

Le mécanisme d'interaction est installé ensuite : on associe les trois identificateurs d'événements `<Button-1>`, `<Button1-Motion>` et `<Button1-ButtonRelease>` concernant le *widget* canevas, aux noms des trois méthodes choisies comme gestionnaires d'événements.

Lorsque l'utilisateur enfonce le bouton gauche de sa souris, la méthode `mouseDown()` est donc activée, et le système d'exploitation lui transmet en argument un objet `event`, dont les attributs `x` et `y` contiennent les coordonnées du curseur souris dans le canevas, déterminées au moment du clic.

Nous mémorisons directement ces coordonnées dans les variables d'instance `self.x1` et `self.x2`, car nous en aurons besoin par ailleurs. Ensuite, nous utilisons la méthode `find_closest()` du *widget* canevas, qui nous renvoie la référence du dessin le plus proche. (Note : cette méthode bien pratique renvoie toujours une référence, même si le clic de souris n'a pas été effectué à l'intérieur du dessin).

Le reste est facile : la référence du dessin sélectionné est mémorisée dans une variable d'instance, et nous pouvons faire appel à d'autres méthodes du *widget* canevas pour modifier ses caractéristiques. En l'occurrence, nous utilisons les méthodes `itemconfig()` et `lift()` pour épaissir son contour et le faire passer à l'avant-plan.

Le « transport » du dessin est assuré par la méthode `mouseMove()`, invoquée à chaque fois que la souris se déplace alors que son bouton gauche est resté enfoncé. L'objet `event` contient cette fois encore les coordonnées du curseur souris, au terme de ce déplacement. Nous nous en servons pour calculer les différences entre ces nouvelles coordonnées et les précédentes, afin de pouvoir les transmettre à la méthode `move()` du *widget* canevas, qui effectuera le transport proprement dit.

Nous ne pouvons cependant faire appel à cette méthode que s'il existe effectivement un objet sélectionné, et il nous faut veiller également à

mémoriser les nouvelles coordonnées acquises.

La méthode `mouseUp()` termine le travail. Lorsque le dessin transporté est arrivé à destination, il reste à annuler la sélection et rendre au contour son épaisseur initiale. Ceci ne peut être envisagé que s'il existe effectivement une sélection, bien entendu.

Python Mega Widgets

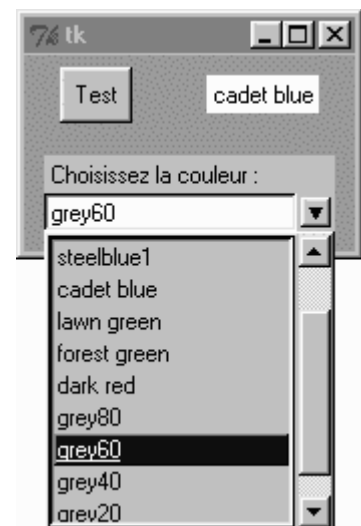
Les modules *Pmw* constituent une extension intéressante de *Tkinter*. Entièrement écrits en Python, ils contiennent toute une bibliothèque de *widgets* composites, construits à partir des classes de base de *Tkinter*. Dotés de fonctionnalités très étendues, ces *widgets* peuvent se révéler fort précieux pour le développement rapide d'applications complexes. Si vous souhaitez les utiliser, sachez cependant que les modules *Pmw* ne font pas partie de l'installation standard de Python : vous devrez donc toujours vérifier leur présence sur les machines cibles de vos programmes. Il existe un grand nombre de ces méga-widgets. Nous n'en présenterons ici que quelques-uns parmi les plus utiles. Vous pouvez rapidement vous faire une idée plus complète de leurs multiples possibilités, en essayant les scripts de démonstration qui les accompagnent (lancez par exemple le script `all.py`, situé dans le répertoire `.../Pmw/demos`).

« Combo Box »

Les méga-widgets s'utilisent aisément. La petite application ci-après vous montre comment mettre en œuvre un *widget* de type `ComboBox` (boîte de liste combinée à un champ d'entrée). Nous l'avons configuré de la manière la plus habituelle (avec une boîte de liste déroulante).

Lorsque l'utilisateur de notre petit programme choisit une couleur dans la liste déroulante (il peut aussi entrer un nom de couleur directement dans le champ d'entrée), cette couleur devient automatiquement la couleur de fond pour la fenêtre maîtresse.

Dans cette fenêtre maîtresse, nous avons ajouté un libellé et un bouton, afin de vous montrer comment vous pouvez accéder à la sélection opérée précédemment dans le *ComboBox* lui-même (le bouton provoque l'affichage du nom de la dernière couleur choisie).



```

from Tkinter import *           #1
import Pmw                     #2
                                #3

```

```

def changeCoul(col):                                #4
    fen.configure(background = col)                 #5
                                                    #6
def changeLabel():                                  #7
    lab.configure(text = combo.get())              #8
                                                    #9
couleurs = ('navy', 'royal blue', 'steelblue1', 'cadet blue', #10
            'lawn green', 'forest green', 'dark red',   #11
            'grey80', 'grey60', 'grey40', 'grey20')    #12
                                                    #13
fen = Pmw.initialise()                              #14
bou = Button(fen, text ="Test", command =changeLabel) #15
bou.grid(row =1, column =0, padx =8, pady =6)        #16
lab = Label(fen, text ='néant', bg ='ivory')         #17
lab.grid(row =1, column =1, padx =8)                #18
                                                    #19
combo = Pmw.ComboBox(fen, labelpos = NW,            #20
                     label_text = 'Choisissez la couleur :', #21
                     scrolledlist_items = couleurs,    #22
                     listheight = 150,                #23
                     selectioncommand = changeCoul)    #24
combo.grid(row =2, columnspan =2, padx =10, pady =10) #25
                                                    #26
fen.mainloop()                                     #27

```

Commentaires

- Lignes 1 & 2 : On commence par importer les composants habituels de *Tkinter*, ainsi que le module *Pmw*.
- Ligne 14 : Pour créer la fenêtre maîtresse, il faut utiliser de préférence la méthode `Pmw.initialise()`, plutôt que d'instancier directement un objet de la classe `Tk()`. Cette méthode veille en effet à mettre en place tout ce qui est nécessaire afin que les widgets esclaves de cette fenêtre puissent être détruits correctement lorsque la fenêtre elle-même sera détruite. Cette méthode installe également un meilleur gestionnaire des messages d'erreurs.
- Ligne 12 : L'option `labelpos` détermine l'emplacement du libellé qui accompagne le champ d'entrée. Dans notre exemple, nous l'avons placé au-dessus, mais vous pourriez préférer le placer ailleurs, à gauche par exemple (`labelpos = w`). Notez que cette option est indispensable si vous souhaitez un libellé (pas de valeur par défaut).
- Ligne 14 : L'option `selectioncommand` transmet un argument à la fonction invoquée : l'item sélectionné dans la boîte de liste. Vous pourrez également retrouver cette sélection à l'aide de la méthode `get()`, comme nous le faisons à la ligne 8 pour actualiser le libellé.

Remarque concernant l'entrée de caractères accentués

Nous vous avons déjà signalé précédemment que Python est tout à fait capable de prendre en charge les alphabets du monde entier (grec, cyrillique, arabe, japonais, etc. - voir notamment page). Il en va de même pour *Tkinter*. En tant que francophone, vous souhaitez certainement que

les utilisateurs de vos scripts puissent entrer des caractères accentués dans les widgets *Entry*, *Text* et leurs dérivés (*ComboBox*, *ScrolledText*).

Veillez donc prendre bonne note que lorsque vous entrez dans l'un de ces widgets une chaîne contenant un ou plusieurs caractères non-ASCII (tel qu'une lettre accentuée, par exemple), *Tkinter* encode cette chaîne suivant la norme UTF-8. Si votre ordinateur utilise plutôt le codage Latin-1 par défaut (ce qui est très souvent le cas), vous devrez convertir la chaîne avant de pouvoir l'afficher.

Cela peut se faire très aisément en utilisant la fonction intégrée `encode()`.
Exemple :

```

# -*- coding: Latin-1 -*-
from Tkinter import *

def imprimer():
    ch1 = e.get()          # le widget Entry renvoie une chaîne utf8
    ch2 = ch1.encode("Latin-1") # conversion utf8 -> Latin-1
    print ch2

f = Tk()
e = Entry(f)
e.pack()
Button(f, text="afficher", command =imprimer).pack()
f.mainloop()

```

Essayez ce petit script en entrant des chaînes avec caractères accentués dans le champ d'entrée.

Essayez encore, mais en remplaçant l'instruction `print ch2` par `print ch1`.
Concluez.

« Scrolled Text »

Ce *méga-widget* étend les possibilités du *widget* `Text` standard, en lui associant un cadre, un libellé (titre) et des barres de défilement.

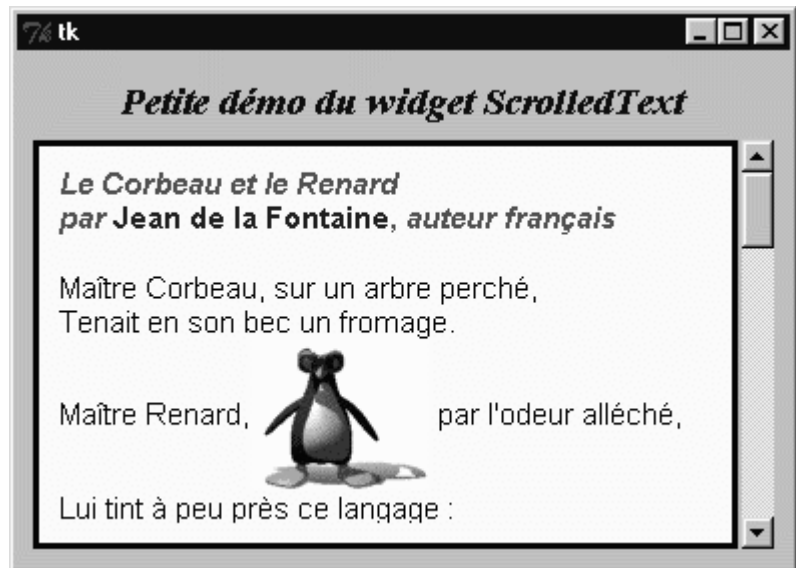
Comme le démontrera le petit script ci-dessous, il sert fondamentalement à afficher des textes, mais ceux-ci peuvent être mis en forme et intégrer des images.

Vous pouvez également rendre « cliquables » les éléments affichés (textes ou images), et vous en servir pour déclencher toutes sortes de mécanismes.

Dans l'application qui génère la figure ci-dessus, par exemple, le fait de cliquer sur le nom « Jean de la Fontaine » provoque le défilement automatique du texte (*scrolling*), jusqu'à ce qu'une rubrique décrivant cet auteur devienne visible dans le *widget* (Voir page suivante le script correspondant).

D'autres fonctionnalités sont présentes, mais nous ne présenterons ici que les plus fondamentales. Veuillez donc consulter les démos et exemples accompagnant *Pmw* pour en savoir davantage.

Gestion du texte affiché :
 Vous pouvez accéder à n'importe quelle portion du texte pris en charge par le *widget* grâce à deux concepts complémentaires, les *indices* et les *balises* :



- Chaque caractère du texte affiché est référencé par un *indice*, lequel doit être une chaîne de caractères contenant deux valeurs numériques reliées par un point (ex : "5.2"). Ces deux valeurs indiquent respectivement le numéro de ligne et le numéro de colonne où se situe le caractère.
- N'importe quelle portion du texte peut être associée à une ou plusieurs *balise(s)*, dont vous choisissez librement le nom et les propriétés. Celles-ci vous permettent de définir la police, les couleurs d'avant- et d'arrière-plan, les événements associés, etc.

Note : Pour la bonne compréhension du script ci-dessous, veuillez considérer que le texte de la fable traitée doit être accessible, dans un fichier nommé *CorbRenard.txt*.

```

from Tkinter import *                                     #1
import Pmw                                               #2
                                                         #3
def action(event=None):                                  #4
    """défilement du texte jusqu'à la balise <cible>""" #5
    index = st.tag_nextrange('cible', '0.0', END)      #6
    st.see(index[0])                                     #7
                                                         #8
# Instanciation d'une fenêtre contenant un widget ScrolledText : #9
fen = Pmw.initialise()                                   #10
st = Pmw.ScrolledText(fen,                               #11
    labelpos =N,                                        #12
    label_text ="Petite démo du widget ScrolledText", #13
    label_font ='Times 14 bold italic',                 #14
    label_fg = 'navy', label_pady =5,                   #15
    text_font='Helvetica 11 normal', text_bg ='ivory', #16
    text_padx =10, text_pady =10, text_wrap ='none',   #17
    borderframe =1,                                     #18
    borderframe_borderwidth =3,                         #19
    borderframe_relief =SOLID,                          #20
    usehullsize =1,                                     #21
    hull_width =370, hull_height =240)                  #22
    
```

```

st.pack(expand =YES, fill =BOTH, padx =8, pady =8) #23
#24
# Définition de balises, liaison d'un gestionnaire d'événement au clic de souris : #25
st.tag_configure('titre', foreground ='brown', font ='Helvetica 11 bold italic') #26
st.tag_configure('lien', foreground ='blue', font ='Helvetica 11 bold') #27
st.tag_configure('cible', foreground ='forest green', font ='Times 11 bold') #28
st.tag_bind('lien', '<Button-1>', action) #29
#30
titre ="" Le Corbeau et le Renard #31
par Jean de la Fontaine, auteur français #32
\n"" #33
auteur ="" #34
Jean de la Fontaine #35
écrivain français (1621-1695) #36
célèbre pour ses Contes en vers, #37
et surtout ses Fables, publiées #38
de 1668 à 1694."" #39
#40
# Remplissage du widget Text (2 techniques) : #41
st.importfile('CorbRenard.txt') #42
st.insert('0.0', titre, 'titre') #43
st.insert(END, auteur, 'cible') #44
# Insertion d'une image : #45
photo =PhotoImage(file= 'Penguin.gif') #46
st.image_create('6.14', image =photo) #47
# Mise en oeuvre dynamique d'une balise : #48
st.tag_add('lien', '2.4', '2.23') #49
#50
fen.mainloop() #51

```

Commentaires

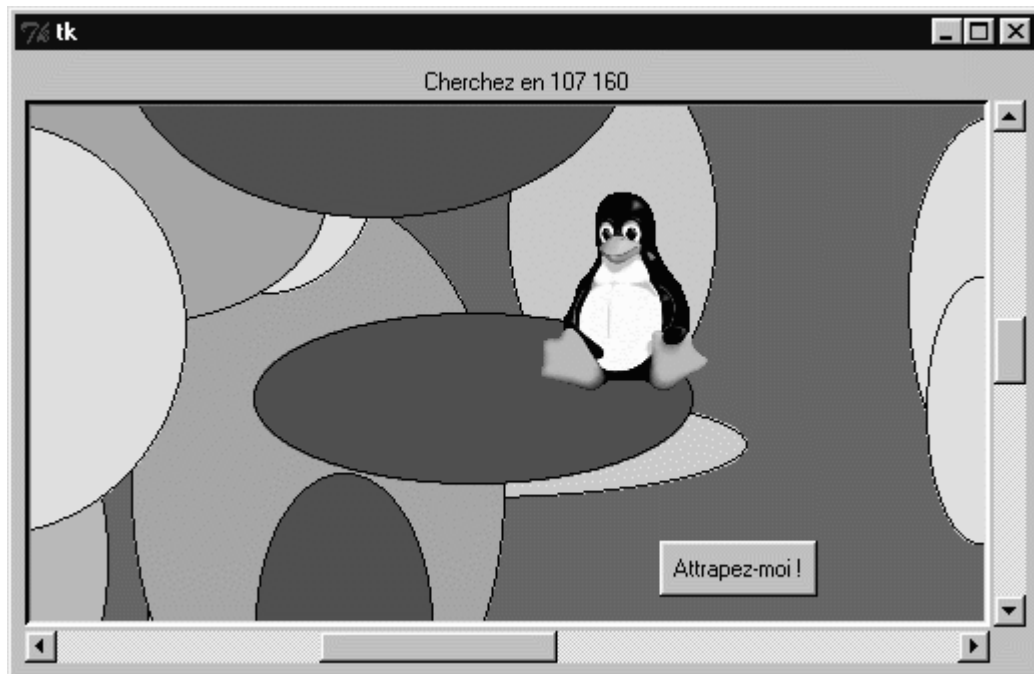
- Lignes 4-7 : Cette fonction est un gestionnaire d'événement, qui est appelé lorsque l'utilisateur effectue un clic de souris sur le nom de l'auteur (cf. lignes 27 et 29). A la ligne 6, on utilise la méthode `tag_nextrange()` du *widget* pour trouver les *indices* de la portion de texte associée à la *balise* `cible`. La recherche de ces index est limitée au domaine défini par les 2^e et 3^e arguments (dans notre exemple, on recherche du début à la fin du texte entier). La méthode `tag_nextrange()` renvoie une liste de deux indices (ceux des premier et dernier caractères de la portion de texte associée à la balise « cible »). À la ligne 7, nous nous servons d'un seul de ces index (le premier) pour activer la méthode `see()`. Celle-ci provoque un défilement automatique du texte (*scrolling*), de telle manière que le caractère correspondant à l'index transmis devienne visible dans le *widget* (avec en général un certain nombre des caractères qui suivent).
- Lignes 9 à 23 : Construction classique d'une fenêtre destinée à afficher un seul *widget*. Dans le code d'instanciation du *widget*, nous avons inclus un certain nombre d'options destinées à vous montrer une petite partie des nombreuses possibilités de configuration.
- Ligne 12 : L'option `labelpos` détermine l'emplacement du libellé (titre) par rapport à la fenêtre de texte. Les valeurs acceptées s'inspirent des lettres utilisées pour désigner les points cardinaux (N, S, E, W, ou encore NE, NW, SE, SW). Si vous ne souhaitez pas afficher un libellé, il vous suffit tout simplement de ne pas utiliser cette option.

- Lignes 13 à 15 : Le libellé n'est rien d'autre qu'un *widget* `Label` standard, intégré dans le *widget* composite `ScrolledText`. On peut accéder à toutes ses options de configuration, en utilisant la syntaxe qui est présentée dans ces lignes : on y voit qu'il suffit d'associer le préfixe `label_` au nom de l'option que l'on souhaite activer, pour définir aisément les couleurs d'avant- et d'arrière-plans, la police, la taille, et même l'espacement à réserver autour du *widget* (option `padx`).
- Lignes 16-17 : En utilisant une technique similaire à celle qui est décrite ci-dessus pour le libellé, on peut accéder aux options de configuration du *widget* `Text` intégré dans `ScrolledText`. Il suffit cette fois d'associer aux noms d'option le préfixe `text_`.
- Lignes 18 à 20 : Il est prévu un cadre (un *widget* `Frame`) autour du *widget* `Text`. L'option `borderframe = 1` permet de le faire apparaître. On accède ensuite à ses options de configuration d'une manière similaire à celle qui a été décrite ci-dessus pour `label_` et `text_`.
- Lignes 21-22 : Ces options permettent de fixer globalement les dimensions du *widget*. Une autre possibilité serait de définir plutôt les dimensions de son composant `Text` (par exemple à l'aide d'options telles que `text_width` et `text_height`), mais alors les dimensions globales du *widget* risqueraient de changer en fonction du contenu (apparition/disparition automatique de barres de défilement). Note : le mot *hull* désigne le contenant global, c'est à dire le *méga-widget* lui-même.
- Ligne 23 : Les options `expand = YES` et `fill = BOTH` de la méthode `pack()` indiquent que le *widget* concerné pourra être redimensionné à volonté, dans ses 2 dimensions horiz. et verticale.
- Lignes 26 à 29 : Ces lignes définissent les trois *balises* `titre`, `lien` et `cible` ainsi que le formatage du texte qui leur sera associé. La ligne 29 précise en outre que le texte associé à la balise `lien` sera « cliquable », avec indication du gestionnaire d'événement correspondant.
- Ligne 42 : Importation de texte à partir d'un fichier. Note : Il est possible de préciser l'endroit exact où devra se faire l'insertion, en fournissant un index comme second argument.
- Lignes 43-44 : Ces instructions insèrent des fragments de texte (respectivement au début et à la fin du texte préexistant), en associant une balise à chacun d'eux.
- Ligne 49 : L'association des balises au texte est dynamique. A tout moment, vous pouvez activer une nouvelle association (comme nous le faisons ici en rattachant la balise « lien » à une portion de texte préexistante). Note : pour « détacher » une balise, utilisez la méthode `tag_delete()`.

« Scrolled Canvas »

Le script ci-après vous montre comment vous pouvez exploiter le *méga-widget*

`ScrolledCanvas`, lequel étend les possibilités du *widget* `Canvas` standard en lui associant des barres de défilement, un libellé et un cadre. Notre exemple constitue en fait un petit jeu d'adresse, dans lequel l'utilisateur doit réussir à cliquer sur un bouton qui s'esquive sans cesse. (Note : si vous éprouvez vraiment des difficultés pour l'attraper, commencez d'abord par dilater la fenêtre).



Le *widget* `Canvas` est très versatile : il vous permet de combiner à volonté des dessins, des images *bitmap*, des fragments de texte, et même d'autres *widgets*, dans un espace parfaitement extensible. Si vous souhaitez développer l'un ou l'autre jeu graphique, c'est évidemment le *widget* qu'il vous faut apprendre à maîtriser en priorité.

Comprenez bien cependant que les indications que nous vous fournissons à ce sujet dans les présentes notes sont forcément très incomplètes. Leur objectif est seulement de vous aider à comprendre quelques concepts de base, afin que vous puissiez ensuite consulter les ouvrages de référence spécialisés dans de bonnes conditions.

Notre petite application se présente comme une nouvelle classe `FenPrinc()`, obtenue par dérivation à partir de la classe de *méga-widgets* `Pmw.ScrolledCanvas()`. Elle contient donc un grand canevas muni de barres de défilement, dans lequel nous commençons par planter un décor constitué de 80 ellipses de couleur dont l'emplacement et les dimensions sont tirés au hasard.

