formes et transformations
éléments
de programmation

UNIVERSITÉ PARIS 8

SYNTHÈSE DE COURS

EDF1ILPA

2011

formes et transformations
éléments de programmation

Jym Feat

5e édition (novembre 2011)

Université Paris 8
Département Informatique
UFR MITSIC
Institut d'enseignement à distance

1 The back-cover text is: "*This manual has been developed at the University of Paris 8 Vincennes-Saint Denis, France*".

formes et transformations
éléments de programmation

Ce cours est fondamental, en ce sens qu'on y définit des concepts et des techniques qui vont permettre de programmer le traitement de l'information. S'adressant à des débutants, il ne requiert aucune connaissance préalable, mais exige une certaine rigueur, qui va de pair avec l'inflexibilité des langages formels. Son contenu correspond à un fascicule d'une centaine de feuillets, incluant quelques dizaines de lignes de code. Le code en question est rédigé dans un langage de programmation simple : il est proposé à titre d'exemple et sert de base aux exercices personnels ; le cours n'est donc pas un manuel de référence du langage en question.

Cette introduction aux langages de programmation est proposée en parallèle à un autre cours de programmation, mais dans un autre langage : LISP. Vous y découvrirez deux nouveaux points de vue :

- celui de l'approche dite « fonctionnelle » ou « applicative » de la programmation ;
- celui de la représentation interne des données, dont la représentation externe (celle que l'interprète veut bien vous montrer) devrait déjà vous être familière.

La raison pour laquelle ces points de vue ne sont pas vraiment développés dans ce cours-ci tient à une décision délibérée de privilégier, pour l'instant, le DESIGN des programmes. Une fois acquises les notions fondamentales, vous accéderez sans difficulté aux cours plus spécialisés du semestre suivant.

sommaire

à vol d'oiseau

0 fondements : interaction avec le système
les rudiments pour se débrouiller avec le terminal, et une vague notion de ce qu'est une interface ;
concepts d'instruction, d'expression, d'évaluation, de valeur, d'effet, et de contexte…
notion de programme interprété et de script ;
aperçu de la programmation graphique et de la modification du code source au moyen d'un éditeur de texte, d'où la notion de fichier en tant que mémoire persistante

1 données élémentaires
approche théorique du traitement de l'information ; notions de type, valeur, symbole, et d'opération sur des valeurs ; principe de l'évaluation

2 manipulation de séquences
3 listes, accès indexé
4 dictionnaires, accès par clé
ces trois chapitres condensent tout ce qu'il faut savoir pour une approche pratique du traitement de l'information ; données structurées ; manipulation de séquences (chaînes, listes) et de tables d'associations ; notions de fonction, de méthode, de programmation fonctionnelle ; expressions booléennes ; programme autonome ; opérations sur fichiers

5. interfaces : fenêtres et boutons
éléments de programmation graphique et de construction d'interfaces ; *widget*s et méthodes, contrôle de la souris, de l'affichage et de la communication entre l'utilisateur et le programme proprement dit

À partir de ce point, vous savez tout ce qu'il faut savoir, sauf qu'il peut paraître parfois difficile de mettre en pratique des connaissances théoriques abstraites ; le reste du cours a donc pour objet de vous immerger dans la réalisation de véritables programmes, avec une démarche un peu plus professionnelle, exigeant plus de rigueur dans l'analyse des problèmes posés, ainsi que pour la conception et l'inception des solutions…

En même temps, cette partie déborde du cadre d'une stricte initiation, proposant une réflexion sur l'approche des problèmes, l'adéquation des outils conceptuels déjà à votre disposition, et sur la facon de les mettre en œuvre pour programmer de façon efficace.

6 architectures de programmes
méthodologie de la conception de programmes à travers un exemple pratique

7 infrastructures logicielles
organisations du logiciel ; rudiments de l'approche orientée objet ; modèle client | serveur ; framework

8. prototypage d'applications
principes de développement rapide et de conception de programmes, techniques de programmation dirigée par les données, dirigée par les événements ; réalisation de prototypes

9. annexes fourre-tout
modules ; programmation shell…

glossaire : terminologie de l'informatique, condensée

objectif

Ce cours propose une réflexion sur le traitement analytique de l'information et sur la façon d'appréhender la structure même de cette information, en vue de la transformer.

Comme il s'adresse à des néophytes, il ne fait appel qu'à des concepts intuitifs, notamment ceux qui découlent de notre perception du texte et de l'image (le son, pourtant omniprésent dans notre vie quotidienne, a été laissé pour compte parce que traiter du signal acoustique numérisé n'est pas si intuitif que ça).

Le regard propre au traitement automatique implique par contre de nouveaux concepts, appelés structures de contrôle. Qu'on se rassure, il n'y en a qu'une petite poignée, et ils font eux aussi référence, implicitement, à des aspects intuitifs de notre comportement :

- séquence d'actions, d'instructions
- itération d'une séquence
- actions conditionnelles
- gestion d'exceptions

Le cours aurait donc 2 objectifs :

- camper le décor en présentant les structures de données et comment les manipuler
  chapitres 1, 2, 3, 4 et 5
- présenter un panorama de ce qu'on peut faire avec
  chapitres 6, 7 et 8

Ce qui signifie que seuls les 5 premiers chapitres [sans compter le 0] sont véritablement coûteux pour ce qui est des concepts techniques, les autres n'en étant que des exemples d'application, amalgamés avec des principes d'un autre ordre, ceux de la conception de programmes, de leur architecture, et de l'art de coder le minimum pour qu'il fasse le maximum.

Au terme de ce cours, l'étudiant pourra aborder de façon autonome l'apprentissage de n'importe quel autre langage de programmation, tant il est vrai que, dans le principe, ils se ressemblent tous.

Ce cours de programmation s'appuie sur un langage particulier, mais ne doit pas être perçu comme spécifique de ce langage : tous les concepts utilisés peuvent être exprimés de façon similaire dans d'autres langages, encore que les mots et les règles de syntaxe vont certainement varier – l'important, c'est d'apprivoiser les notions qui fondent l'approche programmatique et les concepts techniques ; après, il n'y aura pas grande difficulté pour s'adapter à un autre environnement…

guide de lecture

À la différence d'un cours en présentiel, où tout est rabâché 2 fois, un support de cours écrit ne présente que très peu de redondance : on la remplace autant que possible par des renvois à d'autres sections… Il en suit que le texte est relativement dense : au moment de sa rédaction, chaque mot a été pondéré pour éviter toute ambiguïté et garantir qu'il n'y aurait pas de contresens possible.

Conséquence directe, une seule lecture ne peut pas suffire ! Prenez la peine de relire plusieurs fois les passages qui vous paraissent difficiles, et de prendre des notes sur ce qui vous paraît important à votre niveau : c'est une façon de vous approprier l'essence du cours, donc de l'intégrer ; en complément, les exercices sont une autre manière d'assimiler les principes illustrés dans le support de cours : commentez-en chaque instruction si vous n'êtes pas vraiment sûr de retenir ce vous avez compris sur le moment…

supports

La digestibilité du cours dépend bien sûr de vos dispositions, mais l'expérience montre qu'il nécessite entre 20 et 30 heures d'assimilation, et peut-être plus si vous avez à cœur de réaliser tous les exercices recommandés. Cette estimation grossière peut varier en fonction du niveau initial de l'étudiant et de son assiduité.

Les exemples commentés dans le cours doivent être recopiés sous l'interprète de sorte qu'ils soient, autant que possible, évalués instruction par instruction : non seulement pour en comprendre la pertinence avec le cours, mais aussi pour prendre conscience des mécanismes à l'œuvre derrière l'interprétation d'expressions formelles.

Et pour le débutant absolu, nombreux sont les pièges ; le plus banal étant le manque de rigueur : il faut admettre pour principe que le mode d'emploi (le support de cours) doit être rigoureusement suivi à la lettre (y compris les espaces), sinon le code programmé peut n'avoir plus aucun sens pour la machine…

exercices

Des exercices supplémentaires sont proposés pour éprouver votre compréhension du cours et votre capacité à adapter les notions acquises à des problèmes différents : conçus pour servir d'auto-évaluation, ce ne sont jamais que des petites variantes des exemples déjà expliqués, et ils supposent simplement l'acuité nécessaire pour distinguer la légère différence de forme entre les données de l'exemple et celles de l'exercice.

En effet, la programmation est un art, et au début, il faut apprendre à faire ses gammes — la méthode la plus gratifiante reste l'exploration interactive. Il semble qu'il y ait deux attitudes :

1. l'explorateur teste des expressions directement dans l'interprète jusqu'à ce que ça fasse le boulot demandé
2. l'expert est un ancien explorateur : il est déjà passé par là tellement souvent qu'il peut anticiper les réponses de l'interprète, donc programmer directement la séquence d'instructions qui devrait aboutir au résultat

Alors que le second n'a qu'à écrire dans un fichier le code qu'il faut, le premier doit vérifier, en exécutant ses lignes de code avec PYTHON, qu'il n'a pas écrit que des bêtises ; il y a donc deux manières d'utiliser l'interprète : en dialoguant interactivement, ou en exécutant des scripts, rédigés sous la forme de fichiers.

L'exercice est conçu pour concrétiser un principe, une notion théorique, et installer la compréhension : à force d'accumuler ces petites expériences, vous construisez votre compétence. Et si un exercice dépasse votre capacité, ce n'est pas grave : la lumière se fera peut-être plus loin dans le chapitre, ou même dans le chapitre suivant.

Autre approche : lire le chapitre en entier, avant d'attaquer les exercices ; c'est bien aussi de savoir où on vous conduit avant qu'on ne vous y emmène… Vous allez ainsi prendre conscience que tout se tient, que les fragments de connaissance s'emboîtent les uns dans les autres pour reconstituer progressivement un puzzle dont la cohérence globale ne vous apparaîtra véritablement qu'après plusieurs années d'exploration.

En fonction de votre progression, des exercices complémentaires pourront être suggérés par votre tuteur pour faciliter l'assimilation d'un point du cours ou élargir l'horizon vers d'autres applications du même principe.

L'étudiant est invité à discuter le cours, son contenu ou sa progression pédagogique, en s'exprimant sur les forums. Toutes les critiques sont bienvenues, sauf celles qui suggèrent implicitement que soient privilégiés des outils commerciaux, autrement dit, onéreux.

⓪ FONDEMENTS : INTERACTION AVEC LE SYSTÈME

Tout commence par le terminal : non, ce n'est pas un jeu de mot, mais la prise de conscience qu'il n'y a qu'une manière de véritablement contrôler sa machine, c'est de la commander à partir du terminal, aussi appelé la console de commande…

Cette section est un bref aperçu du SHELL, le programme qui permet d'interagir directement avec le système d'exploitation. Le mot SHELL dénote une abstraction : celle de « carapace » qui protège les organes les plus vulnérables, mais, du même coup, joue le rôle d'interface entre le système d'exploitation et celui qui en prend le contrôle.

En fait, c'est une abstraction générique : le terme de SHELL ne désigne pas un programme particulier, mais plutôt une fonction, qui peut être remplie par toute une catégorie de programmes dont la particularité est de permettre l'accès à des fonctions intimes du système. Il existe donc plusieurs SHELLS, qui constituent autant de langages de programmation.

L'objectif premier du SHELL est en effet de permettre l'automatisation de procédures, autrement dit, la réalisation de programmes, sous la forme de séquences d'instructions à exécuter. Ces programmes n'ont parfois qu'une seule instruction, une « commande » tapée directement sur la ligne de commande du *terminal*, ou *console*. Presser la touche ↵ (appelée return, enter ou *entrée*) aura alors pour effet « d'envoyer la commande », c'est-à-dire de la soumettre au SHELL qui va se mettre en devoir de l'interpréter en fonction du *contexte*.

C'est de cette notion de *contexte*, techniquement appelée *environnement*, que vous devez vous imprégner dès le début : votre programme peut tourner dans un certain environnement, alors que dans un autre, il sera totalement incompréhensible.

sommaire

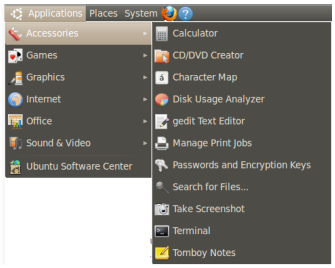
L'objectif de cette section est la prise en main du langage qui va nous servir tout au long de l'année, et jusqu'à la fin de la licence. Au passage, on y aborde quelques uns des concepts qui fondent l'aspect « mécanique » de la programmation, en montrant comment on peut concevoir un programme comme un mécanisme qui utilise d'autres mécanismes de niveau plus élémentaire.

0.1 UTILISER L'INTERPRÈTE PYTHON

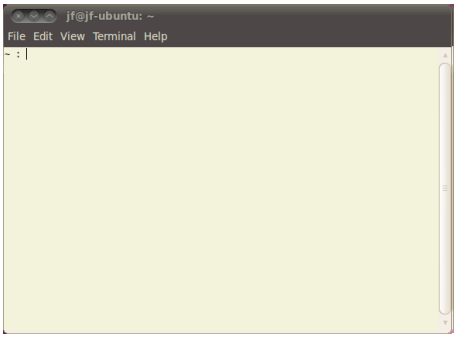
Ici, on entre par la fenêtre : ce n'est pas une boutade, c'est une nécessité — PYTHON est un interprète qui traduit (à la volée) les instructions qu'on lui donne, et les transmet au processeur sous une forme qu'il comprend, autrement dit, dans un autre langage.

Et PYTHON est un programme qu'on ne peut lancer que dans une fenêtre de terminal. Là où ça paraît se compliquer, c'est que le terminal est lui-même un programme, et qu'il faut le lancer pour lui faire ouvrir une fenêtre où nous pourrons lancer un 2e niveau de programme, PYTHON, qui va nous permettre d'exécuter notre propre programme à un 3e niveau. Mais nous rediscuterons plus tard de cet aspect des choses…

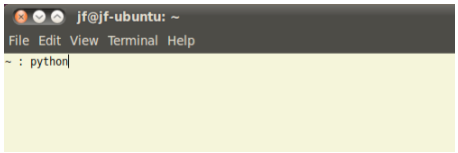
Pour ouvrir un terminal, accéder au menu APPLICATIONS ▷ ACCESSOIRES ▷ TERMINAL, et relâcher le bouton de la souris…



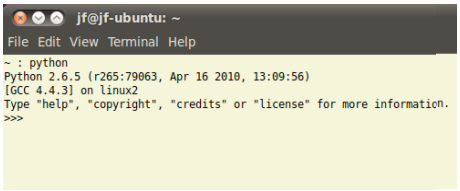
La fenêtre qui s'ouvre ainsi est constituée de 2 parties : la barre d'outils, qui (mis à part les 3 boutons standard de contrôle) ne contient que des menus déroulants, et la ligne de commande qui débute par une invite à taper une commande ; cette invite (en anglais, PROMPT), on peut la « customiser », et moi, je me suis arrangé pour qu'elle soit aussi discrète que possible ; on en reparlera plus tard…



Et puisque l'invite m'y invite, je vais taper quelque chose : le nom du programme à lancer :

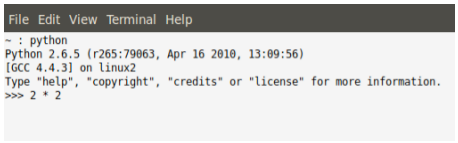


À ce stade, rien n'est définitif : je pourrais entièrement gommer avec la touche ⌫ ce que je viens de taper, ou repositionner le curseur avec ← et → pour corriger une faute d'orthographe ; la commande ne sera prise en compte qu'au moment où j'appuierai sur la touche ↵ pour la valider : allons-y !



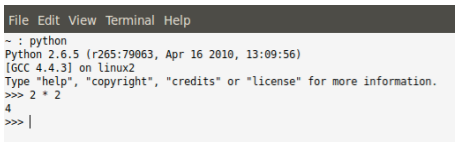
Le message qui s'affiche m'indique quelle version de python a été trouvée ; d'ailleurs il en profite pour me rappeler que je peux obtenir des renseignements supplémentaires en entrant l'une des quatre instructions proposées… à la suite de l'invite, qui a maintenant changé, parce que le contexte a lui-même changé !

Mais ce qui m'intéresse ici, c'est de voir comment python réagit quand je veux traiter de l'information ; par exemple, étant donnés deux nombres, en calculer le produit :

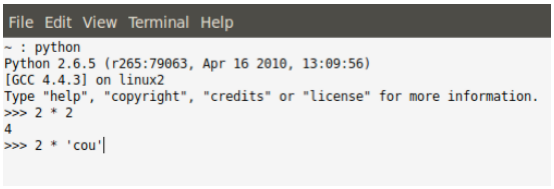


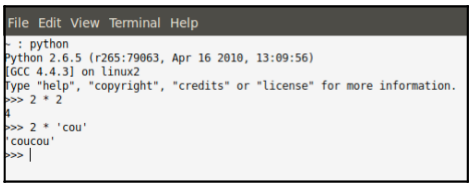
Là aussi, il ne se passe rien tant que je n'envoie pas l'instruction à l'interprète, ce qui me laisse toute latitude pour vérifier que ma syntaxe est correcte, et que les mots sont compréhensibles.

Maintenant, l'appui sur la touche ↵ va provoquer l'évaluation de mon expression arithmétique, évaluation qui retourne une valeur, laquelle valeur est automatiquement affichée par l'interprète en réponse à ma question :



À partir de là, il faut connaître ce langage pour savoir ce qu'on peut dire, ou demander, et ce qu'il saura interpréter ou pas ; je peux, par exemple, calculer le double du mot 'cou'… à condition de le présenter comme une *chaîne de caractères*, c'est-à-dire en prenant soin de l'insérer entre guillemets parce que ce n'est pas un mot du langage PYTHON.





Le résultat est absolument rassurant, mais en même temps, il dénote un fonctionnement qui n'a rien à voir avec l'évaluation d'une expression arithmétique ; et pourtant, c'est le même opérateur * : tout se passe comme s'il « voyait » ce qu'il doit multiplier, et décidait d'opérer différemment selon la nature de l'opérande ; on comprendra plus tard pourquoi et comment…

0.2 UTILISER LES FONCTIONS D'UN MODULE IMPORTÉ

Le langage PYTHON n'est pas grand'chose d'autre qu'un interprète d'instructions ; en fait, toute sa puissance vient des modules qui s'y ajoutent (automatiquement chargés au démarrage) ou que j'y ajoute « à la main » : ici, je vais charger le module turtle, c'est-à-dire *tortue*, qui permet d'émuler quelques fonctionnalités d'un autre langage, LOGO, originellement destiné à l'enseignement de la programmation aux enfants des écoles primaires. En gros, il s'agit de manipuler un mobile à l'écran, et lui faire dessiner des trucs…

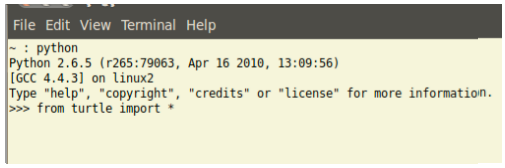
Toute la subtilité d'un tel apprentissage réside dans cette découverte de l'aspect *procédural* de la programmation (on parle aussi d'approche *impérative*), qui met en évidence qu'un programme est constitué d'une séquence ordonnée d'instructions, autrement dit, une procédure : la notion de séquence est fondamentale dans l'approche procédurale — on a même vite fait de s'apercevoir qu'une séquence peut elle-même être constituée de multiples sous-séquences, impliquant l'idée de répétition, et, du même coup, de généralisation.

Ce dont je vais faire ici la démonstration ne peut fonctionner correctement que si vous avez suivi scrupuleusement les consignes données dans la section 5.2 du *document de prise main*[2], en particulier, si vous avez bien installé le paquet python tk ; pour le savoir, essayez ce qui suit…

Si ce n'est déjà fait, ouvrez une fenêtre de terminal, lancez PYTHON, puis, comme ci-contre, tapez l'instruction :

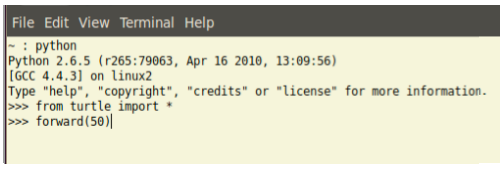
```
from turtle import *
```

en respectant, bien sûr, les espaces entre les mots.

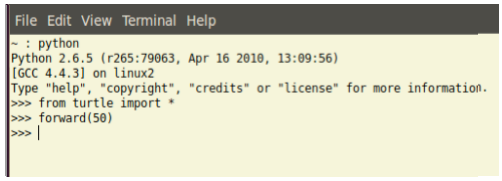


Exécuter cette ligne ne provoque aucune réaction apparente : ce qui ne veut pas dire qu'il ne se passe rien ; en fait, si l'importation échouait, il y aurait un message d'erreur… s'il n'y en a pas, c'est que jusqu'ici, l'interprète a bien compris ce qu'on lui demandait, et a fait son boulot sans baver ;

À ce stade, pour aller plus loin, il faudrait savoir quelles fonctions sont disponibles dans ce module, et à quoi elles servent ; elles sont documentées ici : <http://docs.python.org/library/turtle.html#module-turtle>, mais on ira voir plus tard ; pour le moment, je dis tout ce qu'il y a à savoir…



La fonction forward() fait avancer la tortue d'un certain nombre de pixels ; comment sait-elle de combien de pixels avancer ? On lui en passe le nombre entre parenthèses, et c'est ce qu'on appelle l'argument de la fonction ; dans forward(50), ce 50 est l'argument de forward().

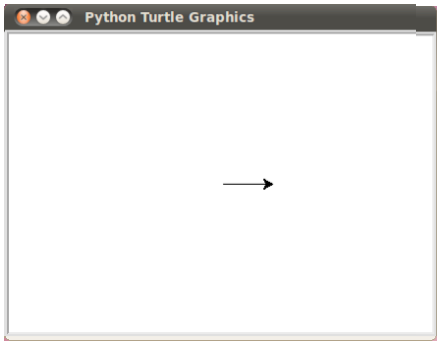


2. cf. <http://foad.iedparis8.net/claroline/claroline/document/goto/index.php/ubuntu%2C%20prise%20en%20main.pdf?cidReq=ELI000_&gidReset=true>

0.3 PROGRAMMATION EN PAS-À-PAS

Tant que je n'ai pas envoyé la ligne, je peux encore reculer avec le curseur, et changer le 50 en 65 ; ce n'est qu'au moment où j'envoie que l'interprète exécute l'instruction, mais là encore, rien ne me dit, dans la fenêtre du terminal, que ça a marché…

Cependant, turtle a ouvert une nouvelle fenêtre, et maintenant je vois l'effet de la fonction : partant du centre de la fenêtre, turtle a avancé de 50 pixels vers l'est, en laissant 50 points noirs sur le chemin parcouru ; la tortue proprement dite est figurée par la pointe de la flèche : je vois donc en même temps sa localisation et son orientation.



> À ce stade, je pourrais changer la fenêtre de place, ou même la redimensionner : elle ne dépend pas de la fenêtre du terminal, mais du programme qui est en train de s'exécuter, lui, dans la fenêtre du terminal, programme que je construis moi-même, instruction par instruction…

Tiens, à propos d'orientation, il y a une fonction pour la changer : left() tourne la tortue de n degrés vers la gauche, donc left(90) effectuera une rotation de 90 ° ;

sitôt dit, sitôt fait, mais encore une fois, l'interprète ne retourne aucun résultat dans la fenêtre du terminal ; ce qui compte, c'est l'effet, visible dans la fenêtre graphique : observez qu'en fait, rien n'a changé, si ce n'est que la tortue est désormais orientée vers le nord…

```
File  Edit  View  Terminal  Help
~ : python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from turtle import *
>>> forward(50)
>>> left(90)
>>>
```

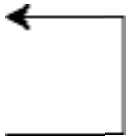
```
File  Edit  View  Terminal  Help
~ : python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from turtle import *
>>> forward(50)
>>> left(90)
>>> forward(50)
```

Maintenant, je voudrais de nouveau avancer de 50 pixels ; je pourrais retaper la même chose qu'avant, mais puisque, justement, c'est la même chose, je peux aussi presser la touche ↑ pour remonter dans le temps, et retrouver sous mes doigts la ligne tapée tout à l'heure…

À ce stade, je pourrais l'éditer (la modifier), mais elle me convient telle qu'elle est, et je n'ai qu'à presser la touche ↵ pour exécuter cette instruction ; et la tortue se déplace, toujours de 50 pixels, mais cette fois-ci vers le nord :

```
File  Edit  View  Terminal  Help
~ : python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from turtle import *
>>> forward(50)
>>> left(90)
>>> forward(50)
>>> left(90)
>>> forward(50)
>>>
```

Encore une fois, pareil, je vais de nouveau tourner de 90 °, puis progresser de 50 pixels ; regardez :



D'un point de vue « mécanique », ce sont les mêmes instructions, et l'effet de principe est toujours le même : ce qui change, c'est le contexte dans lequel est exécutée la paire d'instructions forward() suivie de left(), ce qui fait que l'effet est, globalement, différent…

Maintenant, je vous laisse deviner quelles sont les instructions qui manquent pour finir le carré : bon, c'est un peu facile, mais c'est justement ça qui nous intéresse — arriver à concevoir le programme comme une procédure qui fait elle-même appel à d'autres procédures plus élémentaires.

Et si vous avez saboté votre image, fermez simplement la fenêtre graphique : ça obligera le programme à en ouvrir une nouvelle toute propre, sans les effets des instructions antérieures.

0.4 J'ÉDITE AVEC GEDIT

Le programme de tout à l'heure, une fois testé, on peut avoir besoin d'en garder une trace : en particulier s'il s'agit d'un exercice à rendre… Les menus du terminal permettent de sélectionner le contenu de la fenêtre et de le copier : lancez maintenant l'éditeur de textes GEDIT (par le menu où vous avez trouvé le terminal) et collez-y la copie que vous venez de faire. Votre programme peut maintenant être manipulé comme un texte normal, il ne vous reste donc plus qu'à faire un petit peu de ménage pour ne conserver que ce que vous avez effectivement entré comme instructions :

- supprimer les 3 lignes du haut, affichées par python ;
- utiliser la fonction de remplacement pour enlever tous les '>>> ', espace compris ;

Il ne doit donc plus rester que ce que vous avez effectivement tapé…

Maintenant, si vous sélectionnez ce bloc de texte (bien nettoyé) et que vous le copiez pour le coller dans le terminal, tout se passe comme si vous aviez entré ces instructions à la main : PYTHON ne fait pas la différence, et le programme s'exécute tout pareil !

Programmer en pas-à-pas, c'est très utile, surtout au début, pour être sûr que c'était bien cette instruction-là qu'on voulait exécuter ; avec un peu d'expérience, on est capable de prédire la valeur d'une expression, ou l'effet d'une instruction ; et donc de coder d'un seul tenant tout le programme. D'ailleurs, pour fixer les idées, le texte tout propre pourrait maintenant ressembler à :

```
from turtle import *
forward(50) ; left(90)
forward(50) ; left(90)
forward(50) ; left(90)
forward(50) ; left(90)
```

sachant que j'ai regroupé (en les séparant par un point-virgule) les paires d'instructions qui mettent en évidence l'aspect itératif (répétitif) de ce programme à quatre côtés. Remarque : ce fichier doit être enregistré sur disque pour y être mémorisé ; on lui donne le nom qu'on veut, mais par convention, on y ajoute le suffixe '.py' pour qu'il soit évident, au premier coup d'œil, qu'il s'agit d'un script PYTHON.

0.5 LE MODE IMPÉRATIF

On peut se demander, comme évoqué plus haut, pourquoi il y a des instructions qui affichent une valeur en retour, et d'autre pas : c'est parce ces dernières sont utilisées pour leur effet, comme, par exemple, import ; en revanche, une expression comme 2 * 2 n'a pas d'effet, mais elle a une valeur ; et c'est cette valeur que va afficher PYTHON, en réponse.

Pour clarifier : import est un ORDRE (impératif) qui oblige PYTHON à faire un boulot, mais ça ne produit pas de valeur… alors que 2 * 2 serait plutôt une QUESTION qui attend une réponse.

0.6 POUR COMMENCER AVEC LE TERMINAL

D'après ce que nous venons de voir, le terminal serait une sorte particulière de fenêtre ; à la différence d'une page web, où on ne peut que lire, ou même d'un programme de courrier électronique, dont la fenêtre permet aussi d'écrire, la fenêtre de terminal, elle, est active, en ce sens qu'elle réagit à ce qu'on écrit dedans.

Ouvrir le terminal lance implicitement un programme appelé bash, et c'est lui qui interprète les commandes ; c'est donc lui qui regarde ce qu'il y a sur la ligne qu'on vient de valider en appuyant sur la touche ↵ et qui l'analyse, mot par mot, avant de « comprendre » ce qu'elle veut dire.

En fait, bash n'est que l'un des interprètes possibles : il en existe une floppée d'autres, qu'on désigne sous le nom générique de SHELL, c'est-à-dire *carapace*, pour évoquer son rôle d'interface entre l'utilisateur aux doigts gourds et les entrailles bien vulnérables du système d'exploitation.

Bon : il n'est pas indispensable, pour le moment, d'en savoir plus sur le contrôle de l'ordinateur par le biais du terminal, mais vous trouverez, dans le chapitre 9, une section qui lui est consacrée.

RÉCAPITULATION

En réalité, les quelques pages qui précèdent sont destinées à tester que votre système est correctement configuré pour effectuer les exercices de la suite du cours.

Le but inavoué de cette section était de vous pousser dans la piscine sans se poser la question de savoir si vous saviez déjà nager ; en pratique même si vous faites très attention, vous avez sans doute commis quelques étourderies, fait des fautes d'orthographe, oublié l'espace ou la parenthèse…

C'est normal, et il faut même qu'il en soit ainsi — nul n'apprend à marcher s'il n'est jamais tombé, et chaque pas est une chute contrôlée ; de la même façon, chaque instruction est un saut dans le vide, et il faut l'avoir exécuté au moins une fois pour en comprendre le résultat, que ce soit un effet ou une valeur, et l'exploiter dans un programme différent.

Petit lexique des concepts survolés :

AFFICHAGE : effet d'une instruction, ou d'une commande
ARGUMENT : l'opérande d'une fonction
BASH : le shell par défaut dans l'environnement ubuntu (et aussi MAC OS X)
COMMANDE : manière de dire « instruction » quand on opère sous un SHELL
ÉDITEUR : programme spécialisé dans la modification d'un texte de programme
EFFET : un changement d'état pour le programme
ENVIRONNEMENT : contexte opératoire
ÉVALUATION : interprétation du « calcul » exprimé dans une expression
EXPRESSION : un assemblage (opérateur, opérandes) qui produirait une valeur
FICHIER : données enregistrées sur disque
FONCTION : nom symbolique qu'on utilise pour déclencher une procédure
GEDIT : éditeur de texte dans le monde LINUX
IMPÉRATIF : un mode de communication qui attend un effet, mais pas forcément de réponse
IMPORT : instruction pour charger un module
INSTRUCTION : un ordre donné à l'impératif
INTERPRÉTATION : traduction à la volée d'un langage dans un autre
LANGAGE : un ensemble de mots et de règles de syntaxe
LEXIQUE : ensemble de mots
LOGO : langage interprété, conçu en 1966 pour l'apprentissage de la programmation
MODULE : un programme tout fait dont on peut utiliser les fonctionnalités en l'important
OPÉRATEUR : symbole (ou code sous-jacent) utilisé pour déclencher une opération
OPÉRANDE : ce sur quoi on opère
PIXEL : un point de l'écran
PROCÉDURE : voir PROGRAMME
PROGRAMME : une séquence ordonnée d'instructions
PYTHON : langage interprété conçu en 1991 pour allier la puissance de C à la souplesse de LISP
SHELL : interface pour l'interprétation de commandes et le lancement de programmes
SYNTAXE : ordre des mots pour produire des expressions bien formées
TERMINAL : un programme qui permet de lancer d'autres programmes
TURTLE : module d'émulation du langage LOGO
VALEUR : résultat d'une évaluation

Notons que toutes ces définitions sont volontairement vagues, voire même parfois inexactes : l'objectif n'est pas de vous faire apprendre quoi que ce soit, et ce lexique se veut seulement une manière de récapituler ce que nous venons d'expérimenter…

En même temps, cette expérience est un AUTO-TEST : si cette introduction vous a déplu, peut-être faut-il reconsidérer votre orientation, ou vos motivations ?

Alors si vous avez des questions complémentaires, rendez-vous sur le forum !

① DONNÉES ÉLÉMENTAIRES

Ce chapitre introduit les rudiments de la programmation : les types élémentaires de données et les concepts qui permettent d'effectuer des opérations sur ces données.

Qu'un type de donnée soit élémentaire ou non est affaire de point de vue : à ce stade, nous considérerons comme élémentaire une donnée qui peut être manipulée en une seule opération simple, sans entrer dans le détail de sa véritable nature ou de sa représentation en machine.

Le texte du cours est bourré d'exemples qu'il ne sert à rien de lire : tous les exemples doivent être mis en pratique immédiatement, sous peine de ne pas comprendre le texte lui-même, qui n'a de sens que par rapport au résultat de l'exemple ; et de toute façon, tester l'exemple est le seul moyen de vous l'approprier, en l'incorporant dans votre expérience de programmeur.

Par ailleurs, les exemples modifient le contexte, et c'est dans ce contexte que les exemples suivant doivent eux-mêmes être testés : hors contexte, ces exemples n'ont pas de sens.

sommaire

Dans ce qui va suivre, on suppose déjà familier le concept d'ordinateur numérique, machine passablement complexe qui met en œuvre des ressources matérielles (processeur, mémoire) et logicielles (micro-code, système d'exploitation), structurées sur plusieurs niveaux et interagissant aussi efficacement que le permet une technologie dont la mise au point a commencé il y a déjà plus de 60 ans – et qui semble loin d'être terminée.

Pour ceux qui veulent se documenter plus précisément, un cours complémentaire, « introduction à l'architecture des ordinateurs », propose de descendre dans les détails, mais ce n'est pas absolument nécessaire pour comprendre ce qui va suivre…

1.0 PRÉLIMINAIRES CULTURELS

Si la vocation première d'un ordinateur est la représentation et le traitement de l'information, programmer, c'est manipuler cette information pour la représenter de façon différente, la transformer, en extraire certains aspects, ou en extrapoler de nouvelles informations.

PROGRAMMER, C'EST QUOI ?

Programmer, c'est exprimer une séquence de directives sous une forme compréhensible, donc exécutable – par une machine.

La recette de cuisine (non exécutable pour l'instant) serait l'exemple type de programme :

1. jeter les pâtes dans l'eau bouillante salée
2. laisser cuire 1 minute

Ces directives impliquent des actions (et des tests) sur des objets explicites (pâtes, eau, sel) ou non (chaleur, temps). C'est peut-être moins évident à première vue, mais l'état des objets peut avoir son importance : pour appliquer la directive 1, il faut que l'eau soit salée, et qu'elle soit bouillante. Par ailleurs, la directive 2 n'a de sens que si la 1 a été appliquée correctement. Enfin, remarquons que cette recette implique d'autres actions élémentaires totalement implicites, comme prendre une casserole, la remplir d'eau, et la faire chauffer ; prendre une casserole est en fait un programme relativement complexe, impliquant à la fois perception sensorielle (visuelle, tactile) et contrôle moteur, mettant en jeu quelques dizaines de muscles, ou même plus s'il faut aller l'acheter.

Cet exemple met ainsi en évidence trois propriétés fondamentales du programme :

- la séquence présente les directives dans un ordre qui doit être suivi scrupuleusement
- chaque directive doit être compréhensible, sinon elle ne saurait être appliquée
- une directive peut être elle-même un programme

C'est justement la complexité d'une tâche qui la rend difficile à programmer. C'est pourquoi l'informatique ne permet de faire que des choses simples. Mais du fait qu'une directive peut être un programme, il devient possible de faire un programme *simple* constitué de directives complexes. Autrement dit, un programme constitué de programmes.

Quelques remarques :

- le nombre d'éléments dans la séquence n'est pas contraint, mais on peut difficilement parler de programme s'il y a 0 instruction ; et la séquence peut ne comporter qu'une seule directive : c'est quand même un programme…
- on pourrait dire *instruction* au lieu de *directive* sans changer fondamentalement le sens de la définition ci-dessus ; les programmeurs parlent aussi de commandes, d'opérations ou de fonctions ; bien que ces termes puissent, en contexte, prendre un *sens technique* extrêmement précis, nous pouvons, pour le moment, les considérer comme équivalents ;

LANGAGES DE PROGRAMMATION

Ainsi, programmer, ce serait décrire une séquence d'opérations à effectuer sur des objets, et ordonner à la machine d'exécuter cette séquence. Le problème, c'est qu'une machine ne sait faire, à la base, que

quelques opérations fondamentales, comme mémoriser de l'information et la manipuler de manière rudimentaire : en gros, quelques opérations arithmétiques et logiques. Il se trouve qu'en combinant ces opérations rudimentaires (par programme) on peut réaliser des opérations beaucoup plus complexes, et plus proches de notre intuition : comme dessiner un cercle de plusieurs milliers de points en indiquant simplement les coordonnées de son centre et la mesure de son rayon.

Ce deuxième niveau de complexité est directement accessible par le biais de langages de programmation, qui disposent implicitement de ce *savoir-faire* préprogrammé. Beaucoup de langages (et donc de programmes écrits dans ces langages) peuvent être directement interprétés par un programme spécifique, justement appelé *interprète*. D'autres doivent d'abord être traduits sous une forme compréhensible par la machine avant de pouvoir être exécutés, et c'est là le rôle d'un *compilateur*, autre programme spécifique.

Dans cette introduction, nous n'utiliserons que des langages du premier type : les langages interprétés. L'intérêt de cette approche est qu'elle permet de travailler de manière interactive : un ordre donné à l'interprète est immédiatement exécuté, et si son résultat est inattendu, il est possible de le corriger sur le champ. Dans le cas de programmes compilés, il faut passer par une étape supplémentaire pour voir le résultat du programme, ce qui n'encourage pas la programmation *incrémentale*, la seule méthode qui assure la maîtrise totale du programme, élément par élément.

L'intérêt des langages compilés, c'est qu'ils sont traduits sous une forme directement compréhensible par le processeur de la machine, et qu'ils s'exécutent donc théoriquement plus vite que lorsqu'ils sont interprétés. Mais comme tout programme doit d'abord passer par une phase de mise au point, la vitesse d'exécution n'est, en définitive, que secondaire…

Ainsi, la priorité est donnée aux phases 0 et 1 : conception, mise au point. Du point de vue industriel, ces phases sont coûteuses, et ne rapportent rien tant que le produit n'est pas commercialisable. La tendance est donc de favoriser les outils qui permettent le RAD, ou développement rapide de prototypes, pour réduire au minimum l'investissement. C'est ce qui explique la montée en puissance des langages interprétés depuis une quinzaine d'années.