Nous y ajoutons également un petit clin d'œil sous la forme d'une image *bitmap*, destinée avant tout à vous rappeler comment vous pouvez gérer ce type de ressource.

Nous y installons enfin un véritable *widget* : un simple bouton, en

l'occurrence, mais la technique mise en œuvre pourrait s'appliquer à n'importe quel autre type de *widget*, y compris un gros *widget* composite comme ceux que nous avons développés précédemment. Cette grande souplesse dans le développement d'applications complexes est l'un des principaux bénéfices apportés par le mode de programmation « orientée objet ».

Le bouton s'anime dès qu'on l'a enfoncé une première fois. Dans votre analyse du script ci-après, soyez attentifs aux méthodes utilisées pour modifier les propriétés d'un objet existant.

```

from Tkinter import * #1
import Pmw #2
from random import randrange #3
#4
Pmw.initialise() #5
coul = ['sienna', 'maroon', 'brown', 'pink', 'tan', 'wheat', 'gold', 'orange', 'plum', #6
        'red', 'khaki', 'indian red', 'thistle', 'firebrick', 'salmon', 'coral'] #7
#8
class FenPrinc(Pmw.ScrolledCanvas): #9
    """Fenêtre principale : canevas extensible avec barres de défilement""" #10
    def __init__(self): #11
        Pmw.ScrolledCanvas.__init__(self, #12
            usehullsize =1, hull_width =500, hull_height =300, #13
            canvas_bg = 'grey40', canvasmargin =10, #14
            labelpos =N, label_text = 'Attrapez le bouton !', #15
            borderframe =1, #16
            borderframe_borderwidth =3) #17
        # Les options ci-dessous doivent être précisées après initialisation : #18
        self.configure(vscrollmode = 'dynamic', hscrollmode = 'dynamic') #19
        self.pack(padx =5, pady =5, expand =YES, fill =BOTH) #20
#21
        self.can = self.interior() # accès au composant canevas #22
        # Décor : tracé d'une série d'ellipses aléatoires : #23
        for r in range(80): #24
            x1, y1 = randrange(-800,800), randrange(-800,800) #25
            x2, y2 = x1 + randrange(40,300), y1 + randrange(40,300) #26
            couleur = coul[randrange(0,16)] #27
            self.can.create_oval(x1, y1, x2, y2, fill=couleur, outline='black') #28
        # Ajout d'une petite image GIF : #29
        self.img = PhotoImage(file = 'linux2.gif') #30
        self.can.create_image(50, 20, image =self.img) #31
        # Dessin du bouton à attraper : #32
        self.x, self.y = 50, 100 #33
        self.bou = Button(self.can, text = "Start", command =self.start) #34
        self.fb = self.can.create_window(self.x, self.y, window =self.bou) #35
        self.resizescrollregion() #36
#37
    def anim(self): #38
        if self.run ==0: #39
            return #40
        self.x += randrange(-60, 61) #41
        self.y += randrange(-60, 61) #42
        self.can.coords(self.fb, self.x, self.y) #43
        self.configure(label_text = 'Cherchez en %s %s' % (self.x, self.y)) #44
        self.resizescrollregion() #45
        self.after(250, self.anim) #46
#47
    def stop(self): #48

```

```

        self.run =0                                     #49
        self.bou.configure(text ="Restart", command =self.start) #50
                                                         #51
    def start(self):                                     #52
        self.bou.configure(text ="Attrapez-moi !", command =self.stop) #53
        self.run =1                                     #54
        self.anim()                                     #55
                                                         #56
##### Main Program #####                               #57
                                                         #58
if __name__ == '__main__':                             #59
    FenPrinc().mainloop()                               #60

```

Commentaires

- Ligne 6 : Tous ces noms de couleurs sont acceptés par *Tkinter*. Vous pourriez bien évidemment les remplacer par des descriptions hexadécimales, comme nous l'avons expliqué page



À faire...

- Lignes 12 à 17 : Ces options sont très similaires à celles que nous avons décrites plus haut pour le *widget ScrolledText*. Le présent *méga-widget* intègre un composant `Frame`, un composant `Label`, un composant `Canvas` et deux composants `Scrollbar`. On accède aux options de configuration de ces composants à l'aide d'une syntaxe qui relie le nom du composant et celui de l'option par l'intermédiaire d'un caractère « souligné ».
- Ligne 19 : Ces options définissent le mode d'apparition des barres de défilement. En mode « *static* », elles sont toujours présentes. En mode « *dynamic* », elles disparaissent si les dimensions du canevas deviennent inférieures à celles de la fenêtre de visualisation.
- Ligne 22 : La méthode `interior()` renvoie la référence du composant `Canvas` intégré dans le *méga-widget* `ScrolledCanvas`. Les instructions suivantes (lignes 23 à 35) installent ensuite toute une série d'éléments dans ce canevas : des dessins, une image et un bouton.
- Lignes 25 à 27 : La fonction `randrange()` permet de tirer au hasard un nombre entier compris dans un certain intervalle (Veuillez vous référer aux explications de la page



À faire...

).

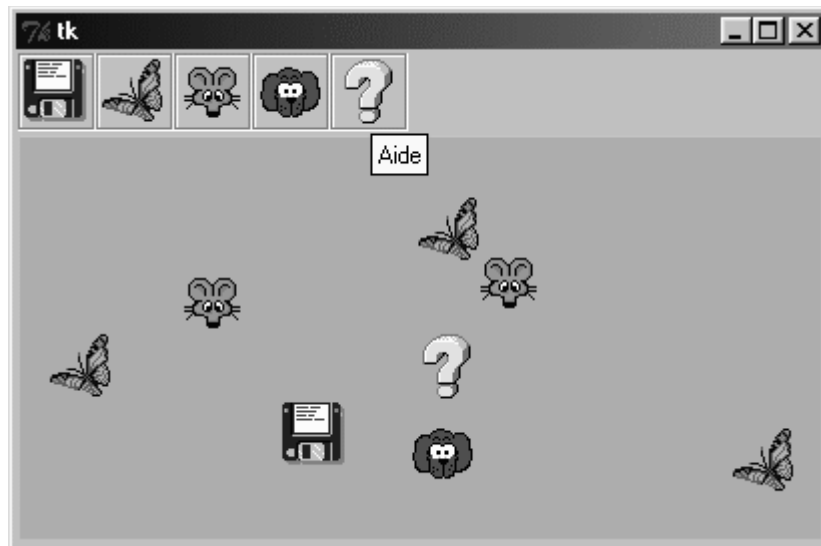
- Ligne 35 : C'est la méthode `create_window()` du *widget* `Canvas` qui permet d'y insérer n'importe quel autre *widget* (y compris un *widget* composite). Le *widget* à insérer doit cependant avoir été défini lui-même au préalable comme un esclave du canevas ou de sa fenêtre maîtresse. La méthode `create_window()` attend trois arguments : les coordonnées X et Y du point où l'on souhaite insérer le widget, et la référence de ce widget.
- Ligne 36 : La méthode `resizescrollregion()` réajuste la situation des barres de défilement de manière à ce qu'elles soient en accord avec la portion du canevas actuellement affichée.
- Lignes 38 à 46 : Cette méthode est utilisée pour l'animation du bouton. Après avoir repositionné le bouton au hasard à une certaine distance de sa position précédente, elle se ré-appelle elle-même après une pause de 250 millisecondes. Ce bouclage s'effectue sans cesse, aussi longtemps que la variable `self.run` contient une valeur non-nulle.
- Lignes 48 à 55 : Ces deux gestionnaires d'événement sont associés au bouton en alternance. Ils servent évidemment à démarrer et à arrêter l'animation.

Barres d'outils avec bulles d'aide - expressions lambda

De nombreux programmes comportent une ou plusieurs « barres d'outils » (*toolbar*) constituées de petits boutons sur lesquels sont représentés des pictogrammes (icônes). Cette façon de faire permet de proposer à l'utilisateur un grand nombre de commandes spécialisées, sans que celles-ci n'occupent une place excessive à l'écran (un petit dessin vaut mieux qu'un long discours, dit-on).

La signification de ces pictogrammes n'est cependant pas toujours évidente, surtout pour les utilisateurs néophytes. Il est donc vivement conseillé de compléter les barres d'outils à l'aide d'un système de bulles d'aide (*tool tips*), qui sont des petits messages explicatifs apparaissant automatiquement lorsque la souris survole les boutons concernés.

L'application décrite ci-après comporte une barre d'outils et un canevas. Lorsque l'utilisateur clique sur l'un des boutons de la barre, le pictogramme qu'il porte est recopié dans le canevas, à un emplacement choisi au hasard :



Dans notre exemple, chaque bouton apparaît entouré d'un sillon. Vous pouvez aisément obtenir d'autres aspects en choisissant judicieusement les options `relief` et `bd` (bordure) dans l'instruction d'instanciation des boutons. En particulier, vous pouvez choisir `relief = FLAT` et `bd = 0` pour obtenir des petits boutons « plats », sans aucun relief.

La mise en place des bulles d'aide est un jeu d'enfant. Il suffit d'instancier un seul objet `Pmw.Balloon` pour l'ensemble de l'application, puis d'associer un texte à chacun des *widgets* auxquels on souhaite associer une bulle d'aide, en faisant appel autant de fois que nécessaire à la méthode `bind()` de cet objet.

```

from Tkinter import *
import Pmw
from random import randrange

# noms des fichiers contenant les icônes (format GIF):
images = ('floppy_2', 'papi2', 'pion_1', 'pion_2', 'help_4')
textes = ('sauvegarde', 'papillon', 'joueur 1', 'joueur 2', 'Aide')

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        # Création d'un objet <bulle d'aide> (un seul suffit) :
        tip = Pmw.Balloon(self)
        # Création de la barre d'outils (c'est un simple cadre) :
        toolbar = Frame(self, bd =1)
        toolbar.pack(expand =YES, fill =X)
        # Nombre de boutons à construire :
        nBou = len(images)
        # Les icônes des boutons doivent être placées dans des variables
        # persistantes. Une liste fera l'affaire :
        self.photoI =[None]*nBou

        for b in range(nBou):
            # Création de l'icône (objet PhotoImage Tkinter) :
            self.photoI[b] =PhotoImage(file = images[b] +'.gif')

            # Création du bouton.:
            # On utilise une expression "lambda" pour transmettre
            # un argument à la méthode invoquée comme commande :

```

```

        bou = Button(toolbar, image =self.photoI[b], relief =GROOVE,
                    command = lambda arg =b: self.action(arg))
        bou.pack(side =LEFT)

        # association du bouton avec un texte d'aide (bulle) :
        tip.bind(bou, textes[b])

    self.ca = Canvas(self, width =400, height =200, bg ='orange')
    self.ca.pack()
    self.pack()

def action(self, b):
    "l'icône du bouton b est recopiée dans le canevas"
    x, y = randrange(25,375), randrange(25,175)
    self.ca.create_image(x, y, image =self.photoI[b])

Application().mainloop()

```

Métaprogrammation. Expressions lambda :

Vous savez qu'en règle générale, on associe à chaque bouton une *commande*, laquelle est une méthode ou une fonction particulière qui se charge d'effectuer le travail lorsque le bouton est activé. Or dans l'application présente, tous les boutons doivent faire à peu près la même chose (recopier un dessin dans le canevas), la seule différence entre eux étant le dessin concerné.

Pour simplifier notre code, nous voudrions donc pouvoir associer l'option `command` de tous nos boutons avec une seule et même méthode (ce sera la méthode `action()`), mais en lui transmettant à chaque fois la référence du bouton particulier utilisé, de manière à ce que l'action accomplie puisse être différente pour chacun d'eux.

Une difficulté se présente, cependant, parce que l'option `command` du *widget* `Button` accepte seulement une *valeur* ou une *expression*, et non une *instruction*. Il est donc permis de lui indiquer la référence d'une fonction, mais pas de l'invoquer véritablement en lui transmettant des arguments éventuels (c'est la raison pour laquelle on indique le nom de cette fonction sans lui adjoindre de parenthèses).

On peut résoudre cette difficulté de deux manières :

li> Du fait de son caractère *dynamique*, Python accepte qu'*un programme puisse se modifier lui-même*, par exemple en définissant de nouvelles fonctions au cours de son exécution (c'est le concept de *métaprogrammation*). Il est donc possible de définir à la volée une fonction qui utilise des paramètres, en indiquant pour chacun de ceux-ci une valeur par défaut, et ensuite d'invoquer cette même fonction sans arguments là où ceux-ci ne sont pas autorisés. Puisque la fonction est définie en cours d'exécution, les valeurs par défaut peuvent être les contenus de variables, et le résultat de l'opération est un véritable transfert d'arguments. Pour illustrer cette technique, remplacez les lignes 27 à 31 du script par les suivantes :

```

# Création du bouton.:

```

```
# On définit à la volée une fonction avec un paramètre, dont
# la valeur par défaut est l'argument à transmettre.
# Cette fonction appelle la méthode qui nécessite un argument :
def agir(arg = b):
    self.action(arg)
# La commande associée au bouton appelle la fonction ci-dessus :
bou = Button(toolbar, image =self.photoI[b], relief =GROOVE,
              command = agir)
```

- Tout ce qui précède peut être simplifié en faisant appel à une expression *lambda*. Ce mot réservé Python désigne une *expression* qui renvoie un objet fonction, similaire à ceux que vous créez avec l'instruction `def`, mais avec la différence que `lambda` étant une expression et non une instruction, on peut l'utiliser comme interface afin d'invoquer une fonction (avec passage d'arguments) là où ce n'est normalement pas possible. Notez au passage qu'une telle fonction est anonyme (elle ne possède pas de nom). Par exemple, l'instruction `: lambda ar1=b, ar2=c : bidule(ar1,ar2)` renvoie la référence d'une fonction anonyme qui aura elle-même invoqué la fonction `bidule()` en lui transmettant les arguments `b` et `c`, ceux-ci étant utilisés comme valeurs par défaut dans la définition des paramètres de la fonction. Cette technique utilise finalement le même principe que la précédente, mais elle présente l'avantage d'être plus concise, raison pour laquelle nous l'avons utilisée dans notre script. En revanche, elle est un peu plus difficile à comprendre : `command = lambda arg =b: self.action(arg)` Dans cette portion d'instruction, la commande associée au bouton se réfère à une fonction anonyme dont le paramètre `arg` possède une valeur par défaut : la valeur de l'argument `b`. Invoquée sans argument par la commande, cette fonction anonyme peut tout de même utiliser son paramètre (avec la valeur par défaut) pour faire appel à la méthode cible `self.action()`, et l'on obtient ainsi un véritable transfert d'argument vers cette méthode.

Nous ne détaillerons pas davantage ici la question des expressions `lambda`, car elle déborde du cadre que nous nous sommes fixés pour cet ouvrage d'initiation. Si vous souhaitez en savoir plus, veuillez donc consulter l'un ou l'autre des ouvrages de référence cités dans la bibliographie.

Fenêtres avec menus

Nous allons décrire à présent la construction d'une fenêtre d'application dotée de différents types de menus « déroulants », chacun de ces menus pouvant être « détaché » de l'application principale pour devenir lui-même une petite fenêtre indépendante, comme dans l'illustration ci-dessous.

Cet exercice un peu plus long nous servira également de révision, et nous le réaliserons par étapes, en appliquant une stratégie de programmation que l'on appelle *développement incrémental*.

Comme nous l'avons déjà expliqué précédemment^[1], cette méthode consiste à commencer l'écriture d'un programme par une ébauche, qui ne comporte que

quelques lignes seulement mais qui est déjà fonctionnelle. On teste alors cette ébauche soigneusement afin d'en éliminer les *bugs* éventuels. Lorsque l'ébauche fonctionne correctement, on y ajoute une fonctionnalité supplémentaire. On teste ce complément jusqu'à ce qu'il donne entière satisfaction, puis on en ajoute un autre, et ainsi de suite...

Cela ne signifie pas que vous pouvez commencer directement à programmer sans avoir au préalable effectué une analyse sérieuse du projet, dont au moins les grandes lignes devront être convenablement décrites dans un cahier des charges clairement rédigé.



Remarque : il reste également impératif de commenter convenablement le code produit, au fur et à mesure de son élaboration. S'efforcer de rédiger de bons commentaires est en effet nécessaire, non seulement pour que votre code soit facile à lire (et donc à maintenir plus tard, par d'autres ou par vous-même), mais aussi pour que vous soyez forcés d'exprimer ce que vous souhaitez vraiment que la machine fasse (cf. *Erreurs sémantiques*).

Cahier des charges de l'exercice

Notre application comportera simplement une barre de menus et un canevas. Les différentes rubriques et options des menus ne serviront qu'à faire apparaître des fragments de texte dans le canevas ou à modifier des détails de décoration, mais ce seront avant tout des exemples variés, destinés à donner un aperçu des nombreuses possibilités offertes par ce type de *widget*, accessoire indispensable de toute application moderne d'une certaine importance.

Nous souhaitons également que le code produit dans cet exercice soit bien structuré. Pour ce faire, nous ferons usage de deux classes : une classe pour l'application principale, et une autre pour la barre de menus. Nous voulons procéder ainsi afin de bien mettre en évidence la construction d'une application type incorporant plusieurs classes d'objets interactifs.

Première ébauche du programme :

Lorsque l'on construit l'ébauche d'un programme, il faut tâcher d'y faire apparaître le plus tôt possible la structure d'ensemble, avec les relations entre

les principaux blocs qui constitueront l'application définitive. C'est ce que nous nous sommes efforcés de faire dans l'exemple ci-dessous :

```

from Tkinter import * #1
#2
class MenuBar(Frame): #3
    """Barre de menus déroulants""" #4
    def __init__(self, boss=None): #5
        Frame.__init__(self, borderwidth =2) #6
        #7
        ##### Menu <Fichier> ##### #8
        fileMenu = Menubutton(self, text='Fichier') #9
        fileMenu.pack(side =LEFT) #10
        # Partie "déroulante" : #11
        me1 = Menu(fileMenu) #12
        me1.add_command(label='Effacer', underline =0, #13
            command = boss.effacer) #14
        me1.add_command(label='Terminer', underline =0, #15
            command = boss.quit) #16
        # Intégration du menu : #17
        fileMenu.configure(menu = me1) #18
        #19
class Application(Frame): #20
    """Application principale""" #21
    def __init__(self, boss=None): #22
        Frame.__init__(self) #23
        self.master.title('Fenêtre avec menus') #24
        mBar = MenuBar(self) #25
        mBar.pack() #26
        self.can = Canvas(self, bg='light grey', height=190, #27
            width=250, borderwidth =2) #28
        self.can.pack() #29
        self.pack() #30
        #31
    def effacer(self): #32
        self.can.delete(ALL) #33
        #34
if __name__ == '__main__': #35
    app = Application() #36
    app.mainloop() #37

```

Veillez donc encoder ces lignes et en tester l'exécution. Vous devriez obtenir une fenêtre avec un canevas gris clair surmonté d'une barre de menus. A ce stade, la barre de menus ne comporte encore que la seule rubrique « Fichier ».

Cliquez sur la rubrique « fichier » pour faire apparaître le menu correspondant : l'option « Effacer » n'est pas encore fonctionnelle (elle servira à effacer le contenu du canevas), mais l'option « Terminer » devrait déjà vous permettre de fermer proprement l'application.



Comme tous les menus gérés par *Tkinter*, le menu que vous avez créé peut être

converti en menu « flottant » : il suffit de cliquer sur la ligne pointillée apparaissant en-tête de menu. Vous obtenez ainsi une petite fenêtre satellite, que vous pouvez alors positionner où bon vous semble sur le bureau.

Analyse du script

La structure de ce petit programme devrait désormais vous apparaître familière : afin que les classes définies dans ce script puissent éventuellement être (ré)utilisées dans d'autres projets par importation, comme nous l'avons déjà expliqué précédemment^[2], le corps principal du programme (lignes 35 à 37) comporte l'instruction classique : `if __name__ == '__main__':`

Les deux instructions qui suivent consistent seulement à instancier un objet `app` et à faire fonctionner sa méthode `mainloop()`. Comme vous le savez certainement, nous aurions pu également condenser ces deux instructions en une seule.

L'essentiel du programme se trouve cependant dans les définitions de classes qui précèdent :

La classe `MenuBar()` contient la description de la barre de menus. Dans l'état présent du script, elle se résume à une ébauche de constructeur.

- Ligne 5 : Le paramètre `boss` réceptionne la référence de la fenêtre maîtresse du *widget* au moment de son instantiation. Cette référence va nous permettre d'invoquer les méthodes associées à cette fenêtre maîtresse, aux lignes 14 & 16.
- Ligne 6 : Activation obligatoire du constructeur de la classe parente.
- Ligne 9 : Instantiation d'un *widget* de la classe `Menubutton()`, défini comme un « esclave » de `self` (c'est-à-dire l'objet composite « barre de menus » dont nous sommes occupés à définir la classe). Comme l'indique son nom, ce type de *widget* se comporte un peu comme un bouton : une action se produit lorsque l'on clique dessus.
- Ligne 12 : Afin que cette action consiste en l'apparition véritable d'un menu, il reste encore à définir celui-ci : ce sera encore un nouveau *widget*, de la classe `Menu()` cette fois, défini lui-même comme un « esclave » du *widget* `Menubutton` instancié à la ligne 9.
- Lignes 13 à 16 : On peut appliquer aux *widgets* de la classe `Menu()` un certain nombre de méthodes spécifiques, chacune d'elles acceptant de nombreuses options. Nous utilisons ici la méthode `add_command()` pour installer dans le menu les deux items « Effacer » et « Terminer ». Nous y intégrons tout de suite l'option `underline`, qui sert à définir un *raccourci clavier* : cette option indique en effet lequel des caractères de l'item doit apparaître souligné à l'écran. L'utilisateur sait alors qu'il lui suffit de frapper ce caractère au clavier pour que l'action correspondant à cet item soit activée (comme s'il avait cliqué dessus à l'aide de la souris).
L'action à déclencher lorsque l'utilisateur sélectionne l'item est désignée par

l'option *command*. Dans notre script, les commandes invoquées sont toutes les deux *des méthodes de la fenêtre maîtresse*, dont la référence aura été transmise au présent *widget* au moment de son instantiation par l'intermédiaire du paramètre *boss*. La méthode *effacer()*, que nous définissons nous-même plus loin, servira à vider le canevas. La méthode prédéfinie *quit()* provoque la sortie de la boucle *mainloop()* et donc l'arrêt du réceptionnaire d'événements associé à la fenêtre d'application.