LANGUE NATURELLE, LANGAGE FORMEL

Les langues naturelles, par nature ambiguës, sont utilisées par des êtres intelligents qui perçoivent en fait bien plus qu'il n'est dit – comme dans l'exemple de la recette ci-dessus ; à l'opposé, les langages de programmation sont des langages *formels* destinés à des machines sans intelligence, et sont par nature non-ambigüs : ils ne s'appuient jamais sur des implicites non-énoncés dans le contexte. De ce fait, ils sont terriblement contraignants, en ce sens que changer la place d'une virgule peut faire une différence radicale entre deux instructions.

Comme tout langage, les langages de programmation sont interprétés à trois niveaux :

- le niveau lexical – aussi appelé *lexique* ou *vocabulaire* – est celui des mots, qui représentent des verbes ou des noms, c'est-à-dire des actions ou des objets ; plus formellement, ils jouent le rôle d'opérandes, et sont manipulés par des opérateurs ;
- le niveau syntaxique est celui de l'organisation des mots en phrases – les programmeurs parleront plutôt d'*expressions* – et l'ordre des mots est évidemment crucial, puisque l'eau ne peut pas cuire *dans* des pâtes bouillantes ;
- le niveau sémantique est celui du résultat de l'interprétation de ces mots dans le contexte de la phrase : il existe par exemple des langages où un même mot peut aussi bien être un verbe qu'un nom, en fonction de sa place dans la phrase ; ou encore, un même opérateur peut avoir différents effets selon le type d'objet auquel il s'applique.

Comme tout langage, les langages de programmation manipulent des symboles et non les objets qu'ils représentent réellement : le *nom* n'est pas la *chose*, la *carte* n'est pas le *territoire*. Par exemple, l'expression 3 + 4 contient deux symboles de constantes arithmétiques et un symbole d'opérateur : le symbole 3 représente la *quantité* iii, qui, ajoutée à la *quantité* représentée par 4 donnera une nouvelle *quantité*, représentée cette fois par le symbole 7.

COMBIEN EXISTE-T-IL DE LANGAGES DE PROGRAMMATION ?

Plusieurs milliers ; et ce, pour des raisons historiques et pratiques. On ne programme plus aujourd'hui comme il y a cinquante ans parce qu'on ne programme plus les mêmes choses, et certains langages permettent de programmer certaines tâches plus facilement que d'autres, ou qui seraient même impossibles autrement. C'est d'ailleurs pour cette raison qu'il s'en crée de nouveaux chaque année[3] : en 2006, il y en avait déjà 8512…

Chaque langage a son propre *lexique* prédéfini, ainsi qu'une *syntaxe* spécifique, et bien entendu l'interprétation *sémantique* qui va avec. Beaucoup de langages ont la propriété d'être extensible, au moins pour ce qui est du lexique : on définira de nouveaux symboles en fonction de combinaisons de symboles déjà existants. Effectivement, si je peux donner un nom à une expression symbolique, l'utilisation de ce nom sera désormais équivalente à l'expression elle-même. Ainsi vais-je pouvoir créer des *abstractions* de haut niveau et manipuler des données en invoquant simplement le nom du programme complet.

Mais avant d'en arriver là, le premier problème que rencontrent les débutants, c'est *comment dire ce qu'on veut faire*… Autrement dit : comment puis-je, avec le lexique de ce langage, construire des expressions syntaxiquement bien formées qui expriment précisément le traitement que je veux appliquer à l'information dont je dispose.

1.1 SHELL PYTHON

PYTHON, langage interprété, a été choisi d'une part à cause de sa diffusion (logiciel libre, disponible sur toutes les plate-formes) et d'autre part pour ses qualités intrinsèques : simple, puissant, et structurant du fait de son orientation objet.

Okay, ce n'est ni le plus simple des langages[4], ni le plus puissant, mais ici, il représente un excellent compromis pour des utilisations qui couvrent de nombreux domaines du traitement de l'information : du multimédia aux applications numériques scientifiques, en passant par l'administration de systèmes d'exploitation, de sites WEB, ou de forums.

ÉVALUATION EN MODE INTERACTIF

Dans un environnement UNIX standard, on appelle l'interprète par son nom, en *minuscules*, dans une fenêtre de TERMINAL : python

```
~ : python
Python 2.5.2 (r252:60911, Feb 22 2008, 07:57:53)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

L'interprète répond avec son n° de version (ici 2.5.2) puis avec le symbole triple >>> qui est une invitation à entrer une expression à évaluer : s'il comprend ce qu'on lui demande, il retourne en réponse le résultat de l'évaluation, sinon il affiche un message d'erreur détaillé.

Ce SHELL est un *environnement* de programmation, et se superpose à l'environnement courant ; pour revenir à l'environnement précédent, la commande standard est ^D – il s'agit d'enfoncer *simultanément* la touche de contrôle (étiquetée ctrl) et la touche D – CONTRÔLE-D est aussi parfois appelé EOT : *end of transmission*.

Si cette commande est inopérante, il est encore possible de suspendre le processus avec la commande ^Z ; et dans tous les cas, on devrait pouvoir SORTIR de l'interprète en tapant cette instruction, qu'on expliquera plus tard : exit(). Ou encore, de fermer la fenêtre, tout bêtement…

PREMIERS PAS

Lancez PYTHON comme ci-dessus, et utilisez l'interprète pour évaluer, en séquence, les quatre expressions ci-après :

```
2 + 3 * 4
'a' + 'b' * 4
"au" + "jour" + "d'hui"
a + b
```

Pour chaque expression, vérifiez la réponse de PYTHON : est-ce bien celle que vous attendiez ? Sinon, quelle est votre erreur ?

3 http://hopl.murdoch.edu.au/
4 cf. LISP

- La première réponse n'est peut-être pas, pour des raisons syntaxiques, celle que vous attendiez : elle s'explique par la priorité des opérateurs, le *produit* étant évalué avant la *somme* ; il est toujours possible de parenthéser une expression pour forcer l'évaluation voulue ; ainsi : `(2 + 3) * 4` devrait retourner `20`…
- La deuxième réponse met en évidence un aspect purement sémantique : les opérateurs se comportent différemment selon que leurs opérandes sont des nombres ou des *chaînes de caractères*, c'est-à-dire des mots.
- La troisième montre qu'une séquence (ou *chaîne*) de caractères peut être aussi bien délimitée par des guillemets simples que par des doubles : en effet, il ne serait pas possible d'exprimer la chaîne 'd'hui' parce que le deuxième guillemet serait considéré comme fermant le premier – et le reste n'aurait bien sûr aucun sens.
- La quatrième est une erreur lexicale : remarquez que jusque-là, nous n'avions utilisé que des *constantes littérales*, autrement dit des symboles dont la valeur est donnée littéralement ; ils ont leur propre valeur (on les prend *au pied de la lettre*) et ils s'évaluent à eux-mêmes… Une chaîne est un littéral, un nombre est un littéral. Mais dans ce contexte, `a` et `b` sont des *symboles* inconnus de l'interprète ; et utiliser des symboles non définis au préalable, ça n'a pas de sens : l'interprète *ne sait pas* évaluer une telle expression.

Remarque : l'interprète PYTHON intègre en standard un éditeur de texte rudimentaire ; sans rentrer dans les détails, sachez que vous pouvez retrouver la ligne que vous venez d'entrer en pressant la touche curseur-haut ; vous avez alors le loisir de la modifier avant de la renvoyer.

SYMBOLES ET ÉVALUATION

La définition de nouveaux symboles est une opération fondamentale pour le programmeur : elle lui permet de créer des abstractions, en l'occurrence des objets qui en représentent d'autres. Essayez :

```
a = 2
b = 3
a + b
```

Un tel objet est appelé *variable* (sa valeur peut varier) et sert à mémoriser un certain état, pendant le déroulement du programme ; si maintenant j'essaye d'évaluer `a + b`, l'interprète évalue d'abord `a` pour découvrir qu'il représente la quantité `2`, puis fait de même pour `b`, avant d'appliquer l'opérateur `+` et d'obtenir la quantité `5`, représentée par ce symbole dans la réponse.

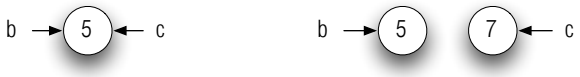
Notez que le résultat de l'évaluation `a + b` n'est pas mémorisé : il est calculé, affiché, puis aussitôt oublié. Il est fréquent d'évaluer des expressions juste pour voir ce que ça donne, et c'est le fondement de l'approche exploratoire dans l'apprentissage d'un langage. En fait, j'aurais pu utiliser un nouveau symbole pour mémoriser, donc représenter le résultat de cette évaluation, autrement dit, de ce calcul :

```
c = a + b
```

Gardons désormais à l'esprit que les variables ne sont que des *représentants* : des symboles qui représentent quelque chose d'autre – souvent une valeur littérale, mais pas forcément (nous y reviendrons). Comprenez bien que lorsque PYTHON interprète l'expression

```
b = c
```

il évalue d'abord `c`, trouve sa valeur actuelle, et change alors la valeur de `b` (l'ancienne valeur est oubliée : si on avait voulu la garder, il aurait fallu la mémoriser), de sorte que `b` représente maintenant la même valeur que `c` ; et si `c` venait à changer de valeur, `b` n'en saurait rien, et continuerait imperturbablement à représenter ce qu'on lui a dit de représenter. Ce qu'on pourrait illustrer ainsi :



Remarque 1 : notez que, dans l'expression `c = a + b`, la *valeur* de la variable `c` dépend de celles de `a` et de `b` au moment de leur évaluation ; essayez maintenant :


```
b = c                                   # où c vaut déjà 5
c = a + b
```

qui « attribue » la nouvelle valeur 7 à la variable c ; il faut comprendre ici que le symbole = représente un opérateur, et qu'il y a, là aussi, des règles syntaxiques ; l'expression située à droite est évaluée en premier, ce qui permet d'écrire des choses comme :

```
c = a + b + c
```

de sorte que mon c vaut maintenant 14, c'est-à-dire la somme de a et b, plus la valeur qu'il avait *avant* la modification.

```
>>> a = 2
>>> b = 3
>>> a + b
5
>>> c = a + b
>>> c
5
>>> b = c
>>> c = a + b
>>> c
7
>>> c = a + b + c
>>> c
14
```

Remarque 2 : la définition d'une variable par l'évaluation d'une expression telle que c = a + b ne retourne pas de valeur affichable dans le terminal, mais, comme illustré ci-dessus, il est toujours possible d'évaluer directement un objet pour lui faire cracher sa valeur courante.

DOCUMENTATION D'UN PROGRAMME

Comme on l'a vu ci-dessus, même un programme simplet peut être difficile à suivre pour un humain, qui ne procède pas de façon aussi systématique et rigoureuse que la machine. Pour cette raison, tout langage de programmation permet de documenter du code en y ajoutant des commentaires, qui ne seront *jamais* évalués. Dans la plupart des langages de script, bash et tcsh compris, un commentaire commence par le symbole # (en dehors d'une chaîne, évidemment) et se termine à la fin de la ligne :

```
(2 + 3) * 4                             # pas pareil que 2 + (3 * 4)
```

Nous, on comprend très bien tout ce qui suit le #, mais pour l'interprète, ce serait *totalement* inintelligible… puisque les symboles pas, pareil et que ne sont pas définis en PYTHON !

Maintenant, essayez ceci :

```
7 / 2                                   # division euclidienne
```

Comprenez- vous pourquoi le résultat est 3 ? La division entière (ou euclidienne) retourne une valeur entière… Autrement dit, le résultat est arrondi à la valeur entière immédiatement *inférieure* au résultat réel ; essayez :

```
7 / -2                                  # résultat arrondi à l'entier inférieur
```

Ah ! Qui aurait pu prédire que cette expression s'évaluerait à -4 ? Si ce résultat vous paraît choquant, tant mieux : souvenez-vous que les machines ne réfléchissent *jamais*, elle se contentent d'exécuter du code. Apprenez simplement à les utiliser à votre profit. Essayez par exemple :

```
7 / 2.                                  # résultat réel : ici, le diviseur est un réel, et non pas un entier
```

La valeur retournée maintenant, 3.5, correspond mieux à notre intuition ; on aurait d'ailleurs obtenu le même résultat avec l'expression 7. / 2 parce qu'il suffit que *l'un* des opérandes soit réel pour que l'opérateur change de comportement : une petite différence syntaxique, en l'occurrence un point, provoque une différence sémantique majeure – remarquons ici que 7. ou 7.0 sont équivalents : c'est toujours la quantité 7 considérée comme un réel et non un entier.

Enfin, essayez de prédire le résultat de l'expression :

```
7. / -2
```

Si vous aviez prédit -3.5, c'est parfait : en fait, vous êtes en train de pratiquer la méthode *incrémentale* évoquée plus tôt ; le programmeur expérimenté a fait tellement d'expériences qu'il est à même d'anticiper, de prédire le résultat de son code ; il acquiert une conscience aiguë du fonctionnement du langage et l'intègre à sa propre façon de penser. Une fois que tout ceci est devenu inconscient, il devient rapide et efficace : en fait, c'est horrible à dire, mais il se met à penser comme la machine.

EXERCICES

Voici les premiers exercices à programmer par vous-même : il est impératif de maîtriser tout ce qui précède par la pratique, parce que, même si on a bien compris l'aspect théorique, c'est l'émotion que provoque l'erreur qui va laisser l'empreinte la plus profonde, comme un enfant qui ne sait marcher que parce qu'il est tombé — et comme disent nos amis allemands, c'est la *pratique* qui fait le *maître*.

[px01-1] définir deux variables, valant respectivement votre année de naissance et l'année en cours, et programmer le calcul de votre âge en soustrayant l'une de l'autre.

[px01-2] définir les variables suivantes :

```
espace = " "                                    # attention : espace entre guillemets doubles
apostrophe = "'"                                # attention : apostrophe entre guillemets doubles
mot1 = "au"
mot2 = "jour"
mot3 = 'd'                                      # les guillemets simples marchent pareil
mot4 = mot1 + mot2 + mot3 + apostrophe + 'hui'
```

Vérifiez, en évaluant chaque variable, qu'elles ont bien la valeur attendue, puis coder l'expression qui permettra, à partir de ces variables, d'obtenir la nouvelle chaîne « *au jour d'aujourd'hui* ».

Remarque : en principe, le nom des variables ne fait rien à l'affaire : on aurait aussi bien pu les appeler *truc*, *machin*, et *chose* ; mais j'ai préféré des noms *auto-documentaires* : le nom dit quelque chose de l'utilisation de la variable. Et ça peut aider quand on lit le code. Mais c'est vrai que pour l'interprète, les définitions suivantes :

```
p = 'pomme' ; d = 'de' ; t = 'terre'                # 3 définitions distinctes
```

permettent ensuite d'évaluer l'expression, parfaitement valide : `p + d + t`

[px01-3] essayez ce programme, et évaluez cette dernière expression... Oh ! Avez-vous, comme moi, oublié d'espacer les mots ? Ou d'insérer des tirets ? À l'évidence, moins le code est lisible, plus on fait d'erreurs : et du code, ça n'est *jamais vraiment* lisible. Nous y reviendrons...

[px01-4] définir les variables `trait` et `point`, et les utiliser pour coder en morse le message "SOS", où "S" est représenté par trois points, et "O" par trois traits : avez-vous spontanément défini ces variables avec les valeurs respectives "." et "-" ? Ou l'inverse ? Qu'en concluez-vous ?

NOTES

On prendra l'habitude, à la fin de chaque section, de se donner le temps de récapituler, d'organiser ce qu'on vient de comprendre sous une forme structurée : ces notes pourront alors servir de référence pour retrouver très vite une définition, ou un détail particulier. Notre « productivité » est à ce prix…

Ces notes sont également l'occasion de prendre conscience de ce qu'on a pas assez bien compris pour le reformuler simplement. Le forum est alors l'endroit idéal pour poser une question : les réponses multiples y ont souvent des points de vue multiples, et sont donc autant d'éclairages différents sur un point obscur.

Les objets qu'on peut manipuler en PYTHON :

- valeurs littérales
  - des mots, des phrases (séquences de caractères)
  - des nombres (suites de symboles numériques)
    - nombres entiers
    - nombres réels
- variables
  - une variable est un symbole représentant une valeur
  - la valeur peut être un des types de données mentionnés ci-dessus
- symboles d'opération (cf. ci-dessous)

NB : les valeurs littérales sont aussi appelées constantes, par opposition à variable. On appelle chaîne de caractères un littéral constitué de caractères entre guillemets, techniquement appelés quotes : PYTHON admet aussi bien les guillemets simples (single quotes), les guillemets doubles (double quotes) et les guillemets triples (triple quotes) ; ces derniers permettent, par exemple, d'exprimer une chaîne de caractère contenant elle-même des guillemets simples (apostrophes) ou doubles : '''elle m'a dit "salut", c'est tout !'''.

Les manipulations qu'on peut faire en PYTHON :

- construire une expression avec des opérateurs et leurs opérandes
  - opérateurs sur les chaînes : +, *
  - opérateurs arithmétiques : +, *, /
  - opérandes : des constantes littérales ou des variables
- évaluer une telle expression pour calculer une nouvelle valeur
- définir un symbole pour représenter une valeur (littérale ou calculée)

Une expression produit une valeur quand elle est évaluée ; le simple fait d'entrer une expression sous l'interprète suffit à l'évaluer :

`"oh !"` évalue à `'oh !'`
`123` évalue à `123`
`3 * "-"` évalue à `'---'`

Une instruction a un effet sur l'environnement, mais ne produit pas de valeur :

```
x = 1
```

Cet effet est invisible, et c'est ce qui fait la difficulté de la programmation ; mais on peut toujours le mettre en évidence en examinant immédiatement l'environnement :

```
>>> x = 2 + x ; x
3
```

On peut mettre plusieurs expressions ou instructions sur la même ligne en les séparant par des points virgule ; ça n'est pas toujours facile à relire, mais des fois, c'est plus commode à manipuler :

```
>>> x = 2 + x ; x
5
>>> x = 2 + x ; x
7
```

On aurait pu dire la même chose avec un opérateur d'affectation cumulatif :

```
x += 2
```

qui va chercher la valeur de `x`, lui ajoute 2, et remplace l'ancienne valeur par la nouvelle.

1.2 APPLICATION DE FONCTIONS

L'interprète permet l'application de *fonctions* ; une fonction est un nom symbolique qui représente un fragment de code indépendant, supposé accomplir une fonction particulière ; par exemple, la fonction len() appliquée à une séquence de caractères calcule le nombre d'éléments de cette séquence :

```
len('azertyuiop')                                    # retourne la valeur 10
```

On appelle *argument* la valeur qu'on passe à une fonction, autrement dit, ce à quoi on veut l'appliquer : ici, la valeur 'azertyuiop' est donnée littéralement, mais on aurait pu mettre, à la place, une variable représentant cette même valeur…

L'argument de la fonction len() ne peut pas être un nombre, simplement parce que ça n'a pas de sens : la représentation interne d'un nombre n'est pas une séquence de symboles. Mais la fonction str() convertit un nombre en string, donc en séquence de caractères ; ainsi, essayez d'évaluer len(str(12345)) et vous obtiendrez sans doute 5.

Nous verrons encore plein d'autres fonctions, mais l'objectif n'est pas de faire un catalogue des ressources intrinsèques de PYTHON – il existe déjà des manuels de référence très bien faits…

TYPES DE VALEURS

D'une manière générale, les langages de programmation détectent le type des valeurs qu'ils manipulent ; essayez ceci :

```
type(123)
type('123')
type(u'123')
type(1.23)
type(compile)
type(())
type({})
type([])
```

Comme vous le constatez, pour chacun de ces exemples, la fonction type() retourne une valeur différente, qui dépend de la valeur de son argument.

remarque pratique : chaque fois que possible, copiez le texte du programme proposé, et collez-le dans le terminal ; comme ça, s'il y a des erreurs ce sera de ma faute, et non de la vôtre… Mais attention, FIREFOX a un bug avec le cut+paste et le drag+drop ! Utilisez KONQUEROR.

TYPE DES VARIABLES

Les variables, elles, ne sont *jamais* typées ; si on a défini, par exemple, z = 123, appliquer la fonction type() à z retournera le type de la *valeur* de z – pour vous en convaincre, essayez :

```
z = 123 ; type(z)
z = '123' ; type(z)
```

Maintenant, si je demande à évaluer *la valeur* de z – donc double évaluation, ce que j'obtiens n'est *plus* du type str mais du type int ; essayez :

```
w = eval(z)                                          # → 123
type(w)                                              # → <type 'int'>
type(eval(z))                                        # → <type 'int'>
```

Sachant cela, il peut être utile de définir une variable de sorte qu'elle *représente* une expression évaluable ; on en verra l'intérêt dans la section suivante où un programme lit une expression au clavier, donc forcément sous forme de texte, et doit ensuite en calculer la valeur ; par exemple :

```
z = '1 + 2 + 3'
```

Ici, le type de la valeur de z est, bien sûr, str, mais si on l'évalue :

```
eval(z)
```

on obtient la valeur 6 ; parce qu'en pratique, le mécanisme d'évaluation (détaillé ci-dessous) ne fait pas de différence formelle entre évaluer '123' et évaluer '1 + 2 + 3'.

Notez bien qu'on a ici affaire à une *double* évaluation :

- la 1ère est implicite ; il s'agit de retrouver la valeur de z : '1 + 2 + 3'
- la 2ème est due à l'application explicite de eval() et retourne la valeur 6

1.3 MÉCANIQUE DE L'ÉVALUATION

La raison en est que la fonction eval() est implicitement utilisée par l'interprète chaque fois que vous entrez une expression ; autrement dit, l'évaluation est un mécanisme implicite ; il obéit à quelques règles simples :

- évaluer un *littéral* retourne ce littéral comme valeur ;
- évaluer une *variable* retourne la *valeur* par laquelle elle a été définie ;
- évaluer une *définition de variable* consiste à évaluer l'expression qui la définit, et lier cette valeur à la variable (la variable elle-même n'est pas évaluée, puisqu'elle est sur le point d'être définie) ; cette opération ne retourne pas de valeur ;
- évaluer une *expression* consiste à évaluer chaque constituant de gauche à droite :
  - si c'est une variable ou un littéral, le mécanisme est décrit ci-dessus ;
  - si c'est un opérateur, ses opérandes sont évalués de gauche à droite avant de leur appliquer l'opération que *représente* cet opérateur – le moins unaire s'applique immédiatement à son opérande : -10.

> Les opérateurs binaires se présentent la plupart du temps sous forme *infixée*, c'est-à-dire qu'ils attendent un premier opérande à gauche et un second à droite. Les fonctions, de la forme f(x), sont aussi des opérateurs, mais sous forme *préfixée* ; dans ce cas, leurs opérandes sont appelés *arguments* de la fonction, et leur nombre caractérise l'*arité* de la fonction – l'arité d'une fonction dépend de la façon dont elle a été codée : il n'y a pas de limite théorique au nombre d'arguments.

Ce mécanisme d'évaluation, similaire à celui de LISP, permet des facilités « impossibles » pour d'autres langages. Par exemple :

```
taille = len                    # définition d'un synonyme
taille('azerty')                # taille() se comporte exactement comme len()
```

retournera la valeur 6…

Ou encore :

```
x = '1 + 2 + 3 + 4'
z ='x'
eval(eval(z))
```

produira la valeur 10…

L'utilité de cette fonctionnalité deviendra évident lorsque, plus tard, nous manipulerons les mots de la ligne de commande, où tout est du texte, bien qu'il n'y ait de guillemets nulle part.

En tant que processeur de commandes, le SHELL ne voit que du texte, qu'il passe au programme appelé : et c'est ce programme qui est responsable de la conversion de la séquence « 123 » en la quantité 123…

D'ailleurs, grâce à la fonction int(), ça ne devrait présenter aucune difficulté – essayez :

```
>>> z = '123'
>>> 3 * int(z)
369
```

NOTES

Comme l'annonce PYTHON aussitôt qu'il est activé, on peut demander de l'aide à tout moment, en évaluant la fonction help() ; par exemple :

```
>>> help(len)
len(...)
    len(object) → integer
   Return the number of items of a sequence or mapping
```

Ceci nous dit précisément :

- la syntaxe de l'utilisation de len()
- le type de la valeur retournée : INTEGER signifie « entier »
- une brève description de son comportement
- à quels types d'objets on peut l'appliquer

NB : les types SEQUENCE et MAPPING seront étudiés dans les prochains chapitres...

Une autre façon de retrouver comment une fonction s'utilise est de garder un marque-page sur la documentation qui décrit toutes les fonctions intrinsèques, en ordre alphabétique :

http://docs.python.org/library/functions.html

RÉCAPITULATION

Le thème implicite de ce chapitre est la notion de *valeur*, notion centrale dans le traitement symbolique de l'information :

- l'information *est* la valeur
- on peut la REPRÉSENTER par un symbole – la variable ;
- on dit alors que ce symbole est une RÉFÉRENCE à cette valeur ;
- une valeur peut aussi être exprimée « virtuellement » comme le résultat d'un calcul :
  - valeur d'une expression symbolique
  - valeur d'une fonction appliquée à des arguments

> Comme montré au chapitre 0, certaines expressions sont utilisées pour leur effet, et n'ont pas de valeur ; si l'effet est externe, comme pour l'affichage, la valeur n'a d'ailleurs guère d'importance ; s'il est interne, comme dans la définition d'une fonction, on ne verra rien… puisqu'il est interne !

Une valeur peut être associée à plusieurs variables, mais une variable ne peut représenter qu'une valeur ; ce serait une erreur de penser qu'une variable est un conteneur et que son contenu est la valeur : une variable n'est qu'un symbole accroché à une valeur, un nom qui étiquette un objet…

Une valeur dépend forcément d'un contexte :

- la valeur d'une variable dépend du contexte où elle a été définie (ou redéfinie) ; en particulier si sa définition dépend de valeurs elles-mêmes dépendantes du contexte ;
- on a vu (page 22) un exemple où la valeur d'une variable `c` dépendait de celle d'autres variables `a` et `b`, et où une même expression `a + b` retournait une valeur différente selon le résultat de l'évaluation de `a` et de `b` ;
- la valeur d'une fonction dépend aussi d'un contexte, celui qu'on lui passe au moment où on l'appelle : autrement dit, ses arguments…

La valeur de `f(x)` dépend de la valeur de `x` :

```
x = 'qsdfghjklm' ; len(x)                    # → 10
x = 'wxcvbn' ; len(x)                        # → 6
```

La représentation d'une valeur par une expression symbolique est une abstraction :

- une variable est l'abstraction d'une valeur ;
- une fonction est l'abstraction d'un calcul ou d'un traitement ;
- une valeur représentant une expression symbolique peut être évaluée ;
- l'abstraction facilite les manipulations par programme ;

Programmer, c'est exprimer des transformations – autrement dit, construire des expressions qui réalisent une extraction d'information, par filtrage, ou recombinent des fragments pour construire une nouvelle information, par composition :

- composition : opération fondamentale qui consiste à assembler des bouts pour faire un tout ; les exercices de la série px01 ont tous une solution de ce type
- filtrage : autre opération fondamentale, introduite au chapitre 2

Autrement dit, tout est forme, et tout peut être transformé : d'où le titre donné à ce cours…

② MANIPULATION DE SÉQUENCES

Ce chapitre manipule essentiellement des séquences de caractères, aussi appelées chaînes. Au passage, la section 2 en profite pour présenter la définition de fonction. C'est également un prétexte (section 3) pour introduire notre 3e structure de contrôle, l'évaluation conditionnelle, les deux premières (la séquence d'instructions et l'appel de fonction) ayant déjà été introduites de façon totalement implicite.

Les séquences ne se laissent pas réduire aux chaînes de caractères : nous verrons plus tard que beaucoup d'opérations applicables aux chaînes peuvent aussi être appliquées aux listes, aux tables d'associations (dictionnaires) et aux ensembles. En pratique, du point de vue des techniques de programmation, la démarche est la même pour tout type de séquence.

La section 4 présentera, sous la forme de travaux pratiques, une application des techniques maîtrisées en section 3, et introduira une quatrième structure de contrôle, l'itération.

En section 5, une brève incursion dans le monde de la programmation objet permettra d'apprivoiser la notion de méthode spécifique d'une classe d'objets, ce qui ne présente pas grande difficulté pour quiconque a déjà admis le principe qu'un même opérateur pouvait avoir un comportement différent selon le type de ses opérandes.

sommaire

Une séquence est l'organisation de données sous la forme d'éléments distincts collectés en une seule structure à laquelle on peut donner un nom symbolique : cette organisation peut être implicitement ordonnée, ce qui se reflète alors dans sa représentation interne (en anglais, ARRAY), auquel cas on peut y accéder séquentiellement, un élément après l'autre ; c'est le cas des *chaînes* de caractères ou des *listes* ; pour d'autres représentations, l'ordre n'a aucune importance parce que l'accès aux données est indépendant de leur position dans la structure, comme c'est le cas, par exemple, pour les *dictionnaires* ou les *ensembles*.

Quand il s'agit de représenter les données en vue de traiter l'information, beaucoup de langages favorisent la structuration en séquence, pour trois raisons :

- une séquence peut être vide, façon évidente de représenter une information manquante
- un élément de la séquence peut lui-même être une séquence, ce qui permet de structurer l'information à plusieurs niveaux de profondeur
- un programme [bien] conçu pour traiter un élément de la séquence peut facilement être généralisé pour traiter tout élément de cette séquence

2.1 CHAÎNES DE CARACTÈRES ET INDEX

Une chaîne de caractère est une *séquence* : ce qui veut dire que chaque caractère occupe dans la chaîne une position (séquentielle) distincte – techniquement, on préfère parler d'index, mais ça revient [presque] au même. Il est possible d'accéder aux éléments d'une séquence en disant précisément ce qu'on veut en extraire.

Prenons par exemple, la chaîne 'petit chat'. Je peux demander le premier élément (premier caractère), de cette séquence en disant :

```
'petit chat' [0]                                    # élément 0 → 'p'
```

Et bien sûr, ça marcherait tout pareil avec des variables :

```
mots = 'petit chat' ; mots[0]                       # élément 0 → 'p'
```

L'opérateur [] est appelé *opérateur d'indexation*. Il est obligatoirement utilisé sous forme postfixée, c'est-à-dire *après* l'expression dont on veut extraire une information. De la même façon, mots[1] produit la valeur 'e', alors que mots[2] me donnerait 't'.

Remarquons qu'en PYTHON, comme d'ailleurs dans beaucoup de langages de programmation, les *index* commencent à 0 (on dit aussi qu'ils sont ZERO-BASED). Et *là* est la petite différence entre *index* et *position* : la première position est à l'index 0, la deuxième à l'index 1, et ainsi de suite…

Il est possible d'extraire des segments (SLICES) de séquence, en indiquant l'intervalle souhaité, c'est-à-dire l'index de *départ* et l'index de *fin* de segment. Si je veux les 5 premiers éléments de la séquence, je peux dire :

```
'petit chat'[0:5]                                   # éléments 0, 1, 2, 3 et 4
```

Ce qui est bizarre ici, c'est que si je demandais 'petit chat'[5], j'obtiendrais l'espace, c'est-à-dire le 6ème élément : j'en déduis que la borne droite de l'intervalle n'est pas incluse dans l'intervalle, autrement dit, qu'il faut donc spécifier le segment en ajoutant 1 au véritable index de fin de segment (ce n'est pas une incohérence, mais c'est pas facile à expliquer).

PYTHON permet de ne spécifier que l'une des deux bornes de l'intervalle : dans ce cas, la borne non spécifiée prend comme valeur par défaut l'index de début ou de fin, c'est selon…

```
'petit chat'[:5]                                    # segment du début jusqu'à l'index 5 non-compris
'petit chat'[6:]                                    # segment depuis l'index 6 jusqu'à la fin
'petit chat'[:]                                     # segment du début jusqu'à la fin
```

Encore plus fort : PYTHON permet d'utiliser des index négatifs ; dans ce cas, l'index part de la fin de la séquence (en fait, l'interprète fait la somme *index + taille de la séquence*) :

```
'petit chat'[-1]                                  # (comme 10 -1) le caractère 't' : dernier élément de la séquence
'petit chat'[-4]                                  # (comme 10 -4) l'élément 'c', en 6e position à partir de la fin
'petit chat'[-4:]                                 # 'chat' : depuis l'élément 'c' jusqu'à la fin
'petit chat'[:-5]                                 # 'petit' : la séquence sans ses 5 derniers éléments
```

Enfin, on peut spécifier une 3e valeur, appelée *stride* (français, *enjambée*) ; par défaut c'est 1, mais on peut mettre n'importe quelle valeur (sans risque d'erreur), même négative :

```
>>> 'petit chat'[::6]
'pc'
>>> 'petit chat'[::-6]                            # extraction de 6 en 6, en partant de la fin
'ti'
```

[px02-1] soit mot = 'cheval' ; exprimer le pluriel de ce mot en remplaçant le 'l' final par 'ux'.

[px02-2] soit z = 'abracadabra' ; définir a et c à partir de z, tel que a ait comme valeur 'abra' et c ait comme valeur 'cad' ; coder l'expression qui permettrait de reconstituer la valeur de z en utilisant uniquement a et c ; sans a ni c, construire 'cadabracad' à partir de z.

[px02-3] en utilisant uniquement a et c, coder l'expression qui construit 'abracadabracadabracadabra'.

[px02-4] pareil, mais sans a ni c : uniquement à partir de z.

[px02-5] soit p = 'pommedeterre' ; coder en *une seule* expression l'insertion des traits d'union là où ils sont nécessaires ; peut-on effectuer l'opération p[7] = '-' ? Qu'en conclure ?

[px02-6] soit x = 'Agence France Presse' ; coder l'expression du sigle 'AFP' à partir de x ; redéfinir x avec comme valeur, 'Pari Mutuel Urbain' et modifier l'expression précédente pour obtenir 'PMU' ; comparez les deux expressions : ne pourrait-on pas obtenir ces deux résultats avec une seule même expression ? Quelle serait cette expression ?

2.2 DÉFINITION DE FONCTION

On appelle *fonction* un symbole qui fait référence à un calcul programmé en fonction d'un certain nombre de paramètres – ce nombre peut d'ailleurs être nul. Une fonction s'utilise en l'appelant par son nom et en mettant entre parenthèses l'argument (ou les arguments) qu'elle attend ; essayez, par exemple :

```
len('petit chat')                                 # calcul de la longueur d'une séquence
```

Rappel : la fonction len() prend une séquence en argument et en retourne le nombre d'éléments. Ainsi, dans le cas ci-dessus, elle compte combien il y a de caractères dans la chaîne passée en argument, et devrait donc retourner la valeur 10.

On définit une fonction en spécifiant la variable qui représentera l'argument, puis en calculant une valeur à partir de cette variable, pour enfin retourner cette valeur, grâce au mot-clé return :

```
def double(truc) : return truc * 2
```

> Attention : une définition de fonction doit être suivie d'une *ligne vide* qui fait comprendre à l'interprète que la définition est terminée ; en effet, au premier alinéa après la définition, vous verrez apparaître un … de continuation : pressez une seconde fois la touche ↵ ; cf. section *9.2*.

Ici, la variable truc, représente l'argument (c'est-à-dire, la valeur) qui sera véritablement utilisé lors de l'appel de la fonction double() :

```
double(11)                                        # → 22, puisque truc vaut 11
double('do')                                      # → 'dodo', car truc vaut 'do'
```

La variable truc est appelée *paramètre formel de la fonction* (*formel* parce que ce n'est qu'une forme vide qui ne prendra de valeur que lorsque la fonction sera activée) ; en d'autres termes, le paramètre, c'est l'argument, mais vu de l'intérieur de la fonction ; alors que l'argument, c'est la valeur que vaudra le paramètre lorsque la fonction sera activée.

On parle de l'*arité* d'une fonction pour dire le nombre d'arguments qu'elle accepte : une fonction *unaire* prend exactement un argument, et ce serait une faute de l'appeler avec deux arguments, ou sans argument du tout ; idem pour les fonctions binaires, ternaires…

- l'argument, c'est la *valeur* avec laquelle la fonction est appelée
- le paramètre, c'est la *variable* qui représente cette *valeur* dans la définition

Petite précision : la variable qui nomme le paramètre est dite *locale* à la fonction ; ça veut dire qu'une définition de fonction définit en même temps un nouveau contexte propre à la fonction, et invisible de l'extérieur d'icelle.

En conséquence, vous pourriez très bien avoir déjà défini une variable truc, et pourtant, l'appel de double() ne modifiera pas votre truc à vous, qui se trouve dans un autre *espace de noms* que celui de la fonction. Autrement dit, peu importe le nom du paramètre, il ne peut pas créer de conflits avec d'autres symboles… Essayez :

```
>>> truc = 100
>>> double('pa')
'papa'
>>> truc
100
```

Récapitulation : formellement, une définition commence avec l'opérateur def suivi du nom de la fonction et d'une liste de paramètres entre parenthèses, puis un ":", et enfin une séquence d'instructions qui définit les calculs effectués par la fonction, terminée par une instruction return suivie de l'expression de la valeur à retourner, s'il y a lieu.

Essayons de définir interactivement la fonction triple(), qui accepte un argument quelconque et retourne 3 fois la valeur qu'on lui passe ; n'oubliez pas le mot-clé return :