- Ligne 18 : Lorsque les items du menu ont été définis, il reste encore à reconfigurer le *widget* maître *Menubutton* de manière à ce que son option « menu » désigne effectivement le *Menu* que nous venons de construire. En effet, nous ne pouvions pas déjà préciser cette option lors de la définition initiale du *widget* *Menubutton*, puisqu'à ce stade le *Menu* n'existait pas encore. Nous ne pouvions pas non plus définir le *widget* *Menu* en premier lieu, puisque celui-ci doit être défini comme un « esclave » du *widget* *Menubutton*. Il faut donc bien procéder en trois étapes comme nous l'avons fait, en faisant appel à la méthode *configure()*. (Cette méthode peut être appliquée à n'importe quel *widget* préexistant pour en modifier l'une ou l'autre option).

La classe *Application()* contient la description de la fenêtre principale du programme ainsi que les méthodes gestionnaires d'événements qui lui sont associées.

- Ligne 20 : Nous préférons faire dériver notre application de la classe *Frame()*, qui présente de nombreuses options, plutôt que de la classe primordiale *Tk()*. De cette manière, l'application toute entière est encapsulée dans un *widget*, lequel pourra éventuellement être intégré par la suite dans une application plus importante. Rappelons que de toute manière, *Tkinter*instanciera automatiquement une fenêtre maîtresse de type *Tk()* pour contenir de cette *Frame*.
- Lignes 23-24 : Après l'indispensable activation du constructeur de la classe parente, nous utilisons l'attribut *master* que *Tkinter* associe automatiquement à chaque *widget*, pour référencer la fenêtre principale de l'application (la fenêtre maîtresse dont nous venons de parler au paragraphe précédent) et en redéfinir le bandeau-titre.
- Lignes 25 à 29 : Instantiation de deux *widgets* esclaves pour notre *Frame* principale. La « barre de menus » est évidemment le *widget* défini dans l'autre classe.
- Ligne 30 : Comme n'importe quel autre *widget*, notre *Frame* principale doit être mise en place.
- Lignes 32-33 : La méthode servant à effacer le canevas est définie dans la classe présente (puisque l'objet canevas en fait partie), mais elle est invoquée par l'option *command* d'un *widget* esclave défini dans l'autre classe. Comme nous l'avons expliqué plus haut, ce *widget* esclave reçoit la référence de son widget maître par l'intermédiaire du paramètre *boss*. Toutes ces références sont hiérarchisées à l'aide de la qualification des noms par points.

Ajout de la rubrique « Musiciens »

Continuez le développement de ce petit programme, en ajoutant les lignes suivantes dans le constructeur de la classe `MenuBar()` (après la ligne 18) :

```
##### Menu <Musiciens> #####
self.musi = Menubutton(self, text = 'Musiciens')
self.musi.pack(side = LEFT, padx = '3')
# Partie "déroulante" du menu <Musiciens> :
me1 = Menu(self.musi)
me1.add_command(label = '17e siècle', underline = 1,
                foreground = 'red', background = 'yellow',
                font = ('Comic Sans MS', 11),
                command = boss.showMusi17)
me1.add_command(label = '18e siècle', underline = 1,
                foreground = 'royal blue', background = 'white',
                font = ('Comic Sans MS', 11, 'bold'),
                command = boss.showMusi18)
# Intégration du menu :
self.musi.configure(menu = me1)
```

... ainsi que les définitions de méthodes suivantes à la classe `Application()` (après la ligne 33) :

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

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

Lorsque vous y aurez ajouté toutes ces lignes, sauvegardez le script et exécutez-le.

Votre barre de menus comporte à présent une rubrique supplémentaire : la rubrique « Musiciens ».

Le menu correspondant propose deux items qui sont affichés avec des couleurs et des polices personnalisées. Vous pourrez vous inspirer de ces techniques décoratives pour vos projets personnels. À utiliser avec modération !

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



Analyse du script

Les seules nouveautés introduites dans ces lignes concernent l'utilisation de polices de caractères bien déterminées (option `font`), ainsi que de couleurs pour l'avant-plan (option `foreground`) et le fond (option `background`) des textes affichés.

Veillez noter encore une fois l'utilisation de l'option `underline` pour désigner les caractères correspondant à des raccourcis claviers (en n'oubliant pas que la numérotation des caractères d'une chaîne commence à partir de zéro), et surtout que l'option `command` de ces *widgets* accède aux méthodes de l'autre classe, par l'intermédiaire de la référence mémorisée dans l'attribut `boss`.

La méthode `create_text()` du canevas doit être utilisée avec deux arguments numériques, qui sont les coordonnées X et Y d'un point dans le canevas. Le texte transmis sera positionné par rapport à ce point, en fonction de la valeur choisie pour l'option `anchor` : Celle-ci détermine comment le fragment de texte doit être « ancré » au point choisi dans le canevas, par son centre, par son coin supérieur gauche, etc., en fonction d'une syntaxe qui utilise l'analogie des points cardinaux géographiques (NW = angle supérieur gauche, SE = angle inférieur droit, CENTER = centre, etc.)

Ajout de la rubrique « Peintres » :

Cette nouvelle rubrique est construite d'une manière assez semblable à la précédente, mais nous lui avons ajouté une fonctionnalité supplémentaire : des menus « en cascade ». Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe `MenuBar()` :

```
##### Menu <Peintres> #####
self.pein = Menubutton(self, text = 'Peintres')
self.pein.pack(side = LEFT, padx = '3')
# Partie "déroulante" :
me1 = Menu(self.pein)
me1.add_command(label = 'classiques', state=DISABLED)
me1.add_command(label = 'romantiques', underline = 0,
                 command = boss.showRomanti)
# Sous-menu pour les peintres impressionistes :
me2 = Menu(me1)
me2.add_command(label = 'Claude Monet', underline = 7,
                 command = boss.tabMonet)
me2.add_command(label = 'Auguste Renoir', underline = 8,
                 command = boss.tabRenoir)
me2.add_command(label = 'Edgar Degas', underline = 6,
                 command = boss.tabDegas)
# Intégration du sous-menu :
me1.add_cascade(label = 'impressionistes', underline=0, menu = me2)
# Intégration du menu :
self.pein.configure(menu = me1)
```

... et les définitions suivantes dans la classe `Application()` :

```
def showRomanti(self):
```

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

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

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

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

Analyse du script

Vous pouvez réaliser aisément des menus en cascade, en enchaînant des sous-menus les uns aux autres jusqu'à un niveau quelconque (il vous est cependant déconseillé d'aller au-delà de 5 niveaux successifs : vos utilisateurs s'y perdraient).

Un sous-menu est défini comme un menu « esclave » du menu de niveau précédent (dans notre exemple, `me2` est défini comme un menu « esclave » de `me1`). L'intégration est assurée ensuite à l'aide de la méthode `add_cascade()`.

L'un des items est désactivé (option `state = DISABLED`). L'exemple suivant vous montrera comment vous pouvez activer ou désactiver à volonté des items, par programme.

Ajout de la rubrique « Options » :

La définition de cette rubrique est un peu plus compliquée, parce que nous allons y intégrer l'utilisation de variables internes à *Tkinter*.

Les fonctionnalités de ce menu sont cependant beaucoup plus élaborées : les options ajoutées permettent en effet d'activer ou de désactiver à volonté les rubriques « Musiciens » et « Peintres », et vous pouvez également modifier à volonté l'aspect de la barre de menus elle-même.



Veillez donc ajouter les lignes suivantes dans le constructeur de la classe `MenuBar()` :

```
##### Menu <Options> #####
optMenu = Menubutton(self, text = 'Options')
optMenu.pack(side = LEFT, padx = '3')
# Variables Tkinter :
self.relief = IntVar()
self.actPein = IntVar()
self.actMusi = IntVar()
# Partie "déroulante" du menu :
self.mo = Menu(optMenu)
self.mo.add_command(label = 'Activer :', foreground = 'blue')
self.mo.add_checkbutton(label = 'musiciens',
                        command = self.choixActifs, variable = self.actMusi)
self.mo.add_checkbutton(label = 'peintres',
                        command = self.choixActifs, variable = self.actPein)
self.mo.add_separator()
self.mo.add_command(label = 'Relief :', foreground = 'blue')
for (v, lab) in [(0, 'aucun'), (1, 'sorti'), (2, 'rentré'),
                (3, 'sillon'), (4, 'crête'), (5, 'bordure')]:
    self.mo.add_radiobutton(label = lab, variable = self.relief,
                           value = v, command = self.reliefBarre)

# Intégration du menu :
optMenu.configure(menu = self.mo)
```

... ainsi que les définitions de méthodes suivantes (toujours dans la classe `MenuBar()`) :

```
def reliefBarre(self):
    choix = self.relief.get()
    self.configure(relief = [FLAT, RAISED, SUNKEN, GROOVE, RIDGE, SOLID][choix])

def choixActifs(self):
    p = self.actPein.get()
    m = self.actMusi.get()
    self.pein.configure(state = [DISABLED, NORMAL][p])
    self.musi.configure(state = [DISABLED, NORMAL][m])
```

Analyse du script

Menu avec « cases à cocher »

Notre nouveau menu déroulant comporte deux parties. Afin de bien les mettre en évidence, nous avons inséré une ligne de séparation ainsi que deux « faux items » (« Activer : » et « Relief : ») qui servent simplement de titres. Nous faisons apparaître ceux-ci en couleur pour que l'utilisateur ne les confonde pas avec de véritables commandes.

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

```
self.mo.add_checkbox(label = 'musiciens', command = choixActifs,
                    variable = mbu.mel.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 ne peut pas être une variable Python ordinaire, parce que les classes de la bibliothèque *Tkinter* sont écrites dans un autre langage. Et par conséquent, on ne pourra accéder à une telle variable interne qu'à travers une interface. Cette interface, appelée « *variable Tkinter* », est en fait un objet, que l'on crée à partir d'une classe particulière, qui fait partie du module *Tkinter* au même titre que les classes de *widgets*. L'utilisation de ces « objets-variables » est relativement simple :

- La classe `IntVar()` permet de créer des objets équivalents à des variables de type entier. On commence donc par créer un ou plusieurs de ces objets-variables, que l'on mémorise dans notre exemple comme de nouveaux attributs d'instance :

```
self.actMusi =IntVar()
```

Après cette affectation, l'objet référencé dans `self.actMusi` contient désormais l'équivalent d'une variable de type entier, dans un format spécifique à *Tkinter*.

- Ensuite, on associe l'option `variable` de l'objet `checkboxbutton` à la variable *Tkinter* ainsi définie :

```
self.mo.add_checkbox(label = 'musiciens', variable =self.actMusi)
```

- Il est nécessaire de procéder ainsi en deux étapes, parce que *Tkinter* ne peut pas directement assigner des valeurs aux variables Python. Pour une raison similaire, il n'est pas possible à Python de lire directement le contenu d'une *variable Tkinter*. Il faut utiliser pour cela une méthode spécifique de cette

classe d'objets : la méthode `get()`^[3] :

```
m = self.actMusi.get()
```

Dans cette instruction, nous affectons à `m` (variable ordinaire de Python) le contenu d'une *variable Tkinter* (laquelle est elle-même associée à un *widget* bien déterminé).

Tout ce qui précède peut vous paraître un peu compliqué. Considérez simplement qu'il s'agit de votre première rencontre avec les problèmes d'*interfaçage* entre deux langages de programmation différents, utilisés ensemble dans un projet composite.

Menu avec choix exclusifs

La deuxième partie du menu « Options » permet à l'utilisateur de choisir l'aspect que prendra la barre de menus, parmi six possibilités. Il va de soi que l'on ne peut activer qu'une seule de ces possibilités à la fois. Pour mettre en place ce genre de fonctionnalité, on fait classiquement appel à des *widgets* de type « boutons radio ». La caractéristique essentielle de ces *widgets* est que plusieurs d'entre eux doivent être associés à une seule et même *variable Tkinter*. À chaque bouton radio correspond alors une valeur particulière, et c'est cette valeur qui est affectée à la variable lorsque l'utilisateur sélectionne le bouton.

Ainsi, l'instruction :

```
self.mo.add_radiobutton(label='sillon', variable=self.relief,
                        value=3, command=self.reliefBarre)
```

configure un item du menu «Options» de telle manière qu'il se comporte comme un bouton radio.

Lorsque l'utilisateur sélectionne cet item, la valeur 3 est affectée à la *variable Tkinter* `self.relief` (celle-ci étant désignée à l'aide de l'option `variable` du *widget*), et un appel est lancé en direction de la méthode `reliefBarre()`. Celle-ci récupère alors la valeur mémorisée dans la *variable Tkinter* pour effectuer son travail.

Dans le contexte particulier de ce menu, nous souhaitons proposer 6 possibilités différentes à l'utilisateur. Il nous faut donc six « boutons radio », pour lesquels nous pourrions encoder six instructions similaires à celle que nous avons reproduite ci-dessus, chacune d'elles ne différant des cinq autres que par ses options `value` et `label`. Dans une situation de ce genre, la bonne pratique de programmation consiste à placer les valeurs de ces options dans une liste, et à parcourir ensuite cette liste à l'aide d'une boucle `for`, afin d'instancier les *widgets* avec une instruction commune :

```
for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
```

```
(3,'sillon'), (4,'crête'), (5,'bordure')]:
self.mo.add_radiobutton(label =lab, variable =self.relief,
                        value =v, command =self.reliefBarre)
```

La liste utilisée est une liste de six tuples (valeur, libellé). A chacune des 6 itérations de la boucle, un nouvel item `radiobutton` est instancié, dont les options `label` et `value` sont extraites de la liste par l'intermédiaire des variables `lab` et `v`.

Dans vos projets personnels, il vous arrivera fréquemment de constater que vous pouvez ainsi remplacer des suites d'instructions similaires, par une structure de programmation plus compacte (en général, la combinaison d'une liste et d'une boucle, comme dans l'exemple ci-dessus).

Vous découvrirez petit à petit encore d'autres techniques pour alléger votre code : nous en fournissons encore un exemple dans le paragraphe suivant. Tâchez cependant de garder à l'esprit cette règle essentielle, qu'un bon programme doit avant tout rester *lisible* et *commenté*.

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

Veillez à présent considérer la définition de la méthode `reliefBarre()` :

À la première ligne, la méthode `get()` nous permet de récupérer l'état d'une *variable Tkinter* qui contient le numéro du choix opéré par l'utilisateur dans le sous-menu « Relief : ».

À la seconde ligne, nous utilisons le contenu de la variable `choix` pour extraire d'une liste de six éléments celui qui nous intéresse. Par exemple, si `choix` contient la valeur 2, c'est l'option `SUNKEN` qui sera utilisée pour reconfigurer le *widget*.

La variable `choix` est donc utilisée ici comme un index, servant à désigner un élément de la liste. En lieu et place de cette construction compacte, nous aurions pu programmer une série de tests conditionnels, comme par exemple :

```
if choix ==0:
    self.configure(relief =FLAT)
elif choix ==1:
    self.configure(relief =RAISED)
elif choix ==2:
    self.configure(relief =SUNKEN)
...
etc.
```

D'un point de vue strictement fonctionnel, le résultat serait exactement le même. Vous admettez cependant que la construction que nous avons choisie est d'autant plus efficace, que le nombre de possibilités de choix est élevé. Imaginez par exemple que l'un de vos programmes personnels doive effectuer une sélection dans un très grand nombre d'éléments : avec une construction du type ci-dessus, vous seriez peut-être amené à encoder plusieurs pages de « `elif` » !

Nous utilisons encore la même technique dans la méthode `choixActifs()`. Ainsi l'instruction :

```
self.pein.configure(state =[DISABLED, NORMAL][p])
```

utilise le contenu de la variable `p` comme index pour désigner lequel des deux états `DISABLED`, `NORMAL` doit être sélectionné pour reconfigurer le menu « Peintres ».

Lorsqu'elle est appelée, la méthode `choixActifs()` reconfigure donc les deux rubriques « Peintres » et « Musiciens » de la barre de menus, pour les faire apparaître « normales » ou « désactivées » en fonction de l'état des variables `m` et `p`, lesquelles sont elles-mêmes le reflet de variables *Tkinter*.

Ces variables intermédiaires `m` et `p` ne servent en fait qu'à clarifier le script. Il serait en effet parfaitement possible de les éliminer, et de rendre le script encore plus compact, en utilisant la composition d'instructions. On pourrait par exemple remplacer les deux instructions :

```
m = self.actMusi.get()
self.musi.configure(state =[DISABLED, NORMAL][m])
```

par une seule, telle que :

```
self.musi.configure(state =[DISABLED, NORMAL][self.actMusi.get()])
```

Notez cependant que ce que l'on gagne en compacité se paie d'une certaine perte de lisibilité.

Pré-sélection d'une rubrique

Pour terminer cet exercice, voyons encore comment vous pouvez déterminer à l'avance certaines sélections, ou bien les modifier par programme.

Veillez donc ajouter l'instruction suivante dans le constructeur de la classe `Application()` (juste avant l'instruction `self.pack()`, par exemple) :

```
mBar.mo.invoke(2)
```

Lorsque vous exécutez le script ainsi modifié, vous constatez qu'au départ la rubrique « Musiciens » de la barre de menus est active, alors que la rubrique « Peintres » ne l'est pas. Programmées comme elles le sont, ces deux rubriques devraient être actives toutes deux par défaut. Et c'est effectivement ce qui se passe si nous supprimons l'instruction :

```
mBar.mo.invoke(2)
```

Nous vous avons suggéré d'ajouter cette instruction au script, pour vous montrer comment vous pouvez effectuer *par programme* la même opération que celle que l'on obtient normalement avec un clic de souris.

L'instruction ci-dessus *invoque* le *widget* `mBar.mo` en actionnant la commande associée au *deuxième* item de ce *widget*. En consultant le listing, vous pouvez vérifier que ce deuxième item est bien l'objet de type `checkboxbutton` qui active/désactive le menu « Peintres » (Rappelons encore une fois que l'on numérote toujours à partir de zéro).

Au démarrage du programme, tout se passe donc comme si l'utilisateur effectuait tout de suite un premier clic sur la rubrique « Peintres » du menu « Options », ce qui a pour effet de désactiver le menu correspondant. Mais si vous voulez le réactiver, vous devrez éteindre votre ordinateur et le tour sera joué !

Notes

1. Voir page : Recherche des erreurs et expérimentation
2. Voir page : Modules contenant des bibliothèques de classes
3. Pour *écrire* dans une variable `Tkinter`, il faudrait utiliser la méthode `set()`.

Exemple :

```
self.actMusi.set(45)
```


Turtle

Turtle est un module graphique du langage de programmation Python. Il est inspiré de la programmation Logo et permet de déplacer une tortue sur l'écran.

Fonctions disponibles

Fonctions principales

Pour utiliser une fonction il faut inscrire son nom et lui donner une valeur dans les parenthèses.

Les principales fonctions mises à votre disposition dans le module *turtle* sont les suivantes :

- `reset()`

-> **Efface l'écran, recentre la tortue et remet les variables à zéro**

- `forward(distance)`

-> **Avance d'une distance donnée**

- Type de donnée : pixel

- `backward(distance)`

-> **Reculer d'une distance donnée**

- Type de donnée : pixel

- `left(angle)`

-> **Pivote vers la gauche**

- Type de donnée : angle

- `right(angle)`

-> **Pivote vers la droite**

- Type de donnée : angle

▪ `up()`

-> **Relève le crayon (pour pouvoir avancer sans dessiner)**

▪ `down()`

-> **Abaisse le crayon (pour recommencer à dessiner)**

▪ `goto(x, y)`

-> **Va à l'endroit de coordonnées (x, y)**

- Type de donnée : pixel

/!\ Ne pas oublier d'utiliser la fonction `up()` avant d'utiliser `goto()` car sinon il tracera le parcours effectué

Autres fonctions

▪ `color(couleur)`

-> **Détermine la couleur du tracé (noir par défaut)**

- Type de donnée : chaîne de caractère

- Couleurs proposées :

> "blue" => Bleu

> "red" => Rouge

> "green" => Vert

> "yellow" => Jaune

> "brown" => Marron

> "black" => Noir

> "white" => Blanc

> "pink" => Rose

> "orange" => Orange

> "purple" => Violet

> "grey" => Gris

▪ width(épaisseur)

-> **Choisit l'épaisseur du tracé**

- Type de donnée : pixel

▪ speed(vitesse)

-> **Choisit la vitesse à laquelle se déplace le curseur**

- Type de donnée : chaîne de caractère

- Vitesses proposées :

> "slowest" => Le plus lent

> "slow" => Lent

> "normal" => Normal

> "fast" => Rapide

> "fastest" => Le plus rapide

▪ circle(rayon, angle)

-> **Trace un cercle de rayon donné. L'argument facultatif *angle* indique**

l'angle de l'arc de cercle (par défaut 360, soit un cercle complet)

- Type de donnée :

° rayon : pixel

° angle : angle

- `title(titre)`

-> **Donne un titre à la fenêtre (par défaut le titre est Turtle Graphics)**

- Type de donnée : chaîne de caractère

- `write(texte)`

-> **Écrit du texte**

- Type de donnée : chaîne de caractère

- `fill(1)`

-> **Remplit un contour fermé à l'aide de la couleur sélectionnée**

Détails des fonctions

color()

La couleur peut être :

- une couleur prédéfinie précisé entre apostrophes droites (exemples : 'red', 'yellow', 'green', 'blue', 'brown', 'violet', 'purple', etc.)
- une couleur RVB avec trois variables r, v, et b comprises entre 0 et 1 (exemple : 1,0,0 pour le rouge)

Fill

Le remplissage peut se faire de deux manières. Par exemple pour remplir un carré :

```
begin_fill()|fill(1)
forward(100)|forward(100)
left(90)    |left(90)
```

```
forward(100);forward(100);
left(90) ;left(90)
forward(100);forward(100);
left(90) ;left(90)
forward(100);forward(100);
end_fill();fill(0)
```

Exemples

- Python 3 par l'exemple/Turtle

Compléments

- Le module **turtle** s'appuie sur le module **tkinter** (**Tkinter** pour les versions de python antérieures à 3), ça permet d'en utiliser les fonctionnalités. Par exemple pour enregistrer sa production au format postscript :

```
import turtle
#.....
#.....
turtle.getcanvas().postscript(file="monoeuvre.eps")
```

- Le module **turtle** permet de manipuler plusieurs tortues.