```
>>> def triple(x) : return x + x + x                    # pareil que 3 * x
...
>>> 'et tra' + triple(double('la'))
'et tralalalalalala'
```

Comme le montre cet exemple, les fonctions qu'on se définit sont utilisables dans les mêmes conditions que les fonctions primitives déjà disponibles en PYTHON :

- dans des expressions complexes, comme 'eli' + str(112), ou encore 'f'+ triple('*') + ' you'
- on peut en définir un synonyme : duo = double
- la valeur d'une fonction peut être l'argument d'une fonction, comme pour len(str('12345'))

[px03-1] définissez la fonction unaire cube() qui permet d'élever au cube un nombre passé en argument, de telle sorte que cube(3) retourne 27, cube(4) retourne 64, et cube(100) retourne 1000000 ; utilisez-la pour vérifier le cube de 2 et le cube de 5.

[px03-2] définir la fonction grogne() d'arité 0, qui retourne la chaîne 'grrrrr'.

[px03-3] définir la fonction unaire respire() qui prend une chaîne en argument, et retourne cette même chaîne après avoir ajouté un espace avant et un espace après.

[px03-4] en utilisant la fonction double(), définir la fonction unaire quadruple() qui multiplie son argument par quatre ; la tester avec des nombres et des chaînes.

RÉCAPITULATIF

comme pour les nombres, il n'est pas possible de modifier un élément d'une chaîne, mais on peut toujours en fabriquer une autre en recopiant uniquement l'information qui nous intéresse...

accès aux éléments d'une séquence

l'opérateur `[ ]`, obligatoirement postfixé, permet l'extraction d'information
- il attend l'expression d'un index `i`, possiblement négatif : `séquence[i]`
  - index positif valide : part de `0` et croît jusqu'à `len(séquence) - 1`
  - index négatif valide : part de `-1` et décroît jusqu'à `-len(séquence)`
- l'index peut être complexe, de la forme `i : j`
  - il spécifie alors un intervalle : `séquence[début : fin]`
  - les bornes peuvent être omises : elles sont alors prises par défaut
  - début en `0` : `séquence[: fin]`
  - fin en `len(séquence) - 1` : `séquence[début :]`
    - extrait les 2 derniers éléments : `'wxcvbn'[-2 :]`
    - copie la séquence entière : `'wxcvbn'[:]`
- on peut encore spécifier une 3e valeur : l'enjambée
  - exemple qui copie la séquence de 2 en 2 : `'wxcvbn'[:: 2]`
  - exemple qui copie la séquence en partant de la fin : `'wxcvbn'[:: -1]`

référence : http://docs.python.org/reference/expressions.html#id7

définition de fonction

l'instruction `def` ne retourne rien :
- elle est utilisée pour son effet
- comme l'opérateur `=`, elle modifie des représentations internes

l'instruction `def` attend :
- un symbole : le nom de la fonction
- une liste de paramètres
  - encadrée de parenthèses
  - possiblement vide
- un ':' précédant la séquence d'instructions qui définit le calcul à effectuer
- le mot-clé `return` suivi de la valeur à retourner

une ligne vide termine la définition

le paramètre représente la valeur de l'argument
son nom est local à la fonction : pas de confusion possible

lors de l'appel, il doit y avoir autant d'arguments que de paramètres en définition

référence : http://docs.python.org/reference/compound_stmts.html#function-definitions

2.3 DISTINGUER LE VRAI DU FAUX

évaluation d'expressions logiques

Toute expression, quand elle considérée d'un point de vue logique, a une valeur dans l'ensemble {*vrai, faux*}, et cette valeur est exclusive : si ce n'est pas vrai, c'est que c'est faux, et d'ailleurs, tout ce qui n'est pas faux est forcément vrai.

Il n'y pas ici de place pour l'incertitude : il s'agit d'une logique booléenne. Par abus de langage, on parlera d'*expression booléenne* pour dire que le seul point de vue qui nous intéresse est sa valeur logique – ou valeur de vérité.

Et ce type d'expression est souvent appelé aussi *prédicat* : ça dépend des langages, mais aussi des programmeurs – ou plutôt de l'historique de leur formation… Ainsi, on parle volontiers de prédicats en LISP, PYTHON ou PROLOG, mais rarement en ANSI-C ou en PERL.

Comme LISP avec equal et eq, PYTHON dispose d'un prédicat d'*égalité*, l'opérateur ==, et d'un prédicat d'*identité*, le verbe is, qui s'utilisent comme suit :

```
111 == 10 ** 2 + 10 ** 1 + 10 ** 0          # bon sang, mais c'est bien sûr !
'black' is not 'white'                      # ah, oui...
1 + 1 is 2                                  # évalue à True, c'est-à-dire vrai
'z' in 'yamaha'                             # False, c'est-à-dire faux
```

On dispose également d'opérateurs logiques, qui permettent de construire des prédicats complexes, ou expressions booléennes : la conjonction (and) et la disjonction (or), qui sont *binaires* (opèrent sur 2 opérandes), et la négation (not), qui est unaire :

x or y	si x évalue à faux, retourne eval(y) ; sinon retourne eval(x)
x and y	si x évalue à faux, retourne eval(x) ; sinon retourne eval(y)
not x	si x évalue à faux, retourne True ; sinon retourne False

[px04-1] calculez la valeur de vérité des *prédicats littéraux* suivants, en vous assurant que vous comprenez parfaitement la réponse de l'interprète – au besoin, utilisez la fonction bool(), évaluez d'abord les sous-expressions, puis l'expression entière ; commentez ces résultats :

```
bool('')                                    # bool() convertit son argument en valeur booléenne
1 + 2 is 3 and 3 is 4 - 1
not 0
'azertyuiop' > 'wxcvbn'
'œ' in 'œil de bœuf'                        # l'élément fait-il partie de l'ensemble ?
1 + 1 != 3                                  # != est l'inverse de ==, comme 'is not'
not (1 is 3 and 1 is 2)
not 3 is not 2                              # scope (portée) de la négation
(not 3) is (not 2)
'xy'[0] is 'z' or 'z'
1 is 1 and 2 is 2 or 3 is not 3
-1 * -1 > 0
3 > (5 < 4)
```

Pour compléter ce panorama, PYTHON dispose d'instructions conditionnelles, c'est-à-dire d'instructions qui ne sont évaluées que si un prédicat (une condition logique) a été auparavant vérifié ; ce sont des instructions *complexes*, qui peuvent éventuellement prendre plusieurs lignes, autrement dit un bloc de code {cf. 112:9.2, indentation}.

Exemple idiot :

```
if 5 > 4 : feu = 'rouge'                    # toujours évalué
```

Ici, tout ce qui est compris entre le mot-clé if et le ':' est considéré comme un prédicat logique qui contraint l'évaluation de l'instruction qui suit ; comme 5 est, par définition, plus grand que 4, l'instruction suivante, feu = 'rouge', sera forcément évaluée.

L'instruction if peut être assortie de son complémentaire else, instruction qui ne sera évaluée que dans le cas où le prédicat du if n'était pas vrai :

```
if feu is 'vert' : passe = True
else : passe = False
```

Notez que l'expression du else doit venir *immédiatement* après celle du if dans la séquence.

Le prédicat qui contrôle l'évaluation peut être aussi complexe que nécessaire, et faire intervenir les opérateurs logiques vus plus haut :

```
if passe and feu is 'rouge' : passe = not passe
```

Cette instruction doit examiner la valeur de la variable passe et celle de la variable feu (passe est ici utilisée comme une expression booléenne, pour sa valeur de vérité) et la condition ne sera vérifiée que si la *conjonction* des deux sous-prédicats évaluent à True.

Une instruction comme passe = not passe a un effet de bascule, ou TOGGLE : si passe est vrai et que le feu est au rouge, passe devient faux (la négation bascule la valeur de vérité). Cette même instruction aura l'effet contraire (faux → vrai) si on évalue la ligne ci-dessous :

```
if not passe and feu is 'vert' : passe = not passe
```

Remarquez que si passe était vrai au moment de son évaluation en tant que prédicat, la conjonction not passe and feu is 'vert' aurait donné faux, donc passe n'aurait pas changé.

Les évaluations conditionnelles sont quelque chose de difficile à maîtriser parce que ça fait intervenir le contexte du programme et qu'on n'a pas toujours conscience de l'état dans lequel sont les variables au moment où le programme doit prendre une décision. Voici, pour récapituler, le fragment de programme qui règle la circulation :

```
feu = 'vert'
if feu == 'vert' : passe = True                          # passe devient vrai
else : passe = False

feu = 'rouge'
if passe and feu == 'rouge' : passe = not passe          # passe devient faux

feu = 'vert'
if not passe and feu == 'vert' : passe = not passe       # passe devient vrai
```

Remarquez les lignes vides : elles sont dues au fait qu'après un if, l'interprète me propose son invite … de continuation, et que je dois presser la touche ↵ pour lui faire comprendre que je n'ai rien à ajouter (voir l'annexe 9.2 pour plus de détails sur ce mécanisme).

Les exercices suivants ne sont pas innocents ; l'objectif est de comprendre qu'une fonction peut être utilisée comme prédicat – autrement dit, qu'elle peut remplacer une ou plusieurs expressions prédicatives littérales.

[px04-2] coder une fonction unaire vert() qui retourne vrai si son argument a comme valeur 'vert' et faux sinon ; coder de même la fonction rouge() et la fonction orange()

[px04-3] coder une fonction unaire roule() avec un paramètre x, de sorte qu'elle ne retourne vrai que si vert(x) retourne vrai, ou que orange(x) retourne vrai

[px04-4] coder la fonction freine() avec un paramètre x, de sorte qu'elle ne retourne vrai que si vert(x) n'est pas vrai

[px04-5] coder la fonction unaire change() qui examine la valeur de son argument, et retourne 'vert' s'il vaut 'rouge' ou 'orange' s'il vaut 'vert', mais 'rouge' pour toute autre valeur ; testez cette fonction en évaluant la ligne suivante à plusieurs reprises :

```
feu = change(feu) ; feu
```

vérifiant que la couleur du feu passe bien par le cycle 'vert' → 'orange' → 'rouge' → 'vert'…
NB : feu doit être initialisé AVANT de pouvoir être utilisé comme argument...

2.4 FILTRAGE

Je voudrais vous proposer une petite séance de travaux pratiques : maintenant que nous nous savons en quoi consiste le traitement de l'information, voici comment mettre en pratique sur un problème *grandeur nature* les connaissances acquises à travers les exercices…

Ainsi, on a vu comment extraire un caractère d'une chaîne, mais comment savoir si ce caractère est une voyelle ou une consonne ? Avant de lire ce qui suit, réfléchissez au moyen de détecter si un caractère donné est une voyelle. On pourrait, par exemple, imaginer de définir une fonction comme :

```
def voyelle(x) :
   if x is 'a' : return True
   if x is 'e' : return True
   if x is 'y' : return True
   if x is 'u' : return True
   if x is 'i' : return True
   if x is 'o' : return True
   return False
```

Lorsque qu'une définition est sur plusieurs lignes, toutes les lignes, hormis la première doivent être décalée d'un cran vers la droite, parce que c'est comme ça que l'interprète comprend que ces lignes forment un *bloc* ou groupe d'instructions. Le même principe doit être observé avec les expressions conditionnelles : il peut y avoir plusieurs instructions à exécuter si la condition logique se vérifie, et l'interprète se fonde sur le décalage à droite pour considérer comme un bloc toutes les lignes qui sont dépendantes de cette condition. Par ailleurs, le décalage doit être le même pour toutes les lignes : si on a commencé un bloc en décalant, par exemple, de deux espaces, il faut que ce soit également deux espaces pour *toutes* les lignes de ce bloc — cf. section 112:9.2, INDENTATION.

Que la forme détermine la structure n'est pas unique à ce langage, mais tous les autres utilisent des caractères délimiteurs appairés (comme les parenthèses en LISP) pour exprimer formellement la structure ; ce qui est unique à PYTHON, c'est que c'est la *présentation graphique* qui détermine la *structure de contrôle*. À rapprocher de la prédiction de Donald E. Knuth, l'auteur de « THE ART OF PROGRAMMING », il y a déjà plus de 30 ans :

> We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language.

Donald E. Knuth
"Structured Programming with goto Statements"
Computing Surveys, Vol 6 No 4, Dec. 1974

Revenons à la fonction voyelle(), remarquez que les 6 première lignes retournent immédiatement une valeur, ce qui signifie que l'évaluation de la séquence est obligatoirement *abandonnée* en ce point ; et si le caractère passé en argument n'est pas l'une des six voyelles testées, on aboutit forcément à la dernière instruction, donc on peut avec certitude retourner la valeur *faux*. Certitude d'ailleurs toute relative, puisque j'ai cru bon de faire l'économie des voyelles accentuées… J'ai même fait le sacrifice de quelque chose d'encore plus important, comme on va le voir tout à l'heure. Et heureusement que je n'ai pas choisi de tester les consonnes, car il m'aurait fallu alors plus de 20 lignes de code !

En fait, il y a bien plus simple : il suffit de créer un ensemble, puis de tester *l'appartenance* à cet ensemble… Testons-en le principe, *interactivement*, avec l'interprète :

```
'o' in 'aeyuio'                                    # oui
'p' in 'aeyuio'                                    # non
```

Maintenant qu'on est sûr que ça marche, nous pouvons le coder sous la forme d'une fonction :

```
def voyelle(x) :
   if x in 'aeyuio' : return True
   return False
```

Ceci montre qu'en matière de traitement de données il y a souvent plusieurs façons de faire, dont certaines plus coûteuses que d'autres. Ici, ça paraît évident, mais ça ne le sera pas toujours… La question de par quel bout attaquer un problème est non-triviale, et demande réflexion. On reproche souvent aux débutant de se précipiter sur leur clavier sitôt le problème posé ; et il y a des chances pour que la complexité de leur solution soit inversement proportionnelle au temps de réflexion.

Bon, ça va mieux ; mais suis-je vraiment certain qu'on ne peut pas faire encore plus simple ? Si je me pose la question, c'est parce que je sais que plus *gros* est le code, plus *grands* sont les risques d'erreur : les statistiques montrent qu'un programmeur professionnel n'écrit en moyenne, par jour, que 20 lignes de code sans erreur. Par *jour* !

Et là, je m'aperçois que ma définition est terriblement redondante : la valeur de l'expression qui me sert de *condition* est précisément la valeur que je retourne ! J'en ai écrit 3 fois trop pour ce que ça fait, et je pourrais ramener cette fonction à *une seule* ligne de code (encore un *one-liner*) :

```
def voyelle(x) : return x in 'aeyuio'
```

[px05-1] définissez la fonction consonne() de la façon la plus économique possible, en supposant que son argument sera uniquement constitué de caractères alphabétiques.

ITÉRATION SUR UNE SÉQUENCE

Maintenant, j'aimerais pouvoir traiter un mot entier, et l'exploser en 2 chaînes distinctes : une pour les voyelles qu'il contient, et une autre pour les consonnes.

> *Ne vous méprenez pas sur l'apparente futilité des traitements proposés : les cas d'école représentent, en essence, l'application de principes généraux, et préparent à la maîtrise de l'outil ; sachant qu'il s'agit là d'un des outils les plus complexes jamais inventés, la maîtrise globale passe forcément par la maîtrise des éléments.*

Ici, le problème posé est celui de la répétition, techniquement appelée *itération* : si je sais faire pour un, et que ma solution est aussi générale que possible, je dois pouvoir faire pour tous. Toute la difficulté du traitement automatique est condensé dans cette formule : généraliser les solutions. En effet, il est toujours possible de traiter chaque cas en particulier, mais la vie est trop courte…

S'il s'agit d'une séquence, ce qui est souvent le cas des traitements automatiques, il existe une instruction primitive d'itération qui prend schématiquement la forme suivante :

```
for x in <séquence> : <fais ça avec> x
```

Le mécanisme mis en œuvre ici va se charger de “scanner”, *éplucher* la séquence de gauche à droite en s'arrangeant pour que la variable x prenne à chaque fois pour valeur l'élément courant de la séquence. Essayez cette application triviale :

```
for x in 'SNCF' : print x
```

Voyez-vous comment l'instruction print s'est vu passer à chaque fois un argument différent ? Et comment elle s'arrête automatiquement une fois la séquence épuisée ? Essayez quelque chose d'un peu plus complexe, qui affiche le code de chaque caractère, c'est à dire sa représentation interne :

```
for x in 'SNCF' : print ord(x)
```

Ce programme d'une ligne nous en apprend beaucoup sur ce qui se passe dans les cuisines du palais : il semble que depuis le début on nous fait croire que nous manipulons des *caractères*, alors qu'il s'agit purement de *codes* numériques, interprétés *en contexte* comme des symboles de notre alphabet. En fait, le roi est nu… Mais là est la puissance de l'outil informatique : tout est virtuel ; et tout est *interprété*, de sorte que nous puissions l'adapter à la représentation de ce qui nous intéresse.

Ici réside le bien-fondé des types de données : un nombre entier, par exemple, n'est jamais qu'une *interprétation*, une *représentation* sous forme de symboles numériques familiers, d'un codage en réalité faits de bits, c'est-à-dire de la juxtaposition des tensions électriques de minuscules éléments électroniques qui constituent la mémoire d'une machine. Et pour nous donner la représentation qu'on attend, le programme a besoin de connaître le type des données que nous manipulons. Nous y reviendrons…

Mais soudain un doute m'assaille : si l'interprète fait la différence entre les minuscules et les majuscules, c'est qu'il doit y avoir une différence de codage ! Essayons :

```
for x in 'SNCF' : print ord(x),
for x in 'sncf' : print ord(x),
```

Avez-vous remarqué que, comme avec toutes les instructions complexes, un bloc for doit être suivi d'une ligne vide ? Ceci au cas où notre bloc ferait plus d'une ligne. Attention ici à la virgule en fin de print ; cette étrange syntaxe dit à print qu'il ne faut pas aller à la ligne à chaque fois, de sorte que je vais pouvoir comparer les deux codages chacun sur sa ligne :

majuscules	'SNCF'	83	78	67	70
minuscules	'sncf'	115	110	99	102

Tiens, tiens, tiens ! Non seulement les codes des majuscules sont inférieurs à ceux des minuscules, mais en plus, on observe une constance dans la différence : la minuscule est codée comme la majuscule + 32. Je vais faire une petite expérience : partant du code du 'A', je devrais obtenir 'Z' en ajoutant 25, et 'z' en rajoutant encore 32.

La converse de ord() s'appelle chr(), ce qui nous permet de coder le programme :

```
d = ord('A')                                                    # début
f = d + 25                                                      # fin
for x in range(d, f) : print chr(x), chr(x + 32)
```

En effet, c'est bien ce j'attendais : et ceci laisse présager que la fonction voyelle() définie plus haut ne fonctionnera correctement que si son argument est minuscule ; s'il est majuscule, il sera automatiquement assimilé à une consonne… Embêtant n'est-il pas ?

Nous verrons bientôt comment résoudre ce problème, mais vous pouvez, en attendant réfléchir à votre propre idée de la solution. Par exemple, vous pouvez tester si le code d'un caractère est compris entre ord('A') et ord('Z'), et le convertir en minuscules en ajoutant 32 à ce code, n'est-ce pas ? Ou toute autre astuce qui vous viendrait à l'esprit : tout est permis, puisque vous ne risquez pas de casser quoi que ce soit – et d'ailleurs personne ne vous regarde.

Ceci explique le résultat du 3e test de [px04-1] qui compare 2 chaînes et retourne faux alors que la première est plus « grande » que la deuxième : c'est en fait une comparaison lexico-graphique, fondée sur le *codage* ; et si 'a' < 'w', alors 'azertyuiop' > 'wxcvbn' est forcément faux.

Remarque : le programme ci-dessus utilise une variante d'expression itérative : il s'agit toujours d'une séquence, et il s'agit toujours bien de scanner cette séquence élément par élément, sauf qu'ici la séquence est la valeur retournée par la fonction range(). Le mot “range” signifie *intervalle*, et cette fonction, ici, est binaire : elle prend 2 arguments qui constituent le point de départ et celui d'arrivée, et retourne une séquence de nombres entiers.

Vous pouvez, si vous le souhaitez, visualiser une telle séquence en évaluant directement la fonction range() sous l'interprète :

```
range(10)
range(10, 20)
```

Est-ce là ce que vous attendiez ? La séquence a-t-elle bien le nombre d'éléments prévus ? Par pure curiosité, essayez aussi :

```
range(10, 20, 2)
range(20, 10, -2)
```

Même si vous n'avez pas encore résolu le problème de casse de caractère, nous en savons déjà assez pour programmer l'explosion d'un mot :

- il nous faut, au départ, deux séquences *vides*, une pour les voyelles, l'autre pour les consonnes ;

- nous allons alors scanner le mot, en ajoutant le caractère courant à l'une ou l'autre séquence, selon le résultat retourné par la fonction voyelle()
- une fois le traitement terminé, nous devrions avoir les caractères répartis selon leur statut dans deux chaînes distinctes

Simple, n'est-ce pas ? Alors, passons au codage proprement dit :

```
mot = 'abracadabra'
v = c = ''                                          # attention : chaîne vide
for x in mot :
   if voyelle(x) : v = v + x
   else : c = c + x

v ; c
```

il ne nous reste plus qu'à examiner les valeurs respectives de v et de c pour vérifier que tout s'est bien passé (et c'est que fait la dernière ligne) ; d'ailleurs l'expression :

```
len(v) + len(c) is len(mot)
```

retourne *vrai*, donc je n'ai perdu aucun caractère en route !

Maintenant que je suis sûr que tout fonctionne correctement, je peux définir une fonction avec ce bout de code, de sorte que je pourrais l'essayer avec d'autres mots sans avoir à me taper de tout retaper à chaque fois. La fonction n'a besoin que d'un argument, le *mot* ; je ne vais donc lui mettre qu'un seul paramètre :

```
def explose(mot) :
   v = c = ''                                       # attention : chaîne vide
   for x in mot :
      if voyelle(x) : v += x                        # pareil que v = v + x
      else : c += x
   return v, c
```

Remarque : cette fonction retourne une valeur *double*, représentée entre parenthèses : il s'agit d'un type de séquence un peu particulier {cf. tuple}, sur lequel nous aurons l'occasion de revenir. Pour l'instant, il nous suffit de l'exploiter comme une séquence ordinaire :

```
valeur = explose('cheval')
valeur                                              # → ('ea', 'chvl')
valeur[0]                                           # élément 0 → 'ea'
valeur[1]                                           # élément 1 → 'chvl'
```

Remarquez qu'on aurait pu écrire directement :

```
voyelles, consonnes = explose('cheval')
```

ce qui a pour effet de *distribuer* la double valeur aux deux variables situées en partie gauche de l'expression. Je ne l'avais pas mentionné avant, mais cette syntaxe vaut en fait pour tout type de séquence. Ainsi :

```
x, y, z = 'abc'
```

a pour effet de donner à x la valeur 'a', à y la valeur 'b', et à z la valeur 'c'.

[px05-2] essayez cette explosion avec d'autre mots, comme *désespéré*, ou *crève-cœur*...

L'itération est un moyen de traiter l'information élément par élément : c'est une application du principe « diviser pour régner » : dès lors que je peux déstructurer la donnée (la décomposer en éléments semblables), je peux concevoir le traitement qui convient pour un élément, et l'appliquer alors à tous les éléments…

Il existe plusieurs formes d'itérations. Celle que nous venons de pratiquer est explicite, mais on a déjà manipulé des itérations implicites comme dans [px04-1] 't' in 'azertyuiop'… Nous en verrons encore d'autres formes, et il en est une de remarquable pour son expression : elle a recours à elle-même pour se définir.

Supposons que nous n'ayons pas de fonction pour calculer la taille d'une séquence : on en a déjà eu besoin,

alors il aurait bien fallu la définir… La conception d'une telle fonction est d'une simplicité extraordinaire ; elle doit pouvoir traiter deux cas de figure : si la séquence est vide, la valeur à retourner est 0 ; dans tous les autres cas, la taille de la séquence c'est 1 + la taille du reste de la séquence ; ce qui s'exprime ainsi :

```
def taille(s) :
   if not s : return 0
   return 1 + taille(s[1:])
```

En effet, regardez comment est traité le 2e cas : la séquence n'est pas vide (sinon, la fonction aurait déjà retourné 0) donc sa taille est au moins égale à 1, plus la taille du reste de la séquence sans son 1er élément, ce qui s'exprime par s[1:]. Par curiosité, insérez un print repr(s) au début, juste avant le if, et regardez ce qui se passe lorsqu'on demande taille('zut') :

```
>>> taille('zut')
'zut'
'ut'
't'
''
3
```

NB : la fonction repr() retourne une *représentation interne* PYTHON de l'objet passé en argument, ce qui a ici pour effet d'afficher les quotes qui délimitent la chaîne ; on aurait pu s'en passer, mais je tenais à ce que vous constatiez de vos yeux ce que voit réellement la fonction quand elle est appelée : la toute dernière fois, l'expression s[1:] ne retourne plus rien (une chaîne vide), donc not s évalue à True, et c'est précisément le cas où la fonction retourne la valeur 0.

Or cette valeur était attendue par une expression 1 + taille(s[1:]), dont l'évaluation avait été différée par la nécessité d'évaluer d'abord la sous-expression taille(s[1:]). Maintenant qu'elle a obtenu la valeur 0, elle peut enfin y ajouter 1 pour retourner cette valeur à l'appelant, et ainsi de suite… d'où la valeur finale 3.

Certes, il y a d'autres manières de calculer le nombre d'éléments d'une séquence, mais aucune n'exprime ce calcul d'une manière aussi simple et aussi concise. D'autant que, codée par un GEEK, la fonction n'est plus qu'un tout petit ridicule bout de code, un ONE LINER :

```
def taille(s) : return s and 1 + taille(s[1:]) or 0
```

Car à y regarder de près, les if et les else ne sont jamais que des prédicats logiques manipulés par une syntaxe plus facile pour l'œil :

```
def taille(s) : return 1 + taille(s[1:]) if s else 0
```

Le if ne serait donc qu'une *conjonction* déguisée, tandis que le else serait une *disjonction*. Mais attention : on ne vous demande ni d'apprendre, ni de retenir cet aspect des choses, mais de savoir que ça existe – histoire de paraître cultivé.

GÉOMÉTRIE ITÉRATIVE

Si itérer c'est répéter, il serait cependant inutile de répéter exactement la même chose : en fait, comme dans le cas de 'SNCF', le contexte est différent à chaque itération. Toute l'astuce consiste donc à programmer ce qui va faire une différence tout en faisant pareil. Par exemple, si je veux dessiner un carré (donc 4 côtés égaux), je vais tracer un trait d'une certaine longueur, puis tourner de 90°, tracer un second trait de même longueur, etc. Et c'est la quadruple répétition de ces actions élémentaires qui aboutit à la représentation finale. Et notez au passage comment est ici appliqué le principe diviser pour régner :

- *analyse* du problème pour le ramener à une série d'actions
- *factorisation* de ces actions pour les programmer économiquement

Pour montrer ce qu'on attend d'un *analyste programmeur*, à ceci près que tout n'est pas aussi facile à analyser – et qu'il faut également savoir programmer, reprenons le programme du 112:0.4 :

```
from turtle import *                                    # cf. 9.1 en annexe
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
```

La toute première ligne intègre dans l'interprète un ensemble de fonctions, prédéfinies dans un fichier qu'on appelle un *module* ; le module turtle est, en principe, distribué avec toutes les versions de PYTHON. Sont définies dans ce module, la fonction forward(), qui trace un trait en alignant n points, et left() qui effectue une rotation de m degrés.

Ce programme prouve que l'analyse est bonne. Maintenant, il s'agit d'identifier ce qui peut être factorisé : d'un coup d'œil, vous voyez qu'après avoir tracé un trait, on effectue une rotation d'un quart de tour pour préparer le trait suivant ; c'est vrai qu'on a négligé de le faire après le dernier trait, puisque ce n'était pas la peine, mais si on l'ajoutait on aurait quatre paires d'instructions absolument identiques. À partir d'une séquence d'actions, nous avons caractérisé le procédé : nous en avons fait une *abstraction*, prête à être programmée.

```
for x in range(4) :
   forward(100)
   left(90)
```

Et à ce stade, on pourrait objecter que le programme reste quand même très particulier, puisqu'il ne sait faire que des carrés noirs de 100 points de côté… D'accord, mais rien ne m'empêche de définir une variable taille = 100, et de m'en servir comme argument de forward(), de sorte que je puisse régler la taille du carré comme je veux ; quant à la couleur, il me suffit d'appeler la fonction color() en lui passant, par exemple, la valeur 'red' ou 'green'…

[px06-1] définissez taille, appelez color('blue') et exécutez le code ci-dessus où la constante 100 serait remplacée par la variable taille ; pour repartir d'une fenêtre propre, appelez clear() ou reset() ; appeler help('turtle') pour plus d'information

[px06-2] coder une fonction binaire qui accepte deux arguments : une valeur numérique (la taille) et une chaîne (la couleur), et qui trace un carré de cette taille et de cette couleur

Le carré n'est jamais qu'un cas particulier de polygone : pour que les 4 traits se rejoignent, l'angle est 90° ; mais qel serait l'angle pour tracer un polygone à 3 côtés ? Pouvez-vous généraliser pour le tracé d'un pentagone, hexagone, et ainsi de suite jusqu'au dodécagone ?

[px06-3] coder une fonction générale polygone() qui peut tracer un polygone quelconque de taille quelconque et de couleur quelconque : le nombre de côtés étant paramétré, la fonction devrait donc aussi pouvoir faire, entre autres, un carré ou un triangle

Ces polygones sont dit *convexes simples*, mais il en est d'autres dont les côtés se « croisent » comme l'étoile ci-contre ; quel serait l'angle à chaque pointe d'une étoile à 5 pointes ? Peut-on généraliser pour n'importe quel nombre de pointes ? Que se passe-t-il pour un nombre pair de pointes ?

[px06-4] coder une fonction générale star() qui peut tracer une étoile quelconque de taille quelconque et de couleur quelconque, et un nombre de branches quelconque — mais supérieur à 4

2.5 NOTION DE MÉTHODE

Comme on l'a mentionné plus haut, PYTHON est un langage orienté objet : sans rentrer dans les détails, disons que chaque *type* d'objet est muni de méthodes spécifiques. Par exemple, on a vu que les nombres entiers et les nombres réels ont des méthodes différentes de division, qui produisent des résultats différents. On a vu également que la somme de deux chaînes (ou séquences de caractères) utilise une méthode de *concaténation* (mise en chaîne) alors que la somme de deux nombres utilise une méthode d'*addition* arithmétique.

Les chaînes sont munies en standard d'une trentaine de méthodes, parmi lesquelles split() (en français,

sépare) qui effectue la segmentation automatique de la séquence en fonction d'un caractère donné, par exemple l'espace.

Essayez le programme suivant :

```
mots = 'petit chat'
espace = ' '
mots.split(espace)                                  # → ['petit', 'chat']
```

Le résultat est une séquence d'un nouveau type appelée *liste*, délimitée (encadrée) par un '[' au début et un ']' à la fin (les listes seront étudiées dans la section suivante). Cette liste comprend ici deux éléments, séparés par une virgule, et chacun de ces éléments est une chaîne ; remarquez que le caractère spécifié comme *séparateur* a disparu de la séquence.

On voit ici que la méthode split() s'utilise en spécifiant l'objet (ici la chaîne), suivi d'un point et du nom de la méthode. L'exemple ci-dessus manipulait la valeur d'une variable de type str, mais il est tout à fait possible de citer une chaîne littérale ; le résultat sera le même…

```
'Agence France Presse'.split(' ')                   # → ['Agence', 'France', 'Presse']
'P.M.U.'.split('.')                                 # → ['P', 'M', 'U', '']
```

Remarquons que, comme dans ce dernier cas, des choses étranges peuvent se produire si le caractère de segmentation se trouve tout au début, ou tout à la fin de la chaîne : pour 'P.M.U.', par exemple, la méthode fabrique une liste avec les éléments qui se trouvent de part et d'autre d'un point, mais comme il n'y a rien à droite du dernier point, elle ajoute un quatrième élément, une chaîne vide, matérialisée par 2 *apostrophes* avec *rien* entre les deux…

Essayez par exemple de segmenter :

```
'.barre'.split('.')
```

[px07-1] si le caractère de segmentation n'est pas dans la chaîne, split() produit quand même une liste ; mais que met-il dedans ? Essayez :

```
'a + b'.split('-')
'univ-paris8'.split('*')
```

[px07-2] si le caractère de segmentation n'est pas spécifié, split() produit quand même une liste ; mais que met-il dedans ? Essayez :

```
'azertyuiop qghjklm'.split()
```

[px07-3] si la chaîne originale contient des nombres, est-ce que split() les garde sous forme de chaîne, ou bien les convertit-il en objets numériques ? Essayez :

```
x = '1 23 456'.split()
x[0] + x[1]                                         # le résultat dépend du type des opérandes…
```

[px07-4] les méthodes upper() et lower() convertissent une chaîne en majuscules et minuscules : recodez la fonction voyelle() pour qu'elle soit indifférente à la casse de son argument

VU D'AVION

valeur de vérité
- valeur booléenne : vraie (`True`) ou fausse (`False`)
- interprétation numérique binaire : `False` équivaut à 0, tandis que `True` équivaut à 1
  - `int('101', 2)` → `(True * (2 ** 2)) + (False * (2 ** 1)) + (True * (2 ** 0))` → `5`
  - `'ab'[False]` → `'a'`
  - `'ab'[True]` → `'b'`

`bool()` : fonction à peu près inutile qui convertit son argument en valeur de vérité

prédicat
- se dit d'une expression dont la valeur est interprétée d'un point de vue logique
- toute expression symbolique, même atomique, peut être utilisé comme prédicat
- pour tout type de donnée, l'élément neutre est interprété comme `False`
- ceci inclut `False`, `None`, `0`, `séquence vide`, etc…

`turtle`
- module graphique de la distribution standard
- on peut en lire le code source dans `/usr/lib/python2.5/turtle.py`
- requiert l'installation du module `Tkinter` pour fonctionner

méthode
- se dit d'une fonctionnalité propre à un type d'objet
- existe sous forme de
  - opérateur binaire : `+`, `*`, `%`
  - simili-fonction : s'utilise avec une syntaxe particulière, comme dans `'abc'.upper()`

le coin des curieux

- méthodes de chaînes, manuel de référence à http://docs.python.org/library/stdtypes.html#id4
- définir une méthode suppose maîtrisée l'approche orientée objet, esquissée section 6.4
- expression booléenne `x if <condition> else y`
  - voir http://docs.python.org/reference/expressions.html#boolean-operations

RÉCAPITULATION

SÉQUENCES

On a compris les rudiments de la manipulation de séquences, à partir d'exemples sur les séquences de caractères – il existe un autre type de séquence, étudié dans le prochain chapitre, qui se manipule de la même façon : par indexation.

Et, de façon totalement implicite, on a aussi compris que la séquence d'instructions est une forme de structure de contrôle…

PRÉDICATS

On sait maintenant qu'un programme peut prendre des décisions à partir de prédicats, expressions dont la valeur est interprétée dans le domaine {VRAI, FAUX}, celui de la logique binaire.

On peut inverser la valeur d'un prédicat en lui appliquant l'opérateur not de négation logique, et on peut construire un prédicat complexe en utilisant l'opérateur and de conjonction et l'opérateur or de disjonction.

DÉCISIONS

Un prédicat peut être exploité dans une structure de contrôle if pour décider de la conduite à tenir (roule ou freine) ou de la valeur à donner à une variable ; une fonction qui retourne vrai ou faux peut tout à fait être utilisée comme prédicat. En fait, n'importe quoi peut être utilisé comme prédicat quand on sait que tout ce qui n'est pas False, 0 ou la séquence vide, est forcément vrai.

L'opérateur in, par exemple, retournera vrai si son opérande gauche est l'une des valeurs de son opérande droit (point de vue ensembliste), alors qu'utilisé dans une expression for, il donnera tour à tour chacune des valeurs de la séquence donnée comme opérande droit.

ITÉRATIONS

Le for introduisait le concept d'itération, permettant de répéter l'exécution d'une séquence d'instructions, nécessité évidente pour ceux qui veulent programmer de manière compacte.

OBJETS + MÉTHODES

Enfin, on sait désormais qu'à part les symboles d'opérateur, tout symbole manipulé par ce langage est un objet, et que chaque type d'objet dispose de méthodes spécifiques, qui n'ont de sens que pour ce type.

Et comme la séquence est, en soi, un type, ça explique pourquoi tous les types de séquence ont en des méthodes en commun – comme la façon d'adresser un élément…

③ LISTES : ACCÈS INDEXÉ

Une liste est une structure de données qui permet d'agréger un ensemble de valeurs et de les manipuler en bloc ou en détail. Elles servent à représenter des données structurées : elles sont extensibles à volonté, et peuvent donc accommoder des représentations dynamiques, qui se construisent au fur et à mesure du traitement de l'information.

Les listes ne sont qu'un *cas particulier* de séquence : tout ce qui a été exposé dans le chapitre précédent s'applique donc également à la manipulation de listes, y compris les opérateurs fondamentaux comme `in`, `+` et `*`, mais aussi les techniques d'accès, par index ou par tranches d'index, autrement dit, par intervalles. Par contre, certaines méthodes typiques de chaînes, comme `upper()` ou `lower()`, ne sont pas applicables aux listes, qui ont, elles, leurs méthodes spécifiques.

La section 2 est une brève incursion dans le monde de la programmation fonctionnelle, monde qui fait l'objet d'un cours spécifique, c'est pourquoi je n'en retiens ici que l'aspect technique le plus, euh… fonctionnel.

En section 3, nous verrons comment créer un exécutable autonome, utilisable à la console : le schéma de principe est celui de tous les programmes unix, qui prennent leurs arguments depuis la ligne de commande.

La section 4 traite du codage des textes. Il n'est pas nécessaire de tout comprendre, ni même de tout lire : il suffit de savoir que si PYTHON exhibe la représentation interne des données, c'est qu'il ne peut pas faire autrement. Le sujet aurait probablement été mieux intégré dans le chapitre 6 qui expose quelques techniques de traitement de données textuelles ; il est donc un peu prématuré, mais ces explications visent à satisfaire immédiatement une demande assez générale…

sommaire

Une liste est une séquence, une collection ordonnée (une série) d'objets *arbitraires* consécutifs accessibles par index ; arbitraire ici signifie qu'une liste peut contenir n'importe quel type d'objet, et ces objets peuvent être hétérogènes, c'est-à-dire de types différents ; à la différence des chaînes et des nombres, les listes sont *mutables*, jargon technique pour dire qu'on peut en modifier les éléments.

Les listes, en tant que séquences, exhibent un comportement similaire à celui des chaînes : on peut les concaténer avec l'opérateur +, les répliquer avec l'opérateur *, et on accède à leurs éléments exactement de la même façon ; évaluez ces expressions, pour voir :

```
jurons = ['saperlipopette', 'mille sabords']
jurons                                          # → ['saperlipopette', 'mille sabords']
jurons + ['zut']                                # → ['saperlipopette', 'mille sabords', 'zut']
jurons                                          # → ['saperlipopette', 'mille sabords']
jurons[1]                                       # → 'mille sabords'
jurons[0]                                       # → 'saperlipopette'
jurons[-1]                                      # → 'mille sabords'
```

Comme le montre la 3e expression ci-dessus, concaténer un nouvel élément à une liste existante ne modifie pas la liste, il faut une instruction explicite :

```
jurons = jurons + ['zut']
jurons                                          # → ['saperlipopette', 'mille sabords', 'zut']
```

Remarque : l'élément ajouté doit être lui-même une liste – d'où la forme ['zut'] – parce que l'opérateur + attend deux objets de même type, et ne fait pas de conversion implicite. La construction d'une liste peut donc être programmée :

```
L = []                                          # une liste vide
for x in range(1, 9) : L = L + [x]              # ajoute les éléments 1 par 1
```

Ajouter un nouvel élément peut aussi se faire *directement* avec la méthode append() :

```
jurons.append('enfer')
jurons                                          # → ['saperlipopette', 'mille sabords', 'zut', 'enfer']
```

Et pour insérer *directement* un nouvel élément, on peut utiliser la méthode insert() :

```
jurons.insert(2, 'tonnerre')
jurons                                          # → ['saperlipopette', 'mille sabords', 'tonnerre', 'zut', 'enfer']
```

Et notez que ces deux méthodes prennent comme argument un élément de liste, et non une liste, à la différence de l'opérateur + ; mais on peut aussi insérer une liste, si on veut.

Comme on l'a dit plus haut, les listes sont mutables, on peut en changer les éléments :

```
jurons[0] = 'ciel'                              # → ['ciel', 'mille sabords', 'tonnerre', 'zut', 'enfer']
jurons[2] += ' de Brest'                        # → ['ciel', 'mille sabords', 'tonnerre de Brest', 'zut', 'enfer']
```

Pour toute redéfinition de variable, on peut utiliser ce qu'on appelle des opérateurs d'affectation cumulatifs (*augmented assignment operators* – cf. notes de la section 1.1) :

```
jurons[2] += ' de Brest'
```

a pour effet de retrouver la valeur de jurons[2], c'est-à-dire 'tonnerre', de concaténer cette valeur avec celle qui est donnée en partie droite de l'instruction, 'tonnerre' + ' de Brest', et de remplacer l'ancienne valeur par la nouvelle. On aurait pu dire la même chose avec :

```
jurons[2] = jurons[2] + ' de Brest'
```

mais ça fait moins à coder, donc ça diminue les risques d'erreurs ; et il se trouve que le code est plus efficace, donc que ça peut faire une différence de performance, en particulier lors d'une itération. Cette syntaxe idiomatique se retrouve dans d'autres langages, comme le C, alors autant la maîtriser tout de suite.

On peut manipuler une expression de liste sans qu'il soit besoin d'en faire une variable :

```
10 + int('1 23 456'.split()[1])                    # → 33, mais bien tordu
```

Puisque (comme on l'a dit plus haut) une liste peut contenir n'importe quel type de donnée, on peut rassembler des données de type *texte* avec, par exemple, des *nombres* :

```
['année', 1945]
```

Cette liste a deux éléments : une chaîne et un nombre entier. Pour en extraire l'année (l'élément de rang 1), il suffit de demander :

```
['année', 1945][1]                                 # → 1945
```

Notons que ça marcherait aussi bien avec des variables ; on peut ainsi écrire :

```
date = ['année', 1945] ; x = 1 ; date[x]           # → 1945
```

Parce qu'une liste est une séquence, on peut itérer sur cette séquence pour accéder à chacun de ses éléments successivement ; et on peut le faire en séquentiel aussi bien qu'en indexé :

```
for d in date : print d                            # accès séquentiel aux valeurs

for x in range(len(date)) : print date[x]          # accès indexé aux valeurs
```

Ceci dit, une liste peut aussi contenir des listes :

```
date = [['année', 1945], ['mois', 'octobre'], ['jour', 15]]
```

La liste date a trois éléments, qui sont des listes – et chaque sous-liste a deux éléments… Pour extraire, par exemple, le jour du mois, il me faudra coder :

```
date[2][1]                                         # élément d'index 1 de l'élément d'index 2
```

Il n'y a pas de limite à la longueur des listes, ni à leur profondeur. On peut donc s'en servir pour représenter n'importe quelle information structurée :

```
emplettes = [['boulangerie', ['baguettes', 3], ['croissants', 6]],
             ['superette', ['yaourts', 2], ['autres trucs', 4]]]
```

Cette liste de profondeur 3 est la représentation, en machine, de ma liste de courses à faire :

boulangerie	baguettes : 3 croissants : 6
superette	yaourts : 2 autres trucs : 4

La structure des données est contrainte par le parenthésage des crochets ; cette liste a ici 2 éléments (des listes) qui ont chacun 3 éléments : le commerçant et deux listes ayant chacune deux éléments : un produit et la quantité à acheter. Maintenant, si je veux savoir combien il me fallait de croissants déjà, il suffit de demander :

```
emplettes[0][2][1]                                 # → 6 ← élément 1 de l'élément 2 de l'élément 0
```

Et pour les yaourts :

```
emplettes[1][1][1]                                 # → 2 ← élement 1 de l'élément 1 de l'élément 1
```

Techniquement, de telles expressions sont parfaitement utilisables quand on sait d'avance quelle position occupe l'élément qui nous intéresse ; on verra plus tard une autre façon de structurer l'information de sorte qu'on puisse accéder directement au nombre de croissants.

Les séquences sont dites « itérables » : il est donc possible d'itérer sur une liste de taille et de profondeur quelconque ; par exemple, pour emplettes :

```
for commerce in emplettes :                          # niveau 1
   for com in commerce :                             # niveau 2
      print com
      if type(com) is type([]) :                     # niveau 3
         for produit in com : print produit
```

Remarquez la précaution de la 4e ligne : à ce niveau, je peux rencontrer des séquences de caractères ou des nombres, et seules les premières (en tant que séquences) sont itérables ; le prédicat vérifie donc qu'il s'agit bien d'une liste avant de descendre dedans…

Les *types* de données ont été vus au chapitre 1 : pour chaque type, il existe une fonction qui fabrique un élément neutre de ce type ; par exemple, l'élément neutre pour l'addition d'entiers est 0, et int(), sans argument, retourne 0 ; de même str() retourne une chaîne vide, et list(), une liste vide, et ainsi de suite. On pourrait donc définir ainsi un prédicat is_list() :

```
def is_list(x) : return type(x) is type(list())
```

et l'utiliser à la place du prédicat littéral, comme pour les exercices de la série [px04].

Les exercices ci-dessous ont pour objectif d'illustrer la manipulation de listes ; il n'est pas absolument nécessaire d'en maîtriser tous les aspects, mais ça peut vous aider, plus tard, pour manipuler des arrangements (*arrays*) similaires dans des langages comme ANSI-C, qui utilisent cette même syntaxe pour accéder aux éléments d'un vecteur, d'une table ou d'une matrice.

[px08-1] soit la liste erreurs = [['Hiroshima', 1945, 'août', 6], ['Nagasaki', 1945, 'août', 9]]. Sachant que cette liste a deux éléments, quelle expression permet d'en extraire le premier élément ? Et le deuxième ? Quelle expression retrouve l'année pour *Nagasaki* ? Comment retrouver le jour, et quelle expression calculerait, à partir de erreurs, le nombre de jours entre les deux dates ?

[px08-2] quelle expression rassemblerait les noms des deux villes dans une nouvelle liste ?

[px08-3] on dit qu'une liste est *plate* lorsqu'elle ne contient pas de sous-liste : quelle expression permettrait d'aplatir erreurs, i.e. d'en faire une liste plate de 8 éléments ?

Les listes plates (de profondeur 1) sont utilisées, en calcul numérique, pour représenter les *vecteurs* ; la dimension d'un vecteur V est alors len(V).

[px09-1] soit V = [] ; quelles instructions permettraient de remplir ce vecteur avec les seize premiers entiers positifs, de 0 à 15 ?

[px09-2] soit V = [] ; quelles instructions permettraient de remplir ce vecteur avec les carrés des seize premiers entiers positifs, de 1 à 16 ?

[px09-3] soit V = [] ; quelles instructions permettraient de remplir ce vecteur avec les seize premières puissances de 2, de 15 à 0, décroissantes de la gauche vers la droite ?

[px09-4] comme pour toute séquence, on peut extraire d'une liste les éléments d'un intervalle donné : quelle expression extrairait de V les 8 puissances les plus faibles ?

[px09-5] soit G un vecteur de gros mots ; quel est le programme le plus simple pour *remplacer* chaque insulte ou juron de ce vecteur par le double de cet élément ?

Pour représenter une *matrice* de valeurs numériques, on utilisera une liste de profondeur 2 ; les dimensions d'une matrice M sont respectivement len(M) et len(M[0])

[px10-1] soit Z = [] ; codez, à partir de Z, la construction de M, la matrice [[1, 2, 3], [4, 5, 6]] ; généralisez pour n'importe quelle matrice de n lignes par m colonnes ;

[px10-2] codez une fonction col2() qui permet d'extraire de M tous les éléments d'index 2 ; généralisez la fonction col() pour n'importe quelle colonne d'une matrice de n lignes par m colonnes ;

[px10-3] codez la transposition (*lignes* → *colonnes*) de la matrice M → [[1, 4], [2, 5], [3, 6]] ; généralisez pour n'importe quelle matrice de n lignes par m colonnes ;

[px10-4] soit M = [] ; codez, à partir de M, la construction de la matrice

```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024],
 [3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049],
 [4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576],
 [5, 25, 125, 625, 3125, 15625, 78125, 390625, 1953125, 9765625] ]
```

3.1 CALCUL DU PLURIEL

Pour bien comprendre l'intérêt de structurer les données, je voudrais vous proposer une nouvelle petite séance de travaux pratiques sur les séquences : chaînes de caractères et listes. Voici le problème…

Presque tout le monde sait qu'en français, le pluriel d'un mot se forme en y ajoutant un « *s* » terminal, à quelques exceptions près. Justement, j'aimerais que vous me fassiez une petite fonction qui prend comme argument le mot à mettre au pluriel, et retourne ce mot au pluriel ; je pourrai donc l'utiliser ainsi :

```
pluriel('clou')
```

et le résultat serait "clous"… Pas bien difficile, n'est-ce pas ?

```
def pluriel(mot) : return mot + 's'
```

Fort bien, mais que va-t-il se passer pour des mots comme « *souris* » ou « *corps* » qui ont déjà un « *s* » au singulier ? Ah, me direz-vous, il suffit d'ajouter un petit test !

```
def pluriel(mot) :
   if mot[-1] is 's' : return mot
   return mot + 's'
```

D'accord, mais j'aimerais quand même bien pouvoir utiliser cette fonction pour d'autres mots courants, comme « *final* » ou « *bijou* », et qu'elle me renvoie un pluriel correct… Oh, bien sûr, vous pourriez me rajouter deux autres petits tests – et puis aussi un pour « *cheval* » et un pour « *genou* »… Mais alors, que ferez-vous de « *seau* » ou de « *riz* » ?

Je vous laisse réfléchir… Nous abordons ici un point délicat : celui du design d'un programme. Nous voyons bien qu'il y a plusieurs problèmes pour une fonction unique, mais ce qui n'est pas clair, c'est quelle approche adopter :

- l'approche brutale – un test pour chaque exception… Je ne sais pas combien il y en a, mais à vue de nez, j'ai l'impression que ça va faire beaucoup…
- l'approche explosée – une fonction pour chaque type d'exception ; car il y a des règles spécifiques pour des classes de mots : *travail* est comme *corail*, et ressemble d'ailleurs beaucoup à *journal*, qui pourtant ne se comporte pas comme *chacal* ; et *ripou* se termine comme *mou*, mais leur pluriel diffère ; et n'oublions pas les *pneus* qui sont *bleus*, ni les *beaux landaus* et autres *cadeaux*, sans parler des mots comme *doux* qui ont déjà un « *x* » au singulier.
- là où ça se complique, c'est qu'il y a, simultanément, des classes de mots qui ont, dans l'ensemble :
  - un pluriel *régulier* – sauf quelques exceptions qui ont un pluriel différent ;
  - un pluriel *irrégulier* – sauf quelques exceptions qui ont un pluriel régulier en « *s* ».

Puisqu'on parle d'ensembles, la solution PYTHON serait entre les deux : il s'agirait de créer des ensembles de mots, ou plutôt de *finales* de mots, un ensemble pour chaque classe d'exceptions. Par exemple, tous les mots à terminaison « *ou* » ont un pluriel en « *s* », sauf *bijou*, *caillou*, *chou*, *genou*, *hibou*, *pou* et *ripou* ; à l'inverse, tous les mots en « *eu* » ont un pluriel en « *x* » sauf *bleu* et *pneu*…

> Si j'écris tout ça, c'est pour retarder le moment où je publierai la solution : je vous prie de ne *pas regarder plus bas* pour l'instant, mais de réfléchir à la solution la plus économique, celle qui vous permettrait d'écrire le moins possible de code, tout en gérant le maximum d'exceptions.

Parce que c'est justement en cela que programmer est un art : une technique qui consiste à abstraire les caractéristiques essentielles de l'information à traiter, de façon à ce que le traitement soit aussi simple que possible. Ceci reste vrai quel que soit le type d'information à manipuler : traitement du signal, cryptographie ou reconnaissance de formes… autant de domaines où il faut faire un effort d'abstraction pour imaginer des solutions généralisables.

D'ailleurs c'est ça le mot-clé : *généraliser* ! Il est toujours (relativement) facile d'écrire un programme *ad hoc*, mais les problèmes se posent dès qu'on veut le faire évoluer pour étendre ses capacités de traitement…

Ainsi, ce que je propose ici, c'est de définir quelques ensembles, comme :

```
ail = 'bail corail émail soupirail travail vantail vitrail'.split()
```

Il s'agit des mots en « *ail* » qui forment leur pluriel en « *aux* » – tous les autres, comme *détail*, ayant un pluriel régulier. Un bête test comme `if mot in ail : return mot [0 : -2] + 'ux'` me permettrait alors de retourner le radical du mot (c'est-à-dire tout sauf le « *il* » final) en y rajoutant « *aux* » ; et le tour est joué !

De la même façon, on définira `ou = 'hibou chou genou caillou pou bijou'.split()`, la liste des « *ou* » qui deviennent « *oux* » ; et on testera l'appartenance du mot à cet ensemble de la même façon.

Et celle des « *eu* » qui ne deviennent *pas* « *eux* » : `eu = 'pneu bleu'.split()` ; et puis des « *au* » qui ne deviennent *pas* « *aux* » : `au = 'landau sarrau'.split()` ; hmm… deux lignes de code de plus !

Il ne reste plus qu'à se débarrasser des *invariants* comme les *riz*, les *roux* et les *gris* ; ainsi que de quelques monstres comme *œil* qui devient *yeux*, et *ail* qui devient *aulx*, et que d'ailleurs personne n'utilise, mais c'est pour faire bien – pour ceux-là je ne vois pas d'autre solution que le test *ad hoc*…

Si vous m'avez suivi jusque-là, vous aurez compris que la programmation est une discipline à double facette :

- l'*analyse* qui permet de décomposer le problème en sous-problèmes ;
- le *codage* qui décide de la *forme* des données à traiter, et du même coup de la *façon* de les traiter.

Et comme tout se tient, une erreur de *design* en un point de cette démarche conditionne tout le reste de la réalisation.

[px11-1] coder une fonction `pluriel()` pour former correctement le pluriel d'à peu près n'importe quel nom ou adjectif, majuscule ou minuscule ; testez-la, entre autres, sur les mots suivants : hors-d'œuvre, bal, régal, banal, canal, étal, pascal, portail, bétail, travail, œil, écœuré, hameçon, ex-æquo, niño, mépris, landau, rideau, jeu, bleu, clou, genou, prix… et tout ce qui vous passe par la tête !

3.2 PROGRAMMATION FONCTIONNELLE

Des langages comme LISP, HASKELL, O'CAML ou SCHEME ont de la programmation une approche *fonctionnelle* ; sans descendre dans les détails, voici une petite idée de ce que ça donne.

Mettons qu'on a une fonction comme `pluriel()` et qu'on a besoin de l'appliquer systématiquement à une liste de mots – c'est d'ailleurs le problème posé dans l'exercice ci-dessus ; supposons que ça m'arrangerait de récupérer une nouvelle liste avec les mêmes éléments, mais au pluriel. Bon, mettons que la liste de mots s'appelle justement `mots` ; il me suffit de coder :

```
[pluriel(x) for x in mots]
```

et la valeur résultante est une liste de même longueur, dont les éléments sont le résultat de l'application de la fonction `pluriel()` aux éléments de la liste originale.

[px11-2] définir la liste `mots`, et lui appliquer *votre* fonction `pluriel()` comme dans l'exemple ci-dessus.

De la même façon, l'expression `[z.upper() for z in 'abcd']` me retournerait :

```
['A', 'B', 'C', 'D']
```

La syntaxe d'une telle expression parle d'elle-même, pour peu qu'on sache l'analyser :

- les crochets à chaque extrémité sont l'indication du type de résultat retourné – une liste, et ceci quel que soit le type de la donnée traitée par l'expression
- vient ensuite une première sous expression qui décrit le calcul élémentaire (à effectuer sur un élément de la donnée), ici, c'est l'application de la méthode `upper()`, dont on sait qu'elle convertit un caractère minuscule en majuscule
- en dernier, on exprime la donnée itérable à traiter, que ce soit une constante littérale, comme ici, ou une variable
- mon tout est la liste collectant les résultats élémentaires

Le `z` de `z.upper()` est le `z` construit par `for z in` : il représente donc, tour à tour, chaque élément de la séquence

en cours de scan, en partant de la gauche. L'expression se comprend alors comme « *fais-moi la liste des résultats obtenus en convertissant* 'abcd' *en majuscules* ».

J'insiste sur le fait que la donnée doit être itérable : ça ne peut pas marcher sur un nombre, qui n'est pas une séquence de symboles, mais une quantité arithmétique. Par contre, ça marchera pour une séquence d'un seul caractère, encore que ça n'ait pas grand sens…

Notons qu'une telle construction syntaxique est équivalente au programme suivant :

```
L = []
for z in 'abcd' : L += [z.upper()]
```

Mais l'économie de cette approche se révèle avec [x * y for x in [2, 3] for y in [5, 6, 7]], qui génère la liste plate [10, 12, 14, 15, 18, 21], c'est-à-dire le produit cartésien de la 1e par la 2e liste… la même technique pouvant aussi servir pour un produit cartésien n-aire, comme, disons, X × Y × Z, pour faire simple.

[px12-1] convertir l'expression ci-dessus en une construction for standard générant exactement le même résultat

Mais le plus fort, c'est que de telles expressions peuvent être imbriquées, comme, par exemple, [[x * y for x in [2, 3]] for y in [5, 6, 7]] qui génère la matrice [[10, 15], [12, 18], [14, 21]].

La concision n'est pas le seul avantage de ce genre d'expression : il se trouve qu'elles ont aussi une bien meilleure performance qu'une construction for standard, ce qui en fait l'outil de choix pour les calculs numériques, mais aussi pour les jeux à base de damiers, ou toute autre application qui nécessite des représentations matricielles, typiquement le traitement d'image.

[px12-2] convertir l'expression ci-dessus en une construction for standard capable de générer exactement le même résultat
[px12-3] recoder les exercices [px10-1] et [px10-3] selon l'approche fonctionnelle
[px12-4] recoder l'exercice [px10-4] selon l'approche fonctionnelle

La génération de données n'est pas exclusivement réservée aux applications numériques : soit la liste ps des pronoms personnels sujets, 'je tu il nous vous ils'.split(), et la liste ip1 des terminaisons des verbes du 1er groupe à l'indicatif présent, 'e es e ons ez ent'.split(). La mécanique appelée fort justement *conjugaison* consiste à combiner un pronom, le radical d'un verbe v et la terminaison appropriée, par exemple : ps[0] + ' ' + v[:-2] + ip1[0] ; c'est précisément ce que je propose pour l'exercice ci-dessous ; alors que le suivant est, mine de rien, une sorte de produit cartésien qui ne dit pas son nom…

[px13-1] soient ps, ip1 (définis ci-dessus) et v = 'tirer' ; définir conjugue() qui prend une chaîne telle que v et un nombre entre 1 et 6 (la « *personne* »), et qui génère la forme conjuguée correspondante ; par exemple : conjugue('tirer', 4) → 'nous tirons'
[px13-2] soient ps, ip1 et v tels que définis ci-dessus ; coder la conjugaison optimisée de v, générant la liste des 6 formes ['je tire', 'tu tires', 'il tire', …]
[px13-3] adapter [px13-2] pour qu'il puisse traiter une liste V = 'filer rigoler conjuguer'.split(), ou toute autre liste (tant que les éléments en sont des verbes du 1er groupe), le résultat du programme devant être une liste de listes des 6 formes pour chaque verbe
[px13-4] juste pour le fun, recoder l'exercice [px13-3] selon l'approche fonctionnelle

3.3 PROGRAMME AUTONOME

Tout ce que nous avons fait jusque-là était développé *interactivement* sous l'interprète, mais il vient un moment où le programme est [à peu près] fini, et où on aimerait pouvoir s'en servir directement à partir du SHELL. C'est le cas du programme du pluriel, notre premier script qui puisse prétendre au statut d'utilitaire [presque] utile.

Il y a, vous l'avez déjà compris, deux manières d'évaluer un script :
- ligne par ligne, comme nous l'avons fait jusque-là
- en tant que fichier – à condition qu'il soit rédigé dans un fichier

Dans ce deuxième cas, il y a quatre façons de demander à PYTHON d'évaluer un script nommé, par exemple,

« plusieurs » :

- depuis PYTHON : execfile('plusieurs') charge et interprète le fichier exactement comme si on l'avait tapé ligne par ligne, mais n'affiche *pas* le résultat des évaluations ;
- depuis le SHELL : python plusieurs dit à PYTHON de charger et interpréter le fichier plusieurs ;
- depuis le SHELL : si le fichier plusieurs est en mode d'accès *exécutable* {cf. chmod}, et à condition que la première ligne du script (genre #! /usr/bin/env python) spécifie l'interprète à utiliser, il suffit de l'appeler par son nom : plusieurs ; il invoque alors implicitement l'interprète spécifié (qui se charge de façon transparente) et fait ce qu'on lui a dit de faire, puis s'arrête en faisant place nette (pour plus de détails, voyez man python).
- il existe encore une autre façon de faire, qui consiste à *importer* ce script, ce qui le convertit automatiquement en module, mais gardons ceci pour plus tard…

Nous allons utiliser, pour cet exercice, la 3e méthode : il vous faudra donc utiliser la commande chmod u+x plusieurs pour donner à votre script plusieurs le statut d'exécutable.

On a vu ce que devait être la première ligne du script, et on sait pourquoi. Maintenant, si le code ou les données contiennent des caractères accentués, vous devez choisir un *encodage* et le déclarer, comme un commentaire, sur la 2e ligne, avec une syntaxe particulière :

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

Ici, deux remarques s'imposent :

- bien qu'elles commencent par le symbole #, ces deux lignes ne sont pas véritablement des commentaires, mais des *directives de comportement* interprétées par le SHELL
- la 2e doit être cohérente avec la façon dont le fichier est réellement encodé (décision que vous avez prise au moment où le fichier a été enregistré), faute de quoi PYTHON se fera une représentation interne erronée du ç et autres caractères accentués ; si, par exemple, le fichier est enregistré en ISO-8859-1, la ligne 2 devrait être # -*- coding: latin-1 -*-

Ensuite vient le code de la fonction pluriel() ; laissez une ligne vide avant (et, bien sûr, une autre après) pour que la définition saute aux yeux :

```
def pluriel(mot) : return mot + 's'
```

J'ai inséré ici ma toute première définition, qui ne vaut pas tripette, mais je suppose que vous aller lui substituer votre propre définition réalisée pour l'exercice [px11-1].

Maintenant, c'est la partie magique : ajoutez les lignes suivantes :

```
import sys
if len(sys.argv) > 1 : print pluriel(sys.argv[1])
else : exit('argument manquant : mot au singulier')
```

L'importation du module sys nous donne accès à de nouvelles fonctionnalités, et celle qui nous intéresse ici, c'est de voir ce qui a été tapé sur la ligne de commande dans la console : le programme appelé, et ses arguments ; en effet, tout ce qu'il y avait sur la ligne de commande, au moment où le script a été appelé, a été collecté dans une liste appelée argv[5] (qu'on utilise préfixée du nom du module originel).

Cette liste contient donc les mots que vous avez entrés : le premier mot est forcément plusieurs, puisqu'il faut bien appeler le programme, mais ici, ce mot ne nous intéresse pas…

Voilà le *pourquoi* du prédicat len(sys.argv) > 1 : vous devez appeler ce programme en lui donnant au moins un mot à mettre au pluriel, sinon, ce n'était pas la peine… Comme il arrive qu'on oublie comment utiliser un programme, l'instruction else rappelle aimablement le format des données attendues par le programme : un mot au singulier.

Ainsi, une utilisation correcte du script plusieurs serait :

```
plusieurs clou
```

5 le nom « *argv* » est l'abrégé de « *argument vector* », conventionnellement utilisé par les programmeurs C

La liste sys.argv contient alors deux éléments, et l'expression len(sys.argv) > 1 évalue à vrai. Comme l'élément 0 ne nous intéresse pas, nous ne prendrons de la séquence que le 2e élément.

Attention : en mode exécutable un script se fait aussi discret que possible et ne montrera donc aucun résultat d'évaluation, normalement visibles en mode interactif – seul sera visible, à la console, l'effet des instructions print insérées dans le programme.

Notez que rien ne vous oblige à nommer votre script plusieurs : il pourrait s'appeler machin.chose ou même abracadabra.magic, c'est comme on veut – les linuxeurs pros préfèrent les noms opaques, comme plsrs.

Dans le monde UNIX, les exécutables n'ont pas de suffixe particulier, et l'usage est de ne pas en mettre du tout, que ce soit pour python ou zip, aussi bien que plusieurs. Un exécutable se reconnaît à ses attributs de fichier (cf. man ls), et on peut aussi utiliser la commande file :

```
file plusieurs
plusieurs: a python script text executable
```

De cette façon, on pourrait lister tous les exécutables – mais ça peut prendre du temps :

```
find . -type f -exec file '{}' \; | grep exec
```

Du fait que la ligne de commande est intégralement recopiée dans la liste sys.argv, il est possible d'utiliser ce mécanisme pour passer plusieurs mots à traiter. Dans ce cas, il ne s'agit plus de traiter seulement le mot sys.argv[1] : il faut itérer sur toute la liste, sauf, bien sûr, son premier élément, le nom du programme – encore que rien ne s'oppose à ce qu'on le mette lui aussi au pluriel, mais ce n'est pas terriblement intéressant.

Ainsi, en mettant plusieurs arguments à la suite du nom du programme plusieurs, il suffirait d'une construction for standard pour qu'une instruction print affiche, à la console, le pluriel de chacun :

```
plusieurs clou bleu mais rouillé
clous bleus mais rouillés
```

[px11-3] adapter le script plusieurs pour qu'il soit capable de traiter plusieurs mots passés en arguments sur la ligne de commande ; le tester abondamment pour garantir qu'il fonctionne correctement dans tous les cas de figure.

Si vous éprouvez des difficultés pour cette réalisation, jetez-donc un coup d'œil à la section iLP:9.1, p. 142

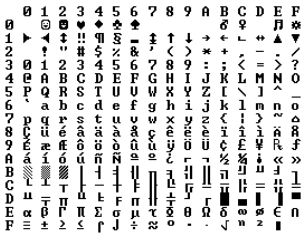
Et pour régler vos listes d'exceptions avec un maximum de précision, allez voir la page ouaibe de WIKIPEDIA : http://fr.wikipedia.org/wiki/Pluriels_irréguliers_en_français

3.4 UNICODE

L'objectif ici n'est pas de discuter les concepts du codage des caractères : il s'agit plutôt de démonter le mécanisme qui vous permet d'utiliser, dans vos programmes, l'*utf-8*, un codage parmi d'autres, mais qui a la particularité d'avoir, depuis ces dix dernières années, émergé en tant que standard.

Avant l'*utf-8*, il y avait l'ASCII (*american standard code [for] information interchange*), un système de codage des caractères conçus par (et pour) les nord-américains : comme on le voit ci-contre, il ne prend en compte que les caractères de contrôle, la ponctuation, les chiffres, et l'alphabet latin *de base* en usage dans les pays anglo-saxons… Les nombres hexadécimaux sur la ligne du haut sont la valeur des bits de poids faible, tandis que ceux de la colonne de gauche sont celles des bits de poids fort.

Rien n'étant prévu pour accommoder les caractères accentués européens, sans parler du grec, du cyrillique, où même de l'IPA (*international phonetic alphabet*), quelqu'un chez IBM eut l'idée d'étendre ce codage sur 8 bits, pour représenter 2^8 (soit 256) caractères — et d'en profiter pour y faire figurer quelques symboles supplémentaires, dont quelques caractères accentués (mais pas systématiquement), plus d'autres symboles d'une utilité discutable… mais à l'époque, ça paraissait quand même mieux que rien !



Parce que la mémoire se manipule, au minimum, par mots de 8 bits, la nécessité d'utiliser un octet pour représenter un caractère a conduit les concepteurs de langages, et, par là, les programmeurs, à considérer qu'un octet *est* un caractère, donc que l'espace nécessaire en mémoire pour représenter n caractères équivaut à n octets. D'ailleurs, les vieux langages ont encore un type de donnée appelé char, occupant 8 bits, supposés représenter un caractère.

Mais, même 256 codes, c'est insuffisant pour beaucoup de langues lues et écrites par quelques milliards d'individus qui n'ont pas grand'chose à voir avec les conventions occidentales. D'où l'idée de l'*utf-8*, qui redéfinit le concept de *code de caractère*, le renommant code point, et lui donnant une taille variable de 8 à 32 bits, selon les besoins.

La bonne nouvelle, pour les anglo-saxons, c'est que l'*utf-8* conserve les codes ASCII originels sous la forme de code points sur 7 bits, de sorte que beaucoup de vieux programmes puissent continuer à tourner sans problèmes — et ça, du point de vue conceptuel, c'est très fort : cf. <http://www.cl.cam.ac.uk/~mgk25/unicode.html>.

BYTE STRING

Mais pour les autres, ça veut dire qu'une chaîne de caractères ne peut pas être correctement interprétée si on ne sait pas comment elle a été codée : heureusement, une convention s'est spontanément établie, de fait, pour utiliser l'*utf-8* sur les systèmes POSIX, c'est-à-dire les variantes de UNIX et LINUX.

Et ceci signifie que les logiciels qui tournent sur votre machine codent, par défaut, le texte en *utf-8* : que Konsole, le terminal KDE, accepte du texte ainsi codé, et que gedit, open office, ou kate lisent et écrivent du texte sous forme *utf-8*.

En pratique, ça veut dire que vous pouvez entrer en PYTHON (en interactif), une chaîne de caractères non-ASCII, et que PYTHON la reconnaîtra automatiquement comme codée en *utf-8* :

```
quoi = 'œuf de pâques à noël'
```

Mais, il faut savoir qu'un glyphe[6] comme « œ » va, pour les besoins de la représentation interne, être transformé en un code point : $C593_{16}$, une valeur sur 16 bits mémorisée comme 2 octets consécutifs ; si on demande, la valeur de quoi, ce que PYTHON nous montrera, c'est la *représentation interne* de cette séquence, en termes d'octets — cette chaîne est, en fait, de type BYTE-STRING :

```
'\xc5\x93uf de p\xc3\xa2ques \xc3\xa0 no\xc3\xabl'
```

6. symbole graphique

Pour ce type d'objet, les valeurs hors de l'intervalle (32, 128) sont représentées par des *séquences d'échappement* (encore une convention où le back-slash introduit un code numérique), ici en hexadécimal, dont nous pouvons déduire que le glyphe « *â* » aurait comme code point $C3A2_{16}$, le glyphe « *à* », $C3A0_{16}$, et le glyphe « *ë* », $C3AB_{16}$. Selon les réglages de votre console, il est aussi possible que cette représentation interne apparaisse en octal, ou base 8, strictement équivalente à ce qu'on vient de voir :

```
'\305\223uf de p\303\242ques \303\240 no\303\2531'
```

En effet, 305 en base 8, c'est 197 en base 10, ou C5 en base 16 : évaluez donc les expressions oct(197) et hex(197) pour vous en convaincre !

Vous aurez l'occasion de rencontrer d'autres séquences d'échappement, les plus courantes étant '\n' pour le new-line ou *saut de ligne*, et '\t' pour la tabulation. Qu'une telle séquence ne représente qu'un caractère, mais occupe effectivement deux octets de données ne devrait plus vous surprendre.

Ici, l'expression len(quoi) retournera donc 24, bien qu'on soit tenté de dire qu'il n'y a jamais que 20 caractères dans cette chaîne… mais dont quatre sont unicodés. En fait il y a 20 glyphes codés en interne sur 24. La seule façon de retrouver ces 20 glyphes est de demander une *sortie* :

```
>>> print quoi
'œuf de pâques à noël'
```

Mais notez que certaines expressions (typiquement celles de données non élémentaires) forcent print à en révéler, de toute façon, la représentation interne :

```
>>> print quoi.split()
['\xC5\x93uf', 'de', 'p\xC3\xA2ques', '\xC3\xA0', 'no\xC3\xAB1']
```

alors que pour exactement les mêmes données, mais affichées élément par élément :

```
>>> for x in quoi.split() : print x,
œuf de pâques à noël
```

Ne vous laissez donc pas démonter par ces deux aspects : la représentation externe, c'est ce qu'on sert au client du restaurant ; l'interne, c'est ce qu'on manipule à la cuisine. Et d'ailleurs, le cuisinier ne devrait pas être gêné par les contraintes de la représentation, puisque les codes points peuvent, pour des opérations superficielles, être manipulés comme des unités élémentaires :

```
>>>'œil'.startswith('œ')
True
>>> 'œil de bœuf'.count('œ')
2
>>> 'œ' in 'meilleurs vœux' and 'ç' in "garçon, l'addition !"
True
>>> 'noël'.find('ë')                                    # retourne l'index où commence la sous-chaîne
2
```

DÉCODAGE AUTOMATIQUE

Mais il n'y a pas que les manipulations directes à la console : il y a aussi des contraintes sur les fichiers que l'interprète peut être amené à lire, que ce soit des scripts, ou des fichiers de données textuelles (ce deuxième cas sera examiné de plus près dans un autre chapitre).

On a dit plus haut que gedit enregistrait (encodait) ses fichiers en *utf-8* par défaut ; mais ça, PYTHON n'en sait rien ; il faut donc le lui dire, en déclarant le codage sur la première ou la deuxième ligne du script :

```
# -*- coding: utf-8 -*-
```

Évidemment, cette déclaration doit être cohérente avec le véritable codage du fichier : si vous enregistrez ce fichier en iso-8859-1,remplacez utf-8 par latin-1… L'important, c'est que PYTHON sache de quel codage il s'agit pour pouvoir le décoder et l'interner au format qui convient.

Ainsi, pour toute version de PYTHON antérieure à 3.0, ne rien mettre du tout serait équivalent à :

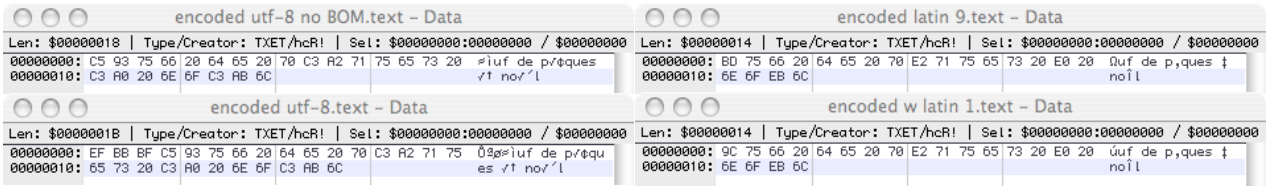
```
# -*- coding: ascii -*-
```

qui est l'encodage par défaut de PYTHON. Pour vous en convaincre, essayez donc ce petit programme :

```
import sys                                          # cf. annexe 9.1.2
sys.getdefaultencoding()
```

qui vous dira quel est effectivement le CODEC[7] par défaut de l'interprète : la version 3, par exemple, est en *utf-8* par défaut… mais il est aussi possible de tripoter le fichier site.py des versions antérieures pour changer ce CODEC – à condition de bien comprendre ce qu'on fait.

Voici la même donnée, « *œuf de pâques à noël* », enregistrée dans quatre formats différents ; les codes que vous voyez dans les fichiers seront ceux de la représentation *utf-8* :



Les formats *utf-8*, avec ou sans BOM[8] (à gauche), ne codent pas les caractères de la même façon que le latin 9 ou le windows latin 1 (à droite) ; et pour être en mesure d'en effectuer correctement le décodage, PYTHON doit savoir, au moment de la lecture, de quel codage il s'agit… On peut, bien sûr, lui mentir, mais le résultat affiché ne sera évidemment pas celui qu'on escomptait.

On peut, dans le terminal, utiliser la commande cat pour voir le contenu d'un fichier (affiché, par défaut, en *utf-8*). Une autre commande SHELL intéressante, c'est hexdump, qui en affiche les codes :

```
hexdump 'œuf de pâques à noël.text'
0000000 c5 93 75 66 20 64 65 20 70 c3 a2 71 75 65 73 20
0000010 c3 a0 20 6e 6f c3 ab 6c
```

Ainsi donc, si votre script contient des données non ASCII, voilà pourquoi il est impératif de le déclarer tout en haut, pour que la représentation externe corresponde à l'interne.

Les choses sont plus simples si vous utilisez la technique du drag and drop[9] pour alimenter la console ; la plupart des programmeurs ouvrent côte à côte, un éditeur et un terminal : on code direct dans l'éditeur, mais en cas de doute, on sélectionne l'expression, et on la tire dans le terminal où elle est évaluée par PYTHON, exactement comme si elle avait été entrée manuellement… Dans un tel cas, pas besoin de spécifier l'encodage puisque l'éditeur aussi bien que le terminal supportent (par défaut) le codage *utf-8*.

La section {6.4.3 : PROBLÈMES ET SOLUTIONS} revisitera la question du décodage dans le contexte de la transmission d'informations par INTERNET, en particulier avec le protocole HTTP.

DÉCODAGE MANUEL

Comme on l'a vu plus haut, les données représentées sous forme de BYTE-STRING peuvent subir un certain nombre de manipulations de surface… mais pas celles qui impliquent une interprétation de la séquence d'octets en termes des glyphes qu'elle représente. Pour ça, il nous faudrait une seconde représentation interne où chaque élément de la séquence correspondrait à un glyphe.

Que le texte soit lu dans un fichier, ou qu'il soit directement donné au programme en tant que valeur d'une variable ne fait aucune différence : pour certaines manipulations, la représentation interne en *utf-8* pose un problème parce que les glyphes qui apparaissent à l'affichage peuvent être, en interne, représentés par 8 ou 16 bits…

On l'a vu lors de la réalisation de l'exercice px05-2 — un mot comme « *écœuré* » supporte mal qu'on le décompose en octets, puisque la moitié de ses caractères sont alors mutilés :