```
fred = turtle.Turtle()
martha = turtle.Turtle()

fred.forward(100)
martha.left(45)
```

Si les méthodes associées aux tortues sont utilisées sans référence à une tortue particulière, elles s'appliquent à une tortue "anonyme" qui est automatiquement créée si nécessaire.

Si aucune fenêtre d'affichage n'existe et qu'on agit sur une tortue, une fenêtre est automatiquement créée et affichée.

Ces automatismes peuvent paraître pratiques mais peuvent aussi être source de confusion. Il vaut mieux les expliciter (subjectif).

```
scene = turtle.Screen()
scene.setup(width=0.5, height=1.0, startx=-1, starty=0)
```

```
# width vraie dimension si entier, proportion de l'écran si décimal
# startx entier, position par rapport à la gauche de l'écran si positif, par rapport à la
```

Exemples

Tracer une spirale quelconque

```
from turtle import *
angle = 5
distance = 10
compteur = 0

while compteur <= 30:
    forward(distance)
    left(angle)
    compteur += 1
    angle += 2
```

Mais aussi de manière plus simple:

```
from turtle import *
speed("fastest")           #Pour l'aider a aller plus vite
rayon = 1                   #Le premier rayon par défaut
rayonSpiral = 100
while(rayon < rayonSpiral):
    circle(rayon, 180)
    rayon += 2               #écartement entre 2 demi-cercle de la spirale
```

Tracer une lanterne

```
from turtle import *
import math

def lanterne(l = 100):
    left(90)
    forward(l)
    right(90)
    forward(l)
    right(90)
    forward(l)
    right(90)
    forward(l)
    right(135)
    forward(l * math.sqrt(2))
    left(90)
    fillcolor("red")
    fill(True)
    forward(l * math.sqrt(2) / 2)
    left(90)
    forward(l * math.sqrt(2) / 2)
    fill(False)
```

```
    left(90)
    forward(l * math.sqrt(2))
    right(45)
lanterne(l = 100)

ht()
exitonclick()
```

Un peu de détente avec le module turtle

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

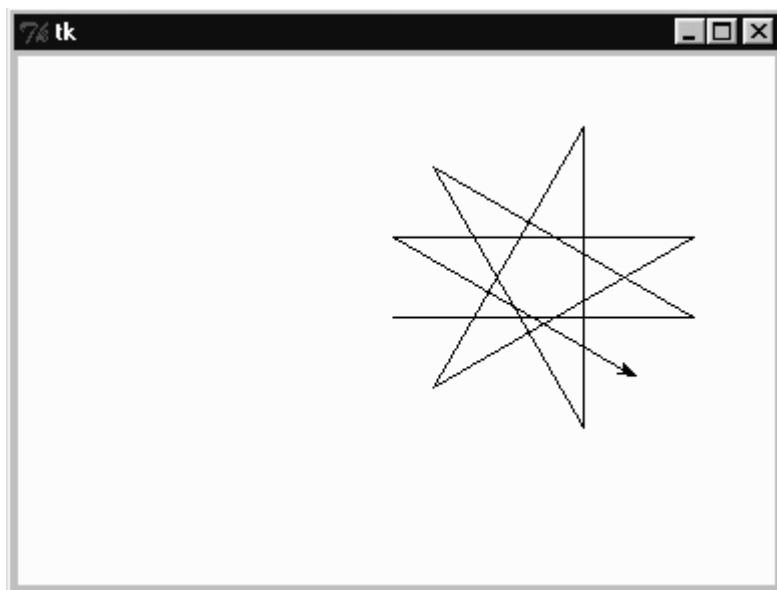
Pour illustrer cela, et nous amuser un peu avec d'autres objets que des nombres, nous allons explorer un module Python qui permet de réaliser des « graphiques tortue », c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont nous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

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

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

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

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



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

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

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

Les threads

Les threads Python sont utilisés pour lancer de multiples processus légers (tâches, appels de fonction) en même temps (mais cela ne veut pas forcément dire sur plusieurs processeurs). Les threads Python n'accélèreront donc pas un programme qui utilise déjà 100 % du temps d'un CPU (voir la programmation parallèle^[1] pour cela).

Les threads Python sont utilisés si une tâche fait attendre les autres. C'est le cas par exemple, des interactions avec un serveur web, ou lors de l'utilisation de la fonction `sleep`.

Exemples

Appel de fonction

Pour afficher les nombres de 1 à 10, en attendant une seconde entre chaque :

```
#!/usr/bin/env python
# coding: utf-8
import threading
import time

def loop1_10():
    for i in range(1, 11):
        time.sleep(1)
        print(i)

threading.Thread(target=loop1_10).start()
```

Objet

```
#!/usr/bin/env python
# coding: utf-8
import threading
import time
from __future__ import print_function

class MyThread(threading.Thread):
    def run(self):
        print("{} started!".format(self.getName())) # affiche "Thread-x started"
        time.sleep(1) # simule un travail d'une seconde
        print("{} finished!".format(self.getName())) # affiche "Thread-x finished"

if __name__ == '__main__':
    for x in range(4): # Quatre fois...
        mythread = MyThread(name = "Thread-{}".format(x + 1)) # ...Instancie un thread et
        mythread.start() # ...Démarre le thread
        time.sleep(.9) # ...Attend 0,9 secondes avant de continuer
```

Résultat :

```
Thread-1 started!  
Thread-2 started!  
Thread-1 finished!  
Thread-3 started!  
Thread-2 finished!  
Thread-4 started!  
Thread-3 finished!  
Thread-4 finished!
```

Note : cet exemple diminue l'IDLE (temps d'inactivité) dans Windows XP, alors que cela fonctionne dans WING IDE, Netbeans, et Eclipse. Cela peut poser problème en remplaçant `sleep(1)` par `(2)`, et `range(4)` par `range(10)`. Le *Thread-2* finit alors en première ligne (avant d'avoir commencé).

Références

1. <http://wiki.python.org/moin/ParallelProcessing>

XML

Il existe dans Python une bibliothèque permettant de manipuler les fichiers XML. Elle implémente la manière SAX (Simple API for XML) et DOM (Document Object Model).

Voyons comment manipuler simplement les fichier XML grâce à la méthode SAX.

La méthode SAX

Cette méthode est basée sur les évènements : une fonction est appelée lorsque l'on ouvre une balise, une autre lorsque le programme rencontre du texte, et une autre encore lorsque la balise se ferme.

Ces évènements sont définis dans une classe nommée **interface**. Cette classe doit dériver de `ContentHandler`, contenu dans le module `xml.sax`, et peut implémenter les fonctions suivantes :

- `startElement()` est la fonction appelée lors de l'ouverture d' une balise. Les deux arguments sont le nom et un dictionnaire contenant les attributs.
- `endElement()` est la fonction appelée lors de la fermeture d' une balise. La fonction prend le nom de la balise en argument.
- `characters()` est appelée lors que le parseur rencontre un caractère en dehors d' une balise. Ce caractère est passé en paramètre.

Une fois cette classe faite, il faut créer un parseur. Cela est fait grâce à la fonction **make_parser()**, située elle aussi dans le module `xml.sax`. On lui associe ensuite une instance de l'interface avec la méthode **setContentHandler()**, et on lance l'analyse du fichier grâce à la méthode **parse**, qui prend le nom du fichier en argument.

Voici donc le code final :

```
from xml.sax.handler import ContentHandler
import xml.sax
import sys

class compteurElements(ContentHandler):
    def __init__(self):
        self.elem={}
    def startElement(self,name,attr):
        if not self.elem.has_key(name):
            self.elem[name] = 0
        self.elem[name] += 1
    def characters(self,ch):
        print ch
    def endElement(self, name):
        print name + ":" + str(self.elem[name])
```

```
parser = xml.sax.make_parser()
handler = compteurElements()
parser.setContentHandler(handler)
parser.parse(sys.argv[1])
```

Exemple 1 : Un compteur d'éléments bavard

La méthode DOM

Le module se nomme `xml.dom`^[1] :

```
import xml.dom
```



Cette section est vide, pas assez détaillée ou incomplète.

Références

1. <https://docs.python.org/2/library/xml.dom.html>

Tests

Tests unitaires

La bibliothèque `unittest`^[1] permet d'automatiser les tests unitaires.

Syntaxe

La méthode `assertEqual()` permet de vérifier la valeur d'un objet :

```
import unittest
...
self.assertEqual(...)
```

Exemples

Le framework Pywikipedia propose toute une série de tests basés sur `unittest`.

1. Le télécharger sur <http://tools.wmflabs.org/pywikibot/>
2. Conformément au manuel [tests/README.rst](#), lancer `python pwb.py tests/api_tests.py -v`.



Cette section est vide, pas assez détaillée ou incomplète.

Références

1. <http://docs.python.org/library/unittest.html>

Exemples de scripts

Exemples de programmes écrits dans le langage de programmation Python dans sa version 3.3

Exemples de codes représentatifs

Une introduction

```

ageDeEricPraline      = 50
ageDeHenryWensleydale = 20

# Test
if 0 > ageDeEricPraline > 150 or ageDeEricPraline > ageDeHenryWensleydale:
    print("l'age de Eric Praline est incorrect")

#echange : swap
ageDeHenryWensleydale, ageDeEricPraline = ageDeEricPraline,ageDeHenryWensleydale
print("age de Henry Wensleydale = %d , age de Eric Praline = %d" %(ageDeHenryWensleydale, a

#>>>
#l'age de Eric Praline est incorrect
#age de Henry Wensleydale = 50 , age de Eric Praline = 20

```

Les structures de données en natif

```

# Les listes :
maListeDeMeubles = ['table','chaise','frigo']
maListeDeMeubles.sort() #Tri de la liste
for unMeuble in maListeDeMeubles:
    print('longueur de la chaîne ', unMeuble, '=', len(unMeuble))

#les listes imbriquées:
laListeDesNombresPairs = [ unNombre for unNombre in xrange(1000) if unNombre % 2 == 0]

#Les dictionnaires :
unAnnuaire = {'Laurent': 6389565, 'Paul': 6356785}
for unNom, x in unAnnuaire.items():
    print("le nom %s a pour numéro de téléphone %d" %(unNom, x))

#Les tuples (n-uplet) : séquence constante
Couleur = ('Rouge', 'Bleu', 'Vert')
print(Couleur[0], Couleur[1], Couleur[2])

PointDeReference = (100, 200)
print(" x0 = %d y0 = %d " %(PointDeReference[0],PointDeReference[1]))

```

Accès a une base de données

```

# Avec MySQL :

```

```
import MySQLdb, pprint
uneConnexionBDD = MySQLdb.connect(host = '192.32.12.10',
                                   user  = 'admin',
                                   passwd = '988',
                                   db    = 'uneBase')
leCurseur      = uneConnexionBDD.cursor()
unAuteur        = "Zola"
leCurseur.execute(""" SELECT title, description FROM books WHERE author = %s """ % (unAuteur))
pprint.pprint(leCurseur.fetchall())
leCurseur.query("update books set title='assommoir' where author='Zola'")
uneConnexionBDD.commit()
```

```
# Par ODBC :
import dbi, odbc
connexion = odbc.odbc('mondsn/monlogin/monpassword')
leCurseur = connexion.cursor()
leCurseur.execute('select clientid, name, city from client')
print(leCurseur.fetchall())
```

Programmation réseau - Internet

```
#Lire une page Web et afficher le source HTML
import urllib
print(urllib.urlopen)

#Usage de FTP : démonstration des paramètres nommés
import ftplib
uneSessionFTP = ftplib.FTP( host = '198.12.45.123',
                             user = 'MonLogin',
                             passwd = 'MonMotDePasse')

unFichier = open('monFichier.txt','rb')
uneSessionFTP.storbinary( 'STOR monFichier.txt', unFichier ) #envoi du fichier
unFichier.close()
uneSessionFTP.quit()
```

Tracé de courbes avec matplotlib

```
from pylab import plot, axis, savefig, show, title
plot( [1,2,3,4],
      [1,4,9,16] )
axis( [1, 4, 0, 16] )
title( 'ma courbe' )
savefig( 'uneCourbeSauvee.png' )
show()
```

Utilitaires de la bibliothèque standard

```
#Encodage d'une image binaire sous forme de chaîne (Pour les protocoles qui n'acceptent qu
lesDonneesImage = open('uneImage.gif', 'rb').read()
laChaineZippee = lesDonneesImage.encode('zlib').encode('base64')
```

Jython : utilisation de Java en Python

```
#Exemple d'une applet scripté
from javax.swing import *
class MyApplet( JApplet ):
    def init( self ):
        unLabel= JLabel("Jython améliore la productivité")
        self.getContentPane().add( unLabel )
```

Mail

```
#Envoi par mail du contenu d'un fichier
import smtplib
from email.MIMEText import MIMEText

IPduServeur = 'localhost'
monFichierMail = open ('monMailAEnvoyer.txt', 'rb') # lecture du fichier
leMessage = MIMEText ( monFichierMail.read() ) # création d'un message text/plain
monFichierMail.close()

leMessage['Subject'] = "Confirmation de l'exigence A441"
leMessage['From'] = "expediteur@phales.com"
leMessage['To'] = "destinataire@ecn.com"

leServeurSMTP = smtplib.SMTP(IPduServeur) # envoi du message
leServeurSMTP.connect()
leServeurSMTP.sendmail('expediteur@phales.com',
                        'destinataire@ecn.com',
                        leMessage.as_string())
leServeurSMTP.quit()
```

Classe

Exemple de classe (voir programmation orientée objet).

```
class Fruit :
    def __init__(self) :
        pass

class Pomme(Fruit):
    """
    Cette classe représente une pomme.
    """
    Mangeurs = ["Jacques", "Nicolas", "Virgile"]
```



```

def __init__(self, couleur):
    """
    Pour construire une Pomme, donnez sa couleur.
    """
    Fruit.__init__(self)
    self._couleur = couleur

def couleur(self):
    """
    Retourne la couleur de la Pomme.
    """
    return self._couleur

def comestible(self, mangeur):
    """
    Dit si la pomme est comestible ou non,
    en fonction du mangeur.
    """
    if mangeur in self.Mangeurs:
        print(mangeur, "mange des pommes")
    else:
        print (mangeur, "n'aime pas les pommes")

petitePomme = Pomme("verte")
petitePomme.comestible("Pierre")      # Pierre n'aime pas les pommes
petitePomme.comestible("Nicolas")     # Nicolas mange des pommes
    
```

On remarque notamment la présence de documentation (optionnelle bien sûr) directement dans le code. La commande `help()` permet d'obtenir, dans l'interpréteur Python, cette aide directement :

```
>>> help(Pomme)
```

donne :

```

Help on class Pomme in module __main__:
class Pomme(Fruit)
| Cette classe représente une pomme.
|
| Methods defined here:
|
| __init__(self, couleur)
|     Pour construire une Pomme, donnez sa couleur.
|
| comestible(self, mangeur)
|     Dit si la pomme est comestible ou non,
|     en fonction du mangeur.
|
| couleur(self)
|     Retourne la couleur de la Pomme.
|
|-----
| Data and other attributes defined here:
|
| Mangeurs = ['Jacques', 'Nicolas', 'Virgile']
    
```

WMI (sous Windows 2000/XP)

Lister les processus et leur nom

```
import wmi
contrôleur = wmi.WMI ()
for unProcessus in contrôleur.Win32_Process ():
    print (unProcessus.ProcessId, unProcessus.Name)
```

Rebooter une machine à distance

```
import wmi
contrôleur = wmi.WMI (computer = "Nom_Machine_Distante",
                    privileges = ["RemoteShutdown"] )
os = contrôleur.Win32_OperatingSystem (Primary=1)[0]
os.Reboot ()
```

Automation Win32

```
#Exemple de pilotage de CATIA par la technologie COM/DCOM
import win32com.client

catiaApplication = win32com.client.Dispatch("CATIA.Application")
monDocument = catiaApplication.ActiveDocument.Product
nombreDeProduit = monDocument.Products.Count

for produits in range(nombreDeProduit):
    produits += 1
    print (monDocument.Products.Item(produits).Name)

#Exemple de pilotage du lecteur multimédia
from win32com.client import Dispatch
leMediaPlayer = Dispatch("WMPlayer.OCX")
uneMorceauDeMusique = leMediaPlayer.newMedia("C:/mesDoc/monTitre.wma") # ou mp3
leMediaPlayer.currentPlaylist.appendItem(uneMorceauDeMusique)
leMediaPlayer.controls.play()
raw_input("appuyez sur la touche ENTRÉE pour arrêter la musique")
leMediaPlayer.leMediaPlayer.stop()
```

ctypes

ctypes permet d'accéder directement à des dll.

```
import ctypes
# modification du fond d'écran de windows
SPI_SETDESKWALLPAPER = 20
ctypes.windll.user32.SystemParametersInfoA(SPI_SETDESKWALLPAPER,
                                            0,
                                            "C:\\Mes documents\\Mes images\\maPhoto.bmp" ,
                                            0)

# Resolution de l'écran
SM_CXSCREEN = 0
SM_CYSCREEN = 1
```

```
taille_x = ctypes.windll.user32.GetSystemMetrics(SM_CXSCREEN)
taille_y = ctypes.windll.user32.GetSystemMetrics(SM_CYSCREEN)
print ("La résolution d'écran est %d par %d" % (taille_x, taille_y))
```

Data Mining - Réseaux bayésiens avec reverend

```
# Les filtres Bayesiens permettent de classifier des données.
# Exemple d'application : reconnaissance automatique de la langue, anti-Spam, cryptographie

from reverend.thomas import Bayes

unReseauBayesiens = Bayes()
uneListeDePropositions = [('math', 'on fait des calculs , des additions, des multi
    ('math', '3 + 4 = 7 ; 10 + 7 = 17 ; 99/11 = 9'),
    ('math', '12 + 10 -5 = 17 ; 70-10 = 60'),
    ('math', 'le carré des 2 cotés du triangle sont liés par
    ('math', 'la fonction sinus ( sin ) sert à représenter l
    ('math', 'sin (PI) = 3.14159 et cosinus (0) = 1' ),
    ('philosophie', 'je disserte sur le sens de la vie'),
    ('philosophie', 'être ou ne pas être, telle est la question'),
    ('philosophie', 'je sais que je ne sais rien'),
    ('philosophie', 'les phénomènes sont réels à condition que nous
    ('philosophie', 'la raison est-elle toujours raisonnable ?'),
    ('philosophie', 'le cerveau peut-il être compris ?'),
    ('philosophie', "l'univers peut-il être l'objet de connaissance"),
    ('philosophie', 'le calcul a-t-il des limites intrinsèques ?'),
    ('philosophie', "une relation peut être durable si l'homme la s

for uneCategorie, uneProposition in uneListeDePropositions:
    unReseauBayesiens.train( uneCategorie, uneProposition ) # entrainement du réseau

phraseAnalyse1 = 'voici un résultat : 66/6 = 11 '
phraseAnalyse2 = "je ne saurais dire s'il pourra tout comprendre ... "
phraseAnalyse3 = "le phénomène de la pluie pourrait être d'origine divine"
phraseAnalyse4 = 'la représentation bourbakiste des chiffres assure leur détermination'

for unePhrase in (phraseAnalyse1, phraseAnalyse2, phraseAnalyse3, phraseAnalyse4) :
    solutions = unReseauBayesiens.guess(unePhrase) # calculs de la catégorie
    categorie = solutions[0][0]
    probabilite = solutions[0][1]
    print ("la phrase '%s' est de catégorie '%s' avec une probabilité de '%d /100' " %(une

# Résultats :
# la phrase 'voici un résultat : 66/6 = 11 ' est de catégorie 'math' avec une probabilité
# la phrase 'je ne saurais dire s'il pourra tout comprendre ... ' est de catégorie 'philos
# la phrase 'le phénomène de la pluie pourrait être d'origine divine' est de catégorie 'ph
# la phrase 'la représentation bourbakiste des chiffres assure leur détermination' est de
```

Implémentation du crible d'Ératosthène

Ce script calcule les nombres premiers inférieurs à 200 en appliquant la méthode du crible d'Ératosthène.