```
>>> mot = 'écœuré' ; mot
'\xc3\xa9c\xc5\x93ur\xc3\xa9'
>>> for x in mot : print x,
  c   u r
```

7. codec = codeur + décodeur ; cf. http://docs.python.org/library/codecs.html#module-codecs pour les détails
8. BOM : byte order mark
9. littéralement « tire et lache »

Chaque fois qu'on prétend ignorer à quoi correspond la représentation interne, on s'expose à des mécomptes — en pratique, il n'est pas possible de traiter avec insouciance du texte sous forme *utf-8*, parce que c'est un codage conçu pour transmettre l'information dans un flux — et non pour la traiter.

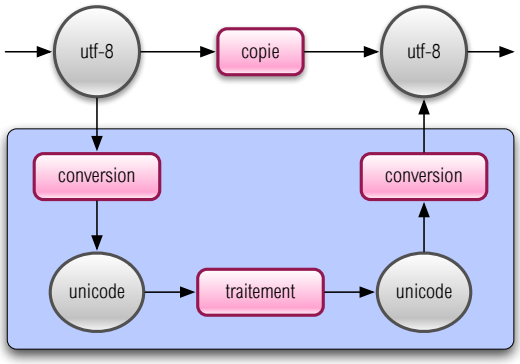
En fait, il nous faut retourner à la figure 7.1 du iAO:7.6 (p. 64), qui aurait dû nous convaincre qu'en matière de données textuelles, il y a trois aspects à considérer :

1. comment ça apparaît en externe : les *glyphes*
   forme de surface, affichable et imprimable, non représentée sur ce diagramme
2. *codage* de l'information : sa représentation interne en *chaîne d'octets*
   forme sur 8 bits, qui ne peut supporter que des manipulations simples…
   comme la copie
3. *encodage* → *chaîne unicode* pour un traitement systématique
   aspect mécanique détaillé dans l'encadré bleu

texte original [iAO:7.6, p. 63] : *un autre système de codage utilise en combinaison*

- *unicode — un codage sur 16 bits pour représenter les informations dans les programmes,*
- *utf (UNICODE TRANSLATION FORMAT) — codage de longueur variable, pour transmettre les informations.*

Notons que le diagramme est à l'inverse du texte : il y a donc lieu de le lire « à l'envers ».



Toujours est-il que si, comme préconisé, on veut pouvoir manipuler du texte sous sa forme *unicode*, il faut d'abord impérativement le convertir :

```
>>> nouveau = mot.decode('utf-8') ; nouveau              # byte string → unicode string
u'\xe9c\u0153ur\xe9'
```

où le u qui préfixe la valeur de nouveau indique que sa représentation interne est maintenant unicode ; c'est donc l'aspect 3 ci-dessus: la séquence n'a plus que 6 éléments au lieu de 9 — encore qu'un print ne montrerait (aspect 1) aucune différence entre la valeur de nouveau et celle de notre mot précédent :

```
>>> print nouveau, len(nouveau), type(nouveau)          # unicode string
écœuré 6 <type 'unicode'>
>>> print mot, len(mot), type(mot)                      # byte string
écœuré 9 <type 'str'>
```

Mais techniquement, il s'agit d'objets de deux types différents : dans une *chaîne unicode*, chaque élément correspond à un glyphe de la représentation de surface ; ce n'est donc qu'à partir de là qu'on peut traiter les éléments de la séquence de façon transparente :

```
>>> list(nouveau)                                       # ce que voit une boucle for
[u'\xe9', u'c', u'\u0153', u'u', u'r', u'\xe9']
>>> for x in nouveau : print x,                         # ce que veulent voir les humains
é c œ u r é
```

Tout le problème vient de ce que nous avons pris l'habitude de confondre la notion de *caractère* (donc de code) avec celle de *glyphe* :

1. pour nous autres, programmeurs, cette distinction est cruciale ;
2. si le texte est représenté par un objet de type *byte string*, il est manipulable par octets, mais pour certains caractères, ça n'a pas de sens
3. seuls les objets de type *unicode string* offrent une correspondance terme à terme entre les glyphes et leur représentation interne

Donc à notre stade, l'important c'est de bien se rendre compte que les programmes doivent manipuler des trucs que l'utilisateur ne voit pas, et donc de bien faire la différence entre la représentation externe des

données et leurs multiples représentations internes, interchangeables par simple transformation…

Et pour faciliter cette transformation, les chaînes de caractères PYTHON sont munies de deux méthodes de conversion, decode() et encode(), auxquelles on spécifie le codage de départ ou d'arrivée comme un CODEC : un procédé de transcodage d'un format à l'autre. En utilisant le même CODEC, la transformation est donc réversible ; ainsi :

```
>>> nouveau.encode('utf-8') == mot                    # compare deux objets de type byte-string
True
```

EN PRATIQUE

Pour revenir à la figure précédente, les boîtes étiquetées « *conversion* » sont à remplacer respectivement par « *decode* » et « *encode* » ; ce qui signifie que pour réaliser correctement le px05-2, il y aura lieu de transformer le mot avant de le passer à la moulinette ; il y a donc deux approches possibles :

1. modifier explose() en y insérant : for x in mot.decode('utf-8')…
2. convertir le mot avant de le passer à explose()

Avantages de la seconde approche :

- ça simplifie le code : explose() n'a pas besoin de prendre des précautions particulières ;
- ça met en évidence, à haut niveau, qu'on travaille avec de l'*unicode*…
  au lieu de l'enfouir au fin fond d'un fragment de code relativement inintelligible !

Naturellement, si la fonction voyelle() ne sait pas reconnaître une voyelle *unicode*, ça va poser un problème ; réglons-le tout de suite, en définissant l'ensemble des voyelles comme un littéral préfixé par un u :

```
voyelles = u'aàâéêeœèëiîoôuùû'                        # les voyelles les plus courantes en français

def voyelle(x) : return x in voyelles                 # teste la présence d'un élément dans l'ensemble
```

Sachant pertinemment que lorsque explose() invoquera voyelle(), c'est un caractère *unicode* qu'il lui passera comme argument… D'où le résultat :

```
>>> for x in explose(u'écœuré') : print len(x), x,    # pour voir les glyphes et les compter du même coup
2 cr 4 éœué
```

Comme on le voit dans les exemples ci-dessus, le préfixe u nous permet de définir directement une chaîne littérale *unicode* ; en fait, les trois expressions suivantes produisent des résultats identiques :

```
>>> u'écœuré'
u'\xe9c\u0153ur\xe9'
>>> 'écœuré'.decode('utf-8')
u'\xe9c\u0153ur\xe9'
>>> unicode('écœuré')
u'\xe9c\u0153ur\xe9'
```

Attention : on peut tout représenter en *utf-8* — et par conséquent en *unicode*, mais l'inverse n'est pas vrai… les codages antédiluviens ont, par définition, des carences irrémédiables :

```
>>> voyelles.encode('latin-1')
 UnicodeEncodeError: 'latin-1' codec can't encode character u'\u0153' in position 6: ordinal not in range(256)
```

Ah, que voilà donc une bonne raison de les éviter !

NOTES

il n'y a jamais que 2 trucs à savoir pour manipuler des listes

- comment les construire : `[10, 11]`
- comment en extraire l'information : `[10, 11][0]`

une liste peut être...
- vide : sa valeur booléenne est alors `False`
- une donnée littérale
  - faite de constantes
    ```
    a= ['allo', 'maman', 'bobo']
    b = [1, 2, 3, 'soleil']
    ```
  - assemblée à partir de variables
    ```
    c = [a, b]
    c = [b] + [a]
    ```
- la valeur d'une fonction ou d'une méthode – donc d'une construction programmée
  ```
  range(10, 20, 3)
  'le train de 18h47'.split()
  ```

on en extrait les éléments avec
- des index
  ```
  c[1][2]
  ```
- des intervalles d'index
  ```
  b[0:: 2]
  a[::-1]
  ```

on peut...
- calculer leur taille
  ```
  len(a)
  ```
- faire le compte de certains éléments
  ```
  [1, 0, 1, 0, 1, 1, 0, 1, 0, 0].count(1)
  ```
- les concaténer
  ```
  a + b
  ```
- les répliquer :
  ```
  a + b * 2
  ```
- la considérer comme un ensemble et y tester la présence d'un élément
  ```
  'soleil' in b
  ```
- en changer un élément
  ```
  a[0] *= 3
  b[2] /= 1.5
  ```
- en modifier un segment
  ```
  a[0:2] = ['hello', 'mom']
  ```
- les scanner :
  ```
  for mot in a : print a
  ```
- y appliquer une transformation à tous les éléments
  ```
  [x.title() for x in a]
  ```

et puis, si une liste ne contient que des chaînes, on peut les réunir
```
x = ['cerf', 'volant']      ; '-'.join(x)      # avec un tiret
x = ['clair', 'de', 'lune'] ; ' '.join(x)      # avec un espace
x = ['bu', 'reau']          ; ''.join(x)       # avec rien du tout
```

QUELQUES LIENS INTÉRESSANTS

lecture vivement recommandée

- UTF-8 and Unicode FAQ for Unix/Linux :
  http://www.cl.cam.ac.uk/~mgk25/unicode.html

documentation

- le module codecs accessible en python :
  http://docs.python.org/library/codecs.html#module-codecs
- pluriels irréguliers :
  http://fr.wikipedia.org/wiki/Pluriels_irréguliers_en_français

et si vous trouvez d'autres liens, signalez-les sur le forum : on les rajoutera ici...

④ DICTIONNAIRES : ACCÈS PAR CLÉ

Les dictionnaires sont une manière d'établir une relation en définissant l'association d'une clé et d'une valeur ; on peut alors accéder à cette valeur grâce à l'index que représente la clé.

Ce type de donnée, aussi appelé TABLE D'ASSOCIATIONS, existe dans de nombreux langages modernes, comme OBJECTIVE C, AWK, JAVA, ou PERL, alors que pour d'autres, plus anciens, il est nécessaire d'émuler sa mise œuvre au moyen de tables de hâchage, ou HASH TABLES.

Puisque PYTHON nous a tout préparé pour qu'il n'y ait plus qu'à se servir, nous n'aurons pas à descendre dans le détail des calculs de HASHING : nous verrons plutôt comment exploiter les propriétés de ce type d'objet pour structurer l'information afin d'en simplifier le traitement. En effet, c'est ici que nous allons prendre conscience d'un principe fondamental du traitement automatique : uniformiser la structure des données aboutit à généraliser l'algorithme qui exploite ces données, simplifiant ainsi la conception du programme.

L'objet de ce chapitre sera donc d'illustrer, par des exemples d'applications, l'intérêt d'une telle structure pour toute manipulation impliquant conversion ou traduction, opérations qui fondent, par exemple, la conception d'interprètes et de compilateurs…

On verra plus loin d'autres exemples d'objets de ce type, notamment dans les chapitres *6* et *8*, où les dictionnaires sont utilisés pour mettre en œuvre des tables de correspondances (LOOK-UP TABLES) qui permettent une exploitation simple de données complexes.

sommaire

Techniquement, un dictionnaire est un « associative array » (*arrangement d'associations*) autrement dit, une collection (non-ordonnée) où chaque élément est double : une *clé* associée à une *valeur* ; une telle collection est délimitée par les symboles { et }, tels que {} dénote un dictionnaire vide ; la clé et sa valeur sont séparées par le symbole :, et la valeur peut être elle-même n'importe quel type de donnée, y compris un dictionnaire… Par exemple :

```
boulangerie = {'baguettes' : 3, 'croissants' : 6}
superette = {'yaourts' : 2, 'autres trucs' : 4}
```

Une fois définie une telle structure, on accède à ses éléments en utilisant comme *clé d'index* l'expression qui précède le deux-points :

```
>>> superette['autres trucs']
4
>>> boulangerie['croissants']
6
```

Bien entendu, ça marche tout pareil avec des variables :

```
>>> x = boulangerie ; y = 'croissants' ; x[y]
6
```

Ce type de donnée correspond à ce qu'on appelle en Lisp une *liste d'associations* (ou a-list, pour faire *short*) et s'utilise de la même manière : avec la fonction assoc(), étant donnée une *clé* d'accès, on récupèrerait directement la *valeur* associée à cette clé.

Les données représentées par les emplettes du chapitre 3 pourraient avantageusement être définies avec une structure de dictionnaire, et voici ce que ça donnerait :

```
emplettes = {'boulanger' : boulangerie, 'superette' : superette}
```

Ici, ne perdez pas de vue que la variable boulangerie est évaluée au moment de l'évaluation de l'expression, donc qu'à la clé 'boulanger' sera associée la valeur de boulangerie ; idem pour superette…

Une expression telle que emplettes['boulanger'] retourne la valeur associée à la clé 'boulanger', c'est-à-dire {'baguettes': 3, 'croissants': 6}, et une expression à clés multiples permet d'extraire une information plus profonde :

```
emplettes['superette']['autres trucs']                # → 4
```

Évidemment, là encore, ceci fonctionne indifféremment avec des variables :

```
boutique = 'superette' ; article = 'autres trucs'
emplettes[boutique][article]                          # → 4
```

On ajoutera un nouvel élément au dictionnaire en évaluant :

```
emplettes['épicier'] = {'sucre' : '1 kg', 'sel' : '500 g'}
```

Et on modifie un élément existant en le redéfinissant, comme si c'était une variable :

```
emplettes['épicier'] = 'poivre'
x = 'boulanger' ; y = 'croissants' ; emplettes[x][y] = 6
```

Adjonction et suppression de clés (ainsi que la modification de valeurs) sont des opérations cruciales : elles font, des dictionnaires, des structures de données *dynamiques*, autrement dit, adaptables à la volée par les programmes qui les manipulent.

Or un dictionnaire a ceci de particulier qu'il représente, d'une certaine manière, un agrégat de variables (et leur définition) ; c'est un peu comme si cette structure de données permettait de définir dynamiquement de nouvelles variables, au fur et à mesure des besoins.

On verra bientôt l'application pratique de ce principe : notons que, puisque toute variable participe au contexte d'exécution d'un programme, ce qu'on appelle pompeusement *algorithme* n'est jamais qu'une procédure de *transformation* des données ; il n'y a pas d'algorithme sans données, et la simplicité de

l'algorithme (donc, son efficacité) dépend étroitement de la façon dont les données sont arrangées, ou structurées…

Un exemple tout simple (qui n'a rien à voir avec les dictionnaires) est le choix du codage des chiffres dans le code ASCII : un rapide coup d'œil au 1er diagramme de la section unicode du chapitre 3 vous montrera que si on masque les 4 bits de poids fort (à gauche) dans le code ASCII d'un chiffre, la valeur obtenue est justement la quantité arithmétique que représente ce chiffre ; ce qui, on s'en doute, contribue grandement à simplifier l'algorithme de conversion d'un chiffre en sa valeur.

Tester la présence d'un élément dans un dictionnaire se fait avec l'opérateur in, ou la méthode has_key(), comme dans l'exemple suivant :

```
'épicier' in emplettes                    # → True
emplettes.has_key('épicier')              # → True
```

Pour afficher *proprement* la valeur d'une donnée structurée complexe (liste, dictionnaire) on aura recours au module pprint :

```
from pprint import pprint                 # charge la fonction de pretty printing
pprint(emplettes)
```

Attention : un dictionnaire n'est pas une séquence, mais une simple *collection* ; et à ce titre, il n'est pas itérable à proprement parler. Cependant (et c'est cohérent avec le prédicat ci-dessus) l'emploi de l'opérateur in est possible dans une expression itérative, mais ce sur quoi on itère n'est pas le dictionnaire lui-même, c'est la liste (non ordonnée) des clés définies :

```
>>> [x for x in emplettes]
[épicier superette boulanger]
```

D'ailleurs, on peut obtenir directement, avec la méthode keys(), la liste des clés déjà définies dans un dictionnaire donné :

```
emplettes.keys()                          # → ['épicier', 'superette', 'boulanger']
emplettes['boulanger'].keys()             # → ['baguettes', 'croissants']
```

Enfin, pour détruire un élément, on évaluera, par exemple :

```
del emplettes['superette']                # instruction utilisée pour son effet : pas de valeur
emplettes.keys()                          # → ['épicier', 'boulanger']
```

…ce qui montre bien que superette n'est désormais plus défini dans le dictionnaire…

Et maintenant, une brouette d'exercices pour se faire la main :

[px14-1] créer un dictionnaire argot avec les associations : pantalon → falzar, et chemise → liquette ; y ajouter flic → poulet et chaussure → pompe ; changer la définition de flic en mettant keuf à la place de poulet ; ajouter voiture → caisse ; changer la définition de chaussure en mettant godasse à la place de pompe ; remplacer caisse par tas de boue ; coder l'expression qui retourne la liste des mots définis

[px14-2] coder une fonction unaire trouve() qui retourne la bonne définition si le mot qu'on lui donne en argument en a une dans le dictionnaire, et la chaîne vide sinon ; trouve('voiture'), par exemple, retournerait 'tas de boue' alors que trouve('ministre') retournerait None.

[px14-3] quelle serait la façon la plus simple d'inverser ce dictionnaire, de sorte que les clés soient en argot, associées à des valeurs en français ?

4.1 EXPLOITATION STATIQUE

L'exemple précédent est en fait applicable à n'importe quel problème de conversion, autrement dit quand il s'agit de passer d'une représentation à une autre ; en particulier quand cette correspondance n'est pas aisément calculable. Le dictionnaire est, dans ce cas, utilisé comme une donnée *statique* : une sorte d'état des lieux qui ne changera pas.

C'est le concept de LOOK-UP TABLE (abrégé LUT), très commode pour trouver, par exemple, la correspondance (conventionnelle) entre un *nom de couleur* et le triplet de valeurs RGB qui affichera cette couleur à l'écran (c'est à ça que sert le fichier /usr/share/X11/rgb.txt) ; mais ce n'est pas tout d'avoir les données : encore faut-il écrire le code qui exploite ces données…

Supposons qu'on ait besoin de construire un convertisseur de nombre romain en décimal : on sait qu'une expression comme {'V' : 5, 'X' : 10}['X'] extrait la valeur qui correspond à clé d'index 'X', c'est-à-dire la valeur 10 ; en d'autres termes, une telle expression joue à la fois le rôle de *look-up table*, et celui de l'algorithme de recherche de l'élément.

Je peux donc me donner une table de tous les chiffres romains ; il n'y en a que sept :

```
conv = {'C': 100, 'D': 500, 'I': 1, 'M': 1000, 'L': 50, 'V': 5, 'X': 10}
```

sachant qu'elle me procure une conversion immédiate, mais, bien sûr, ceci n'est pas suffisant pour convertir un nombre romain.

Il serait techniquement possible de construire une table de conversion pour tous les nombres romains (il n'y en a que 3999), mais s'il faut le faire par programme, autant dépenser son énergie à faire des choses moins fastidieuses, comme, par exemple, coder un convertisseur qui fonctionne pour tout nombre.

> C'est ici qu'intervient le vieux compromis entre l'espace et le temps : les données occupent de l'espace en mémoire, mais requièrent un temps de calcul nul ; inversement, l'information non encore existante occupe un espace nul, mais demande un certain temps de calcul… Si on pouvait représenter tout l'univers en machine, le temps de calcul de n'importe quelle information serait [théoriquement] zéro.

Supposons qu'on me donne un nombre R = 'xcviii' ; j'ai fait exprès ici de le donner en minuscule, pour mettre en évidence la nécessité d'harmoniser la casse des données d'entrée avec celles de la table de conversion, grâce à la méthode upper() ; ceci fait, un moulin *style fonctionnel* va me le convertir en une liste de nombres, en base 10, bien sûr :

```
R = [conv[x] for x in R.upper()]
```

Notez que si, en ce point, la liste n'avait qu'un élément, c'est qu'il n'y avait qu'un seul chiffre dans le nombre à convertir : la conversion du nombre est terminée… Par contre, si la liste a plus d'un élément, il me faut gérer la conversion d'une séquence ordonnée, où l'ordre de la valeur peut s'interpréter de deux façons :

- si $R_i < R_{i+1}$: retranche R_i du résultat ; attention : ce test n'a pas de sens quand R_i est le *dernier* élément de la liste, puisqu'il n'y a, alors, pas de R_{i+1}
- sinon : ajoute R_i au résultat

L'expression du prédicat $R_i < R_{i+1}$ empêche de coder ce traitement avec un simple for x in R, car alors il ne serait pas possibe d'accéder au x suivant… Le moulin qui calcule la valeur du résultat N doit accéder aux valeurs par leur index, donc ressemblera plutôt à quelque chose comme :

```
N = 0
for k in range(len(R) - 1) :                    # sauf R[-1]
   if R[k] < R[k + 1] : N -= R[k]               # cas de 'ix'
   else : N += R[k]                             # cas de 'xi'
```

que j'aurais pu aussi exprimer de façon plus concise – mais aussi plus cryptique :

```
for k in range(len(R) - 1) : N += -R[k] if R[k] < R[k + 1] else R[k]
```

Ceci posé, noublions pas que l'ultime élément de la liste n'a pas été traité ; et comme il est à droite de tous

les autres, il devra obligatoirement être ajouté au résultat :

```
N += R[-1]                                          # autrement dit : R[k + 1]
```

[px15-1] recoder tout ceci comme une fonction unaire RomDec() qui accepte une chaîne de chiffres romains et retourne le résultat de la conversion en décimal ; tester cette fonction avec des représentants de chaque configuration possible, comme ii, iv, vii, ix, XIII, xv, MIX, xvi, xiv, DIX, xxxii, XLII, etc.

[px15-2] coder un *script autonome* appelé romdec+ (ou rd+ pour *coller* au style unix) qui sache lire deux nombres romains — ou même plus — sur la ligne de commande et en afficher la somme à la console, en décimal.

4.2 EXPLOITATION DYNAMIQUE

Une autre façon d'exploiter un dictionnaire, c'est d'utiliser la capacité de créer dynamiquement de nouvelles entrées ; voici un tout petit programme (provenant d'un cours de *L2*) qui génère une information statistique à partir d'un texte : l'histogramme de la fréquence des caractères.

Le problème, en simplifié : je voudrais savoir combien il y a d'occurrences de chaque lettre dans un mot, par exemple, « *Mississippi* ».

Au lieu d'accéder au dictionnaire de la manière habituelle, je vais utiliser la méthode get(), qui a l'avantage de me permettre de suppléer une *valeur par défaut* si l'argument fourni n'est pas une clé du dictionnaire :

```
occurrences = { }
for lettre in 'Mississippi' :
   occurrences[lettre] = occurrences.get(lettre, 0) + 1
   print occurrences                                # pour voir au fil du traitement

print 'occurrences :', occurrences.keys()           # nombre d'entrées ainsi créées
for x in occurrences : print x, ':', occurrences[x] # les éléments du dictionnaire
```

Cette méthode get(), appliquée à un objet de type dict, peut prendre deux arguments : le 1er est bien sûr la *clé* d'accès, et le 2ème est une *valeur par défaut* si cette clé n'a pas été trouvée.

- Ainsi, au tout premier tour de la boucle for, la clé 'M' n'existe pas encore dans mon dictionnaire occurrences, et l'expression occurrences.get(lettre, 0) + 1 retournera la valeur par défaut 0 augmentée de 1 ; avec un effet secondaire (*side effect*) : maintenant, la clé 'M' de occurrences est associée à la valeur 1.
- Aux 2e et 3e tours, ce mécanisme fonctionnera de la même façon, et les clés 'i' et 's' se verront également associée une valeur 1.
- Mais ce qui est intéressant, c'est ce qui se passe au 4ème tour : la clé 's' existe déjà, donc la méthode get() retrouve la valeur 1, qu'elle augmente de 1 avant de modifier cette entrée du dictionnaire occurrences – ainsi, 's' est maintenant associé à 2.

J'ai rajouté deux autres instructions pour clarifier le résultat, mais le fait est qu'à la fin du traitement, le dictionnaire occurrences est la donnée de :

```
{'i': 4, 'p': 2, 's': 4, 'M': 1}
```

[px16-1] coder un programme similaire pour calculer le nombre d'occurrences des caractères du mot « *anticonstitutionnellement* ».

[px16-2] coder la même chose, mais sous la forme d'une fonction à un argument (un *mot*) qui retourne un nouveau dictionnaire garni des occurrences correspondant à ce *mot* ; l'essayer avec tout un tas d'exemples comme : *désespérés*, *salsifis du samedi*, *chaussette à trous*, *pastaga*, ou même *saloperie de nom d'un chien*...

4.3 TRADUCTION

Revenons maintenant à une application plus classique des dictionnaires : un programme capable de traduire des phrases entières d'une langue en une autre.

Je me donne d'abord une liste de *paires* de mots *français* + *anglais* :

```
mots = 'le the petit little chat cat boit drinks du some bon good lait milk'.split()
mots                                                    # pour voir
```

Partant d'un dictionnaire FA (comme *français/anglais*) parfaitement vide

```
FA = { }
```

je sais que pour ajouter un mot au dictionnaire, la syntaxe est très simple :

```
dictionnaire[mot] = définition
```

Tout ce qu'il me reste à faire c'est de scanner ma variable mots en prenant à chaque fois deux mots que je balance dans le dictionnaire FA, jusqu'à *épuisement* des données d'origine. Je partirai donc de l'élément 0 et progresserai de 2 en 2 jusqu'à épuisement de la liste de mots.

```
m = 0
while m < len(mots) :
    FA[mots[m]] = mots[m + 1]
    print FA                                            # preuve inutile que tout marche bien
    m += 2

FA                                                      # affiche-moi le dico
```

J'en profite ici pour présenter une nouvelle structure de contrôle : le *moulin conditionnel*. Il s'introduit avec une instruction while suivie d'un prédicat (la condition pour continuer), et d'un deux-points ; vient ensuite, indentée comme il faut, la *séquence* des instructions qui doivent être répétitivement exécutées tant que le prédicat évalue à vrai.

Ceci n'était bien sûr qu'un prétexte pour vous compliquer la vie : au lieu d'utiliser while, j'aurais pu coder la même chose avec un for en précisant, à la fonction range(), le point de *départ*, celui d'*arrivée*, ainsi que la *foulée* de progression, vulgairement appelée STRIDE ; vérifiez :

```
for m in range(0, len(mots), 2) : FA[mots[m]] = mots[m + 1]
```

Il y a, en fin de chapitre, une 3e technique pour calculer la même chose avec la fonction zip() à partir de deux listes distinctes ; essayez-là : il y a bien des cas où ça peut servir de façon immédiate…

Le résultat :

```
{'lait': 'milk', 'le': 'the', 'chat': 'cat', 'boit': 'drinks', 'petit': 'little', 'bon': 'good',
'du': 'some'}
```

est bien ce que j'attendais, ce qui fait que je peux déjà l'utiliser :

```
FA['boit']                                              # → 'drinks'
FA['chat']                                              # → 'cat'
FA['bon']                                               # → 'good'
```

Maintenant, on va voir comment s'en servir pour traduire une phrase telle que

```
phrase = 'le bon chat boit du lait'
```

D'abord, tout serait plus facile si cette phrase était une liste de mots :

```
mots = phrase.split()                                   # → ['le', 'bon', 'chat', 'boit', 'du', 'lait']
```

Puisque chaque mot figure dans le dictionnaire, une expression comme [FA[x] for x in mots] va construire une liste où chacun des mots de la phrase sera remplacé par son équivalent anglais tiré du dictionnaire :

```
trado = [FA[x] for x in mots]                           # → ['the', 'good', 'cat', 'drinks', 'some', 'milk']
```

Il ne me reste plus qu'à utiliser join() – méthode qui s'applique à une chaîne représentant le séparateur (ici, l'espace), et concatène les éléments d'une liste + le séparateur pour former une nouvelle chaîne :

```
' '.join(trado)                                         # → 'the good cat drinks some milk'
```

À ce stade, il me semble que ce que j'ai détaillé, c'est le principe-même du processus de traduction, mais comme il implique plusieurs opérations de transformation, la sagesse voudrait que j'en fasse une fonction

regroupant toutes ces lignes – en faisant bien sûr abstraction des valeurs littérales :

```
def traduis(dic, phrase) :
   mots = phrase.split()
   trado = [dic[x] for x in mots]                    # ceci traduit la liste d'un seul coup
   return ' '.join(trado)
```

Reste à l'essayer sur mon exemple :

```
traduis(FA, 'le bon chat boit du lait')              # → 'the good cat drinks some milk'
```

En fait, ce que je viens de détailler se coderait aisément en une seule expression, sans qu'il y ait besoin de recourir à des variables intermédiaires superflues :

```
' '.join([FA[x] for x in 'le bon chat boit du lait'.split()])
# -> 'the good cat drinks some milk'
```

Ceci me permet de simplifier (et donc d'accélérer) la définition de traduis() à tel point que ça en devient un « ONE-LINER » complètement cryptique pour les non-initiés :

```
def traduis(d, p) : return ' '.join([d[x] for x in p.split()])
```

Et en effet :

```
traduis(FA, 'le bon chat boit du lait')              # → 'the good cat drinks some milk'
```

Mais attention : cette définition de traduis() n'est valable que si *tous* les mots sont effectivement dans le dictionnaire. Que se passe-t-il s'il en manque un mot – si, par exemple, je voulais traduire "le chien boit du lait" ? Au moment où le programme découvre que 'chien' n'est pas une clé dans ce dictionnaire, il déclenche une EXCEPTION – terme technique désigner les circonstances qui obligent l'interprète à abandonner l'exécution du programme.

De ce fait, mon ONE-LINER tout propre et bien concis ne vaut *rien* : il me faut traiter la phrase mot par mot, et, si une exception se présente, programmer une solution. Ce retour à la case départ est extrêmement fréquent : pratiquement inévitable, même pour un programmeur chevronné, puisqu'il faut avoir prévu toutes les exceptions.

Je vais donc repartir d'une définition aussi élémentaire que possible.

```
def traduis(dic, phrase) :
   trado = []                                        # le résultat est d'abord une liste vide
   phrase = phrase.split()                           # explose la phrase
   for mot in phrase :                               # scanne-la mot par mot
      trado += [dic[mot]]                            # ajoute la traduction au résultat
   return ' '.join(trado)                            # retourne le résultat sous forme de texte
```

Mais, me direz-vous, ça ne change strictement rien : l'accès à dic [mot] va provoquer la même exception ! Bon : rassurez-vous, les objets de type dict comprennent l'opérateur in, et connaissent la méthode has_key() qui retourne *vrai* seulement si l'objet passé en argument figure bien en tant que clé dans le dictionnaire ; je pourrais donc tester :

```
if mot in dic : ...                                  # pareil que if dic.has_key(mot)
```

Et ainsi éviter l'accident… Mais le traitement du mot *Mississippi* m'a appris comment gérer les trous – à cette différence près que, comme il ne s'agit plus de compter, peu importe quelle valeur je mets par défaut :

```
def traduis(dic, phrase) :
   trado = []                                        # le résultat est d'abord une liste vide
   for mot in phrase.split() :                       # scanne la phrase explosée, mot par mot
      x = dic.get(mot, '<?>')                        # calcule une valeur
      trado += [x]                                   # traduis
   return ' '.join(trado)                            # et retourne du texte
```

Résultat des courses : il n'y a plus d'exception – que des '<?>' :

```
>>> traduis(FA, 'le gros chat boit du lait')
'the <?> cat drinks some milk'
```

Notez la syntaxe de l'expression trado += [x] : ici, ce x est l'élément que je voudrais rajouter à la liste mais l'opérateur + ne peut pas concaténer une *liste* avec une *chaîne*, il faut donc que mon x soit une liste, contrainte que j'aurais pu contourner en appliquant la méthode trado.append(x), ou simplement comme je l'ai fait en mettant x sous la forme d'une liste : [x]. C'est d'ailleurs pour la même raison que j'avais aussi, plus haut, trado += [dic [mot]]

Okay, mais en pratique, une traduction pleine de trous n'est pas très utile : je vais donc sortir de mon chapeau une nouvelle *structure de contrôle*, qui ressemble beaucoup au if ~ else déjà présenté plus haut, à ceci près qu'elle permet de traiter les *exceptions*… Il s'agit de try ~ except, une forme qui se rabat sur l'évaluation du bloc except quand l'évaluation du bloc try a déclenché une exception :

```
def traduis(dic, phrase) :
   trado = []                                          # le résultat est d'abord une liste vide
   for mot in phrase.split() :                         # scanne la phrase explosée, mot par mot
      try :                                            # essaye pour voir
         trado += [dic [mot]]                          # tout va bien ?
      except :                                         # non : un trou dans le dico
         x = raw_input('traduis-moi "%s" : ' % mot)    # demande
         dic [mot] = x                                 # rajoute au dico
         trado += [dic [mot]]                          # sers-t-en pour traduire
   return ' '.join(trado)                              # et retourne du texte
```

J'ai utilisé ici la fonction raw_input() (renommée simplement input() en PYTHON 3) qui prend un *texte* comme argument et retourne le *texte* entré *interactivement* par l'utilisateur (texte qui peut même comporter plusieurs mots). L'avantage est double, puisqu'ainsi le nouveau mot :

- est simultanément ajouté au dictionnaire
- prend naturellement sa place dans la traduction

Note : l'argument de raw_input() est ici une expression articulée autour de l'opérateur %, qui prend deux opérandes :

- à gauche, une chaîne, dite *de format*, contenant des éléments spécifiant un format typique, comme %s, spécificateur pour le type str ;
- à droite un *tuplet* de valeurs (ou de variables à évaluer), possiblement une seule.

Une nouvelle chaîne est alors formée en substituant, dans l'ordre, chaque *valeur* à chaque *spécificateur* – pour plus ample information, cf. http://docs.python.org/library/stdtypes.html#string-formatting

[px17-1] coder cette fonction, et l'essayer avec diverses phrases, comme, par exemple, traduis(FA, 'le gros chien mange la bonne soupe'), ou n'importe quoi d'autre ; et penser à afficher le dictionnaire FA pour voir si les nouveaux mots ont bien été ajoutés...

Question : maintenant que j'ai un dictionnaire français-anglais, est-ce qu'il est possible de retrouver le mot français quand on connaît le mot anglais ? Réponse : oui, mais ça risque d'être terriblement coûteux, parce qu'il faut parcourir tout le dictionnaire et examiner non pas chaque clé mais chaque valeur jusqu'à trouver celle qui nous intéresse ; et un dictionnaire grandeur nature, ça fait quand même plusieurs dizaines de milliers de mots !… Mais regardez bien ce que nous retournerait l'expression FA.keys() :

```
['lait', 'le', 'chat', 'boit', 'petit', 'de', 'bon', 'gros', 'du']
```

Ceci est la liste des *clés*, c'est-à-dire des mots définis jusqu'à présent dans le dictionnaire FA ; or, si j'évalue FA ['le'], j'obtiens 'the', donc rien ne m'empêcherait de construire l'inverse, autrement dit AF [FA ['le']] = 'le'. Je n'ai donc qu'à *scanner* la liste des clés, et pour chaque clé x, retrouver sa traduction par FA [x] pour construire progressivement AF, le dictionnaire anglais-français ; bien entendu, le dictionnaire AF doit, au départ, être vide :

```
AF = {}                                  # pourrait se dire AF = dict()
for x in FA.keys() : AF[FA[x]] = x       # remplis-le
```

Donc, bien évidemment, c'est le dictionnaire *entier* que cette expression scanne en une ligne de code, mais l'inversion se fait en une seule passe…

[px17-2] construire, à partir de FA, le dictionnaire AF comme expliqué ci-dessus, puis l'essayer avec

diverses phrases, comme, par exemple, traduis(AF, 'the big dog eats the good warm soup'), ou n'importe quoi d'autre ; et penser à afficher le dictionnaire AF pour vérifier que les *nouveaux* mots ont effectivement été ajoutés...

[px17-3] sachant que FA.keys() est la liste des entrées, et que FA.values() est celle des valeurs, voyez-vous comment les combiner avec zip() pour reconstruire le dictionnaire AF en une seule instruction ?

4.3.1 FICHIERS : OPEN, READ, WRITE

Supposons que vous ayez codé les exercices sur les dictionnaires français-anglais et anglais-français : si vous les avez fait consciencieusement, ça représente quand même un gros paquet de données, et ça serait bien commode si on pouvait conserver ces données dans des fichiers qu'il suffirait de relire pour retrouver les données intactes.

La solution est fournie en PYTHON sous la forme de la fonction open(), et des méthodes read(), write() et close().

La fonction open() prend deux arguments, tous deux de type str :
- le nom du fichier, par exemple 'azertyuiop',
- le mode d'accès, 'w', comme WRITE (*écriture*), ou 'r', comme READ (*lecture*),

et retourne un objet de type file qui représente le fichier ouvert.

La méthode write() n'est connue que par les objets de type file :
- elle ne prend qu'un argument : la donnée à écrire dans le fichier ;
- cette donnée est obligatoirement de type str, ce qui signifie que si on veut écrire un objet d'un autre type, quel qu'il soit, il faut d'abord lui donner une représentation sous forme de texte en utilisant la fonction repr() – voir l'utilisation ci-dessous ;

cette méthode n'a pas de valeur de retour.

La méthode read() n'est connue, elle aussi, que par les objets de type file :
- elle peut ne prendre aucun argument, et retourne dans ce cas la totalité du contenu du fichier sous la forme d'une chaîne – à charge pour le programmeur de mettre cette donnée sous la forme appropriée, dans la variable qui lui convient ;
- elle peut aussi prendre un nombre entier comme argument, auquel cas elle se contente de lire le nombre spécifié de BYTES (i.e. *octets*) – et la prochaine utilisation de cette méthode lira le reste du fichier depuis le point où on s'était arrêté.

Pareil pour la méthode close() : elle ne peut être utilisée que par un objet de type file, et a pour effet de clore le fichier ; notez qu'un programme qui s'arrête en laissant en plan un fichier *ouvert en écriture* empêche que le système le répertorie comme il se doit : dans le meilleur des cas, le fichier apparaîtra avec une taille *zéro*. Cette méthode n'a pas de valeur de retour.

Remarque : une fois le fichier complètement lu, toute autre tentative de lecture retournera la chaîne vide ; si on veut le relire, il faut d'abord le refermer, puis le rouvrir : la raison en est que l'objet file créé par open() sait exactement où il en est dans le fichier, et que, à moins de savoir comment manipuler cet index, le plus simple est encore de tout recommencer.

4.3.2 FICHIERS : DONNÉES NON TEXTUELLES

La méthode read() est certainement idéale pour lire un texte entier, par exemple un poème, une page HTML, ou le texte-source d'un programme. Mais un dictionnaire comme ceux qu'on a fabriqués plus haut sont des données structurées… pas vraiment du texte !

À moins que… Retournez voir à la section évaluation en mode interactif, sous-section type des variables : vous souvenez-vous de l'utilité de la fonction eval() ? Elle va nous être ici extrêmement précieuse ! Suivez-moi bien…

Supposons que j'ai un dictionnaire comme celui qui représente les occurrences dans le mot Mississippi :

```
mot = 'Mississippi'
statistiques = occurrences(mot)                    # → {'i': 4, 'p': 2, 's': 4, 'M': 1}
```

Rappel : cette fonction occurrences() était le sujet d'un exercice précédent… Je décide d'en enregistrer les résultats dans un fichier appelé 'Mississippi.stat' :

```
nom = mot + '.stat'
fichier = open(nom, 'w')                           # ouvre-le en mode write
fichier.write(repr(statistiques))                  # écris-y la valeur de statistiques
fichier.close()
```

Si je regardais le contenu de ce fichier (avec la commande SHELL cat, par exemple) voici ce que j'y verrai :

```
{'i': 4, 'p': 2, 's': 4, 'M': 1}
```

Ce n'est pas vraiment une surprise, puisque c'est la *représentation*, sous forme de texte, de mon dictionnaire, autrement dit la valeur retournée par repr(statistiques). Maintenant, si je veux le relire, ce fichier, je code :

```
fichier = open(nom, 'r')                           # ouvre-le en mode read
truc = fichier.read()
fichier.close()
```

En théorie, donc, la variable truc a maintenant comme valeur un texte constitué de tout ce qu'il y avait dans le fichier 'Mississippi.stat'… Vérifions :

```
type(truc)                                         # → <type 'str'>
truc                                               # → "{'i': 4, 'p': 1, 's': 4, 'M': 1}"
```

La valeur de ma variable truc est donc bien un texte, et bien sûr, telle quelle, cette chaîne est inutilisable, mais regardez :

```
type(eval(truc))                                   # → <type 'dict'>
d = eval(truc)                                     # ceci évalue la chaîne pour en faire un dictionnaire
d['s']                                             # → 4
```

Voilà : ce n'est pas plus compliqué que ça… Du moins, tant qu'on n'écrit, et ne relit, qu'une seule valeur – correspondant à une seule variable. Mais comment faire si on veut sauvegarder plusieurs valeurs ?

Imaginons, par exemple, que je veuille enregistrer les données de mon dictionnaire FA et celles de mon dictionnaire AF dans un même fichier : la solution consiste à séparer les valeurs l'une de l'autre en les écrivant sur deux lignes distinctes :

```
fichier = 'dictionnaires'
f = open(fichier, 'w')                             # ouverture en mode write
f.write(repr(FA) + '\n')                           # notez l'ajout du saut de ligne
f.write(repr(AF))
f.close()
```

Bien évidemment, je suis le seul à savoir ce que ces données représentent et dans quel ordre elles ont été enregistrées, donc le seul à pouvoir les relire « intelligemment »…

Et pour me faciliter les choses, je vais utiliser une autre méthode de lecture : elle s'appelle readline() et ne lit qu'une ligne à la fois – y compris le saut de ligne qui va avec.

```
fichier = 'dictionnaires'
f = open(fichier, 'r')                             # ouverture en mode read
FA = f.readline()                                  # ligne 1
AF = f.readline()                                  # ligne 2
f.close()
```

Maintenant, il faut que je reste conscient du fait que la valeur de FA est une chaîne dont le dernier caractère est un *saut de ligne*, '\n', et je devrais théoriquement m'en débarrasser avant d'utiliser eval() pour en faire la conversion en dictionnaire – mais l'expérience prouve que ce *saut de ligne* n'empêche absolument pas eval() de faire son boulot.

Attention : ce *truc* n'est *pas* documenté, ce qui veut dire que ça *pourrait* ne plus être vrai dans une version ultérieure de PYTHON. Dans ce cas, il y aura lieu de se débarrasser de ce caractère en trop en n'utilisant que FA [: -1], c'est-à-dire la chaîne sans son dernier caractère.

En attendant, je peux donc tranquillement faire mes conversions :

```
FA = eval(FA)
AF = eval(AF)
```

et vérifier que mes deux dictionnaires se retrouvent *exactement* dans l'état où ils étaient au moment de leur sauvegarde…

4.3.3 FICHIERS : PICKLE

Il existe une autre manière de sauvegarder des données : c'est d'utiliser le module pickle (en anglais, *pickle* dénote une façon de conserver les aliments). Voici le code pour sauvegarder nos dictionnaires FA et AF :

```
import pickle                                   # module standard
f = open('mes dictionnaires', 'w')              # w : mode write
pickle.dump(AF, f)                              # et de un
pickle.dump(FA, f)                              # et de deux
f.close()
```

Je reviens le lendemain, et j'évalue :

```
import pickle                                   # module standard
f = open('mes dictionnaires', 'r')              # r : mode read
info1 = pickle.load(f)
info2 = pickle.load(f)
f.close()
```

J'ai fait *exprès* de nommer mes variables différemment pour qu'il soit bien clair que seules les valeurs sont envoyées dans le fichier… Ainsi, info1 et info2 auront respectivement les valeurs de AF et FA lorsque j'en ai fait le dump() la veille…

À la différence de la technique précédente, pickle.dump() n'écrit pas du texte : il se contente de transférer dans le fichier la représentation interne de l'objet, celle qu'a en mémoire l'interprète pour l'objet en question ; et pickle.load() recharge cette représentation directement, reconstituant ainsi l'état de la mémoire pour cet objet… Il n'y a donc plus besoin d'évaluer quoi que ce soit.

4.3.4 FICHIERS : ENCORE PLUS

Il n'est pas question ici de détailler toutes les manipulations possibles sur les fichiers… mais savoir lire et écrire est le minimum qu'on puisse demander à un programmeur ; d'ailleurs, pour ce qui est des données purement textuelles, tout peut être réalisé avec les fonctions et méthodes décrites en 4.3.1 : le reste, ce sont des commodités pré-programmées qui permettent d'alléger le code et de gagner du temps…

S'il vous en faut vraiment plus, consultez la documentation :

- la section FICHIERS[10] de la documentation liste toutes les opérations possibles sur ce type d'objet ;
- la section PERSISTANCE DES DONNÉES[11] explique entre autres comment accéder à des bases de données en émulant le langage SQL ainsi que l'interface avec SQLite ;
- la section FORMATS DE FICHIERS[12] vous montrerait comment lire des données dans un format particulier comme CSV ou NETRC, mais ce n'est pas utile à notre niveau.

De toute façon, nous devrons revenir sur ce sujet dans le chapitre ⑥ lorsqu'il faudra vraiment manipuler de l'information en provenance de fichiers, en particulier pour traiter des pages web…

10. http://docs.python.org/library/stdtypes.html#file-objects
11. http://docs.python.org/library/persistence.html
12. http://docs.python.org/library/fileformats.html

4.4 INTERPRÉTATION

Toujours dans l'optique de générer dynamiquement des données structurées, l'exemple suivant montre comment réaliser un interprète pour un langage très simple : le pseudo-langage machine, ou BYTE CODE exposé au début du cours *iAO* : « *INTRODUCTION À L'ARCHITECTURE DES ORDINATEURS* », avec lequel vous avez donc probablement déjà fait connaissance.

Conçu avec une visée purement pédagogique, ce langage a [au moins] deux bonnes propriétés :
- il ressemble, dans ses mécanismes, à un authentique langage machine
- en interpréter les instructions permet véritablement de traiter de l'information

Pour cette raison, ce que je vais en faire ici a une portée bien plus large que nos habituels exercices sur des cas d'école, et les principes que j'utiliserai pour simuler l'exécution d'un programme sont généralisables à d'autres langages plus sophistiqués, tandis que la mise en œuvre que je propose n'est pas spécifique à PYTHON, mais pourrait aisément être portée en d'autres langages de programmation, comme LISP, SQUEAK, ou ANSI-C.

Pour commencer, remettons-nous dans l'ambience avec le programme du *calcul de la racine carrée par approximations successives* proposé à la section *iAO:1.2.4*, l'introduction du cours « *INTRODUCTION À L'ARCHITECTURE DES ORDINATEURS* »…

[px18-1] étant donné le programme original de la section *iAO:1.2.4*, page *10* :

```
début
   r ← 1
   répéter
      r ← (r + n / r) / 2
fin
```

le coder convenablement en PYTHON et le faire tourner ; limiter les répétitions ;

Ceci fait, concentrons-nous maintenant plus précisément sur le BYTE CODE listé page *13*, dans la sous-section TRADUCTION DU PROGRAMME. Le code comporte *6* instructions, toutes sur *4* octets, la première démarrant à l'adresse arbitraire 2997.

Une autre façon de considérer ce programme serait sous la forme d'une table :

adresse	opcode	arg 1	arg 2	arg 3	commentaire
2997	1	1	1000	0	r initial
3001	1	2	1020	0	diviseur
3005	2	2000	1000	1010	n / r
3009	4	2000	1010	1010	r + n / r
3013	2	1010	1020	1000	(r + n / r) / 2
3017	3	3005	0	0	branchement

Ce point de vue m'intéresse parce qu'il correspond justement à la structure d'un dictionnaire. Ainsi, pour représenter ce programme, on commencera par définir un dictionnaire vide, appelé m comme *mémoire*, qu'on remplira instruction par instruction, comme ci-dessous :

```
m = {}                                  # départ à vide
m[2997] = [1, 1, 1000]                  # définition du r initial
m[3001] = [1, 2, 1020]                  # définition du diviseur
m[3005] = [2, 2000, 1000, 1010]         # n / r
m[3009] = [4, 2000, 1010, 1010]         # r + n / r
m[3013] = [2, 1010, 1020, 1000]         # (r + n / r) / 2
m[3017] = [3, 3005]                     # branchement
```

Remarquez ici que :

- il ne serait pas possible d'ajouter une clé à un dictionnaire qui n'existe pas
- la clé d'accès aux données d'une instruction *est* l'adresse de cette instruction
- les données des instructions sont mémorisées sous forme de listes

- certaines instructions n'utilisent pas vraiment 4 octets, ce qui permet d'en simplifier la représentation
- les deux premières instructions devraient manipuler des nombres réels, sinon, en 3005 et 3013, les divisions seraient entières

Il va nous falloir aussi simuler le registre pc (comme PROGRAM COUNTER) qui devra contenir au départ l'adresse du début du programme, c'est-à-dire 2997. On va, lui aussi, l'intégrer aux données de m, rien que pour garder ensemble tous les éléments de la simulation :

```
m['pc'] = 2997
```

À ce stade, notre dictionnaire m contiendrait donc :

```
{  2997 : [1, 1, 1000],
   3001 : [1, 2, 1020],
   3005 : [2, 2000, 1000, 1010],
   3009 : [4, 2000, 1010, 1010],
   3013 : [2, 1010, 1020, 1000],
   3017 : [3, 3005],
   'pc' : 2997 }
```

Notons que, comme dans le cas ci-dessus, rien n'empêche d'avoir des clés numériques en même temps que des clés alphabétiques. En fait, un dictionnaire peut accepter n'importe quelles clés, à la seule condition qu'elles soient immuables *par nature*, comme c'est justement le cas pour les nombres ou les chaînes – entre autres.

Les clés 2997 à 3017 nous donnent donc un accès direct aux instructions du programme, et il ne manque qu'une chose pour que ce programme soit véritablement exécutable : que les codes d'opération effectuent l'opération en question… Pour ce faire, on va définir une fonction pour chaque code d'opération : pour chaque fonction, il faut, évidemment, autant de paramètres que d'opérandes dans l'instruction.

L'écriture d'une valeur en mémoire, codée 1, requiert une valeur à écrire et une adresse à laquelle écrire ; et ça, on sait faire :

```
def affecte(valeur, adresse) : m[adresse] = valeur
```

Ainsi, l'évaluation de l'expression affecte(1, 1000) provoquera la création d'une nouvelle entrée du dictionnaire m pour la clé 1000, qui sera donc associée à la valeur 1 ; idem pour la 2e instruction qui créera la clé 1020 associée à la valeur 2. Si tout fonctionne comme attendu, après exécution des deux premières instructions, notre m aura été dynamiquement augmenté de deux nouvelles clés : {1000 : 1, 1020 : 2}.

La division, codée 2, demande 3 paramètres – la division du premier par le second générant une nouvelle valeur qui doit être mémorisé à l'adresse spécifiée par le troisième :

```
def divise(x, y, adresse) : m[adresse] = m[x] / m[y]
```

L'addition, codée 4, serait tout à fait similaire :

```
def ajoute(x, y, adresse) : m[adresse] = m[x] + m[y]
```

Quant au branchement, codé 3, c'est tout bêtement une altération du registre pc :

```
def encore(adresse) : m['pc'] = adresse
```

Gardez bien conscience que, comme on l'avait déjà vu en *iLP:1.2*, APPLICATION DE FONCTIONS, affecte, divise, ajoute et encore sont des objets d'un type particulier : type(affecte) → <type 'function'>. On va se donner une autre table où chaque code d'opération sera associé à l'opération elle-même, toujours au moyen d'un dictionnaire :

```
op = {1 : affecte, 2 : divise, 3 : encore, 4 : ajoute}
```

Notons que les valeurs pour chaque clé ne sont pas des chaînes de caractères mais des références (vulgairement appelées *pointeurs*) aux fonctions elles-mêmes. Si maintenant, en mode interactif, on demandait à PYTHON d'évaluer op, sa réponse montrerait la véritable nature des valeurs associées : des

adresses, mille tonnerres !

```
>>> op
{1 : <function affecte at 0x717f0>, 2 : <function divise at 0x71830>, 3 : <function encore at
0x718b0>, 4 : <function ajoute at 0x71870>}
```

Bon… Il est maintenant possible de définir un *séquenceur* qui simule l'exécution de ce programme :

```
def run(fois) :
   for n in range(4 * fois) :                    # itération sur les instructions
      a = m['pc']                                # compte tenu du décodage
      apply(op[m[a][0]], m[a][1:])               # valeur du program counter
      if m[a][0] != 3 :                          # exécute l'opcode
         m['pc'] += 4                            # sauf dans le cas d'un branchement inconditionnel
   return m[1000]                                # incrémente le program counter
                                                 # retourne la valeur calculée
```

En fait, ce séquenceur n'est jamais qu'une bête itération sur le décodage et l'exécution de chaque instruction, compte tenu de la valeur courante du registre pc :

- le paramètre fois contrôle le nombre d'itérations
- la variable a (comme *adresse*) est la valeur du pc, extraite du dictionnaire m
- m[a] est donc la liste représentant l'instruction à l'adresse a
- m[a][0] est le code de l'opération, et le reste, m[a][1:], c'est les opérandes
- l'expression op[m[a][0]] retourne la fonction qui correspond au code opération m[a][0]
- pour émuler l'exécution, on utilise ici apply(), qui applique[13] cette fonction à ses arguments, représentés par le reste de la liste.
- et bien entendu, après chaque exécution, le registre pc doit être incrémenté de 4 pour pointer sur l'instruction suivante — sauf dans le cas d'un branchement, code 3…

Ainsi, grâce à apply(),tout se passe comme si run() construisait, à la volée, des *appels de fonctions* :

```
affecte(1, 1000)
affecte(2, 1020)
divise(2000, 1000, 1010)
ajoute(2000, 1010, 1010)
encore(3005)
```

…effectuant, par là-même, l'exécution de l'instruction qu'il vient de décoder.

[px18-2] codez ce programme et faites-le tourner en appelant la fonction run() ; celle-ci prend un argument : le nombre de tours à faire avant d'arrêter le calcul.

- Remarquez que ce nombre de tours va être multiplié par 4 pour tenir compte du fait que ce qui est représenté comme une *simple instruction* en haut de la section iAO:1.2.4 demande en réalité quatre instructions en BYTE CODE… à titre indicatif, il faudrait 13 tours pour obtenir la racine de 10^4.
- Par ailleurs, il y a quelque chose qui n'est donné nulle part, c'est le nombre dont il faut extraire la racine carrée : il doit absolument être défini dans m avec la clé d'index 2000.
- Enfin notez que toutes les divisions devraient être effectuées sur des nombres réels – et non des entiers, comme semble le suggérer l'original : corriger les données, sinon ça ne marchera pas comme il faut – ni même du tout.

[px18-3] adapter ce même simulateur pour interpréter le programme de l'exercice *1.6*, page *14* du même document.

[px18-4] adapter ce même simulateur pour interpréter le programme de l'exercice *1.7*, page *15-16* du même document.

De cette section, il n'y a rien d'autre à retenir que le principe : un programme qui exploite un dictionnaire de fonctions pour interpréter un programme représenté par un dictionnaire.

13 notez que apply() en PYTHON a exactement le même comportement qu'en LISP : essayez (apply '+ '(2 3)) pour le vérifier.

RÉFLEXIONS

simulation ou émulation

un simulateur se contente de faire de l'effet : vu de l'extérieur, il donne une authentique impression d'avoir affaire à l'original parce qu'il produit les mêmes effets dans les mêmes circonstances – simuler une bonne grippe n'est pas trop difficile ;

un émulateur travaille un cran plus bas, en simulant les fonctions qui produisent l'effet : en introduisant des virus morts, le vaccin contre la grippe déclenche les mêmes réactions immunitaires qu'un virus vivant…

les techniques d'émulation peuvent descendre à un niveau de réalisme tel que le processus simulé ne peut plus être discerné de l'original : le programme VirtualBox de SUN MICROSYSTEMS simule tellement bien une véritable machine qu'il est possible d'y installer, et d'y faire tourner, un autre système d'exploitation ;

c'est l'un des créneaux actuellement les plus porteurs de la technologie informatique : virtualiser la réalité ouvre des horizons qui débouchent sur la modélisation, autrement dit la conception de maquettes, de modèles théoriques (plus ou moins réducteurs) de mécanismes physiques, promouvant ainsi l'approche expérimentale dans tous les domaines : robotique, chirurgie, météo, génétique, fin du monde…

interprètes et compilateurs

la technologie du traitement de l'information évolue très vite parce que c'est, justement, un univers où tout est possible, à condition de l'inventer ; la véritable difficulté vient de la capacité à exprimer l'invention, et qui dit « expression » dit « langage » : voilà donc pourquoi on invente de nouveaux langages, et des programmes pour les interpréter – la section précédente était en fait une introduction aux techniques de base de l'interprétation de programmes.

ce thème est suffisamment important pour faire l'objet d'un chapitre entier en iAO, architecture, mais aussi ici, en iLP:8.4, et avec l'approche impérative, en 213:8 ; en 3e année, c'est un cours entier qui lui sera consacré ; d'ici là, il reste un peu de temps pour digérer cette section iLP:4.4…

pour le moment, tout ce qu'il faut retenir, c'est le nécessaire compromis entre

- la complexité des structures de données
- la simplicité du programme proprement dit

autrement dit, données et programmes sont des vases communicants…
simplifier l'un revient à complexifier l'autre – et vice versa

l'ultime simplification du programme aboutit à des langages où il n'y a plus que des données ; c'est pour cette raison qu'on dit qu'ils sont déclaratifs :

- c'est le cas de PROLOG, où les données constituent le programme
- c'est aussi le cas de LISP, qui permet de manipuler les programmes comme si c'était des données
- d'autres langages ont été inventés pour programmer de façon radicalement différente : par exemple ACTOR, qui modélise un micro-univers où la programmation consiste à laisser interagir des « agents » qui s'échangent des messages et les propagent aux autres membres de la « société » – programmer, c'est alors définir les bons agents, et les réponses adéquates aux messages…

mais si on y réfléchit bien, tous les langages sont déclaratifs, puisque ce sont des données interprétées par un moteur, que ce soit au niveau matériel, le processeur, ou au niveau logiciel : l'interprète, le compilateur… tout programme est en réalité des données pour un autre programme, fût-il réalisé avec de la logique digitale.

RÉCAPITULATION

dictionnaire : collection de données, de la forme clé → valeur

il n'y a jamais que 2 malheureux trucs à savoir pour exploiter des dictionnaires

- comment les construire : `{'un' : 'one', 'deux' : 'two'}`
- comment en extraire l'information : `{'un' : 'one', 'deux' : 'two'}['un']`

– un dictionnaire est indexé par ses clés
– la clé permet d'accéder à la valeur
– rien n'empêche que la valeur soit elle-même un dictionnaire

un dictionnaire peut être...

– vide : `{}`
  la fonction `dict()`, sans argument, crée un objet neutre de type `'dict'`, vide
  `bool(dict())` # → `False`
– la donnée littérale des couples clé → valeur
  `{'duo' : 2, 'uno' : 1}`
– construit par programme
  `espagnol = dict(zip(['uno', 'duo'], [1, 2]))` # → `{'duo' : 2, 'uno' : 1}`

à partir d'un dictionnaire, on peut...

– calculer sa taille
  `len(espagnol)` # → `2`
– y tester la présence d'une clé
  `'quatro' in espagnol` # → `False`
– y définir une nouvelle clé
  `espagnol['quatro'] = 4` # → réessayer l'expression précédente
– en supprimer une clé :
  `del espagnol['uno']`
– en extraire la liste des clés :
  `espagnol.keys()`
– en extraire la liste des valeurs :
  `espagnol.values()`

en fait, nous manipulons des dictionnaires depuis le tout début, sans le savoir... par curiosité, définissez donc quelque chose comme `azertyuiop = 789`, puis regardez la valeur de la fonction `locals()` : c'est la même que celle de `globals()` – sauf pendant l'évaluation d'une fonction, où `locals()` retournerait le dictionnaire des objets définis dans cette fonction, autrement dit son contexte local...

un dictionnaire ne peut pas avoir deux mêmes clés pour deux valeurs distinctes : les clés constituent un ensemble (au sens mathématique du terme) où chaque élément est unique ; et voilà pourquoi, dans un contexte donné, une variable ne peut représenter qu'une valeur !

fichiers

il n'y a que trois principes à appliquer pour utiliser un fichier :

– il faut l'avoir ouvert :
  `open()` en mode `r` ou `w`
– on peut alors y lire ou y écrire quelque chose :
  `read()` ou `write()`
  `load()` ou `dump()` – à partir du module `pickle`
– il faut le refermer :
  `close()`

⑤ INTERFACES : FENÊTRES ET BOUTONS

Comprendre ce qui suit requiert les concepts de TYPE et de MÉTHODE, respectivement évoqués aux sections 1.2 et 2.5 : tout comme 123 et 'abracadabra' sont des objets de types différents, les objets graphiques sont eux aussi organisés en types, et sont munis de méthodes : certains objets sauront donc faire des trucs qui ne seront simplement pas possibles avec d'autres…

La plupart des applications modernes mettent en œuvre la métaphore des fenêtres : en fait, une fenêtre n'est jamais qu'une zone rectangulaire (avec une image dedans) qui délimite la « sensibilité » du programme ; ce rectangle est appelé interface, et tout ce qu'on peut y faire, avec la souris ou le clavier, est considéré par le programme comme un événement ; en dehors de cette fenêtre, c'est autre chose, et notre programme y est insensible.

L'intérêt des fenêtres c'est qu'elles ont une dynamique, du fait-même qu'il y a un programme derrière, et que ce programme peut réagir en fonction des événements ; cette réaction peut prendre de nombreuses formes, de l'affichage d'un texte à l'extinction de la machine, en passant par la lecture d'un fichier mp3, ou le lancement d'une autre application dans une autre fenêtre…

Le code du programme n'a pas besoin d'avoir conscience qu'il tourne derrière une interface graphique : nous verrons un exemple qui utilise le programme de calcul du pluriel présenté précédemment, sans en modifier une ligne de code. Autrement dit, pour réaliser une interface graphique, il suffit d'ajouter quelques instructions à un programme existant.

Cependant il est des cas ou la conception du programme s'appuie au contraire sur le fait que les données à manipuler proviennent de l'interaction avec l'utilisateur : et ça, c'est ce qu'on appelle EVENT-DRIVEN PROGRAMMING, ou programmation pilotée par les événements.

sommaire

Le document http://www.pythonware.com/library/tkinter/introduction/ est la référence sur laquelle je me suis appuyé pour élaborer les programmes qui suivent ; en principe, elle s'applique à Tkinter 8.0.4 et Python 1.5, sous LINUX et le X WINDOW SYSTEM, mais elle est toujours valable pour Python 2.6. Un autre document http://infohost.nmt.edu/tcc/help/pubs/tkinter/intro.html présente la même chose avec une approche différente : c'est toujours bien d'avoir deux points de vue sur le même sujet.

Tkinter est un module graphique dont l'architecture s'inspire du modèle CLIENT/SERVEUR : les ressources du SERVEUR sont concentrées en un point, et mises à la disposition du CLIENT sur simple demande[14]. Ce module définit un ensemble d'objets (sous forme de classes) qu'il n'y a plus qu'à instancier pour créer les éléments de l'interface graphique.

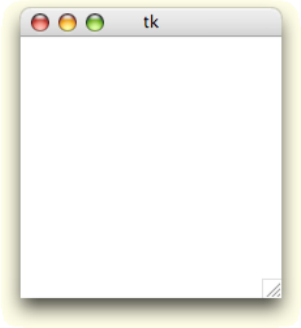
Techniquement, une fenêtre est un *widget* (raccourci pour WINDOW GADGET) qui peut contenir d'autres *widget*s, par exemple des boutons, de type Button, des zones de texte, de type Label, des champs éditables, de type Entry, des images, des menus, etc.

5.1 LABEL

Commençons par un script très simple, capable d'afficher un message fixé à l'avance, par exemple « coucou » : il me faut d'abord créer un premier *widget conteneur* (la fenêtre), qui contiendra un deuxième *widget*, le message.

Puisque PYTHON est un interprète, il est possible d'évaluer ce programme ligne par ligne dans la fenêtre de terminal. Appelons d'abord l'interprète, et importons le module graphique :

```
>>> from Tkinter import *
```



Rien ne se passe… et c'est normal puisque nous n'avons rien fait que charger un module ; mais dès qu'on entre la deuxième ligne, on peut voir une fenêtre vide, dont la taille *par défaut* n'a pas encore été ajustée…

```
>>> top = Tk() # fenêtre-maître
```

La ligne suivante n'a aucun effet apparent : l'instance message de Label a bien été créée, mais elle ne sera pas affichée tant qu'on appellera pas sa méthode pack() ; et en effet, dès l'entrée de la 4e ligne, la fenêtre top recalcule ses dimensions, et s'adapte à la taille du texte de message.


```
>>> message = Label(top, text='coucou', fg='pale violet red')
>>> message.pack()
```

À ce stade on peut fermer la fenêtre graphique, ou en faire une deuxième en répétant les mêmes instructions, ou encore arrêter PYTHON avec contrôle D… L'important ici, c'est le principe que toute instance de Tk se réalise comme une nouvelle fenêtre.

Voilà donc nos 4 lignes de code :

```
from Tkinter import *
top = Tk()                                              # instance de la classe Tk
message = Label(top, text='coucou', fg='pale violet red')
message.pack()
```

et leur explication détaillée :

1. charge le module Tkinter qui permet la création de *widget*s ;
2. crée une instance de la classe Tk, le *widget* top qui sera un conteneur pour les autres *widget*s que j'y placerai ;
3. crée le *widget* message, dans le *widget* top ; message est ici de type Label, ce qui lui donne potentiellement les attributs des objets de ce type ; ici, je n'en utiliserai que deux, son *texte* et sa

14 En réalité, ce module est un FRAMEWORK : le client ne demande rien, il se contente de définir dans son programme des CALLBACKS, c'est-à-dire des ressources qui seront exploitées par le FRAMEWORK quand survient un événement.

couleur d'avant-plan – fg est l'abrégé de FOREGROUND ;

4. compacte le *widget* message en haut et au centre de la fenêtre : il s'agit ici de calculer la taille du texte à afficher, et d'adapter la taille de la fenêtre à la taille du *widget* ;

[px19-1] effectuer la manipulation ci-dessus ; noter que la création d'une nouvelle fenêtre ne détruit pas l'ancienne – on s'en débarrassera en quittant l'interprète, mais on peut aussi cliquer sur le *bouton de fermeture*, ou taper alt+F4.

[px19-2] remplacer le texte du *widget* message par un texte plus long, éventuellement même sur plusieurs lignes, comme, par exemple, "hé ho…\nça va comme ça ?" ; observer comment la fenêtre s'adapte à la nouvelle taille du texte grâce à la méthode pack().

[px19-3] créer le script contenant le texte de ce programme, y ajouter une première ligne de commentaire pour spécifier l'interprète requis ; changer son *mode d'accès* (avec la commande chmod) pour le rendre exécutable, et lancez-le depuis le shell ; et si votre message n'est pas de l'ASCII pur, ne manquez pas d'en spécifier l'encodage…

Notez que les classes sont traditionnellement nommées avec une majuscule initiale, alors que les instances sont en minuscules ; pure convention, à laquelle vous pouvez déroger, au risque de rendre votre code plus difficile à lire pour les autres programmeurs ; remarquez aussi que le nom de mes variables, top et message, ne font rien à l'affaire : j'aurais pu aussi bien les appeler machin et truc…

Autre convention, dont la cohérence m'échappe : lorsqu'on spécifie des valeurs dans les arguments d'une méthode – typiquement, x = Label (f, text='Coucou !', fg='red') – on “colle” l'opérateur = à ses opérandes, sans espace ; je ne sais pas pourquoi : c'est ridicule, et ça ne rend certainement pas le code plus intelligible, mais bon… ce n'est jamais qu'une convention.

Techniquement, les argument de la forme x=y sont possibles quand x est un paramètre muni par définition d'une valeur par défaut ; par exemple, la définition suivante :

```
>>> def grogne(fois=2, r ='r') : return 'G' + fois * r
```

permet d'appeler grogne() comme si elle était 0-aire, 1-aire, ou 2-aire, et même en inversant l'ordre des arguments… à condition de nommer le paramètre :

```
>>> grogne()
'Grr'
>>> grogne(4)
'Grrrr'
>>> grogne(r='R')
'GRR'
>>> grogne(r='R', fois=5)
'GRRRRR'
```

Cette commodité est disponible dans beaucoup de langages modernes : OBJECTIVE C, C++…

5.2 BUTTON

Nous allons, en deux instructions, rajouter un bouton à cette interface ; la syntaxe est similaire à celle de la création d'un Label :


```
bouton = Button(top, text='La ferme !', command=top.destroy)
bouton.pack()
```

où l'appel de la méthode pack() joue le même rôle que précédemment : calculer la taille du nouveau *widget* pour le compacter après le *widget* existant, toujours au centre par défaut.

La différence intéressante, c'est qu'à un bouton je peux associer une *commande* qui sera appelée au moment où, avec la souris, je clique sur le bouton : ici, je me contenterais d'une méthode existante pour faire simple, mais nous verrons plus tard qu'on peut associer à un bouton n'importe quelle fonction qu'on veut bien définir — et même différencier selon qu'il s'agit d'un clic gauche, centre, ou droit, et que le bouton a été relâché ou non…

En fait, j'ai besoin de non pas 2, mais 3 instructions : la troisième est indispensable pour faire de mon

programme un véritable PLUG-IN, et c'est l'appel à la méthode mainloop().

```
from Tkinter import *
top = Tk()                                                    # fenêtre-maître
message = Label(top, text='coucou… ça va ?', fg='red')
message.pack()
bouton = Button(top, text='ferme-la !', command=top.destroy)
bouton.pack()
top.mainloop()
```

Par contraste avec le programme précédent, un clic sur le bouton détruit la fenêtre, arrêtant, du même coup, l'application. Le texte du bouton ne fait rien à l'affaire ; on aurait pu lui coller l'étiquette « *bon arrête, ça suffit comme ça* », du moment que son attribut command reste lié à la méthode destroy().

Et encore une fois, peu importe le nom de mes variables dans la mesure où je reste cohérent ; le code ci-dessous donnerait exactement le même résultat avec une seule variable, f : en fait, message et bouton ne servaient qu'à permettre de décomposer en deux instructions, la définition et l'appel de la méthode pack(), mais c'était, à vrai dire, tout à fait inutile…

```
from Tkinter import *
f = Tk()
Label(f, text='Coucou, patron !', fg='red').pack()
Button(f, text='La ferme !', command=f.destroy).pack()
f.mainloop()
```

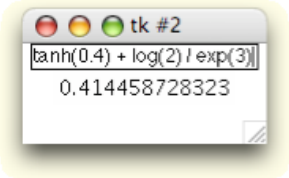
[px20-1] coder ce programme, ligne par ligne, en mode interactif ; expérimenter en modifiant le texte du Label et celui du Button.

[px20-2] changer la couleur du texte en, par exemple, '*dark blue*' ; intervertir l'ordre des définitions du Label et du Button : l'ordre change-t-il l'apparence de la fenêtre ?

Remarque : la liste des couleurs légales se trouve quelque part sur votre disque ; chez moi, c'est rgb.txt, c'est-à-dire [/usr/share/X11/rgb.txt] ; vous devriez aussi pouvoir trouver cette liste sur le web…
par exemple http://web.njit.edu/~kevin/rgb.txt.html ou bien http://sedition.com/perl/rgb.html.

5.3 ENTRY

Nous allons maintenant créer une interface minimale pour une calculette capable d'effectuer n'importe quelle opération mathématique – incluant les fonctions transcendantes comme l'exponentielle, le logarithme, ou le cosinus hyperbolique. Le truc consiste à exploiter la capacité de PYTHON à évaluer du code à la volée, puisque la fonction eval() accepte une chaîne de caractères et calcule la valeur qu'aurait cette expression si on l'avait entrée directement sous l'interprète…



Pour que l'utilisateur puisse entrer l'opération à effectuer, nous aurons besoin d'un *widget* de type Entry, alors que le résultat du calcul sera affiché dans un *widget* de type Label.

Pour afficher ce résultat, nous définirons une fonction valeur() dont le boulot peut être décomposé en ces quatre étapes :

- récupère, avec la méthode get(), le texte de l'expression tapée dans express, le *widget* Entry
- évalue cette expression avec eval(), pour produire une valeur numérique
- convertit ce nombre en texte avec str()
- passe ce texte à val, le *widget* Label, pour l'afficher comme résultat

Ce dernier point demande une explication : on a vu comment on pouvait définir le texte d'un objet de type Label à sa création, mais comment faire pour changer ce texte après qu'il ait été créé ? La réponse est la méthode configure() qui prend comme arguments le nom des attributs à changer et leur nouvelle valeur : exactement comme quand on définit une instance de Label.

Cette fonction valeur() sera un HANDLER lié par bind() et appelé par le FRAMEWORK avec un argument, l'objet qui constitue l'événement qui a provoqué le CALLBACK ; d'où le paramètre event inutilisé :

```
def valeur(event) : val.configure(text=str(eval(express.get())))
```

Ensuite, il nous faut importer le module math qui nous donnera effectivement accès à toutes les fonctions mathématiques standard :

```
from math import *
```

La troisième ligne du programme ne nous surprend pas vraiment :

```
from Tkinter import *
```

Et puis, après définition de top, la fenêtre-maître, et express, destiné à accueillir l'expression, la fonction valeur() sera liée à la touche RETURN en tant que HANDLER pour ce dernier *widget*.

```
top = Tk()

express = Entry(top)
express.bind("<Return>", valeur)
express.pack()

val = Label(top)
val.pack()

top.mainloop()
```

Remarquer la nature des arguments de la méthode bind() ; le premier est une chaîne, un nom d'événement, tandis que le symbole valeur est utilisé ici en tant que référence, exactement comme quand on avait construit une table de fonctions pour l'exercice [px18-2].

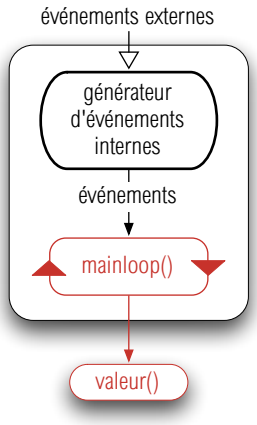
Cette fonction valeur() sera invoquée comme l'était la méthode destroy() utilisée plus haut : on l'appelle un CALLBACK parce qu'elle est appelée par mainloop(), le gestionnaire d'événements. Or mainloop() invoque les HANDLERS liés par bind() en leur passant implicitement comme argument l'objet qui représente l'événement ; c'est pour cette raison que j'ai dû mettre un paramètre à la fonction valeur(), même si moi je n'en fais rien… pour l'instant.

On avait, plus haut, présenté Tkinter comme une sorte de serveur, mais le mécanisme de la méthode bind() montre qu'il s'agit véritablement d'un FRAMEWORK :

> le client ne demande rien, il se contente de définir dans son programme des CALLBACKS, c'est-à-dire des ressources qui seront appelées par le FRAMEWORK quand surviendra un événement.

Ici, la fonction valeur() se comporte comme un PLUG-IN utilisé par Tkinter pour gérer l'événement externe « enfoncement de ↵ », transformé en événement interne, un objet, par le générateur.

Voilà pourquoi un HANDLER ne peut être utilisé que pour son effet, et ne retourne pas de valeur…



On vient de voir qu'il y aurait deux manières de donner à une fonction le statut de HANDLER :

- en la liant en tant que valeur de l'attribut command
- en la liant avec la méthode bind()

Dans le premier cas, le HANDLER peut gérer des événements multiples, tels que button-press suivi de button-release : un tel HANDLER ne prend pas d'argument du tout (lequel devrait-il prendre), et sa définition est donc celle d'une fonction 0-aire.

Dans le deuxième cas, le HANDLER sera invoqué avec, comme argument, l'événement qui a déclenché le CALLBACK : un tel HANDLER doit obligatoirement être défini avec un paramètre qui représente l'événement en question, qu'il soit utilisé ou non…

[px21-1] coder et exécuter le programme complet ; entrer une expression dans le champ de texte du

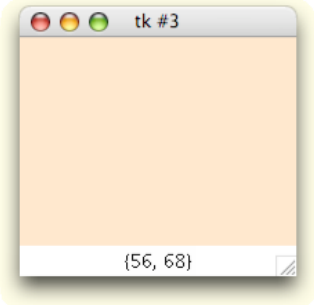
widget Entry, et la réponse s'affichera dans le *widget* Label ; toutes les opérations arithmétiques habituelles et toutes les fonctions du module math[15] sont disponibles ; en cas d'erreur de syntaxe, une exception (avec TRACEBACK) sera affichée dans le terminal : c'est au programmeur de blinder son programme.

[px21-2] modifier ce programme pour qu'il calcule le *pluriel d'un mot* et non la valeur d'une expression mathématique : au lieu d'importer le module math, ajouter simplement votre définition de la fonction pluriel(), et arrangez-vous pour qu'elle soit invoquée quelque part dans le handler valeur() à la place du mécanisme qui, jusque là, évaluait l'expression mathématique…

[px21-3] simplifier la définition de la fonction valeur(), sachant que la fonction str() est devenu redondante – encore que bien qu'inutile, elle n'est pas vraiment nuisible, puisque str() appliquée à une chaîne retourne cette chaîne sans altération aucune...

5.4 FRAME

On comprend qu'on puisse cliquer sur un bouton pour déclencher une action, mais comment l'interprète sait-il que c'est là qu'on a cliqué, et pas ailleurs ?



Dans un but expérimental, imaginez une application où la fenêtre entière serait une zone sensible aux clics de la souris :

chaque clic de souris dans cette zone constitue un événement capable de déclencher l'appel d'une fonction qui recevra les coordonnées de cet événement.

- pour mettre ceci en évidence, je vais matérialiser, par sa couleur, un conteneur Frame créé dans le conteneur fenêtre
- et pour vérifier où j'ai cliqué, j'afficherai l'abscisse et l'ordonnée du clic juste en dessous, dans un Label

```
def position(click) : afficheur ['text'] = '%s, %s' % (click.x, click.y)

from Tkinter import *

top = Tk()

zone = Frame(top, width=200, height=150, bg="bisque")
zone.bind("<Button-1>", position)
zone.pack()

afficheur = Label(top)
afficheur.pack()

top.mainloop()
```

La fenêtre top contient donc ici deux *widget*s :
- zone délimite la zone cliquable
- afficheur n'est là que pour afficher les coordonnées du clic

Le *widget* zone, dont les dimensions sont fixées par les paramètres width et height, définit une association entre le *bouton gauche* de la souris et une fonction position(), qui, de ce fait, ne gère que les événements provoqués par le bouton gauche. Remarquez que la fonction position() doit déjà exister au moment du LIAGE : c'est pour ça que je l'ai définie en haut du programme, mais j'aurais pu aussi placer cette définition *juste avant* d'appeler la méthode bind()…

La fonction position() est ce qu'on appelle techniquement un HANDLER : dans cette définition, le paramètre nommé click représente l'événement envoyé au HANDLER par mainloop() au moment où on clique ; un événement est en fait un *objet muni de propriétés* : en particulier, les valeurs numériques entières correspondant à l'abscisse et l'ordonnée d'un tel événement.

Alors l'expression '%s, %s' % (click.x, click.y) – comme on l'avait vu précédemment dans la section 4.3 TRADUCTION – convertit en texte ces deux valeurs, de sorte qu'il ne reste plus qu'à l'utiliser pour modifier le

15 évaluer help(math) pour voir la documentation des fonctions importées — arrêt par la touche q

texte du Label, l'objet afficheur.

Comme vous l'avez peut-être déjà deviné, tous ces objets ont une représentation interne de type dict, ce qui explique qu'on puisse manipuler directement la valeur associée à la clé 'text' de afficheur… comme d'ailleurs n'importe quelle clé de la liste retournée par afficheur.keys().

Une autre façon plus standard de faire la même chose serait d'utiliser la méthode configure() dont l'avantage est qu'elle permet de modifier, en une seule invocation, de multiples paramètres.

Remarquez au passage que, contrairement à l'usage dans les représentations orthonormées, l'origine (0, 0) se trouve en haut à gauche, et que l'ordonnée, y, grandit vers le bas ; par contre l'abscisse, x, grandit vers la droite, comme d'habitude : notez bien que jamais l'abscisse, ni l'ordonnée ne peuvent être négatives !

[px22-1] coder ce programme ligne par ligne, en mode interactif ; expérimenter en changeant les dimensions de la zone, et sa couleur, et en liant <Button-2> ou <Button-3> plutôt que <Button-1>.
[px22-2] ajouter un bouton pour stopper l'application.
[px22-3] définir un second LIAGE pour zone, avec <Button-3> et un HANDLER légèrement différent, qui afficherait les coordonnées en hexadécimal (%x et non %s) pour rigoler.

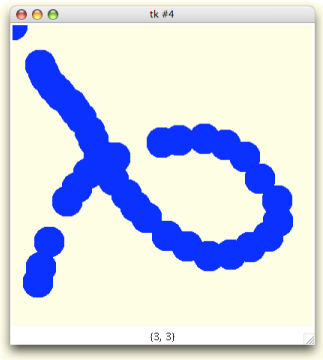
5.5 Canvas

Les objets de type Canvas sont pourvus de méthodes pour dessiner : il suffirait de créer une fenêtre Tk, d'y inclure un Canvas et de lui demander create_oval() pour tracer un cercle de couleur.

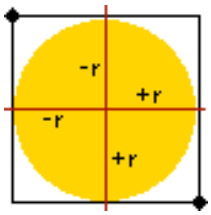
Le code ci-dessous est une extension du programme précédent : puisqu'on peut détecter le point de clic, on peut, à cet endroit précis, dessiner quelque chose, par exemple un disque bleu dont le centre serait au point de clic.

Ainsi, le dernier disque dessiné ici a pour centre, comme indiqué, le point de coordonnées (3, 3).

Remarquez d'ailleurs que ce n'est pas une erreur de dessiner en dehors de la fenêtre : ce n'est tout simplement pas visible…



5.5.1 CREATE_OVAL()



Ici, on crée top, le *master-widget* comme une instance de Tk. Mais au lieu d'un Frame nous allons créer une instance de Canvas, une classe qui connaît des méthodes de dessin : entre autres, la méthode create_oval(). Cette méthode dessine en fait une ellipse inclue dans un rectangle *implicite*, et attend donc au moins 4 arguments : les coordonnées du coin supérieur gauche et celles du coin inférieur droit du rectangle en question.

Ainsi, pour dessiner un *cercle* de rayon r, étant donnés x et y, coordonnées du *point de clic*, on déterminera les 4 arguments attendus en retranchant r à x et y pour le coin supérieur gauche, et en ajoutant ce même r à x et y pour l'autre coin du rectangle.

Dessiner un disque n'est pas fondamentalement différent : un disque n'est qu'un cercle plein, dont la couleur est contrôlée par l'attribut fill, et le contour, par l'attribut outline.