```
# calcul des nombres premiers inférieurs à N
# initialisation
```

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
N = 200
liste = range(2, N) # liste de 2 à N
nombre = 2
while nombre*nombre <= N: # tant que le nb premier < à la
    # racine carrée de N
    for i in liste[liste.index(nombre) + 1:]: # parcourt la liste à partir de ce nombre
        if i % nombre == 0: # un multiple du nombre est trouvé
            liste.remove(i) # ou "del liste[ liste.index(i) ]" :
            # on le raye de la liste
        nombre = liste[liste.index(nombre) + 1] # on prend le nombre suivant non rayé

print (liste) # affichage du résultat
```

Voici une version algorithmiquement équivalente mais beaucoup plus rapide qui exploite la puissance du type liste de Python :

```
N = 1000 # on cherche les nombres premiers inférieurs à N
liste = range(N+1) # création de la liste des entiers de 0 à N
liste[1] = 0 # 1 n'est pas premier
nombre = 2

while nombre ** 2 <= N: # tant que le nombre à examiner est inférieur à
    # la racine carrée de N
    liste[nombre*2 :: nombre] = [0] * ((N // nombre) - 1) # éliminer tous les
    # multiples du nombre
    # passer au nombre non examine suivant
    nombre += 1
    while not liste[nombre]:
        nombre += 1

liste = filter (None, liste)
print (liste) # et à la fin, on affiche le résultat
```

Que l'on peut enfin réduire à :

```
N, nombre = 1000, 2
liste, liste[1] = range(N+1), 0

while nombre**2 <= N:
    liste[nombre*2 :: nombre] = [0]*len( liste[nombre*2 :: nombre] )
    nombre += 1
print (filter(None, liste))
```

Les fonctions, les types et les classes sont des objets

```
#Les fonctions sont des objets
def uneFonction ( unEntier, uneChaine):
    "documentation de la fonction"
    print ("unEntier : " + str( unEntier ))

#Vous pouvez d'ailleurs ajouter des attributs à la fonction:
```

```

uneFonction.compteur = 10

print ("la fonction a pour nom : " + uneFonction.__name__)
print ("la fonction a pour documentation : " + uneFonction.__doc__)
print (uneFonction.compteur)

>>la fonction a pour nom :uneFonction
la fonction a pour documentation : documentation de la fonction
10

#Les types de base sont des objets
valeurOpposee = -25 .__neg__() # equivalent à - (-25)
print ("la valeur opposée de -25 est : " + str ( valeurOpposee ))

#Les classes sont des objets
class Insecte : pass # On definit une classe mère, qui ne fait rien pour l'inst
class Fourmi(Insecte): # Definition d'une classe dérivée
    def marcher (self):
        print ("je marche")

print ("la classe Fourmi a pour classe mère : " + str ( Fourmi.__bases__))

#l'opposé de -25 est : 25
#la classe Fourmi a pour classe mère :(<class __main__.Insecte at 0x02B69A20>,)

```

Graphique

Tkinter

Cet exemple montre comment se servir de la bibliothèque Tkinter pour créer une interface graphique.

```

from Tkinter import *
fen = Tk()
text1 = Label(fen, text='Bonjour !')
entr1 = Entry(fen)
bout1 = Button(fen, text='Quitter', command=fen.quit)
text1.grid( row=1, column=1)
entr1.grid( row=1, column=2)
bout1.grid( row=2, column=1)
fen.mainloop()

```

WxPython

Cet exemple crée en WxPython une fenetre et un boite modale sur appui bouton - l'utilisation d'editeur WYSIWYG comme glade ou boa constructor est recommandée pour les grandes applications

```

from wxPython.wx import *
ID_OK_BOUTON=100

class PanneauPerso( wxPanel ):
    def __init__(self, parent, id):

```

```

wxPanel.__init__(self, parent, id)
self.unChampText = wxStaticText(self, -1, "TextAvantLeClic", wxPoint(5, 5))
self.unChampBouton = wxButton(self, ID_OK_BOUTON, "CliquerMoi", wxPoint(100, 5))
EVT_BUTTON(self, ID_OK_BOUTON, self.OnClick) #enregistrement de la callBack

def OnClick(self,event):
    self.unChampText.SetLabel('ApresLeClic') #sur appui bouton, modifier le champ
    b = wxMessageDialog( self, "Une Boite de dialogue","titre", wxOK)
    b.ShowModal() #montrer un e boite de dialogue
    b.Destroy()

monApplication = wxPySimpleApp()
maFenetre = wxFrame(None, -1, "Titre de l Application")
PanneauPerso ( maFenetre , -1)
maFenetre.Show(1)
monApplication .MainLoop()

```

Analyse de programmes concrets

Dans ce chapitre, nous allons nous efforcer d'illustrer la démarche de conception d'un programme graphique, depuis ses premières ébauches jusqu'à un stade de développement relativement avancé. Nous souhaitons montrer ainsi combien la programmation orientée objet peut faciliter et surtout sécuriser la stratégie de développement incrémental que nous préconisons.

L'utilisation de classes s'impose, lorsque l'on constate qu'un projet en cours de réalisation se révèle nettement plus complexe que ce que l'on avait imaginé au départ. Vous vivrez certainement vous-même des cheminements similaires à celui que nous décrivons ci-dessous.

Jeu des bombardes

Ce projet de jeu^[1] s'inspire d'un travail similaire réalisé par des élèves de terminale.

Il est vivement recommandé de commencer l'ébauche d'un tel projet par une série de petits dessins et de schémas, dans lesquels seront décrits les différents éléments graphiques à construire, ainsi qu'un maximum de *cas d'utilisations*. Si vous rechignez à utiliser pour cela la bonne vieille technologie papier/crayon (laquelle a pourtant bien fait ses preuves), vous pouvez tirer profit d'un logiciel de dessin technique, tel l'utilitaire *Draw* de la suite bureautique *OpenOffice.org*^[2].

L'idée de départ est simple : deux joueurs s'affrontent au canon. Chacun doit ajuster son angle de tir pour tâcher d'atteindre son adversaire, les obus décrivant une trajectoire balistique.

L'emplacement des canons est défini au début du jeu de manière aléatoire (tout au moins en hauteur). Après chaque tir, les canons se déplacent (afin d'accroître l'intérêt du jeu, l'ajustement des tirs étant ainsi rendu plus difficile). Les coups au but sont comptabilisés.

Le dessin préliminaire que nous avons reproduit à la page précédente est l'une des formes que peut prendre votre *travail d'analyse*. Avant de commencer le développement d'un projet de programmation, il vous faut en effet toujours vous efforcer d'établir un *cahier des charges détaillé*. Cette étude préalable est très importante. La plupart des débutants commencent bien trop vite à écrire de nombreuses lignes de code au départ d'une vague idée, en négligeant de rechercher la structure d'ensemble. Leur programmation risque alors de devenir chaotique, parce qu'ils devront de toute façon mettre en place cette structure tôt ou tard. Il s'apercevront alors bien souvent qu'il leur faut supprimer et ré-écrire des pans entiers d'un projet qu'ils ont conçu d'une manière *trop monolithique* et/ou *mal paramétrée*.

- Trop monolithique : cela signifie que l'on a négligé de décomposer un

problème complexe en plusieurs sous-problèmes plus simples. Par exemple, on a imbriqué plusieurs niveaux successifs d'instructions composées, au lieu de faire appel à des fonctions ou à des classes.

- Mal paramétrée : cela signifie que l'on a traité seulement un cas particulier, au lieu d'envisager le cas général. Par exemple, on a donné à un objet graphique des dimensions fixes, au lieu de prévoir des variables pour permettre son redimensionnement.

Vous devez donc toujours commencer le développement d'un projet par une phase d'analyse aussi fouillée que possible, et concrétiser le résultat de cette analyse dans un ensemble de documents (schémas, plans, descriptions...) qui constitueront le cahier des charges. Pour les projets de grande envergure, il existe d'ailleurs des méthodes d'analyse très élaborées (*UML*, *Merise*...) que nous ne pouvons nous permettre de décrire ici car elles font l'objet de livres entiers.

Cela étant dit, il faut malheureusement admettre qu'il est très difficile (et même probablement impossible) de réaliser dès le départ l'analyse tout à fait complète d'un projet de programmation. C'est seulement lorsqu'il commence à fonctionner véritablement qu'un programme révèle ses faiblesses. On constate alors qu'il reste des cas d'utilisation ou des contraintes qui n'avaient pas été prévues au départ. D'autre part, un projet logiciel est pratiquement toujours destiné à évoluer : il vous arrivera fréquemment de devoir modifier le cahier des charges au cours du développement lui-même, pas nécessairement parce que l'analyse initiale a été mal faite, mais tout simplement parce que l'on souhaite encore ajouter des fonctionnalités supplémentaires.

En conclusion, tâchez de toujours aborder un nouveau projet de programmation en respectant les deux consignes suivantes :

- Décrivez votre projet en profondeur avant de commencer la rédaction des premières lignes de code, en vous efforçant de mettre en évidence les composants principaux et les relations qui les lient (pensez notamment à décrire les différents cas d'utilisation de votre programme).
- Lorsque vous commencerez sa réalisation effective, évitez de vous laisser entraîner à rédiger de trop grands blocs d'instructions. Veillez au contraire à découper votre application en un certain nombre de composants paramétrables bien encapsulés, de telle manière que vous puissiez aisément modifier l'un ou l'autre d'entre eux sans compromettre le fonctionnement des autres, et peut-être même les réutiliser dans différents contextes si le besoin s'en fait sentir.

C'est pour satisfaire cette exigence que la programmation orientée objets est a été inventée.

Considérons par exemple l'ébauche dessinée à la page précédente.

L'apprenti programmeur sera peut-être tenté de commencer la réalisation de ce jeu en n'utilisant que la programmation procédurale seule (c'est-à-dire en

omettant de définir de nouvelles classes). C'est d'ailleurs ainsi que nous avons procédé nous-même lors de notre première approche des interfaces graphiques, tout au long du chapitre 8. Cette façon de procéder ne se justifie cependant que pour de tout petits programmes (des exercices ou des tests préliminaires). Lorsque l'on s'attaque à un projet d'une certaine importance, la complexité des problèmes qui se présentent se révèle rapidement trop importante, et il devient alors indispensable de fragmenter et de compartimenter.

L'outil logiciel qui va permettre cette fragmentation est la *classe*.

Nous pouvons peut-être mieux comprendre son utilité en nous aidant d'une analogie :

Tous les appareils électroniques sont constitués d'un petit nombre de composants de base, à savoir des transistors, des diodes, des résistances, des condensateurs, etc. Les premiers ordinateurs ont été construits directement à partir de ces composants. Ils étaient volumineux, très chers, et pourtant ils n'avaient que très peu de fonctionnalités et tombaient fréquemment en panne.

On a alors développé différentes techniques pour encapsuler dans un même boîtier un certain nombre de composants électroniques de base. Pour utiliser ces nouveaux *circuits intégrés*, il n'était plus nécessaire de connaître leur contenu exact : seule importait leur fonction globale. Les premières fonctions intégrées étaient encore relativement simples : c'étaient par exemple des portes logiques, des bascules, etc. En combinant ces circuits entre eux, on obtenait des caractéristiques plus élaborées, telles que des registres ou des décodeurs, qui purent à leur tour être intégrés, et ainsi de suite, jusqu'aux microprocesseurs actuels. Ceux-ci contiennent dorénavant plusieurs millions de composants, et pourtant leur fiabilité reste extrêmement élevée.

En conséquence, pour l'électronicien moderne qui veut construire par exemple un compteur binaire (circuit qui nécessite un certain nombre de *bascules*), il est évidemment bien plus simple, plus rapide et plus sûr de se servir de bascules intégrées, plutôt que de s'échiner à combiner sans erreur plusieurs centaines de transistors et de résistances.

D'une manière analogue, le programmeur moderne que vous êtes peut bénéficier du travail accumulé par ses prédécesseurs en utilisant la fonctionnalité intégrée dans les nombreuses bibliothèques de classes déjà disponibles pour Python. Mieux encore, il peut aisément créer lui-même de nouvelles classes pour encapsuler les principaux composants de son application, particulièrement ceux qui y apparaissent en plusieurs exemplaires. Procéder ainsi est plus simple, plus rapide et plus sûr que de multiplier les blocs d'instructions similaires dans un corps de programme monolithique, de plus en plus volumineux et de moins en moins compréhensible.

Examinons par exemple notre ébauche dessinée. Les composants les plus importants de ce jeu sont bien évidemment les petits canons, qu'il faudra pouvoir dessiner à différents emplacements et dans différentes orientations, et

dont il nous faudra au moins deux exemplaires.

Plutôt que de les dessiner morceau par morceau dans le canevas au fur et à mesure du déroulement du jeu, nous avons intérêt à les considérer comme des objets logiciels à part entière, dotés de plusieurs propriétés ainsi que d'un certain comportement (ce que nous voulons exprimer par là est le fait qu'il devront être dotés de divers mécanismes, que nous pourrons activer par programme à l'aide de *méthodes* particulières). Il est donc certainement judicieux de leur consacrer une classe spécifique.

Prototypage d'une classe « Canon »

En définissant une telle classe, nous gagnons sur plusieurs tableaux. Non seulement nous rassemblons ainsi tout le code correspondant au dessin et au fonctionnement du canon dans une même « capsule », bien à l'écart du reste du programme, mais de surcroît nous nous donnons la possibilité d'instancier aisément un nombre quelconque de ces canons dans le jeu, ce qui nous ouvre des perspectives de développements ultérieurs.

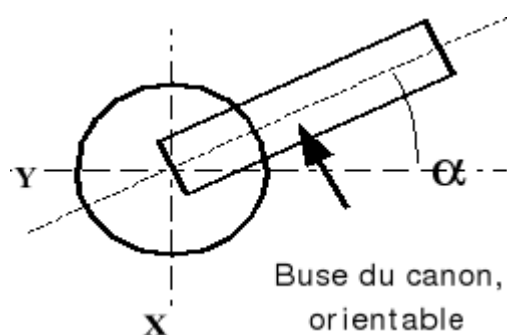
Lorsqu'une première implémentation de la classe `Canon()` aura été construite et testée, il sera également possible de la perfectionner en la dotant de caractéristiques supplémentaires, sans modifier (ou très peu) son interface, c'est-à-dire en quelque sorte son « mode d'emploi » : à savoir les instructions nécessaires pour l'instancier et l'utiliser dans des applications diverses.

Entrons à présent dans le vif du sujet.

Le dessin de notre canon peut être simplifié à l'extrême. Nous avons estimé qu'il pouvait se résumer à un cercle combiné avec un rectangle, celui-ci pouvant d'ailleurs être lui-même considéré comme un simple segment de ligne droite particulièrement épais.

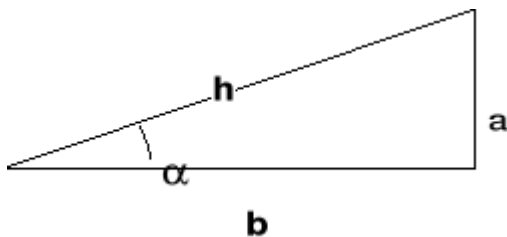
Si l'ensemble est rempli d'une couleur uniforme (en noir, par exemple), nous obtiendrons ainsi une sorte de petite bombarde suffisamment crédible.

Dans la suite du raisonnement, nous admettrons que la position du canon est en fait la position du centre du cercle (coordonnées x et y dans le dessin ci-contre). Ce point clé indique également l'axe de rotation de la buse du canon, ainsi que l'une des extrémités de la ligne épaisse qui représentera cette buse.



Pour terminer notre dessin, il nous restera alors à déterminer les coordonnées de l'autre extrémité de cette ligne. Ces coordonnées peuvent être calculées sans grande difficulté, à la condition de nous remémorer deux concepts fondamentaux de la trigonométrie (le sinus et le cosinus) que vous devez certainement bien connaître :

Dans un triangle rectangle, le rapport entre le côté opposé à un angle et l'hypoténuse du triangle est une propriété spécifique de cet angle qu'on appelle sinus de l'angle. Le cosinus du même angle est le rapport entre le côté adjacent à l'angle et l'hypoténuse.



Ainsi, dans le schéma ci-contre : $\sin\alpha = a/h$ et $\cos\alpha = b/h$.

Pour représenter la buse de notre canon, en supposant que nous connaissions sa longueur l et l'angle de tir α , il nous faut donc tracer un segment de ligne droite épaisse, à partir des coordonnées du centre du cercle (x et y), jusqu'à un autre point situé plus à droite et plus haut, l'écart horizontal Δx étant égal à $l \cdot \cos\alpha$, et l'écart vertical Δy étant égal à $l \cdot \sin\alpha$.

En résumant tout ce qui précède, dessiner un canon au point x, y consistera simplement à :

- tracer un cercle noir centré sur x, y ;
- tracer une ligne noire épaisse depuis le point x, y jusqu'au point $x + l \cdot \cos\alpha, y + l \cdot \sin\alpha$.

Nous pouvons à présent commencer à envisager une ébauche de programmation correspondant à une classe « Canon ». Il n'est pas encore question ici de programmer le jeu proprement dit. Nous voulons seulement vérifier si l'analyse que nous avons faite jusqu'à présent « tient la route », en réalisant un premier *prototype* fonctionnel.

Un prototype est un petit programme destiné à expérimenter une idée, que l'on se propose d'intégrer ensuite dans une application plus vaste. Du fait de sa simplicité et de sa concision, Python se prête fort bien à l'élaboration de prototypes, et de nombreux programmeurs l'utilisent pour mettre au point divers composants logiciels qu'ils reprogrammeront éventuellement ensuite dans d'autres langages plus « lourds », tels que le C par exemple.

Dans notre premier prototype, la classe `Canon()` ne comporte que deux méthodes : un constructeur qui crée les éléments de base du dessin, et une

méthode permettant de modifier celui-ci à volonté pour ajuster l'angle de tir (l'inclinaison de la buse). Comme nous l'avons souvent fait dans d'autres exemples, nous inclurons quelques lignes de code à la fin du script afin de pouvoir tester la classe tout de suite :

```

1 from Tkinter import *
2 from math import pi, sin, cos
3
4 class Canon:
5     """Petit canon graphique"""
6     def __init__(self, boss, x, y):
7         self.boss = boss           # référence du canevas
8         self.x1, self.y1 = x, y    # axe de rotation du canon
9         # dessiner la buse du canon, à l'horizontale pour commencer :
10        self.lbu = 50              # longueur de la buse
11        self.x2, self.y2 = x + self.lbu, y
12        self.buse = boss.create_line(self.x1, self.y1, self.x2, self.y2,
13                                     width =10)
14        # dessiner ensuite le corps du canon par-dessus :
15        r = 15                     # rayon du cercle
16        boss.create_oval(x-r, y-r, x+r, y+r, fill='blue', width =3)
17
18    def orienter(self, angle):
19        "choisir l'angle de tir du canon"
20        # rem : le paramètre <angle> est reçu en tant que chaîne de car.
21        # il faut le traduire en nombre réel, puis convertir en radians :
22        self.angle = float(angle)*2*pi/360
23        self.x2 = self.x1 + self.lbu*cos(self.angle)
24        self.y2 = self.y1 - self.lbu*sin(self.angle)
25        self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)
26
27 if __name__ == '__main__':
28     # Code pour tester sommairement la classe Canon :
29     f = Tk()
30     can = Canvas(f,width =250, height =250, bg ='ivory')
31     can.pack(padx =10, pady =10)
32     c1 = Canon(can, 50, 200)
33
34     s1 =Scale(f, label='hausse', from_=90, to=0, command=c1.orienter)
35     s1.pack(side=LEFT, pady =5, padx =20)
36     s1.set(25)                       # angle de tir initial
37
38     f.mainloop()

```

Commentaires

- Ligne 6 : Dans la liste des paramètres qui devront être transmis au constructeur lors de l'instanciation, nous prévoyons les coordonnées x et y, qui indiqueront l'emplacement du canon dans le canevas, mais également une référence au canevas lui-même (la variable boss). Cette référence est indispensable : elle sera utilisée pour invoquer les méthodes du canevas. Nous pourrions inclure aussi un paramètre pour choisir un angle de tir initial, mais puisque nous avons l'intention d'implémenter une méthode spécifique pour régler cette orientation, il sera plus judicieux de faire appel à celle-ci au moment voulu.
- Lignes 7 et 8 : Ces références seront utilisées un peu partout dans les différentes méthodes que nous allons développer dans la classe. Il faut donc

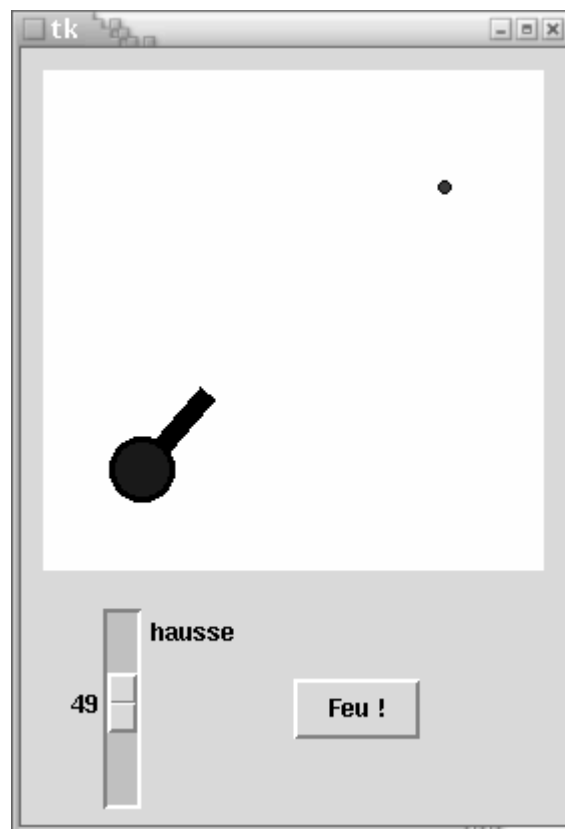
en faire des attributs d'instance.

- Lignes 9 à 16 : Nous dessinons la buse d'abord, et le corps du canon ensuite. Ainsi une partie de la buse reste cachée. Cela nous permet de colorer éventuellement le corps du canon.
- Lignes 18 à 25 : Cette méthode sera invoquée avec un argument « angle », lequel sera fourni en degrés (comptés à partir de l'horizontale). S'il est produit à l'aide d'un widget tel que *Entry* ou *Scale*, il sera transmis sous la forme d'une chaîne de caractères, et nous devons donc le convertir d'abord en nombre réel avant de l'utiliser dans nos calculs (ceux-ci ont été décrits à la page précédente).
- Lignes 27 à 38 : Pour tester notre nouvelle classe, nous ferons usage d'un widget *Scale*. Pour définir la position initiale de son curseur, et donc fixer l'angle de hausse initial du canon, nous devons faire appel à sa méthode `set()` (ligne 36).

Ajout de méthodes au prototype

Notre prototype est fonctionnel, mais beaucoup trop rudimentaire. Nous devons à présent le perfectionner pour lui ajouter la capacité de tirer des obus.

Ceux-ci seront traités plutôt comme des « boulets » : ce seront de simples petits cercles que nous ferons partir de la bouche du canon avec une vitesse initiale d'orientation identique à celle de sa buse. Pour leur faire suivre une trajectoire réaliste, nous devons à présent nous replonger dans notre cours de physique :



Comment un objet laissé à lui-même évolue-t-il dans l'espace, si l'on néglige les

phénomènes secondaires tels que la résistance de l'air ?

Ce problème peut vous paraître complexe, mais en réalité sa résolution est très simple : il vous suffit d'admettre que le boulet se déplace à la fois horizontalement et verticalement, et que ces deux mouvements simultanés sont tout à fait indépendants l'un de l'autre.

Vous allez donc établir une boucle d'animation dans laquelle vous recalculerez les nouvelles coordonnées x et y du boulet à intervalles de temps réguliers, en sachant que :

- Le mouvement horizontal est *uniforme*. À chaque itération, il vous suffit d'augmenter graduellement la coordonnée x du boulet, en lui ajoutant toujours un même déplacement Δx .
- Le mouvement vertical est *uniformément accéléré*. Cela signifie simplement qu'à chaque itération, vous devez ajouter à la coordonnée y un déplacement Δy qui augmente lui-même graduellement, toujours de la même quantité.

Voyons cela dans le script :

Pour commencer, il faut ajouter les lignes suivantes à la fin de la méthode constructeur. Elles vont servir à créer l'objet « obus », et à préparer une variable d'instance qui servira d'interrupteur de l'animation. L'obus est créé au départ avec des dimensions minimales (un cercle d'un seul pixel) afin de rester presque invisible :

```

1      # dessiner un obus (réduit à un simple point, avant animation) :
2      self.obus =boss.create_oval(x, y, x, y, fill='red')
3      self.anim =False                # interrupteur d'animation
4      # retrouver la largeur et la hauteur du canevas :
5      self.xMax =int(boss.cget('width'))
6      self.yMax =int(boss.cget('height'))

```

Les deux dernières lignes utilisent la méthode `cget()` du widget « maître » (le canevas, ici), afin de retrouver certaines de ses caractéristiques. Nous voulons en effet que notre classe `Canon` soit généraliste, c'est-à-dire réutilisable dans n'importe quel contexte, et nous ne pouvons donc pas tabler à l'avance sur des dimensions particulières pour le canevas dans lequel ce canon sera utilisé.

Remarque : `tkinter` renvoie ces valeurs sous la forme de chaînes de caractères. Il faut donc les convertir dans un type numérique si nous voulons pouvoir les utiliser dans un calcul.

Ensuite, nous devons ajouter deux nouvelles méthodes : l'une pour déclencher le tir, et l'autre pour gérer l'animation du boulet une fois que celui-ci aura été lancé :

```

1      def feu(self):
2          "déclencher le tir d'un obus"

```

```

3     if not self.anim:
4         self.anim = True
5         # position de départ de l'obus (c'est la bouche du canon) :
6         self.boss.coords(self.obus, self.x2 -3, self.y2 -3,
7                             self.x2 +3, self.y2 +3)
8         v =15 # vitesse initiale
9         # composantes verticale et horizontale de cette vitesse :
10        self.vy = -v *sin(self.angle)
11        self.vx = v *cos(self.angle)
12        self.animer_obus()
13
14    def animer_obus(self):
15        "animation de l'obus (trajectoire balistique)"
16        if self.anim:
17            self.boss.move(self.obus, int(self.vx), int(self.vy))
18            c = self.boss.coords(self.obus) # coord. résultantes
19            xo, yo = c[0] +3, c[1] +3 # coord. du centre de l'obus
20            if yo > self.yMax or xo > self.xMax:
21                self.anim =False # arrêter l'animation
22            self.vy += .5
23            self.boss.after(30, self.animer_obus)

```

Commentaires

- Lignes 1 à 4 : Cette méthode sera invoquée par appui sur un bouton. Elle déclenche le mouvement de l'obus, et attribue une valeur « vraie » à notre « interrupteur d'animation » (la variable `self.anim` : voir ci-après). Il faut cependant nous assurer que pendant toute la durée de cette animation, un nouvel appui sur le bouton ne puisse pas activer d'autres boucles d'animation parasites. C'est le rôle du test effectué à la ligne 3 : le bloc d'instruction qui suit ne peut s'exécuter que si la variable `self.anim` possède la valeur « faux », ce qui signifie que l'animation n'a pas encore commencé.
- Lignes 5 à 7 : Le canevas *Tkinter* dispose de deux méthodes pour déplacer les objets graphiques. La méthode `coords()` effectue un positionnement absolu ; il faut cependant lui fournir toutes les coordonnées de l'objet (comme si on le redessinaït). La méthode `move()`, utilisée à la ligne 17, provoque quant à elle un déplacement relatif ; elle s'utilise avec deux arguments seulement : les composantes horizontale et verticale du déplacement souhaité.
- Lignes 8 à 12 : La vitesse initiale de l'obus est choisie à la ligne 8. Comme nous l'avons expliqué à la page précédente, le mouvement du boulet est la résultante d'un mouvement horizontal et d'un mouvement vertical. Nous connaissons la valeur de la vitesse initiale ainsi que son inclinaison (c'est-à-dire l'angle de tir). Pour déterminer les composantes horizontale et verticale de cette vitesse, il nous suffit d'utiliser des relations trigonométriques tout à fait similaires à que celles que nous avons déjà exploitées pour dessiner la buse du canon. Le signe - utilisé à la ligne 9 provient du fait que les coordonnées verticales se comptent de haut en bas. La ligne 12 active l'animation proprement dite.
- Lignes 14 à 23 : Cette procédure se ré-appelle elle-même toutes les 30 millisecondes par l'intermédiaire de la méthode `after()` invoquée à la ligne 23. Cela continue aussi longtemps que la variable `self.anim` (notre « interrupteur d'animation ») reste « vraie », condition qui changera lorsque les

coordonnées de l'obus sortiront des limites imposées (test de la ligne 20).

- Lignes 18, 19 : Pour retrouver ces coordonnées après chaque déplacement, on fait appel encore une fois à la méthode `coords()` du canevas : utilisée avec la référence d'un objet graphique comme unique argument, elle renvoie ses quatre coordonnées dans un tuple.
- Lignes 17 et 22 : La coordonnée horizontale de l'obus augmente toujours de la même quantité (mouvement uniforme), tandis que la coordonnée verticale augmente d'une quantité qui est elle-même augmentée à chaque fois à la ligne 24 (mouvement uniformément accéléré). Le résultat est une trajectoire parabolique.

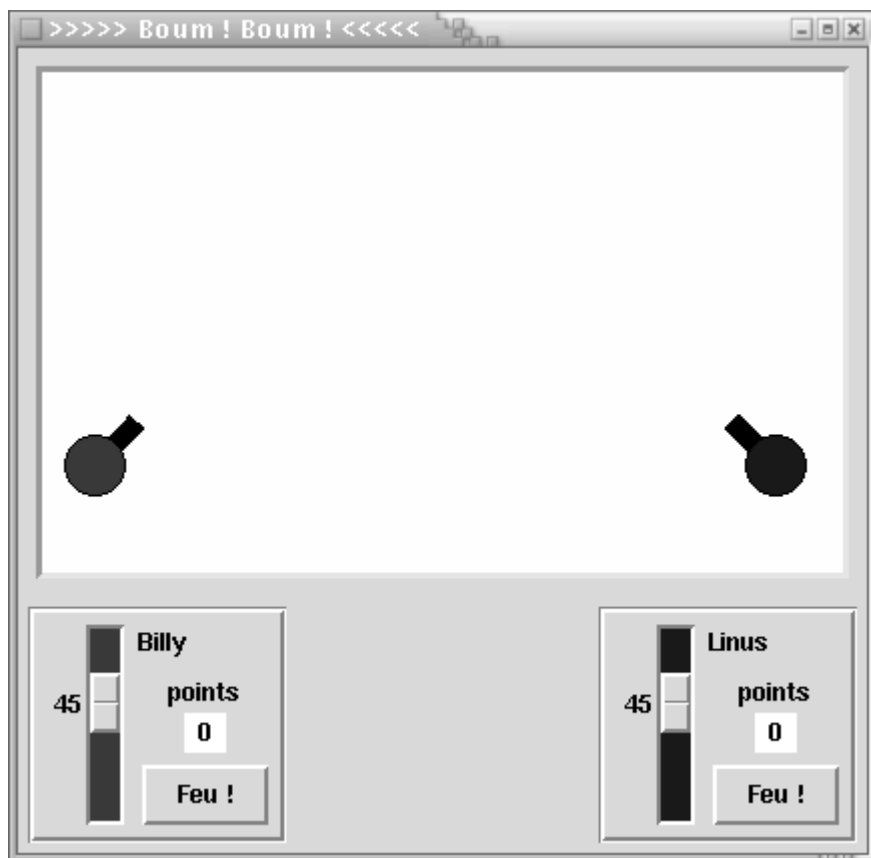
Remarque : l'opérateur `+=` permet d'incrémenter une variable : « `a += 3` » équivaut à « `a = a + 3` ».

Il reste enfin à ajouter un bouton déclencheur dans la fenêtre principale. Une ligne telle que la suivante (à insérer dans le code de test) fera parfaitement l'affaire :

```
Button(f, text='Feu !', command =c1.feu).pack(side=LEFT)
```

Développement de l'application

Disposant désormais d'une classe d'objets « canon » assez bien dégrossie, nous pouvons à présent envisager l'élaboration de l'application proprement dite. Et puisque nous sommes décidés à exploiter la méthodologie de la programmation orientée objet, nous devons concevoir cette application comme *un ensemble d'objets qui interagissent par l'intermédiaire de leurs méthodes*.



Plusieurs de ces objets proviendront de classes préexistantes, bien entendu : ainsi le canevas, les boutons, etc. Mais nous avons vu dans les pages précédentes que nous avons intérêt à regrouper des ensembles bien délimités de ces objets basiques dans de nouvelles classes, chaque fois que nous pouvons identifier pour ces ensembles une fonctionnalité particulière. C'était le cas par exemple pour cet ensemble de cercles et de lignes mobiles que nous avons décidé d'appeler « canon ».

Pouvons-nous encore distinguer dans notre projet initial d'autres composants qui mériteraient d'être encapsulés dans des nouvelles classes ? Certainement. Il y a par exemple le pupitre de contrôle que nous voulons associer à chaque canon : nous pouvons y rassembler le dispositif de réglage de la hausse (l'angle de tir), le bouton de mise à feu, le score réalisé, et peut-être d'autres indications encore, comme le nom du joueur, par exemple. Il est d'autant plus intéressant de lui consacrer une classe particulière, que nous savons d'emblée qu'il nous en faudra deux instances.

Il y a aussi l'application elle-même, bien sûr. En l'encapsulant dans une classe, nous en ferons notre objet principal, celui qui dirigera tous les autres.

Veillez à présent analyser le script ci-dessous. Vous y retrouverez la classe `Canon()` encore davantage développée : nous y avons ajouté quelques attributs et trois méthodes supplémentaires, afin de pouvoir gérer les déplacements du canon lui-même, ainsi que les coups au but.

La classe `Application()` remplace désormais le code de test des prototypes

précédents. Nous yinstancions deux objets `Canon()`, et deux objets de la nouvelle classe `Pupitre()`, que nous plaçons dans des dictionnaires en prévision de développements ultérieurs (nous pouvons en effet imaginer d'augmenter le nombre de canons et donc de pupitres). Le jeu est à présent fonctionnel : les canons se déplacent après chaque tir, et les coups au but sont comptabilisés.

```

1 from Tkinter import *
2 from math import sin, cos, pi
3 from random import randrange
4
5 class Canon:
6     """Petit canon graphique"""
7     def __init__(self, boss, id, x, y, sens, coul):
8         self.boss = boss          # réf. du canevas
9         self.appli = boss.master  # réf. de la fenêtre d'application
10        self.id = id              # identifiant du canon (chaîne)
11        self.coul = coul          # couleur associée au canon
12        self.x1, self.y1 = x, y   # axe de rotation du canon
13        self.sens = sens          # sens de tir (-1:gauche, +1:droite)
14        self.lbu = 30             # longueur de la buse
15        self.angle = 0            # hausse par défaut (angle de tir)
16        # retrouver la largeur et la hauteur du canevas :
17        self.xMax = int(boss.cget('width'))
18        self.yMax = int(boss.cget('height'))
19        # dessiner la buse du canon (horizontale) :
20        self.x2, self.y2 = x + self.lbu * sens, y
21        self.buse = boss.create_line(self.x1, self.y1,
22                                     self.x2, self.y2, width =10)
23        # dessiner le corps du canon (cercle de couleur) :
24        self.rc = 15              # rayon du cercle
25        self.corps = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
26                                     y +self.rc, fill =coul)
27        # pré-dessiner un obus caché (point en dehors du canevas) :
28        self.obus = boss.create_oval(-10, -10, -10, -10, fill='red')
29        self.anim = False        # indicateurs d'animation
30        self.explo = False       # et d'explosion
31
32    def orienter(self, angle):
33        "régler la hausse du canon"
34        # rem: le paramètre <angle> est reçu en tant que chaîne.
35        # Il faut donc le traduire en réel, puis le convertir en radians :
36        self.angle = float(angle)*pi/180
37        # rem: utiliser la méthode coords de préférence avec des entiers :
38        self.x2 = int(self.x1 + self.lbu * cos(self.angle) * self.sens)
39        self.y2 = int(self.y1 - self.lbu * sin(self.angle))
40        self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)
41
42    def déplacer(self, x, y):
43        "amener le canon dans une nouvelle position x, y"
44        dx, dy = x -self.x1, y -self.y1    # valeur du déplacement
45        self.boss.move(self.buse, dx, dy)
46        self.boss.move(self.corps, dx, dy)
47        self.x1 += dx
48        self.y1 += dy
49        self.x2 += dx
50        self.y2 += dy
51
52    def feu(self):

```

```

53     "tir d'un obus - seulement si le précédent a fini son vol"
54     if not (self.anim or self.explo):
55         self.anim = True
56         # récupérer la description de tous les canons présents :
57         self.guns = self.appli.dictionnaireCanons()
58         # position de départ de l'obus (c'est la bouche du canon) :
59         self.boss.coords(self.obus, self.x2 -3, self.y2 -3,
60                          self.x2 +3, self.y2 +3)
61         v = 17 # vitesse initiale
62         # composantes verticale et horizontale de cette vitesse :
63         self.vy = -v *sin(self.angle)
64         self.vx = v *cos(self.angle) *self.sens
65         self.animer_obus()
66         return True # => signaler que le coup est parti
67     else:
68         return False # => le coup n'a pas pu être tiré
69
70     def animer_obus(self):
71         "animer l'obus (trajectoire balistique)"
72         if self.anim:
73             self.boss.move(self.obus, int(self.vx), int(self.vy))
74             c = self.boss.coords(self.obus) # coord. résultantes
75             xo, yo = c[0] +3, c[1] +3 # coord. du centre de l'obus
76             self.test_obstacle(xo, yo) # a-t-on atteint un obstacle ?
77             self.vy += .4 # accélération verticale
78             self.boss.after(20, self.animer_obus)
79         else:
80             # animation terminée - cacher l'obus et déplacer les canons :
81             self.fin_animation()
82
83     def test_obstacle(self, xo, yo):
84         "évaluer si l'obus a atteint une cible ou les limites du jeu"
85         if yo >self.yMax or xo <0 or xo >self.xMax:
86             self.anim =False
87             return
88         # analyser le dictionnaire des canons pour voir si les coord.
89         # de l'un d'entre eux sont proches de celles de l'obus :
90         for id in self.guns: # id = clef dans dictionn.
91             gun = self.guns[id] # valeur correspondante
92             if xo < gun.x1 +self.rc and xo > gun.x1 -self.rc \
93             and yo < gun.y1 +self.rc and yo > gun.y1 -self.rc :
94                 self.anim =False
95                 # dessiner l'explosion de l'obus (cercle jaune) :
96                 self.explo = self.boss.create_oval(xo -12, yo -12,
97                                                    xo +12, yo +12, fill ='yellow', width =0)
98                 self.hit =id # référence de la cible touchée
99                 self.boss.after(150, self.fin_explosion)
100                break
101
102     def fin_explosion(self):
103         "effacer l'explosion ; ré-initialiser l'obus ; gérer le score"
104         self.boss.delete(self.explo) # effacer l'explosion
105         self.explo =False # autoriser un nouveau tir
106         # signaler le succès à la fenêtre maîtresse :
107         self.appli.goal(self.id, self.hit)
108
109     def fin_animation(self):
110         "actions à accomplir lorsque l'obus a terminé sa trajectoire"
111         self.appli.disperser() # déplacer les canons
112         # cacher l'obus (en l'expédiant hors du canevas) :
113         self.boss.coords(self.obus, -10, -10, -10, -10)
114
115

```

```

116 class Pupitre(Frame):
117     """Pupitre de pointage associé à un canon"""
118     def __init__(self, boss, canon):
119         Frame.__init__(self, bd =3, relief =GROOVE)
120         self.score =0
121         self.appli =boss                # réf. de l'application
122         self.canon =canon                # réf. du canon associé
123         # Système de réglage de l'angle de tir :
124         self.regl =Scale(self, from_ =75, to =-15, troughcolor=canon.coul,
125                         command =self.orienter)
126         self.regl.set(45)                # angle initial de tir
127         self.regl.pack(side =LEFT)
128         # Étiquette d'identification du canon :
129         Label(self, text =canon.id).pack(side =TOP, anchor =W, pady =5)
130         # Bouton de tir :
131         self.bTir =Button(self, text ='Feu !', command =self.tirer)
132         self.bTir.pack(side =BOTTOM, padx =5, pady =5)
133         Label(self, text ="points").pack()
134         self.points =Label(self, text=' 0 ', bg ='white')
135         self.points.pack()
136         # positionner à gauche ou à droite suivant le sens du canon :
137         if canon.sens == -1:
138             self.pack(padx =5, pady =5, side =RIGHT)
139         else:
140             self.pack(padx =5, pady =5, side =LEFT)
141
142     def tirer(self):
143         "déclencher le tir du canon associé"
144         self.canon.feue()
145
146     def orienter(self, angle):
147         "ajuster la hausse du canon associé"
148         self.canon.orienter(angle)
149
150     def attribuerPoint(self, p):
151         "incrémenter ou décrémenter le score, de <p> points"
152         self.score += p
153         self.points.config(text = ' %s ' % self.score)
154
155 class Application(Frame):
156     '''Fenêtre principale de l'application'''
157     def __init__(self):
158         Frame.__init__(self)
159         self.master.title('>>>> Boum ! Boum ! <<<<<')
160         self.pack()
161         self.jeu = Canvas(self, width =400, height =250, bg ='ivory',
162                          bd =3, relief =SUNKEN)
163         self.jeu.pack(padx =8, pady =8, side =TOP)
164
165         self.guns ={}                    # dictionnaire des canons présents
166         self.pupi ={}                    # dictionnaire des pupitres présents
167         # Instanciation de 2 'objets canons (+1, -1 = sens opposés) :
168         self.guns["Billy"] = Canon(self.jeu, "Billy", 30, 200, 1, "red")
169         self.guns["Linus"] = Canon(self.jeu, "Linus", 370,200,-1, "blue")
170         # Instanciation de 2 pupitres de pointage associés à ces canons :
171         self.pupi["Billy"] = Pupitre(self, self.guns["Billy"])
172         self.pupi["Linus"] = Pupitre(self, self.guns["Linus"])
173
174     def disperser(self):
175         "déplacer aléatoirement les canons"
176         for id in self.guns:
177             gun =self.guns[id]
178             # positionner à gauche ou à droite, suivant sens du canon :

```

```

179         if gun.sens == -1 :
180             x = randrange(320,380)
181         else:
182             x = randrange(20,80)
183         # déplacement proprement dit :
184         gun.deplacer(x, randrange(150,240))
185
186     def goal(self, i, j):
187         "le canon <i> signale qu'il a atteint l'adversaire <j>"
188         if i != j:
189             self.pupi[i].attribuerPoint(1)
190         else:
191             self.pupi[i].attribuerPoint(-1)
192
193     def dictionnaireCanons(self):
194         "renvoyer le dictionnaire décrivant les canons présents"
195         return self.guns
196
197 if __name__ == '__main__':
198     Application().mainloop()

```

Commentaires

- Ligne 7 : Par rapport au prototype, trois paramètres ont été ajoutés à la méthode constructeur. Le paramètre `id` nous permet d'identifier chaque instance de la classe `Canon()` à l'aide d'un nom quelconque. Le paramètre `sens` indique s'il s'agit d'un canon qui tire vers la droite (`sens = 1`) ou vers la gauche (`sens = -1`). Le paramètre `coul` spécifie la couleur associée au canon.
- Ligne 9 : Il faut savoir que tous les widgets *Tkinter* possèdent un attribut `master` qui contient la référence leur widget maître éventuel (leur « contenant »). Cette référence est donc pour nous celle de l'application principale. (Nous avons implémenté nous-mêmes une technique similaire pour référencer le canevas, à l'aide de l'attribut `boss`).
- Lignes 42 à 50 : Cette méthode permet d'amener le canon dans un nouvel emplacement. Elle servira à repositionner les canons au hasard après chaque tir, ce qui augmente l'intérêt du jeu.
- Lignes 56, 57 : Nous essayons de construire notre classe canon de telle manière qu'elle puisse être réutilisée dans des projets plus vastes, impliquant un nombre quelconque d'objets canons qui pourront apparaître et disparaître au fil des combats. Dans cette perspective, il faut que nous puissions disposer d'une description de tous les canons présents, avant chaque tir, de manière à pouvoir déterminer si une cible a été touchée ou non. Cette description est gérée par l'application principale, dans un dictionnaire, dont on peut lui demander une copie par l'intermédiaire de sa méthode `dictionnaireCanons()`.
- Lignes 66 à 68 : Dans cette même perspective généraliste, il peut être utile d'informer éventuellement le programme appelant que le coup a effectivement été tiré ou non.
- Ligne 76 : L'animation de l'obus est désormais traitée par deux méthodes

complémentaires. Afin de clarifier le code, nous avons placé dans une méthode distincte les instructions servant à déterminer si une cible a été atteinte (méthode `test_obstacle()`).