```
def manager(event) :
    position(event)
    dessine(event)

def position(click) :
    afficheur ['text'] = "%s, %s" % (click.x, click.y)

def dessine(event):
    x, y = event.x, event.y
    ew = event.widget
    r = 20 # le rayon
    ew.create_oval(x - r, y - r, x + r, y + r, fill='blue', outline="")

from Tkinter import *

top = Tk()
C = Canvas(top, width=400, height=400, bg="light yellow")
C.pack()
C.bind("<Button-1>", manager)
afficheur = Label(top)
afficheur.pack()
top.mainloop()
```

L'instance C de Canvas spécifie ses dimensions à la création, ainsi que la couleur du fond, bg (anglais, *background*, c'est-à-dire arrière-plan). Elle est liée comme d'hab au bouton gauche de la souris, de sorte qu'un clic appelle la fonction manager().

La nouveauté, ici, c'est que manager() ne fait rien d'autre que recevoir l'événement et déléguer le véritable travail des fonctions subalternes :

- l'une affiche les coordonnées du clic – c'est d'ailleurs tout à fait superflu, mais rassurant ;
- l'autre dessine effectivement dans le *widget* déclencheur…

On savait qu'un événement était un objet complexe et connaissait ses propres coordonnées ; mais il sait aussi quel est le *widget* déclencheur, propriété essentielle puisque c'est ce *widget* seul qui peut appeler la méthode de dessin. Bien sûr, ew n'est qu'une référence, et j'aurais pu coder directement event.widget.create_oval(), mais ça me paraissait moins intelligible.

Quant à la méthode create_oval(), je lui ai ajouté un argument pour la couleur de remplissage (anglais, *fill*), et un autre pour rendre le contour (anglais, *outline*) transparent.

[px23-1] variante de couleur, fonction de la parité : modifier le code ci-dessus pour afficher un disque de couleur *rouge* si l'abscisse du clic est paire, et *verte* si elle est impaire.

Remarque : PYTHON n'a pas de prédicat pour détecter la parité d'un nombre entier, mais vous pouvez aisément y suppléer… si la valeur logique de n % 2 est False, c'est que n est impair.

[px23-2] modifier encore le code pour dessiner en 4 couleurs selon la conjonction des parités de l'abscisse et de l'ordonnée du clic.

5.5.2 CREATE_RECTANGLE()

[px24-1] étant donnée la méthode create_rectangle() disponible pour les objets de type Canvas, modifier le programme précédent pour qu'il dessine un carré de 21 pixels de côté, centré sur le point de clic.

Comme pour l'ellipse, le rectangle a besoin qu'on lui détermine sa diagonale : si c est une instance de Canvas, alors c.create_rectangle(9, 9, 30, 30, fill='blue', outline='yellow') dessine un carré bleu à bords jaunes ; et mettre une chaîne vide au lieu de 'yellow', rendra le contour invisible.

[px24-2] adapter la couleur du rectangle en fonction de la parité du rang du rectangle : le 1er en rouge, le 2e en vert, le 3e en rouge, le 4e en vert, et ainsi de suite...

[px24-3] modifier le programme précédent pour qu'il prenne en compte deux clics successifs pour déterminer les coordonnées de la diagonale du rectangle.

L'astuce serait d'utiliser une variable logique *globale*, définie comme False au départ, appelée, par exemple, second.

Chaque fois que notre HANDLER manager() est activé, il doit tester la valeur de second :

si second est False, alors manager() se contente de :

- mémoriser les coordonnées du point de clic
- les afficher, pour bien montrer qu'il a quand même compris
- basculer second en not second, c'est-à-dire True

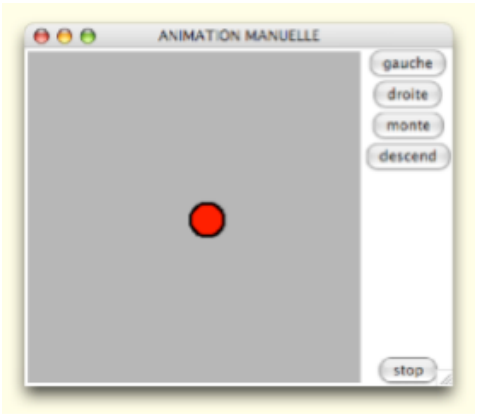
si second est True, alors manager() doit :

- utiliser les coordonnées mémorisées et les nouvelles coordonnées pour appeler dessine(), de sorte que le rectangle soit effectivement dessiné en fonction des deux derniers points de clics
- basculer second en not second, c'est-à-dire False

Ainsi, manager() n'appellera effectivement dessine() qu'une fois tous les deux clics.

5.6 ANIMATION DE SPRITES

Ce petit programme anime un SPRITE, un objet pré-dessiné, dont la position pourra, ici, être changée au moyen de boutons de commande.



Toute l'astuce se trouve dans la définition de la fonction avance(), où C est une instance de Canvas, dans laquelle a été créé un objet nommé boule :

la méthode coords() a pour effet de changer les coordonnées de la boule, et donc la déplace effectivement…

Autrement dit, une expression comme

```
C.coords(boule, X, Y, X + D, Y + D)
```

redessine la boule rouge en redéfinissant le rectangle implicite dans lequel la boule est incluse.

Le programme se laisse décomposer en quatre parties, qui apparaissent en fait à l'envers, pour des raisons techniques expliquées plus bas…

4. Les *widget*s sont les instances d'objets graphiques qui permettent de construire l'interface :

- le *widget maître*, appelé ici top, dont dépendent tous les autres *widget*s
- le *widget* C, instance de la classe Canvas, la seule à savoir dessiner
- le *widget* boule, objet créé dans le Canvas C
- les 4 boutons de contrôle, chacun lié à son HANDLER, sa fonction spécifique
- le bouton stop qui arrête tout

Aucun de ces boutons n'aura jamais besoin d'être référencé dans le programme : je n'ai donc pas besoin de définir de variables – il suffit que le bouton existe et qu'il soit lié à une commande.

3. Les HANDLERS bas(), haut(), droite() et gauche() sont les fonctions appelées par mainloop(), le mécanisme qui gère les liens entre les boutons et la commande qu'ils déclenchent ; mainloop() est la boucle du gestionnaire d'événements, ou EVENT MANAGER.

2. La fonction avance() est une fonction *ancillaire*, c'est-à-dire qu'elle est utilisée par toutes les autres pour

gérer les détails de la réactualisation de l'affichage.

1. Les variables *globales* sont les valeurs accessibles en n'importe quel point du programme, et sont donc définies en premier :

   - le module Tkinter doit être chargé en tout premier lieu, car il définit des variables globales nommées X, Y et D qui doivent absolument être redéfinies par notre programme ;
   - ici, D est le diamètre de la boule, et sert aussi bien lors de la création du *widget* boule que lors de sa manipulation par la fonction avance() ;
   - de même, X et Y représentent les coordonnées de la boule, et doivent persister même quand la fonction avance() n'est pas active – si ces variables étaient locales à la fonction avance(), elles seraient *dynamiquement* créées au moment de l'activation de avance(), et donc disparaîtraient dès que avance() a terminé son boulot.

```
from Tkinter import *

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ variables globales
D = 30          # diamètre
X, Y = 10, 10   # coordonnées initiales

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ animation
def avance(delta_X, delta_Y) :
   global X, Y
   X, Y = X + delta_X, Y + delta_Y
   C.coords(boule, X, Y, X + D, Y + D)

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ handlers
def bas() : avance(0, 10)

def haut() : avance(0, -10)

def droite() : avance(10, 0)

def gauche() : avance(-10, 0)

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ widgets
top = Tk()
top.title("animation manuelle")

C = Canvas(top, bg='dark grey', height=300, width=300)
boule = C.create_oval(X, Y, X + D, Y + D, width=2, fill='red')
C.pack(side=LEFT)

Button(top, text='gauche', command=gauche).pack()
Button(top, text='droite', command=droite).pack()
Button(top, text='monte', command=haut).pack()
Button(top, text='descend', command=bas).pack()
Button(top, text='stop', command=top.destroy).pack(side=BOTTOM)

top.mainloop()
```

La raison pour laquelle le programme est écrit à l'envers, c'est que les HANDLERS doivent être déjà définis au moment où sont créés les *widget*s avec lesquels ils vont être liés : tout se passe, en principe, comme si la partie du programme qui *fait* quelque chose devait être définie avant la partie qui *montre* quelque chose, autrement dit, l'interface.

Les HANDLERS bas(), haut(), droite() et gauche() servent de relais pour transmettre de nouvelles valeurs à la fonction avance() : sachant que cette fonction attend des valeurs *différentielles* en abscisse ou en ordonnée, chaque HANDLER incrémente ou décrémente l'un ou l'autre des arguments.

Quant à la fonction avance(), hormis la déclaration de ses variables globales, elle n'a en fait que deux lignes utiles :

- calcul des nouvelles coordonnées de la boule à partir des anciennes coordonnées et de la valeur des paramètres delta_X et delta_Y
- actualisation du dessin de la boule

Et d'ailleurs, on aurait pu condenser ces deux lignes en une seule, au prix, il est vrai, de leur lisibilité…

Remarquez qu'il est tout à fait possible de « pousser » la boule hors du champ de vision du Canvas. On aurait d'ailleurs pu, comme dans n'importe quelle vraie fenêtre, rajouter des bandes de défilement (ou SCROLL-BARS) pour « dérouler » le *widget* C dans les 4 directions de sorte que notre boule reste visible, mais ce n'était pas le but de cette démonstration.

On peut, bien sûr, créer des SPRITES plus sophistiqués, en utilisant les méthodes appropriés, par exemple create_polygon(), ou même en incorporant directement une image[16] en provenance d'un fichier graphique standard, avec create_image().

[px25-1] coder et exécuter ce programme ; expérimenter en changeant la taille (pas forcément carrée) de l'instance C de Canvas, et sa couleur de fond bg.
[px25-2] expérimenter en changeant le diamètre D de la boule, sa couleur de remplissage fill et l'épaisseur width de son contour (dont la couleur par défaut est 'black').
[px25-3] expérimenter en changeant les valeurs de déplacement dans les HANDLERS.
[px25-4] créer un nouveau bouton qui replace la boule dans le coin en bas à droite.
[px25-5] ajouter une nouvelle fonctionnalité : un clic dans le Canvas déplace la boule en ce point.

5.7 GRAPHES DE FONCTIONS

Cette section montre comment exploiter les fonctionnalités graphiques pour représenter des fonctions ; la différence avec les applications précédentes, c'est qu'il faut pouvoir accommoder des plages de valeurs très variées : la fonction *sinus()*, dont la valeur oscille entre +1 et -1, ou la courbe d'un polynôme de degré n pour lequel il faut calculer les valeurs des coefficients avant de savoir quelle sera la forme et l'amplitude du graphe[17].

Il y a donc là trois problèmes :

- l'abscisse et l'ordonnée croissent vers la droite et vers le bas à partir de l'origine, en haut à gauche : pour une représentation cartésienne, il faut donc décaler les coordonnées ; de plus, les y devront être gérés avec une inversion de signe, sinon le graphe sera inversé ;
- les coordonnées graphiques sont obligatoirement des entiers : une fonction variant continument entre +1 et -1 devra être multipliée par 100 ou 200 pour qu'on commence à y voir quelque chose…
- l'aspect du graphe dépend de l'échelle de la représentation : un véritable grapheur permet de zoomer en avant ou en arrière, et de contrôler l'échelle de chaque axe indépendamment pour que le rendu soit plus évident

5.7.1 DESSINER UN POINT

Dans la panoplie de TKINTER, il y a des fonctions pour créer une ligne, un rectangle, un ovale, mais pas pour créer un point ; ne nous laissons pas démonter, puisqu'un point, c'est le plus petit rectangle, ou le plus petit ovale, ou même la plus petite ligne ; ceci devrait le faire :

```
def pixel(w, x, y, c) : # widget, abs, ord, color
    w.create_rectangle(x, y, x, y, fill=c, outline=c)
```

Le premier paramètre, w, représente le WIDGET dans lequel on veut dessiner. La couleur c s'applique ici aussi bien au contour qu'au remplissage, ce dernier étant probablement inutile…

5.7.2 DESSINER UNE COURBE

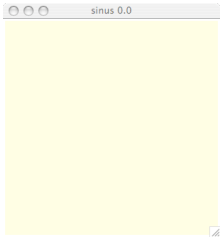
Supposons qu'on veuille représenter la fonction $y = sinus(x)$; cette fonction est disponible sous le nom de sin() dans le module math ; on va donc charger ces deux modules, Tkinter et math, puis camper le décor en définissant un objet de type Canvas qui nous permettra de dessiner le graphe de la fonction :

16 un objet de type PhotoImage, créé par p = PhotoImage(file='xxx'), où xxx serait le chemin d'accès à un fichier gif ou jpeg
17 cf. ici, § 9.3.6 graphe

```
from Tkinter import *
from math import *

T = Tk()  # set up master widget
T.title('sinus 0.0')
W = 300  # width
H = 300  # height
F = Canvas(T, width=W, height=H, bg='light yellow')
F.pack()
```

À ce stade, il ne reste plus qu'à faire varier l'angle x sur une période de 360 degrés et à calculer y = sin(x) pour avoir les coordonnées x, y ; comme sin() s'attend à une valeur en radians, il me faut itérer de 0 à 628 pour effectuer une révolution complète – un radian, c'est à peu près 57.3 °.

```
for angle in xrange(0, int(radians(360) * 100), 5) :
   x = angle / 100.
   y = sin(x)
   pixel(F, x, y, 'red')

T.mainloop()
```

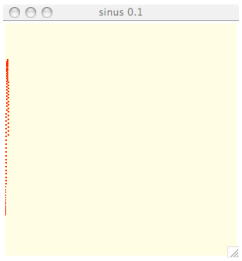


Malheureusement, on ne voit rien dans ma fenêtre graphique : comme on l'a dit plus haut, y varie entre +1 et -1, mais ce qui se passe entre deux pixels est trop petit pour être affichable – il faut donc que je change y d'échelle, en le multipliant au moins par 100. Ici, je vais arrondir sa valeur à 2 décimales, après l'avoir multiplié par le tiers de la hauteur de la fenêtre, soit H/3 ; et pour faire bonne mesure, je vais le décaler de H/2 sorte que le 0 se trouve à mi-hauteur de la fenêtre :

```
def scale(val) : return round(val * H/3, 2)            # scaling
```

et la fin du programme devient donc :

```
for angle in xrange(0, int(radians(360) * 100), 5) :
   x = angle / 100.
   y = scale(sin(x)) + H / 2                           # scaling and offsetting
   pixel(F, x, y, 'red')

T.mainloop()
```

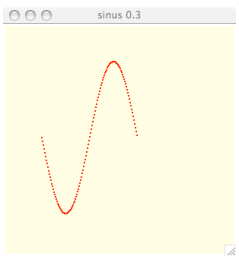


Mais là encore, deux problèmes :

- les x sont tellement sardinés que le graphe n'a aucune ampleur ;
- et que tout est tassé dans les 63 premiers pixels de la fenêtre…

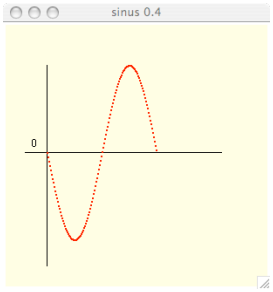
Je vais donc « aérer » les x en les multipliant par 20, en même temps que je décalerai le tout de, disons 50 pixels vers la droite, pour voir :

```
X = W / 6                                              # horizontal offset
Y = H / 2                                              # vertical offset
for angle in xrange(0, int(radians(360) * 100), 5) :
   x = angle / 100.
   y = scale(sin(x)) + Y
   x *= 20
   x += X
   pixel(F, x, y, 'red')

T.mainloop()
```



Maintenant, on voit bien que c'est le graphe d'une sinusoïde… et que comme toute fonction périodique, elle revient à sa valeur de départ une fois que x a effectué une révolution complète autour du cercle trigonométrique.

Et ça me va comme ça, à un détail près : c'est que ce graphe n'a pas grand sens s'il n'y a pas de repère cartésien pour expliciter les variations de y en fonction de x ; alors trois petites lignes de code, et il n'y paraîtra plus :




```
X = W / 6                                          # horizontal offset
Y = H / 2                                          # vertical offset
F.create_line(X/2, Y, W-X, Y)                      # x axis
F.create_line(X, X, X, H - 20)                     # y axis
F.create_text(X/2 + 10, Y - 10, text='0')          # cartesian origin
for angle in xrange(0, int(radians(360) * 100), 5) :  # complete revolution
   x = angle / 100.                                # yields abscissa
   y = scale(sin(x)) + Y                           # and ordinate
   x = x * 20 + X                                  # scaling
   pixel(F, x, y, 'red')                           # plot point

T.mainloop()
```

[px26-1] coder et exécuter ce programme ; expérimenter en jouant avec les valeurs de X et Y, et en changeant le pas de l'angle à *1* et à *10* (gardez celui qui vous plaît le mieux), et utiliser la méthode create_text() pour marquer les valeurs *+1* et *-1* sur l'axe des y ; et remarquez que ce graphe est inversé – problème de signe avec les y…

[px26-2] adapter le [px26-1] pour grapher la fonction *tangente* : math.tan() ; pour fixer les idées, faire une étude préalable de la variation de y avec une petite boucle du genre : for angle in range(-90, 90) : print angle, tan(radians(angle)) pour avoir l'allure générale du graphe et une estimation grossière des plages de valeurs à représenter ;

Pour plus d'information, voir le module MATH : <http://docs.python.org/library/math.html> ; cf. *§ iLP:9.3* pour une démonstration plus poussée, en rapport avec l'ÉC 122 de MATHÉMATIQUES GÉNÉRALES.

RÉCAPITULATION

on avait déjà fait connaissance avec Tkinter, sur lequel est fondé le module turtle

définitions
- BINDING : voir LIAGE
- CALLBACK : appel d'un PLUG-IN ; par extension, le PLUG-IN lui-même
- ÉVÉNEMENT : souris et clavier = périphériques générateurs d'événements
- EVENT HANDLER : une fonction appelée quand un événement se produit
- EVENT MANAGER : fonction mainloop(), chargée d'activer les CALLBACKS
- FRAMEWORK : architecture à PLUG-INS
- LIAGE : association d'un événement et du HANDLER spécifique de cet événement
- WIDGET : window gadget

programmation d'une interface graphique : principes généraux
- penser en termes de LOOK AND FEEL
  - quelle gueule devrait avoir mon interface
  - quels éléments doivent être interactifs
- penser en termes de CONTENEURS
  - l'organisation des WIDGETS est hiérarchique
  - ex : la BOULE est un objet du CANEVAS qui est un objet de la FENÊTRE
- penser en termes de HANDLERS
  - décider comment l'interface sera interactive
  - distinguer COMMAND binding de EVENT binding
    - un liage par bind() ne traite qu'un événement à la fois
      - le HANDLER doit accueillir l'événement déclencheur
      - il peut alors accéder à ses propriétés : widget déclencheur, coordonnées…
    - un liage par l'attribut COMMAND traite d'un coup plusieurs événements
      - ex : button-press + button-release
      - le HANDLER n'attend alors pas d'argument du tout
- penser en termes de fonctions ancillaires
  - pour ce qui peut être factorisé ; ex : avance()
  - pour ce qui n'a rien à voir avec l'interface

et pour plus d'information, voir :

TkInter FAQts	http://www.faqts.com/knowledge_base/index.phtml/fid/264
TkInter	http://wiki.python.org/moin/TkInter
Tkinter reference	http://www.pythonware.com/library/tkinter/introduction/
Tkinter reference: a GUI for python	http://infohost.nmt.edu/tcc/help/pubs/tkinter/
Tkinter Wiki	http://tkinter.unpythonic.net/wiki/
thinking in Tkinter	http://www.ferg.org/thinking_in_tkinter/index.html
python interface to Tcl/Tk	http://docs.python.org/library/tkinter.html
python and Tkinter programming	http://www.manning.com/grayson/

voir aussi le tutoriel de G. Swinnen — en particulier le chapitre 8
http://python.developpez.com/cours/TutoSwinnen/?page=Chapitre8#L8

pour ceux qui veulent aller plus loin :

extensions Tkinter

PYTHON MEGAWIDGETS	http://pmw.sf.net
TIX	http://tix.sf.net
TKZINC, extended Tk canvas type	http://www.tkzinc.org
EASYGUI	http://easygui.sf.net/
TIDE+	http://starship.python.net/crew/mike

il existe d'autres interfaces graphiques plus tendance que TKINTER, dont le look commence à dater ; aucune application récente n'a cet aspect brut de décoffrage, pour la simple raison que tout le monde est maintenant passé au cran supérieur : le look 3D, style OPENGL, avec reliefs, ombres, transparences, couleurs dégradées, et autres friandises cosmétiques...

modules WXWIDGETS

WXPYTHON	http://wxpython.org
BOA CONSTRUCTOR	http://boa-constructor.sf.net
PYTHONCARD	http://pythoncard.sf.net
WXGLADE	http://wxglade.sf.net

mention spéciale pour PYQT, système graphique fonctionnellement similaire à TKINTER, utilisable de manière identique en PYTHON aussi bien qu'en C++, ce qui permet non seulement une transition aisée depuis TKINTER, mais aussi la portabilité vers des langages industriels...

systèmes graphiques GNOME/GTK+

PYGTK	http://pygtk.org
GLADE	http://glade.gnome.org
PYGUI	http://www.cosc.canterbury.ac.nz/~greg/python_gui

systèmes graphiques QT/KDE

PYQT	http://riverbankcomputing.co.uk/pyqt
PYKDE	http://riverbankcomputing.co.uk/pykde
ERIC	http://eric-ide.python-projects.org
PYQTGPL	http://www.quadgames.com/download/pythonqt

autres systèmes graphiques OPEN SOURCE

FXPY	http://fxpy.sf.net
PYFLTK	http://pyfltk.sf.net
PYOPENGL	http://pyopengl.sf.net

PYOPENGL préserve l'interopérabilité avec un grand nombre de bibliothèques GUI pour PYTHON, y compris TKINTER (en installant le TOGL WIDGET), WXPYTHON, FXPY, PYGAME, QT...

⑥ ARCHITECTURE DE PROGRAMMES

Les chapitres précédents montraient comment structurer les données sans jamais dire à quoi ça pouvait être utile… Mais manipuler des structures de données, à travers des structures de contrôle, n'est évidemment pas une fin en soi : ce n'est qu'un moyen de passer d'une forme à une autre, de transformer l'information pour la présenter selon un nouveau point de vue.

La véritable difficulté de la programmation n'est pas d'apprivoiser les idiosyncrasies de tel ou tel langage, mais d'organiser les processus de transformation de manière assez simple pour maîtriser totalement le programme.

L'objectif de ce chapitre est de montrer en détail comment concevoir un programme, et organiser ses ressources, pour que son architecture reste aussi transparente que possible, de façon à faciliter son évolution : son développement d'abord, puis sa maintenance et, enfin, son extension en termes de fonctionnalités.

Il n'y aura donc pas ici de nouveaux concepts : il s'agit juste d'illustrer quelques principes fondamentaux qui ne font l'objet d'aucune théorie, ni d'aucun manuel, parce que ce sont, en essence, des principes trop abstraits pour que ça ait un sens de les formuler sans montrer, en pratique, comment les mettre en œuvre.

Dans la résolution d'un problème, l'angle d'attaque est probablement le point le plus délicat, mais pour le choisir, il n'est pas de recette infaillible : à la question, par quel bout le prendre, la réponse, par le milieu, peut paraître une boutade ; mais nous allons voir qu'il n'en est rien.

sommaire

L'indexation de textes joue un rôle considérable dans l'industrie de la langue : c'est *la* technologie fondamentale des SPIDERS, ces programmes qui parcourent la « *toile* » jour et nuit pour repérer les pages WEB où figure tel ou tel mot-clé : en indexant les pages (référencées par leur URL), ils construisent des bases de données qui permettront ensuite à GOOGLE, YAHOO!, ou GIGABLAST de cracher, en quelques centièmes de seconde, des milliers de références sur un sujet donné.

Cette même technologie est également utilisée pour construire l'index qu'on trouve, la plupart du temps, à la fin des ouvrages de référence ; bien évidemment, dans ce cas, l'index est local à l'ouvrage, et les pages ne sont référencées que par leur numéro.

Par ailleurs, la plupart des programmes de traitement de texte, en particulier ceux qui sont « *orientés ligne* » (éditeurs pour programmeur, mais aussi interprètes et compilateurs), mettent en œuvre cette technologie pour retrouver rapidement à quelle ligne une variable ou une fonction a été définie.

C'est pourquoi je voudrais vous proposer, maintenant, une réflexion sur la conception d'un programme capable de donner la liste de tous les mots d'un texte avec, en référence, toutes les lignes où il apparaît dans le texte. Une fois la technique maîtrisée, l'effort pour l'adapter à un livre entier ou à une collection de pages WEB devrait être minime.

Soit le texte suivant – composé exclusivement en ASCII, en vue d'en simplifier la manipulation et les vérifications de traitement :

```
Un programme qui tourne bien, c'est un plaisir, mais un
programme qui marche mal ou pas du tout, c'est vraiment
une (grosse) frustration. La solution c'est de le coder par
petits bouts, en testant progressivement chaque expression
pour avoir la certitude qu'elle retourne bien la valeur
attendue, avant de s'en servir dans une expression plus
large, autrement dit une "instruction" du programme. On
teste alors chaque instruction avec le contexte voulu, ce
qui, progressivement, garantit le bon fonctionnement global.
```

Si vous voulez « coller » à la démonstration qui va suivre, téléchargez le [texte à indexer][18], et mettez-le dans votre dossier de travail (ou *working directory*).

> Et si votre dossier de travail n'est pas le dossier *par défaut*, utilisez la commande cd pour naviguer jusqu'à ce dossier avant d'appeler PYTHON ; notez que vous pouvez le faire aussi avec PYTHON, en important d'abord le module os, puis en utilisant la fonction chdir() pour vous déplacer dans le dossier voulu {cf. 9.1.2, *module os*}.

À ce stade, il vous faut lire le fichier, comme expliqué dans la section précédente :

```
flux = open('indexe-moi', 'r')                    # ouverture en mode lecture
texte = flux.read()                               # lis tout
flux.close()                                      # c'est fini !
```

Maintenant, la variable texte est une longue chaîne de caractères : la totalité du contenu du fichier, y compris les sauts de lignes… dont le code ASCII est 10, et la représentation en tant que caractère, '\n' comme *new line*. Ce qui a pour conséquence qu'une expression comme :

```
texte.split('\n')
```

retourne, comme valeur, une liste dont chaque élément est le contenu d'une ligne ; ce qui entraîne que texte.split('\n')[0] nous retournera le texte de la première ligne :

```
'Un programme qui tourne bien, c'est un plaisir, mais un'
```

Et le reste à l'avenant…

Maintenant, le problème est d'indexer les mots de *chaque* ligne, en construisant, pour *chaque* mot, une liste des n° de ligne où il apparaît. Souvenez-vous du Mississippi, et pensez en termes de *dictionnaire*, qu'on appellera ici *index* pour des raisons pratiques :

18 il faut être déjà dûment connecté pour télécharger http://foad.iedparis8.net/claroline/courses/E464/document/python-code/indexe-moi

- si le mot n'est *pas encore* dans l'index, je l'ajoute en tant que *clé*, et j'y associe une liste d'un seul élément, le n° de la ligne en cours ;
- si le mot est *déjà* dans l'index, c'est qu'il a déjà une liste de n° de lignes, à laquelle il suffit d'ajouter un nouvel élément, le n° de la ligne en cours.

Pour vérifier que ça peut fonctionner comme ça, donnons-nous un dictionnaire vide qu'on appellerait, justement, index, puis prenons la première ligne, et explosons-la sous la forme d'une liste de *mots* :

```
index = {}                                  # dictionnaire vide
texte = texte.split('\n')                   # liste de lignes
mots = texte[0].split()                     # liste de mots
```

Maintenant, la variable mots vaut :

```
['Un', 'programme', 'qui', 'tourne', 'bien,', "c'est", 'un', 'plaisir,', 'mais', 'un']
```

Il s'agit, ici, de tester qu'il est possible d'ajouter chacun de ces mots au dictionnaire, avec une expression du genre :

```
for mot in mots : index[mot] = [0]          # on est en train de traiter la ligne 0
```

Et l'index résultant a désormais comme valeur :

```
{'bien,': [0], 'mais': [0], 'tourne': [0], "c'est": [0], 'plaisir,': [0], 'Un': [0], 'qui': [0],
'un': [0], 'programme': [0]}
```

Et là, je vois apparaître un premier problème : c'est que le même mot, “un”, apparaît 2 fois, parce que le programme *fait la différence* entre minuscules et majuscules…

Mais voyons ce que la deuxième ligne a dans le ventre :

```
mots = texte[1].split()                     # liste de mots
for mot in mots : index[mot] = [1]          # on est en train de traiter la ligne 1
```

Si maintenant, je regarde la valeur de index, je vois bien que les mots de la deuxième ligne ont été rajoutés. Et là, il me semble qu'il y a deux problèmes :

- d'abord, ça fait bizarre que la ligne 1 soit numérotée 0 ; peut-être que je pourrais ajouter 1 à chaque n ° pour faire plus “standard” ;
- ensuite, un dictionnaire en vrac, c'est un peu pénible à examiner ; je pourrais me faire cuire une petite fonction *print dictionary*, disons prd(), qui m'aiderait à y voir clair :

```
def prd(d) :                                # d : un dictionnaire
   for c in sorted(d) :
      print '\t', c, ':', d[c]
```

Cette fonction prd() accepte un argument de type dict, en trie les *clés* par ordre croissant grâce à la fonction sorted(), puis scanne la liste ainsi créée pour afficher, sur une ligne, la *clé* et la *valeur* qui lui est associée (la séquence '\t' est une convention pour afficher une *tabulation*).

Et maintenant :

```
>>> prd(index)
   Un : [0]
   bien, : [0]
   c'est : [1]
   du : [1]
   mais : [0]
   mal : [1]
   marche : [1]
   ou : [1]
   pas : [1]
   plaisir, : [0]
   programme : [1]
   qui : [1]
   tourne : [0]
```

```
   tout, : [1]
   un : [0]
   vraiment : [1]
```

affiche donc bien lisiblement les associations *clé/n° de ligne* ; ce qui me permet de détecter deux nouveaux problèmes :

- un certain nombre de mots (*bien, plaisir, tout*) ont gardé la virgule qui leur était collée ;
- les mots déjà indexés pour la ligne 0, comme « *programme* » ou « *qui* », ont perdu leur référence originale, remplacée par le n° de ligne 1.

et la différence « Un/un » me chagrine de plus belle…

6.1 ABSTRACTIONS

Il devient clair qu'il ne va pas être facile de faire ce genre de boulot avec une bête expression for : je vais l'abstraire, et en faire une fonction indexe(), comme ça, je serais plus à l'aise pour coder des expressions conditionnelles.

```
def indexe(dex, mots, ligne) :                    # un dictionnaire, une liste de mots, un entier
   for mot in mots :
      if mot in dex : dex[mot] += [ligne]
      else : dex[mot] = [ligne]
   return dex
```

Remarquez l'expression conditionnelle qui teste si l'index lexical a déjà la clé qu'on se propose d'ajouter : si elle y est déjà, on se contente d'ajouter le n° de ligne à celui qui est déjà associé à cette clé ; on obtient alors, avec la définition ci-dessus :

```
index = indexe({}, texte[0].split(), 0)

prd(index)
   Un : [0]
   bien, : [0]
   c'est : [0]
   mais : [0]
   plaisir, : [0]
   programme : [0]
   qui : [0]
   tourne : [0]
   un : [0, 0]
```

Et là, apparaît un nouveau problème : en fait, il y a trois fois le mot “un” dans la première ligne (dont deux fois en minuscule), d'où la double référence [0, 0]… Avez vous une idée de comment régler ce problème de doublon ? Arrêtez-vous un instant, et réfléchissez-y, ça ne devrait pas être bien terrible : il suffirait de tester si le n° de ligne que je suis sur le point d'ajouter se trouve déjà dans la liste associée à cette clé-là :

```
if ligne in dex[mot] : pass
else : dex[mot] += [ligne]
```

L'instruction pass fait exactement ça : *rien*… Bon, c'est sûr, je pourrais *inverser* le test en lui mettant un not et économiser ainsi une ligne de code, mais j'y perdrais en clarté — et pour l'instant, j'ai plus besoin de lumière que d'obscurité.

La fonction indexe() est donc ainsi modifiée :

```
def indexe(dex, mots, ligne) :                    # un dictionnaire, une liste de mots, un entier
   for mot in mots :
      if mot in dex :
         if ligne in dex[mot] : pass
         else : dex[mot] += [ligne]
      else : dex[mot] = [ligne]
   return dex
```

Oh, oh : ça se complique… Pour moi, quand on atteint le troisième niveau d'indentation, c'est le signe que le temps est venu d'en faire une *abstraction*, sous la forme d'une fonction autonome, ici ajoute() :

```
def ajoute(dex, mot, ligne) :                    # un dictionnaire, un mot, un entier
   if mot in dex :
      if ligne in dex[mot] : pass
      else : dex[mot] += [ligne]
   else : dex[mot] = [ligne]
   return dex
```

L'avantage c'est que maintenant, je peux, dans la fonction indexe(), manipuler le mot avant de l'indexer : par exemple, le nettoyer de sa virgule collée, ou tester s'il n'appartient pas à une *stoplist* de mots que j'ai pas envie (ou pas besoin) d'indexer : typiquement les mots grammaticaux, qui n'ont pas grand intérêt pour le lecteur qui s'attend à un index *lexical*.