- Lignes 79 à 81 : Nous avons vu précédemment que l'on interrompt l'animation de l'obus en attribuant une valeur « fausse » à la variable `self.anim`. La méthode `animer_obus()` cesse alors de boucler et exécute le code de la ligne 81.
- Lignes 83 à 100 : Cette méthode évalue si les coordonnées actuelles de l'obus sortent des limites de la fenêtre, ou encore si elles s'approchent de celles d'un autre canon. Dans les deux cas, l'interrupteur d'animation est actionné, mais dans le second, on dessine une « explosion » jaune, et la référence du canon touché est mémorisée. La méthode annexe `fin_explosion()` est invoquée après un court laps de temps pour terminer le travail, c'est-à-dire effacer le cercle d'explosion et envoyer un message signalant le coup au but à la fenêtre maîtresse.
- Lignes 115 à 153 : La classe `Pupitre()` définit un nouveau widget par dérivation de la classe `Frame()`, selon une technique qui doit désormais vous être devenue familière. Ce nouveau widget regroupe les commandes de hausse et de tir, ainsi que l'afficheur de points associés à un canon bien déterminé. La correspondance visuelle entre les deux est assurée par l'adoption d'une couleur commune. Les méthodes `tirer()` et `orienter()` communiquent avec l'objet `Canon()` associé, par l'intermédiaire des méthodes de celui-ci.
- Lignes 155 à 172 : La fenêtre d'application est elle aussi un widget dérivé de `Frame()`. Son constructeur instancie les deux canons et leurs pupitres de pointage, en plaçant ces objets dans les deux dictionnaires `self.guns` et `self.pupi`. Cela permet d'effectuer ensuite divers traitements systématiques sur chacun d'eux (comme par exemple à la méthode suivante). En procédant ainsi, on se réserve en outre la possibilité d'augmenter sans effort le nombre de ces canons si nécessaire, dans les développements ultérieurs du programme.
- Lignes 174 à 184 : Cette méthode est invoquée après chaque tir pour déplacer aléatoirement les deux canons, ce qui augmente la difficulté du jeu.

Développements complémentaires

Tel qu'il vient d'être décrit, notre programme correspond déjà plus ou moins au cahier des charges initial, mais il est évident que nous pouvons continuer à le perfectionner.

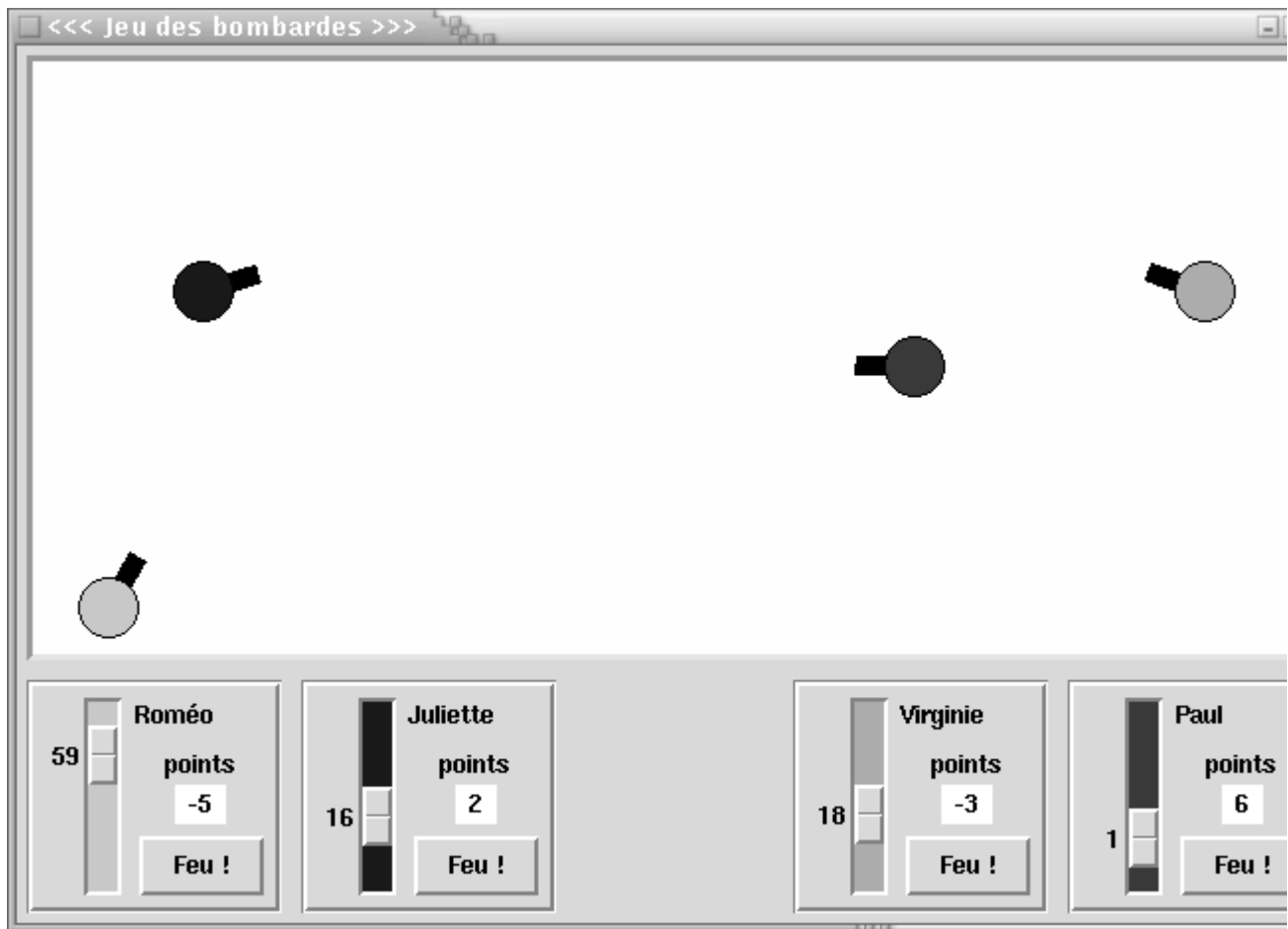
Nous devrions par exemple mieux le paramétrer. Qu'est-ce à dire ? Dans sa forme actuelle, notre jeu comporte un canevas de taille prédéterminée (400 x 250 pixels, voir ligne 161). Si nous voulons modifier ces valeurs, nous devons veiller à modifier aussi les autres lignes du script où ces dimensions interviennent (comme par exemple aux lignes 168-169, ou 179-184). De telles lignes interdépendantes risquent de devenir nombreuses si nous ajoutons

encore d'autres fonctionnalités. Il serait donc plus judicieux de dimensionner le canevas à l'aide de variables, dont la valeur serait définie en un seul endroit. Ces variables seraient ensuite exploitées dans toutes les lignes d'instructions où les dimensions du canevas interviennent.

Nous avons déjà effectué une partie de ce travail : dans la classe `Canon()`, en effet, les dimensions du canevas sont récupérées à l'aide d'une méthode prédéfinie (voir lignes 17-18), et placées dans des attributs d'instance qui peuvent être utilisés partout dans la classe.

Après chaque tir, nous provoquons un déplacement aléatoire des canons, en redéfinissant leurs coordonnées au hasard. Il serait probablement plus réaliste de provoquer de véritables *déplacements* relatifs, plutôt que de redéfinir au hasard des positions absolues. Pour ce faire, il suffit de retravailler la méthode `deplacer()` de la classe `Canon()`. En fait, il serait encore plus intéressant de faire en sorte que cette méthode puisse produire à volonté, aussi bien un déplacement relatif qu'un positionnement absolu, en fonction d'une valeur transmise en argument.

Le système de commande des tirs devrait être amélioré : puisque nous ne disposons que d'une seule souris, il faut demander aux joueurs de tirer à tour de rôle, et nous n'avons mis en place aucun mécanisme pour les forcer à le faire. Une meilleure approche consisterait à prévoir des commandes de hausse et de tir utilisant certaines touches du clavier, qui soient distinctes pour les deux joueurs.



Mais le développement le plus intéressant pour notre programme serait certainement d'en faire une *application réseau*. Le jeu serait alors installé sur plusieurs machines communicantes, chaque joueur ayant le contrôle d'un seul canon. Il serait d'ailleurs encore plus attrayant de permettre la mise en oeuvre de plus de deux canons, de manière à autoriser des combats impliquant davantage de joueurs.

Ce type de développement suppose cependant que nous ayons appris à maîtriser au préalable deux domaines de programmation qui débordent un peu le cadre de ce cours :

- la technique des sockets, qui permet d'établir une communication entre deux ordinateurs ;
- la technique des threads, qui permet à un même programme d'effectuer plusieurs tâches simultanément (cela nous sera nécessaire, si nous voulons construire une application capable de communiquer en même temps avec plusieurs partenaires).

Ces matières ne font pas strictement partie des objectifs que nous nous sommes fixés pour ce cours, et leur traitement nécessite à lui seul un chapitre entier. Nous n'aborderons donc pas cette question ici. Que ceux que le sujet intéresse se rassurent cependant : ce chapitre existe, mais sous la forme d'un complément à la fin du livre (chapitre 18) : vous y trouverez la version

réseau de notre jeu de bombardes.

En attendant, voyons tout de même comment nous pouvons encore progresser, en apportant à notre projet quelques améliorations qui en feront un jeu pour 4 joueurs. Nous nous efforcerons aussi de mettre en place une programmation bien compartimentée, de manière à ce que les méthodes de nos classes soient réutilisables dans une large mesure. Nous allons voir au passage comment cette évolution peut se faire sans modifier le code existant, en utilisant *l'héritage* pour produire de nouvelles classes à partir de celles qui sont déjà écrites.

Commençons par sauvegarder notre ouvrage précédent dans un fichier, dont nous admettrons pour la suite de ce texte que le nom est : *canon03.py*.

Nous disposons ainsi d'un module Python, que nous pouvons importer dans un nouveau script à l'aide d'une seule ligne d'instruction. En exploitant cette technique, nous continuons à perfectionner notre application, en ne conservant sous les yeux que les nouveautés :

```
1 from Tkinter import *
2 from math import sin, cos, pi
3 from random import randrange
4 import canon03
5
6 class Canon(canon03.Canon):
7     """Canon amélioré"""
8     def __init__(self, boss, id, x, y, sens, coul):
9         canon03.Canon.__init__(self, boss, id, x, y, sens, coul)
10
11     def deplacer(self, x, y, rel =False):
12         "déplacement, relatif si <rel> est vrai, absolu si <rel> est faux"
13         if rel:
14             dx, dy = x, y
15         else:
16             dx, dy = x -self.x1, y -self.y1
17         # limites horizontales :
18         if self.sens ==1:
19             xa, xb = 20, int(self.xMax *.33)
20         else:
21             xa, xb = int(self.xMax *.66), self.xMax -20
22         # ne déplacer que dans ces limites :
23         if self.x1 +dx < xa:
24             dx = xa -self.x1
25         elif self.x1 +dx > xb:
26             dx = xb -self.x1
27         # limites verticales :
28         ya, yb = int(self.yMax *.4), self.yMax -20
29         # ne déplacer que dans ces limites :
30         if self.y1 +dy < ya:
31             dy = ya -self.y1
32         elif self.y1 +dy > yb:
33             dy = yb -self.y1
34         # déplacement de la buse et du corps du canon :
35         self.boss.move(self.buse, dx, dy)
36         self.boss.move(self.corps, dx, dy)
37         # renvoyer les nouvelles coord. au programme appelant :
38         self.x1 += dx
```

```

39     self.y1 += dy
40     self.x2 += dx
41     self.y2 += dy
42     return self.x1, self.y1
43
44     def fin_animation(self):
45         "actions à accomplir lorsque l'obus a terminé sa trajectoire"
46         # déplacer le canon qui vient de tirer :
47         self.appli.depl_aleat_canon(self.id)
48         # cacher l'obus (en l'expédiant hors du canevas) :
49         self.boss.coords(self.obus, -10, -10, -10, -10)
50
51     def effacer(self):
52         "faire disparaître le canon du canevas"
53         self.boss.delete(self.buse)
54         self.boss.delete(self.corps)
55         self.boss.delete(self.obus)
56
57 class AppBombardes(Frame):
58     '''Fenêtre principale de l'application'''
59     def __init__(self, larg_c, haut_c):
60         Frame.__init__(self)
61         self.pack()
62         self.xm, self.ym = larg_c, haut_c
63         self.jeu = Canvas(self, width =self.xm, height =self.ym,
64                           bg='ivory', bd =3, relief =SUNKEN)
65         self.jeu.pack(padx =4, pady =4, side =TOP)
66
67         self.guns ={}           # dictionnaire des canons présents
68         self.pupi ={}          # dictionnaire des pupitres présents
69         self.specificites()     # objets différents dans classes dérivées
70
71     def specificites(self):
72         "instanciation des canons et des pupitres de pointage"
73         self.master.title('<<< Jeu des bombardes >>>')
74         id_list = [("Paul", "red"), ("Roméo", "cyan"),
75                  ("Virginie", "orange"), ("Juliette", "blue")]
76         s = False
77         for id, coul in id_list:
78             if s:
79                 sens =1
80             else:
81                 sens =-1
82             x, y = self.coord_aleat(sens)
83             self.guns[id] = Canon(self.jeu, id, x, y, sens, coul)
84             self.pupi[id] = canon03.Pupitre(self, self.guns[id])
85             s = not s           # changer de côté à chaque itération
86
87     def depl_aleat_canon(self, id):
88         "déplacer aléatoirement le canon <id>"
89         gun =self.guns[id]
90         dx, dy = randrange(-60, 61), randrange(-60, 61)
91         # déplacement (avec récupération des nouvelles coordonnées) :
92         x, y = gun.deplacer(dx, dy, True)
93         return x, y
94
95     def coord_aleat(self, s):
96         "coordonnées aléatoires, à gauche (s =1) ou à droite (s =-1)"
97         y =randrange(int(self.ym /2), self.ym -20)
98         if s == -1:
99             x =randrange(int(self.xm *.7), self.xm -20)
100        else:
101            x =randrange(20, int(self.xm *.3))

```

```

102     return x, y
103
104     def goal(self, i, j):
105         "le canon n°i signale qu'il a atteint l'adversaire n°j"
106         # de quel camp font-ils partie chacun ?
107         ti, tj = self.guns[i].sens, self.guns[j].sens
108         if ti != tj :           # ils sont de sens opposés :
109             p = 1               # on gagne 1 point
110         else:                   # ils sont dans le même sens :
111             p = -2              # on a touché un allié !!
112         self.pupi[i].attribuerPoint(p)
113         # celui qui est touché perd de toute façon un point :
114         self.pupi[j].attribuerPoint(-1)
115
116     def dictionnaireCanons(self):
117         "renvoyer le dictionnaire décrivant les canons présents"
118         return self.guns
119
120 if __name__ == '__main__':
121     AppBombardes(650,300).mainloop()

```

Commentaires

- Ligne 6 : La forme d'importation utilisée à la ligne 4 nous permet de redéfinir une nouvelle classe Canon() dérivée de la précédente, tout en lui conservant le même nom. De cette manière, les portions de code qui utilisent cette classe ne devront pas être modifiées (Cela n'aurait pas été possible si nous avions utilisé par exemple : « from canon03 import * »).
- Lignes 11 à 16 : La méthode définie ici porte le même nom qu'une méthode de la classe parente. Elle va donc remplacer celle-ci dans la nouvelle classe (On pourra dire également que la méthode deplacer() a été surchargée). Lorsque l'on réalise ce genre de modification, on s'efforce en général de faire en sorte que la nouvelle méthode effectue le même travail que l'ancienne quand elle est invoquée de la même façon que l'était cette dernière. On s'assure ainsi que les applications qui utilisaient la classe parente pourront aussi utiliser la classe fille, sans devoir être elles-mêmes modifiées. Nous obtenons ce résultat en ajoutant un ou plusieurs paramètres, dont les valeurs par défaut forceront l'ancien comportement. Ainsi, lorsque l'on ne fournit aucun argument pour le paramètre rel, les paramètres x et y sont utilisés comme des coordonnées absolues (ancien comportement de la méthode). Par contre, si l'on fournit pour rel un argument « vrai », alors les paramètres x et y sont traités comme des déplacements relatifs (nouveau comportement).
- Lignes 17 à 33 : Les déplacements demandés seront produits aléatoirement. Il nous faut donc prévoir un système de barrières logicielles, afin d'éviter que l'objet ainsi déplacé ne sorte du canevas.
- Ligne 42 : Nous renvoyons les coordonnées résultantes au programme appelant. Il se peut en effet que celui-ci commande un déplacement du canon sans connaître sa position initiale.
- Lignes 44 à 49 : Il s'agit encore une fois de surcharger une méthode qui existait dans la classe parente, de manière à obtenir un comportement différent : après chaque tir, désormais on ne disperse plus tous les canons

présents, mais seulement celui qui vient de tirer.

- Lignes 51 à 55 : Méthode ajoutée en prévision d'applications qui souhaiteraient installer ou retirer des canons au fil du déroulement du jeu.
- Lignes 57 et suivantes : Cette nouvelle classe est conçue dès le départ de manière telle qu'elle puisse aisément être dérivée. C'est la raison pour laquelle nous avons fragmenté son constructeur en deux parties : La méthode `__init__()` contient le code commun à tous les objets, aussi bien ceux qui seront instanciés à partir de cette classe que ceux qui seront instanciés à partir d'une classe dérivée éventuelle. La méthode `specificites()` contient des portions de code plus spécifiques : cette méthode est clairement destinée à être surchargée dans les classes dérivées éventuelles.

Jeu de Ping

Dans les pages qui suivent, vous trouverez le script correspondant à un petit programme complet. Ce programme vous est fourni à titre d'exemple de ce que vous pouvez envisager de développer vous-même comme projet personnel de synthèse. Il vous montre encore une fois comment vous pouvez utiliser plusieurs classes afin de construire un script bien structuré.

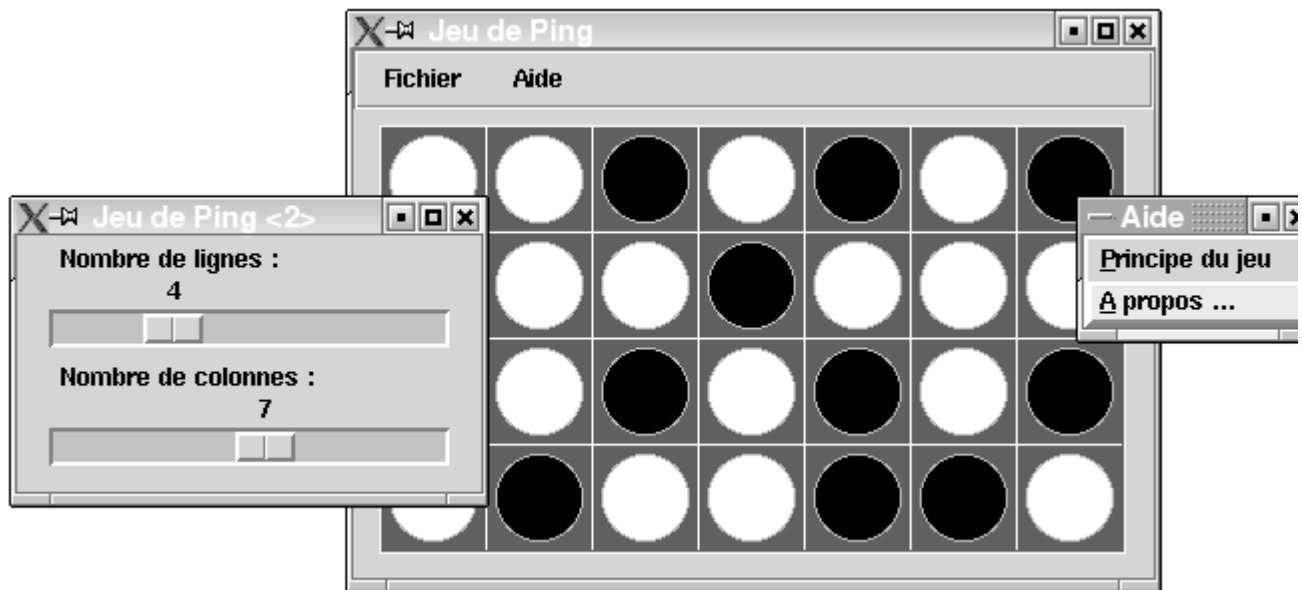
Principe

Le « jeu » mis en œuvre ici est plutôt une sorte d'exercice mathématique. Il se joue sur un panneau ou est représenté un quadrillage de dimensions variables, dont toutes les cases sont occupées par des pions. Ces pions possèdent chacun une face blanche et une face noire (comme les pions du jeu Othello/Reversi), et au début de l'exercice ils présentent tous leur face blanche par-dessus.

Lorsque l'on clique sur un pion à l'aide de la souris, les 8 pions adjacents se retournent.

Le jeu consiste alors à essayer de retourner tous les pions, en cliquant sur certains d'entre eux.

L'exercice est très facile avec une grille de 2 x 2 cases (il suffit de cliquer sur chacun des 4 pions). Il devient plus difficile avec des grilles plus grandes, et est même tout à fait impossible avec certaines d'entre elles. A vous de déterminer lesquelles ! (Ne négligez pas d'étudier le cas des grilles 1 x n)



Remarque : vous trouverez la discussion complète du jeu de Ping, sa théorie et ses extensions, dans la revue « Pour la science » n° 298 - Août 2002, pages 98 à 102.

Programmation

Lorsque vous développez un projet logiciel, veillez toujours à faire l'effort de décrire votre démarche le plus clairement possible. Commencez par établir un cahier des charges détaillé, et ne négligez pas de commenter ensuite très soigneusement votre code, au fur et à mesure de son élaboration (et non après coup !).

En procédant ainsi, vous vous forcez vous-même à exprimer ce que vous souhaitez que la machine fasse, ce qui vous aide à analyser les problèmes et à structurer convenablement votre code.

Cahier des charges du logiciel à développer

- L'application sera construite sur la base d'une fenêtre principale comportant le panneau de jeu et une barre de menus.
- L'ensemble devra être extensible à volonté par l'utilisateur, les cases du panneau devant cependant rester carrées.
- Les options du menu permettront de :
 - choisir les dimensions de la grille (en nombre de cases) ;
 - réinitialiser le jeu (c'est-à-dire disposer tous les pions avec leur face blanche au-dessus) ;
 - afficher le principe du jeu dans une fenêtre d'aide ;
 - terminer.(fermer l'application).
- La programmation fera appel à trois classes :
 - une classe principale ;
 - une classe pour la barre de menus ;

- une classe pour le panneau de jeu ;
- Le panneau de jeu sera dessiné dans un canevas, lui-même installé dans un cadre (*frame*). En fonction des redimensionnements opérés par l'utilisateur, le cadre occupera à chaque fois toute la place disponible : il se présente donc au programmeur comme un rectangle quelconque, dont les dimensions doivent servir de base au calcul des dimensions de la grille à dessiner.
- Puisque les cases de cette grille doivent rester carrées, il est facile de commencer par calculer leur taille maximale, puis d'établir les dimensions du canevas en fonction de celle-ci.
- Gestion du clic de souris : on liera au canevas une méthode-gestionnaire pour l'événement <clic du bouton gauche>. Les coordonnées de l'événement serviront à déterminer dans quelle case de la grille (n° de ligne et n° de colonne) le clic a été effectué, quelles que soient les dimensions de cette grille. Dans les 8 cases adjacentes, les pions présents seront alors « retournés » (échange des couleurs noire et blanche).

```
#####
# Jeu de ping #
# Références : Voir article de la revue #
# <Pour la science>, Aout 2002 #
# #
# (C) Gérard Swinnen (Verviers, Belgique) #
# http://www.ulg.ac.be/cifen/inforef/swi #
# #
# Version du 29/09/2002 - Licence : GPL #
#####

from Tkinter import *

class MenuBar(Frame):
    """Barre de menus déroulants"""
    def __init__(self, boss=None):
        Frame.__init__(self, borderwidth =2, relief =GROOVE)
        ##### Menu <Fichier> #####
        fileMenu = Menubutton(self, text='Fichier')
        fileMenu.pack(side =LEFT, padx =5)
        me1 = Menu(fileMenu)
        me1.add_command(label='Options', underline =0,
            command = boss.options)
        me1.add_command(label='Restart', underline =0,
            command = boss.reset)
        me1.add_command(label='Terminer', underline =0,
            command = boss.quit)
        fileMenu.config(menu = me1)

        ##### Menu <Aide> #####
        helpMenu = Menubutton(self, text='Aide')
        helpMenu.pack(side =LEFT, padx =5)
        me1 = Menu(helpMenu)
        me1.add_command(label='Principe du jeu', underline =0,
            command = boss.principe)
        me1.add_command(label='A propos ...', underline =0,
```

```

        command = boss.aPropos)
    helpMenu.configure(menu = me1)

class Panneau(Frame):
    """Panneau de jeu (grille de n x m cases)"""
    def __init__(self, boss =None):
        # Ce panneau de jeu est constitué d'un cadre redimensionnable
        # contenant lui-même un canevas. A chaque redimensionnement du
        # cadre, on calcule la plus grande taille possible pour les
        # cases (carrées) de la grille, et on adapte les dimensions du
        # canevas en conséquence.
        Frame.__init__(self)
        self.nlig, self.ncol = 4, 4          # Grille initiale = 4 x 4
        # Liaison de l'événement <resize> à un gestionnaire approprié :
        self.bind("<Configure>", self.redim)
        # Canevas :
        self.can =Canvas(self, bg ="dark olive green", borderwidth =0,
            highlightthickness =1, highlightbackground ="white")
        # Liaison de l'événement <clic de souris> à son gestionnaire :
        self.can.bind("<Button-1>", self.clic)
        self.can.pack()
        self.initJeu()
    def initJeu(self):
        "Initialisation de la liste mémorisant l'état du jeu"
        self.etat =[]                      # construction d'une liste de listes
        for i in range(12):                # (équivalente à un tableau
            self.etat.append([0]*12)      # de 12 lignes x 12 colonnes)

    def redim(self, event):
        "Opérations effectuées à chaque redimensionnement"
        # Les propriétés associées à l'événement de reconfiguration
        # contiennent les nouvelles dimensions du cadre :
        self.width, self.height = event.width -4, event.height -4
        # La différence de 4 pixels sert à compenser l'épaisseur
        # de la 'highlightbordure" entourant le canevas)
        self.traceGrille()

    def traceGrille(self):
        "Dessin de la grille, en fonction des options & dimensions"
        # largeur et hauteur maximales possibles pour les cases :
        lmax = self.width/self.ncol
        hmax = self.height/self.nlig
        # Le coté d'une case sera égal à la plus petite de ces dimensions :
        self.cote = min(lmax, hmax)
        # -> établissement de nouvelles dimensions pour le canevas :
        larg, haut = self.cote*self.ncol, self.cote*self.nlig
        self.can.configure(width =larg, height =haut)
        # Tracé de la grille :
        self.can.delete(ALL)              # Effacement dessins antérieurs
        s =self.cote
        for l in range(self.nlig -1):      # lignes horizontales
            self.can.create_line(0, s, larg, s, fill="white")
            s +=self.cote
        s =self.cote
        for c in range(self.ncol -1):      # lignes verticales
            self.can.create_line(s, 0, s, haut, fill ="white")
            s +=self.cote
        # Tracé de tous les pions, blancs ou noirs suivant l'état du jeu :
        for l in range(self.nlig):
            for c in range(self.ncol):
                x1 = c *self.cote +5          # taille des pions =
                x2 = (c +1)*self.cote -5      # taille de la case -10
                y1 = l *self.cote +5          #

```

```

        y2 = (l + 1)*self.cote - 5
        coul = ["white", "black"][self.etat[l][c]]
        self.can.create_oval(x1, y1, x2, y2, outline = "grey",
                             width = 1, fill = coul)

def clic(self, event):
    "Gestion du clic de souris : retournement des pions"
    # On commence par déterminer la ligne et la colonne :
    lig, col = event.y/self.cote, event.x/self.cote
    # On traite ensuite les 8 cases adjacentes :
    for l in range(lig - 1, lig + 2):
        if l < 0 or l >= self.nlig:
            continue
        for c in range(col - 1, col + 2):
            if c < 0 or c >= self.ncol:
                continue
            if l == lig and c == col:
                continue
            # Retournement du pion par inversion logique :
            self.etat[l][c] = not (self.etat[l][c])
    self.traceGrille()

class Ping(Frame):
    """corps principal du programme"""
    def __init__(self):
        Frame.__init__(self)
        self.master.geometry("400x300")
        self.master.title(" Jeu de Ping")

        self.mbar = MenuBar(self)
        self.mbar.pack(side = TOP, expand = NO, fill = X)

        self.jeu = Panneau(self)
        self.jeu.pack(expand = YES, fill = BOTH, padx = 8, pady = 8)

        self.pack()

    def options(self):
        "Choix du nombre de lignes et de colonnes pour la grille"
        opt = Toplevel(self)
        curL = Scale(opt, length = 200, label = "Nombre de lignes :",
                    orient = HORIZONTAL,
                    from_ = 1, to = 12, command = self.majLignes)
        curL.set(self.jeu.nlig) # position initiale du curseur
        curL.pack()
        curH = Scale(opt, length = 200, label = "Nombre de colonnes :",
                    orient = HORIZONTAL,
                    from_ = 1, to = 12, command = self.majColonnes)
        curH.set(self.jeu.ncol)
        curH.pack()

    def majColonnes(self, n):
        self.jeu.ncol = int(n)
        self.jeu.traceGrille()

    def majLignes(self, n):
        self.jeu.nlig = int(n)
        self.jeu.traceGrille()

    def reset(self):

```



```

self.jeu.initJeu()
self.jeu.traceGrille()

def principe(self):
    "Fenêtre-message contenant la description sommaire du principe du jeu"
    msg =Toplevel(self)
    Message(msg, bg ="navy", fg ="ivory", width =400,
            font ="Helvetica 10 bold",
            text ="Les pions de ce jeu possèdent chacun une face blanche et "\
                "une face noire. Lorsque l'on clique sur un pion, les 8 "\
                "pions adjacents se retournent.\nLe jeu consiste a essayer "\
                "de les retourner tous.\n\nSi l'exercice se révèle très facile "\
                "avec une grille de 2 x 2 cases. Il devient plus difficile avec "\
                "des grilles plus grandes. Il est même tout à fait impossible "\
                "avec certaines grilles.\nA vous de déterminer lesquelles !\n\n"\
                "Réf : revue 'Pour la Science' - Aout 2002")\
            .pack(padx =10, pady =10)

def aPropos(self):
    "Fenêtre-message indiquant l'auteur et le type de licence"
    msg =Toplevel(self)
    Message(msg, width =200, aspect =100, justify =CENTER,
            text ="Jeu de Ping\n\n(C) Gérard Swinnen, Aout 2002.\n"\
                "Licence = GPL").pack(padx =10, pady =10)

if __name__ == '__main__':
    Ping().mainloop()

```

Remarque : rappel : Si vous souhaitez expérimenter ces programmes sans avoir à les réécrire, vous pouvez trouver leur code source à l'adresse : <http://www.ulg.ac.be/cifen/info/ref/swi/python.htm>

Notes

1. Nous n'hésitons pas à discuter ici le développement d'un logiciel de jeu, parce qu'il s'agit d'un domaine directement accessible à tous, et dans lequel les objectifs concrets sont aisément identifiables. Il va de soi que les mêmes techniques de développement peuvent s'appliquer à d'autres applications plus « sérieuses ».
2. Il s'agit d'une suite bureautique complète, libre et gratuite, largement compatible avec MS-Office, disponible pour Linux, Windows, MacOS, Solaris ... Le présent manuel a été entièrement rédigé avec son traitement de textes. Vous pouvez vous la procurer par téléchargement depuis le site Web : <http://www.openoffice.org>

Problèmes connus

Si les opérations avec un caractère non-ASCII ne fonctionne pas

Ajouter l'encodage sous le shebang : `# coding: utf-8`.

'ascii' codec can't encode characters in position

Ajouter `.encode('utf-8')` après la variable entière (après concaténation).

Import error: No module named monModule

Il suffit de modifier le PYTHONPATH pour qu'il trouve le module mentionné, par exemple derrière une condition s'assurant que la machine qui exécute le script contient le répertoire du module :

```
import sys, socket
if socket.gethostname() == "MonUbuntu":
    sys.path.append(u'/usr/local/lib/python2.7/dist-packages/requests')
else:
    import requests
```

IndexError: list index out of range

Ajouter un test sur la taille du tableau avant d'y accéder, ex :

```
if len(tableau) > 0: print tableau[0]
```

sre_constants.error: bad character range

Arrive quand on inverse les paramètres dans `re.compile(paramètre1).search(paramètre2)`.

sre_constants.error: multiple repeat

Dans un regex, il y a des symboles de répétition consécutifs tels que `**`, `+*` ou `++`.

sre_constants.error: unmatched group

Survient quand `re.sub()` ne trouve pas un groupe de capture. On peut donc le

rechercher pour lancer le remplacement si la condition est remplie :

```
#!/usr/bin/env python
if re.compile(chaine).search(regex):
    re.sub(regex, chaine)
```

Sinon il y a peut-être une barre verticale non échappée à tort.

TypeError: maMethode() takes no arguments (1 given)

Une méthode de classe doit être déclarée avec l'argument "self" à minima.

TypeError: 'module' object is not callable

Il suffit d'appeler le module sous la forme : NomFichier.NomFonction.

TypeError: slice indices must be integers or None or have an __index__ method

Une chaîne est divisée par une variable inappropriée (ex : x dans c[x:y]).

UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 12: ordinal not in range(128)

Convertir la variable avec `.encode(config.console_encoding, 'replace')`.

UnicodeWarning: Unicode unequal comparison failed to convert both arguments to Unicode - interpreting them as being unequal

L'encodage pose problème.



Cette section est vide, pas assez détaillée ou incomplète.

ValueError: too many values to unpack

Un "for" est mal utilisé, par exemple retirer "clé" dans : "for clé, valeur in tableau".

Références

Ressources externes

Cette partie du livre *Programmation Python* présente les ouvrages et les sites Web ayant permis sa rédaction et permettant au lecteur de poursuivre son étude du sujet.

Bibliographie

En français

- **Gérard Swinnen**, *Apprendre à programmer avec Python*, Eyrolles, (ISBN 2-84177-299-3), dont plusieurs extraits libres de droit figurent dans ce livre.
- *Python en concentré*, par Alex Martelli, traduction d'Éric Jacoboni, Editions O'Reilly, Paris, 2004, 645 p., (ISBN 9782841774524). C'est le premier ouvrage de référence véritable édité en langue française. Une mine de renseignements essentielle.
- *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). Cet ouvrage est une excellente initiation à Python pour ceux qui pratiquent déjà d'autres langages.
- *L'intro Python*, par Ivan Van Laningham, traduction de Denis Frère, Karine Cottreaux 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) et (ISBN 2-84177-360-4)
- *Programmation Python*, par Tarek Ziadé, aux éditions Eyrolles. (ISBN 2-212-11677-2)

En anglais

En langue anglaise, le choix est évidemment beaucoup plus vaste. Nous apprécions personnellement beaucoup *Python : How to program*, par Deitel, Liperi & Wiedermann, Prentice Hall, Upper Saddle River - NJ 07458, 2002, 1300 p., (ISBN 0-13-092361-3) , très complet, très clair, agréable à lire et qui utilise une méthodologie éprouvée, *Core Python programming*, par Wesley J. Chun, Prentice Hall, 2001, 770 p., (ISBN 0-13-026036-3) dont les explications sont limpides, et *Learn to program using Python*, par Alan Gauld, Addison-Wesley, Reading, MA, 2001, 270 p., (ISBN 0-201-70938-4) , qui est un très bon ouvrage pour débutants.

Pour aller plus loin, notamment dans l'utilisation de la bibliothèque graphique *Tkinter*, on pourra utilement consulter *Python and Tkinter Programming*, par John E. Grayson, Manning publications co., Greenwich (USA), 2000, 658 p., (ISBN 1-884777-81-3) , et surtout l'incontournable *Programming Python* (second

edition) de Mark Lutz, Editions O'Reilly, Paris, 2001, 1255 p., (ISBN 0-596-00085-5), qui est une extraordinaire mine de renseignements sur de multiples aspects de la programmation moderne (sur tous systèmes).

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

Si vous souhaitez plus particulièrement exploiter aux mieux les ressources liées au système d'exploitation *Windows*, *Python Programming on Win32*, par Mark Hammond & Andy Robinson, Editions O'Reilly, Paris, 2000, 654 p., (ISBN 1-56592-621-8) est un ouvrage précieux. Référence également fort utile, la *Python Standard Library* de Fredrik Lundh, Editions O'Reilly, Paris, 2001, 282 p., (ISBN 0-596-00096-0)

- **Jeffrey Elkner**, *How to think like a computer scientist*

Livres en ligne

- (français) Apprendre à programmer en Python
- (français) Plongez au cœur de Python
- (anglais) How to Think Like a (Python) Programmer (licence GFDL)

Site Web

- (anglais) Site officiel de Python et son wiki
- (anglais) Convention de mise en forme du code
- (anglais) <https://www.python.org/dev/peps/pep-0008/>
- (français) Association francophone de Python
- (français) Site collaboratif français sur Python
- (français) Recueil de liens commentés
- (français) Tutoriel python
- (français) Python est mon ami

Tableau des opérateurs

Priorité des opérations

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

- **P** pour *parenthèses*. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.
Ainsi $2*(3-1) = 4$, et $(1+1)**(5-2) = 8$.
- **E** pour *exposants*. Les exposants sont évalués ensuite, avant les autres opérations.
Ainsi $2**1+1 = 3$ (et non 4), et $3*1**10 = 3$ (et non 59049 !).
- **M** et **D** pour *multiplication* et *division*, qui ont la même priorité. Elles sont évaluées avant l'*addition* **A** et la *soustraction* **S**, lesquelles sont donc effectuées en dernier lieu.
Ainsi $2*3-1 = 5$ (plutôt que 4), et $2/3-1 = -1$ (Rappelez-vous 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*).
- **A** et **S** pour *addition* et *soustraction*.

Précédence des opérateurs (du plus au moins prioritaire^[1])

Symbole ↕	Nom ↕
{}	Dictionnaire
()	Argument
[]	Partie (opérateur d'indilage)
.	Attribut
**	Puissance
~	inversion de bit
+	Positif
-	Négatif
*	Multiplier
/	Diviser
//	Résultat entier d'une division
%	Modulo
+	Plus
-	Moins
<<	Décalage à gauche
>>	Décalage à droite
&	et logique
^	ou exclusif
	ou logique
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
==	est égal
!=	est différent
is	est
in	appartient
not	non booléen
and	et booléen
or	ou booléen
lambda	expression lambda

Tableau des types

Liste des types	
<i>int</i>	Nombre entier optimisé
<i>long</i>	Nombre entier de taille arbitraire
<i>float</i>	Nombre à virgule flottante
<i>complex</i>	Nombre complexe
<i>str</i>	Chaîne de caractère
<i>unicode</i>	Chaîne de caractère unicode
<i>tuple</i>	Liste de longueur fixe
<i>list</i>	Liste de longueur variable
<i>dict</i>	dictionnaire
<i>file</i>	Fichier
<i>bool</i>	Booléen
<i>NoneType</i>	Absence de type
<i>NotImplementedType</i>	Absence d'implémentation
<i>function</i>	fonction
<i>module</i>	module

Tableau 2 : Liste des types prédéfinis en Python

Tableau des valeurs False

Les valeurs False	
bool	False
int	0
float	0.
string	""
tuple	()
list	[]
dict	{}

Tableau 3 : Liste des valeurs évaluées à False

Tableau des mots réservés

Les mots clés

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

Il faut ajouter les trois valeurs constantes True, False, None en python3 qui sont heureusement des mots clefs. Vous pouvez vous amusez en python2 à affecter True= False et Python ne dit rien ! C'est 1984 !

Autre classement :

Mot	Définition
and	Opérateur ET booléen logique
as	
assert	
break	Sortie de boucle
class	Définition de classe d'objet (Programmation Orientée Objet)
continue	
def	Définition de fonction
del	Suppression de
elif	Condition contraire
else	Contraire
except	Sauf (à utiliser après "try")
exec	
finally	
for	Boucle
from	De
global	Définition (ou utilisation) dans une fonction d'une variable globale
if	Condition
import	Importation de module
in	Contient
is	Est
is not	N'est pas
lambda	Définition d'une fonction Lambda
not	Négation logique
or	Opérateur de choix OU booléen logique
pass	
print	Afficher
raise	
return	Stopper la fonction courante (renvoyer sa valeur)
sort	Classer par ordre alphabétique
try	Essayer (généralement suivi de "except" : sauf)
while	Boucle
yield	S'emploie uniquement dans une fonction, et renvoie son résultat régénéré

Les fonctions

Commande	Définition
help()	Affiche l'aide sur le paramètre
dir()	Affiche les méthodes du paramètre
print()	Affiche le texte en paramètre
input()	Enregistre la saisie de l'utilisateur
raw_input()	Équivalent à input() (sous Python 3, préférer input())
len()	Renvoie la taille du paramètre
range()	Affiche la liste des entiers de l'intervalle du paramètre
ord()	Renvoie l'ordinal associé au caractère en paramètre
locals()	Créer un dictionnaire (objet "dict"), dont le contenu est accessible avec "["]
globals()	Comme locals() mais en incluant les variables globales
str()	Convertit une variable en caractères
int()	Convertit une variable en nombre entier
Fichiers	
open()	Ouvrir un fichier
close()	Fermer un fichier
read()	Lire un fichier
readline()	Lire une ligne
readlines()	Lire les lignes séparées par des "\n,"
tell()	Donne la position d'un objet
seek()	Donne la position d'un objet
write()	Écrire dans un fichier

Les modules

Ils sont importés avant utilisation avec "import".

Module	Définition
anydbm	
array	
atexit	
bisect	
calendar	
cmath	
codecs	
collections	
commands	
ConfigParser	
copy	
ctypes	
datetime	
decimal	
dummy_thread	
dummy_threading	
exceptions	
encodings.aliases	
formatter	
heapq	
gettext	
locale	
linecache	
marshal	
math	Module de fonctions mathématiques (sqrt, sin, cos, tan, etc...)
mmap	
operator	
os	Module de fonction concernant le système d'exploitation (OS)
pickle	
Queue	
re	Regular expressions
shelve	
shutil	

signal	
stat	
string	
StringIO	
struct	
subprocess	
sys	
textwrap	
tempfile	
thread	
threading	
time	
timeit	
traceback	
unicodedata	
xml.sax	
warnings	
whichdb	
_winreg	



GFDL

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page de couverture.

1. https://swaroop.wordpress.com/notes/python_fr-op%C3%A9rateurs_et_expressions/

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Programmation_Python/Version_imprimable&

[oldid=547992](#) »

Cette page a été modifiée pour la dernière fois le 22 avril 2017 à 00:01.
Les textes sont disponibles sous licence Creative Commons attribution partage à l'identique ; d'autres termes peuvent s'appliquer.
Voyez les termes d'utilisation pour plus de détails.