```
def indexe(dex, mots, ligne) :                   # un dictionnaire, une liste de mots, un entier
   for mot in mots :
      mot = nettoie(mot)
      if mot.lower() in stoplist : pass
      else : dex = ajoute(dex, mot, ligne)
   return dex
```

Vous allez sans doute trouver étrange que j'appelle ici une fonction nettoie() qui n'existe pas, ou que je manipule une variable stoplist qu'on avait jamais vu avant… Mais souvenez-vous que définition et exécution sont deux phases distinctes… Et c'est ça qui est magique : d'un coup de baguette, je fais sortir de mon chapeau :

```
stoplist = 'ce de du en le la mais on ou par pas pour qui un une'.split()

def nettoie(x) : return x
```

Ah, la belle affaire, un *stub* : une fonction qui retourne ce qu'on lui envoie ! Oui, mais attendez : pour que je puisse tester mon code vite fait, il *faut* que cette fonction existe ; après j'ai une petite idée de ce qu'elle devrait faire, mais je ne ne veux pas me *disperser* pour le moment.

Remarquez au passage que le test d'appartenance à la stoplist se fait sur la forme *minuscule* du mot : et voilà comment je me débarrasse du « Un »… Voici donc notre programme prêt à être testé sur l'ensemble du texte :

```
def pilote(fichier, dex) :                       # nom du fichier, dictionnaire
   flux = open(fichier, 'r')                     # ouverture en mode lecture
   for n, ligne in enumerate(flux) :             # on a besoin de n
      dex = indexe(dex, ligne.split(), n + 1)    # indexation de la n-ième ligne
   flux.close()                                  # c'est fini !
   prd(dex)                                      # présente les résultats
   return dex                                    # retourne les données

def indexe(dex, mots, ligne) :                   # un dictionnaire, une liste de mots, un entier
   for mot in mots :
      mot = nettoie(mot)
      if mot.lower() in stoplist : pass          # à supprimer
      else : dex = ajoute(dex, mot, ligne)
   return dex

def ajoute(dex, mot, ligne) :                    # un dictionnaire, un mot, un entier
   if mot in dex :
      if ligne in dex[mot] : pass                # à supprimer
      else : dex[mot].append(ligne)
   else : dex[mot] = [ligne]
   return dex

def nettoie(x) : return x

def prd(d) :                                     # un dictionnaire
   for c in sorted(d) :
      print '\t', c, ':', d[c]

stoplist = 'ce de du en le la mais on ou par pas pour qui un une'.split()

pilote('indexe-moi', {})
```

Comme vous le constatez, l'architecture du programme a subi quelques remaniements :

- la fonction pilote() regroupe désormais les instructions de pilotage du programme, autrefois tapées au top-level comme une séquence d'instructions ;

- il ne reste donc plus qu'une seule instruction globale : l'appel de pilote(), qui prend comme argument le nom du fichier à indexer, et un dictionnaire — vide, au départ ;
- et il ne reste plus qu'une variable globale, stoplist, qui doit (pour l'instant) rester accessible à la fonction indexe() ;
- dans ajoute(), l'opération += a été remplacée par la méthode append(), plus efficace ; notez que la prudence voudrait qu'on vérifie que le mot n'est pas vide avant de l'ajouter, bien que ce soit, en principe, totalement inoffensif

Remarquez l'expression for n, ligne in enumerate(flux), dans la fonction pilote() :

- un fichier est une séquence, au même titre qu'une liste, une fois ouvert, je peux itérer sur chaque élément, c'est-à-dire chaque ligne — pas besoin de read(), ni de split()
- la fonction enumerate() s'applique à une séquence et retourne en même temps l'indice de l'élément (ici appelé n) et l'élément proprement dit (ici appelé ligne)
- c'est donc ce n qui me donne le moyen de référencer cet indice en tant que n° de ligne
- et d'y ajouter 1 pour faire plus « *grand public* ».

Pour bien comprendre ce mécanisme, essayez l'instruction ci-dessous :

```
for n, c in enumerate('PMU') : print n, c
```

qui affiche, sur chaque ligne, l'indice du caractère, puis le caractère lui-même :

```
0 P
1 M
2 U
```

[px27-1] coder ce programme et le tester, en l'état, sur le fichier INDEXE-MOI[19].

On fait donc tourner ça sur l'ensemble du texte… Ici, pour limiter l'encombrement, je ne mentionnerais que les entrées indésirables — reportez-vous donc à vos propres résultats :

```
"instruction" : [7]
bien, : [1]
bouts, : [4]
frustration. : [3]
global. : [9]
(grosse) : [3]
large, : [7]
plaisir, : [1]
programme. : [7]
progressivement, : [9]
qui, : [9]
tout, : [2]
voulu, : [8]
```

Je suppose que vous comprenez pourquoi « *bien*, » ne doit pas être entré tel quel : la virgule faisant partie de la clé d'accès, le programme ne peut pas savoir que le mot « *bien* » est déjà là, puisqu'en réalité il n'y est pas — ou du moins, pas sous sa forme attendue, la plus dépouillée…

19 http://foad.iedparis8.net/claroline/courses/E464/document/python-code/indexe-moi

6.2 RÉGLAGES

Et maintenant, je note toutes les « anomalies » que je voudrais nettoyer : comme les double quotes, les virgules et les points collés… Et ça me permet de définir proprement la fonction nettoie() pour rogner ce qui dépasse :

```
ponctuation = '(,.")'

def nettoie(mot) :
    if mot[-1] in ponctuation : mot = mot[:-1]
    if mot[0] in ponctuation : mot = mot[1:]
    return mot
```

Attention : cela ne se produira pas avec ce texte-ci, mais il se pourrait que le mot n'ait qu'un caractère, interdit : une fois nettoyé, mot serait vide, et mot[0] provoquerait une erreur ; voilà donc d'où viennent les bugs… d'un excès de confiance !

Finalement, voici ce que donnera le programme après *nettoyage* des mots par le nettoie() qu'on vient de redéfinir :

```
bien : [1, 5]
bouts : [4]
frustration : [3]
garantit : [9]
global : [9]
grosse : [3]
large : [7]
plaisir : [1]
plus : [6]
programme : [1, 2, 7]
progressivement : [4, 9]
tout : [2]
valeur : [5]
voulu : [8]
vraiment : [2]
```

Remarquez que le 'qui,' a disparu : il faisait pourtant bien partie de la stoplist, mais n'était pas détecté à cause de sa virgule en trop… Et puisqu'on parle de la stoplist, qui est ici bien mince, verriez vous l'intérêt de la définir dans un fichier, de sorte qu'on puisse la modifier de façon externe au programme ? Essayez-donc ceci :

```
flux = open('stop.list', 'w')
flux.write(' '.join(stoplist))                    # écris la liste comme une chaîne
flux.close()
```

Ouvrez le fichier stop.list avec un éditeur de texte, et rajoutez-y quelques mots sans intérêt, comme « *c'est* », « *dans* », « *s'en* » ou « *plus* », puis refermez-le (il n'est pas vraiment crucial que la liste soit triée, tant qu'elle reste de taille modeste). Maintenant, repassez sous PYTHON et tapez ces 3 lignes de code :

```
flux = open('stop.list', 'r')
stoplist = flux.read().split()                    # explose en mots à la lecture
flux.close()
```

et regardez ce que stoplist a maintenant dans le ventre ; a-t-elle bien pris en compte vos modifications ? Est-ce que ça change le comportement du programme ?

Remarquez l'utilisation conjuguée des deux méthodes, dans l'expression flux.read().split() : ça veut dire que la valeur retournée par fread() (de type str) appelle directement la méthode split(), et que stoplist accueille donc directement une liste (sans passer par un état intermédiaire) ; cette écriture condensée est une affaire de style (ici, le style dit *fonctionnel*), et si vous n'y voyez pas clair, il est toujours possible de *décomposer* l'expression en deux instructions distinctes, même si c'est (du strict point de vue de la performance) moins efficace.

6.3 FINALISATION

Peut-être pensez-vous qu'une vraie stoplist serait plus présentable – donc plus facilement modifiable – s'il n'y avait qu'un mot par ligne ? Qu'à cela ne tienne…

[px27-2] adapter le programme [px27-1] de sorte que la stoplist soit enregistrée avec un saut de ligne, c'est-à-dire '\n', après chaque mot.

[px27-3] modifier le programme de sorte que les 3 lignes de code qui enregistrent ou relisent la stoplist soit *abstraites* dans des fonctions spécifiques, appelées, par exemple, put_list() et get_list() – l'idéal serait que put_list(), qui ne retourne rien, soit définie avec un paramètre représentant la liste de mots à enregistrer, et que get_list() retourne une liste, pour qu'on puisse l'utiliser comme stoplist = get_list('stop.list')

Voyez-vous comment cette généralisation permet maintenant d'utiliser ces fonctions dans n'importe quel autre programme ?

Ceci vous permet de remarquer au passage que, *par défaut*, la méthode split() prend en compte, non seulement le blanc (l'espace), mais aussi les caratères « gris » : la tabulation et l'alinéa…

[px27-4] modifier le texte, et mettre le mot final, GLOBAL, entre guillemets : pourquoi la fonction nettoie() ne fait-elle plus son boulot correctement ? Comment traiterait-elle, par exemple, à la ligne 3, le mot GROSSE s'il était à la fois entre guillemets *et* entre parenthèses ? Modifier le code pour remédier à ce problème.

[px27-5] la fonction prd() est très commode pour *afficher* les données, mais ce n'est pas comme ça qu'un typographe professionnel présenterait l'index d'un ouvrage : le lecteur n'a que faire des “[]” ; modifier cette fonction pour que la liste de lignes soit présentée sous la forme : programme : 1-2, 7 ; quand il y a 2 numéros consécutifs ou plus, par exemple, 1, 2, 3, 5, la liste est présentée sous la forme 1-3, 5 : coder cette présentation.

Le moment est venu de mettre le programme à l'épreuve sur des fichiers plus volumineux : peut-on l'appliquer aux textes source de programmes, et comment se comporte-t-il avec le format pdf ou celui d'un fichier html ? Et bien évidemment, pour des raisons pratiques, il serait commode que ce programme soit un *script autonome* qui puisse lire, sur la LDC (la ligne de commande), le nom du fichier à indexer…

[px28-1] adapter le programme [px27-5] pour qu'il prenne le nom du fichier sur la LDC, et l'essayer sur des textes nettement plus volumineux ; découvrez-vous de nouvelles anomalies ? Modifiez le code (ou les données) du programme pour y remédier.

Un problème qui se pose à l'indexation de gros fichiers, c'est qu'il y a beaucoup de mots sans intérêt : pourquoi ne pas inverser le principe de la stoplist pour, au contraire, n'indexer que les mots qui figurent dans une golist ? À titre expérimental, essayez-en le principe avec une liste contenant les mots BOUTS, FRUSTRATION, GLOBAL, PROGRAMME, PROGRESSIVEMENT et VALEUR.

En fait, il faudrait adapter le programme pour qu'il puisse fonctionner indifféremment en mode stop ou en mode go : il suffit de le lui indiquer sur la ligne de commande, au moyen de ce qu'on appelle un switch, une option, c'est-à-dire un argument particulier, conventionnellement précédé d'un tiret, que le programme teste avant de décider comment traiter le fichier…

[px28-2] adapter le programme précédent pour pouvoir, optionnellement, n'indexer que les mots figurant dans une golist, donnée dans un fichier nommé go.list.

Et pour parachever l'ouvrage, déporter dans un module les fonctions indexe(), ajoute(), nettoie(), prd() et les données globales qu'elles utilisent : le script qui exploiterait ce module ne doit donc plus contenir que les instructions d'ouverture du flux, la création du dictionnaire d'index vide, et la boucle d'indexation, autrement dit, la fonction pilote() — cf. {section 9.1, modules}.

[px28-3] adapter ce programme pour en faire un module nommé dexlex avec ses fonctions et les données globales, d'une part, et le reste dans un script autonome, d'autre part.

6.4 EXTENSION

Si l'un des principes fondamentaux de la programmation est de généraliser autant que faire se peut, étendre la capacité de ce programme au traitement de textes accessibles par le réseau ne paraît pas une ambition démesurée : si Google le fait, c'est que c'est possible…

Un moteur de recherche, vu au plus simple, c'est un programme qui indexe des pages WEB, et indexe les mots intéressants, non pas en fonction de la ligne où ils apparaissent, mais de la page où ils figurent ; en fait, dans une architecture typique CLIENT/SERVEUR, cette tâche est dévolue au serveur ; coté client, une fonction supplémentaire permet de saisir la requête et de retourner la liste des pages pertinentes.

Tout ceci est présenté de façon extrêmement simplifiée, mais il est tout à fait possible, selon ces principes, de réaliser un prototype opérationnel. En définitive, côté serveur, un moteur de recherche destiné à trouver du texte sur une page WEB serait constitué de trois composants :

1. quelques lignes de code pour ouvrir et lire une page WEB à la volée
2. une fonction qui, à condition que le type MIME de cette page soit bien TEXT/HTML, indexe la page et unifie le codage des mots selon un dénominateur commun, par exemple le polyvalent *utf-8*
3. une fonction qui construit la liste des liens disponibles sur cette page pour qu'on puisse recommencer à l'étape 1 pour chaque lien

Côté client, les choses ne sont guère plus complexes :

1. un bout de code pour saisir à la volée les mots recherchés : la requête (ou QUERY)
2. une fonction pour collecter les pages indexées pour ces mots
3. une fonction pour présenter la liste des pages répondant à la requête

6.4.1 CÔTÉ SERVEUR

Lire une page WEB n'est guère plus difficile que lire un fichier : il suffit d'ouvrir un flux pour un URL[20] particulier et non pour un fichier local. L'ouverture d'un URL est analogue à celle d'un fichier, à ceci près qu'au lieu de open(), c'est la fonction urlopen() qu'il faut utiliser ; comme argument, on lui donne une chaîne, celle-là même qu'on aurait spécifié dans un BROWSER pour atteindre une page WEB. Cette fonction n'est pas intrinsèque, il faut aller la chercher dans un module :

```
from urllib import urlopen
flux = urlopen('http://www.iedparis8.net/ied/')
```

Une fois le flux ouvert, on utilisera les mêmes méthodes de lecture que pour un fichier :

```
page = flux.read()                                    # toute la page d'un coup
```

Ce qui signifie qu'un programme codé en vue de manipuler du texte en provenance d'un fichier fonctionnera indifféremment avec un fichier local ou distant, c'est à dire en réseau…

Le script préparé pour [ex28-4] aura besoin d'une petite modification, puisque la référence à indexer n'est plus un n° de ligne mais un URL : en pratique, ça veut dire que le 3e argument de la fonction indexe() n'est plus un nombre mais une chaîne, ce qui ne fait, d'ailleurs, aucune différence pour indexe(), donc ne requiert *aucune* modification du module dexlex…

Dans le script ci-dessous, j'ai ajouté le mot MODIF en commentaire pour chaque instruction modifiée ou ajoutée. Ainsi donc, avec *deux lignes* de plus, il suffirait d'un seul script pour deux usages : tester le début du nom du fichier — avec la méthode startswith() — permet de décider s'il s'agit d'un URL ou d'un fichier local ; ensuite, définir une variable url comme l'URL ou rien (c'est-à-dire None) permettra, à l'appel de indexe(), de passer cet URL à la place d'un n° de ligne, par simple évaluation de l'expression booléenne url or n+1.

20 uniform resource locator : localisateur uniforme de ressource

```
from dexlex import indexe, prd                  # module d'indexation
from sys import argv                            # pour lire les arguments
from urllib import urlopen                      # -- modif

def pilote(f, X) :                              # URL ou nom de fichier, dictionnaire
   url = f if f.startswith('http://') else None # -- modif
   flux = urlopen(f) if url else open(f)        # -- modif
   for n, texte in enumerate(flux) :            # on a besoin de n
      X = indexe(X, texte.split(), url or n+1)  # -- modif
   flux.close()                                 # c'est fini !
   return X

if len(argv) > 1 : prd(pilote(argv[1], {}))     # l'argument sur la LDC
else : exit('arg manquant : fichier ou url')
```

C'est maintenant que se justifie la mise en œuvre d'une stoplist ou d'une golist : testez naïvement le programme sur la page WEB ouverte ci-dessus, et vous admettrez que le code HTML est un peu encombrant — sauf, bien sûr, si ce sont justement les balises (ou *tags*) HTML qui nous intéressent.

Et pourquoi nous intéresseraient-elles, les balises ? Parce que justement : si un CRAWLER doit indexer les pages WEB référencées dans la page déjà ouverte, il faut bien qu'il y recherche les *tags* HREF, et plus précisément, l'URL qu'ils spécifient.

Un lien HREF (aussi appelé *hyper-lien*) se présente sous la forme d'une chaîne précédée de HREF=, dans une balise <A> ou une balise <LINK> ; par exemple :

```
<a href="http://docs.python.org/3.0/"</a>
<a href="/ftp/python/3.0/python-3.0.msi"</a>
<a href="/ftp/python/3.0/Python-3.0.tar.bz2"</a>
```

Le début de la chaîne est le protocole d'accès : au vu de l'exemple ci-dessus, il semble évident que seul nous intéressent les accès HTTP, mais comment les retrouver dans le texte de la page ? Facile : la méthode find() d'une chaîne retourne *l'indice* de la sous-chaîne passée en argument, ou la valeur -1 si la sous-chaîne n'est pas là :

```
>>> 'bleu blanc rouge'.find('blanc')
5
>>> 'bleu blanc rouge'.find('vert')
-1
```

Il est donc aisé de programmer une fonction qui explore une ligne de texte :

```
def get_href(ligne) :                           # texte multiligne
   x = ligne.find('href=')                      # trouves-tu ?
   if x < 0 : return                            # non, dégage
   ref = ligne[x+6:].split('"')[0]              # oui, extrais-le lien
   if ref.startswith('http:') : return ref      # et retourne-le
```

et de l'utiliser itérativement, après ouverture d'un URL :

```
hrefs = []
flux = urlopen('http://www.python.org/')
for ligne in flux : hrefs.append(get_href(ligne))
```

Testez-le : pour cette page, il vous construira une liste 48 liens[21] — il n'y a plus qu'à mouliner sur les 48 pages référencées, et sur les hyper-liens qu'elles contiennent, et sur les pages auxquelles font référence ces hyper-liens, etc.

Ce serait donc l'embryon d'une solution au problème du SPIDER (ou CRAWLER) posé plus loin par l'exercice [px29-2].

Et pendant que j'y pense, notez bien que urlopen() peut aussi ouvrir un fichier local…

21 attention : cette fonction get_href() est codée pour ne retourner que le 1er URL rencontré dans chaque ligne

6.4.2 CÔTÉ CLIENT

En supposant que votre serveur ait, la nuit dernière, construit l'index de toutes les pages WEB de la planète, il ne reste plus qu'à l'interroger en lui fournissant les mot-clés recherchés :

- s'il y a un seul mot, il suffit (théoriquement) d'accéder au dictionnaire en lui fournissant cette clé d'accès : la valeur est la liste des pages WEB indexées pour ce mot
- s'il y a plusieurs mots, la chose devient un tout petit peu plus complexe, parce qu'il faut alors faire *l'intersection* des pages WEB indexées pour chacun de ces mots

6.4.3 PROBLÈMES ET SOLUTIONS

Comme dans tout projet de programmation, des problèmes peuvent se poser au cours de la réalisation : le plus délicat (non-traité ici) est celui de la détection de la circularité : cette page a-t-elle déjà été visitée ? Si elle l'a été, n'a-t-elle pas été modifiée entre-temps ? Si oui, alors peut-être faut-il la réindexer… donc supprimer cette référence pour tout mot de l'index.

On devine que la maintenance de l'index est un problème non-trivial, d'autant que cet index risque fort d'être assez volumineux… en tout cas trop volumineux pour être conservé en mémoire : il faudra probablement recourir aux techniques de bases de données…

Dans le cadre d'une maquette, d'une étude de faisabilté, il est un autre problème qu'il nous faut absolument résoudre, celui de la disparité d'encodage des pages WEB ; certaines sont en latin-1, alors que d'autres sont en utf-8 : comment le détecter, et comment ceci peut-il influer sur l'indexation proprement dite ?

Comme on l'a vu à la {section 3.4, *unicode*}, le mot œuf serait codé différemment selon qu'on est en utf-8, en latin-1 ou en latin-9, et comme c'est le *mot* qui sert de clé d'accès au dictionnaire, la conséquence immédiate est que ces 3 codages vont forcément correspondre à 3 entrées distinctes : il faut donc un moyen de déterminer l'encodage de la page et une petite acrobatie pour unifier la représentation interne selon un dénominateur commun, *unicode*, par exemple…

Toute page HTML bien formée comporte normalement une indication du type des données qu'elle contient, sous la forme d'un indicateur CONTENT à l'intérieur d'un *tag* META ; ainsi, dans le cas de la page <http://www.iedparis8.net/ied/>, il y a une ligne qui dit :

```
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
```

D'autres pages publieront ce type d'information avec de légères variantes ; par exemple la 7e ligne du code source de <http://www.python.org/> ajoute un blanc et un / avant le chevron fermant :

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
```

Cette indication, TEXT/HTML, est ce qu'on appelle un type MIME[22] : un standard pour caractériser le contenu du fichier — il existe actuellement 96 types de textes, et tant qu'il s'agit de texte, ça a un sens d'en indexer le contenu…

Mais l'indicateur CONTENT nous révèle aussi le CHARSET, autrement dit l'*encodage* du fichier ; et ici, nous allons retrouver le vieux problème de la mondialisation de l'informatique : les multiples manières de coder les caractères…

Détecter l'encodage d'une page WEB serait donc juste affaire de trouver l'endroit où se trouve l'indicateur CHARSET (dans notre cas, à la ligne 14), puis d'extraire l'information ; un petit moulin de 3 instructions devrait faire le boulot :

```
def get_charset(url) :
   for line in urlopen(url) :
      if line.find('<meta') < 0 : continue
      x = line.find('charset=')
      if not x < 0 : return line[x + len('charset='):-2]
```

22 MIME : multipurpose internet mail extensions ; cf. http://www.webmaster-toolkit.com/mime-types.shtml

Comme on l'a déjà vu plus haut, la méthode find(), appliquée à une chaîne, retourne la position où se trouve la sous-chaîne recherchée (ou la valeur -1 si elle ne s'y trouve pas) ; donc la boucle for, dans cette fonction get_charset(), doit explorer chaque ligne contenant un *tag* META (en fait, il y en a plusieurs), et retourner le CHARSET trouvé.

Et en effet, l'expression get_charset('http://www.iedparis8.net/ied/') retourne l'information : iso-8859-1…

Mais le code, ici, manque singulièrement de robustesse, et présente 4 problèmes potentiels :

1. rien n'oblige personne à indiquer le CHARSET sur la *même ligne* que le *tag* META qui l'inclut ; en fait, la seule contrainte est que cet indicateur doit être un élément de la valeur de l'indicateur CONTENT ; par convention, on le met souvent à côté du type MIME, mais là encore, rien d'obligatoire…
2. de plus, sur une page qui, justement, traiterait de ce sujet (comme, par exemple, celle-ci), il pourrait y avoir beaucoup d'occurrences non pertinentes de la chaîne 'charset='
3. le mot CHARSET n'est pas obligatoirement écrit en minuscules — en fait, le code HTML est souvent typographié en capitales, en particulier quand il est généré automatiquement
4. la valeur du CHARSET n'est pas obligatoirement *tout sauf les 2 derniers caractères* de la ligne — la preuve, la définition pour <http://www.python.org/>, mentionnée ci-dessus

Heureusement, il y a une autre façon d'obtenir cette information ; à condition de savoir que quand on ouvre un URL, le serveur envoie *silencieusement* un paquet de données (*additional information*) que la plupart des *browsers* sait exploiter — mais pas afficher.

Ainsi, la valeur retournée par urlopen() est un pointeur sur un objet ADDITIONAL-INFO-URL :

```
>>> flux = urlopen('http://www.iedparis8.net/ied/')
>>> flux
<addinfourl at 18775656 whose fp = <socket._fileobject object at 0x782ab0>>
```

Cet objet est muni de méthodes spécifiques[23], en particulier la méthode info(), qui retourne un objet httplib.HTTPMessage, ou plus exactement une instance de ce type :

```
>>> flux.info()
<httplib.HTTPMessage instance at 0x11e7dc8>
```

à partir duquel on peut fabriquer un dictionnaire avec cette simple expression :

```
information = dict(flux.info())
```

et information a maintenant comme valeur :

```
{'connection': 'close',
 'content-type': 'text/html; charset=iso-8859-1',
 'date': 'Sun, 01 Feb 2009 20:13:34 GMT',
 'expires': 'Sun, 01 Feb 2009 21:05:31 GMT',
 'last-modified': 'Sun, 01 Feb 2009 19:05:31 GMT',
 'server': 'Apache',
 'vary': 'Cookie,Accept-Encoding',
 'x-powered-by': 'PHP/4.3.10'}
```

où charset est une donnée de content-type. Essayons d'extraire le CHARSET d'un URL quelconque :

```
>>> u = 'http://createdigitalmusic.com/'
>>> dict(urlopen(u).info())['content-type'].split()[1].split('=')[1]
UTF-8
>>> u = 'http://www.iedparis8.net/ied/'
>>> dict(urlopen(u).info())['content-type'].split()[1].split('=')[1]
iso-8859-1
```

Théoriquement, pour tout URL, l'expression dict(urlopen(…url…).info())['content-type'].split()[1].split('=')[1], aussi tordue qu'elle paraisse, *devrait* retourner la valeur attendue — si tant est que charset est effectivement transmis avec le content-type ; et pour des raisons de lisibilité, mieux vaudrait l'abstraire en une fonction — où le paramètre f représenterait un flux déjà ouvert :

23 malheureusement non documentées…

```
def get_charset(f) : return dict(f.info())['content-type'].split()[1].split('=')[1]
```

et qu'on utiliserait ainsi :

```
from urllib import urlopen
flux = urlopen('http://www.iedparis8.net/ied/')
charset = get_charset(flux)
```

En fait, aucune technique n'est infaillible : l'exemple de <http://fr.wikipedia.org/wiki/Distribution_linux> montre que certains serveurs ne donnent pas ce renseignement, alors même qu'il figure dans la section HEAD de la page ; dans ce cas, il faudra recourir à notre toute première méthode…

Connaissant le CHARSET, nous savons maintenant s'il faut convertir l'information pour, avant de l'indexer, uniformiser son encodage, grâce aux méthodes decode() et encode() qui vont permettre de changer l'encodage du mot *avant* son insertion dans l'index — cf. la documentation sur les chaînes[24].

Supposons par exemple que nous ayons déjà lu, et indexé, la page <http://wiki.mandriva.com/fr/>, qui est en utf-8, donc que nous avons donc déjà constitué un index de n mots en utf-8, mais que nous voudrions *augmenter* cet index avec la page <http://www.iedparis8.net/ied/> qui est, elle, en iso-8859-1.

Sur cette page, la première ligne contenant des codes sur 8 bits sera lue ainsi :

```
L = "<title>institut d'enseignement \xe0 distance - universit\xe9 paris 8</title>\n"
```

où le 'à' et le 'é' ont respectivement pour code 224 et 233 en iso-8859-1, le reste étant de l'ASCII.

Et c'est là que les méthodes decode() et encode() vont résoudre notre problème :

```
>>> L.decode('iso-8859-1').encode('utf-8')
"<title>institut d'enseignement \xc3\xa0 distance - universit\xc3\xa9 paris 8</title>\n"
```

Autrement dit, pour chaque ligne d'une telle page, il nous faut explicitement convertir la chaîne iso-8859-1 en une représentation décodée, à partir de laquelle on pourra construire une nouvelle représentation, codée cette fois en utf-8, garantissant ainsi l'uniformité de l'index (l'argument des méthodes decode() et encode() est le nom du CODEC à utiliser pour la conversion).

Alternativement, on pourrait aussi lire toute la page d'un coup et effectuer alors la conversion sur la totalité du texte, comme préconisé dans le cours iAO :7 — ça ne peut pas faire de mal…

La preuve : c'est ce que font, en pratique, tous les BROWSERS[25] bien conçus…

D'ailleurs, ce problème n'est pas spécifique aux pages WEB, et se poserait (se résoudrait) de la même façon pour des fichiers texte locaux, à cette différence qu'ils ne contiennent aucune information d'encodage : il faut le savoir[26] *avant* de les ouvrir.

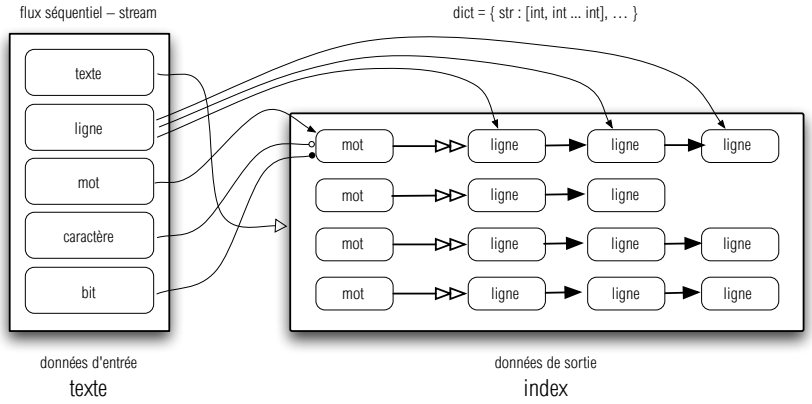
[px29-1] adapter le script [px28-3] pour qu'il puisse indexer consécutivement plusieurs pages WEB, indépendamment de leur codage ; utiliser, par exemple, l'url de l'IED et celui du WIKI de MANDRIVA, mais tester aussi le programme avec plein d'autres URL.

[px29-2] adapter le script [px29-1] pour qu'il puisse *non seulement* indexer une page WEB quel que soit son codage, *mais aussi* pour qu'il construise automatiquement la liste des liens HTTP mentionnés dans la page, et indexe également leur contenu, quelqu'en soit le codage, toujours ; et bien entendu, prévoir l'architecture du programme pour qu'il soit aisé de le faire évoluer en vue de traiter également les liens référencés par les pages liées.

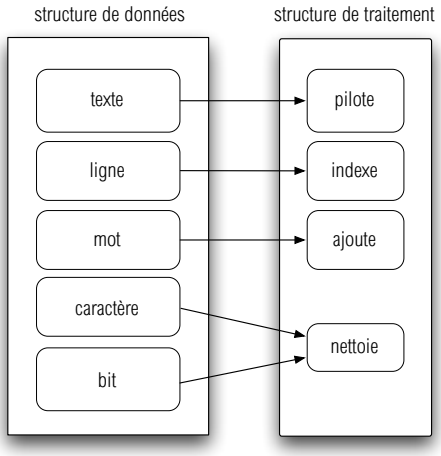
24 <http://docs.python.org/library/stdtypes.html#str.encode> — voir aussi <http://docs.python.org/library/codecs.html#module-codecs>
25 sauf, bien sûr, MICROSOFT INTERNET EXPLORER
26 exception faite des textes en utf-8 + BOM, où le BYTE ORDER MARK constitue les deux premiers octets du fichier

6.5 RÉFLEXIONS

Si maintenant je reprends de l'altitude, je constate que, partant d'un flux séquentiel, j'ai fini par aboutir, par une transformation complexe, à une information structurée de type dict, constituée d'objets de type str (intégrant 3 niveaux) et de listes (list) d'objets de type int :



Ainsi, l'art de programmer, c'est l'art de (a) *projeter* son objectif, et (b) de se donner les moyens d'y parvenir : il suffit, à la limite, d'avoir une conscience claire de l'objectif pour être capable de produire la transformation à la main, mot par mot… Après, il s'agit bien sûr de la programmer ; mais il y a justement quelque chose de tout à fait remarquable ici, c'est que l'organisation-même du programme est, tous comptes faits, *implicite* dans la structure des données de départ :



Le diagramme ci-dessus montre :

- les données qu'il faut traiter, et comment elles se laissent naturellement déstructurer ;
- la structure *formelle* du programme proposé plus haut, qui se présente sous la forme de quatre fonctions, compte non tenu de prd(), dédiée à la présentation des résultats.

Et il apparaît alors évident que chaque niveau structurel de l'information originale est traité par une fonction spécifique, et que la structure *formelle* du programme correspond exactement à la structure des données, à une exception près : la fonction nettoie() accède au niveau de la représentation binaire des caractères par l'entremise de la méthode lower(), et il ne m'a pas semblé *utile* de décomposer le programme à ce point-là.

Bien entendu, le flux du programme lui-même, autrement dit la *structure de contrôle*, respecte strictement la structure des données, puisque les fonctions indexe(), ajoute() et nettoie() sont en effet appelées dans cet ordre, depuis la fonction qui pilote l'ensemble du traitement.

NOTES

la façon dont l'information se laisse naturellement déstructurer peut servir de guide pour le traitement

représenter l'information, c'est
- choisir un type de structure de données pour la représenter
  - liste
  - table d'associations
  - agrégat composite des types ci-dessus
- transformer l'information pour la mettre sous cette nouvelle forme

choisir une structure de données suppose une INTUITION PRÉALABLE
- du genre d'opérations qu'on voudra leur appliquer
- de la commodité de l'accès aux données pour la structure choisie
- et ceci implique un minimum d'expérience

en pratique, il y a TROIS TYPES PRIMITIFS DE STRUCTURES
- accès par INDEX
  - c'est le cas des chaînes (donc des fichiers) et des listes
  - la recherche de l'information est séquentielle
  - le temps pour retrouver l'information dépend de la taille de la structure
- accès par CLÉ
  - c'est le cas des dictionnaires — mais aussi des variables scalaires
  - la recherche de l'information est directe
  - le temps pour retrouver l'information est indépendant de la taille de la structure
- accès par ATTRIBUT, ou propriété d'objet
  - méthodes propres à un type d'objet, accessibles à partir d'une instance
  - un objet définit implicitement un espace de nommage
  - il peut donc aussi avoir des attributs, ou variables locales

la façon de structurer les données implique un compromis entre
- la complexité des structures de données
- la simplicité des algorithmes qui les exploitent

données et programmes sont des vases communicants
- simplifier l'un revient à complexifier l'autre
- et vice versa

pour des raisons d'intelligibilté
- la tendance est de simplifier les algorithmes
- en sophistiquant les structures de données

c'est la tendance des langages à objets : SQUEAK
mais aussi celle des langages déclaratifs : PROLOG
en PROLOG, il n'y a plus d'algorithme explicite : que des données

la création d'un programme s'appuie sur le principe DIVISER POUR RÉGNER

l'esprit humain a du mal à gérer des concepts à plus de 3 niveaux sans faire d'abstractions[27]
c'est pourquoi il est plus facile
- de limiter les boucles à 3 niveaux d'imbrication
- d'exploser le programme en de multiples fonctions
  - même si le traitement serait possible en une séquence MONOLITHIQUE d'instructions
  - même si la fonction ne prend qu'une ligne
  - en pratique, il n'y a pas de différences entre
    - les fonctions définies par le programmeur
    - celles qui sont définies dans des modules
    - celles qui sont intrinsèques au langage

la programmation est essentiellement une ACTIVITÉ CRÉATIVE où il est normal d'inventer

une bonne façon de résoudre un problème serait
- concevoir, grosso modo, les grandes lignes de la solution
  - s'inspirer de ce qu'on ferait à la main
  - tester in vivo la faisabilité d'une telle approche
- partir de l'idée que les sous-problèmes qui se présentent sont déjà résolus
- en faire des abstractions — soigner les noms pour qu'ils soient évocateurs
- descendre alors dans le détail de la résolution des sous-problèmes
- et récursivement…

une fonction est une capsule
- isolée des perturbations extérieures
- ne communique que par les paramètres définis
- son fonctionnement ne dépend que des arguments ainsi transmis
- sa fiabilité est garantie par les tests unitaires, si possible exhaustifs
- par prudence, elle ne devrait donc pas utiliser de variables globales
  - sauf durant la phase de mise au point
  - exemple : la stoplist globale peut être modifiée sans avoir à redéfinir indexe()

toute fonction peut avoir un effet et une valeur
a priori, toute fonction devrait retourner une valeur
- sinon, ça veut dire qu'elle est utilisée pour son effet global
- et non pour la valeur qu'elle calcule

la présentation des résultats est un effet
devrait être l'apanage d'une seule fonction du programme
- même si pour ce faire elle utilise des fonctions auxiliaires

s'il faut changer cette présentation, ce sera plus facile
- de localiser, par son nom, le code responsable
- d'y substituer une nouvelle fonction
  - de tester les nouvelles fonctionnalités
  - sans compromettre les anciennes
- ainsi, on peut toujours revenir sur ses pas

27 par exemple, un hypercube à 4 dimensions

⑦ INFRASTRUCTURES LOGICIELLES

Ce chapitre adopte un point de vue méthodologique, et prend l'altitude nécessaire pour faire le point sur la façon dont nous avons, jusqu'à présent, pratiqué l'activité de programmation.

On découvre alors que nous avons, depuis le début, pensé implicitement en termes d'objets, et nous verrons alors jusqu'où on peut pousser cette approche conceptuelle. Il y a d'autres façons d'aborder la programmation :

- l'approche fonctionnelle a déjà été étudiée à travers LISP
- l'approche objet sera étudiée plus systématiquement dans le cours de SQUEAK
- l'approche procédurale, dite impérative, fera aussi l'objet d'un cours distinct
- l'approche logique est déclarative et pense en termes de relations : elle sera étudiée dans un cours spécifique, avec un langage hautement spécialisé — PROLOG

Alors que la différence entre l'approche purement DÉCLARATIVE et l'approche PROCÉDURALE est très nette – puisque les deux sont mutuellement exclusives – cette dernière est par contre tout à fait compatible avec une attitude fonctionnelle ou une attitude objet ; la programmation en PYTHON se révèle ainsi une approche hybride, avec laquelle, comme nous le verrons plus tard, il serait même possible (au prix d'une petite acrobatie) de programmer de façon déclarative.

L'objectif de ces quelques pages est de caractériser la démarche du programmeur en ce qu'elle dépend de l'infrastructure d'un langage de programmation. Ce point de vue n'est pas innocent parce qu'il implique que notre toute première exposition à un langage de programmation va nous conditionner culturellement, et a toute chance d'influencer notre jugement quant à l'intérêt d'autres langages, d'autres façons de voir le monde et de le structurer : et par là, notre capacité à évaluer la qualité de nos méthodes de développement.

sommaire

7.1 DU CONCEPTUEL À LA MISE EN ŒUVRE

Concevoir le traitement de l'information implique une déstructuration analytique en même temps qu'une projection synthétique — l'énoncé d'un problème de traitement part le plus souvent d'un état des lieux, les données disponibles, et caractérise la forme que doit avoir la solution, forme à laquelle on devrait aboutir après une série de mutations contrôlées par un programme qui manipule les données originales.

La difficulté réside donc dans la conception de cette procédure, généralement décomposée en une séquence d'opérations quand on adhère à l'approche impérative (ou *procédurale*) de la programmation. La projection synthétique est une démarche convergente : si j'avais ceci et cela, je n'aurais plus qu'à les afficher ; mais c'est justement parce que je ne les ai pas que je me pose la question de comment les fabriquer à partir des données originales, d'où la nécessité de les analyser pour détecter comment en extraire l'aspect qui m'intéresse.

La résolution d'un problème en PROLOG ne se pose pas en terme de procédure mais en terme de but à atteindre, donc des conditions qu'il faut réunir pour y arriver : en procédant à reculons, ces conditions sont alors considérées comme des sous-buts, lesquels demandent également la satisfaction d'autres conditions, qui elles-mêmes… récursivement.

Un peu comme si on était des saumons et qu'on remonte le courant : et ceci, d'une certaine façon, ressemble à la démarche du programmeur, décrite ci-dessous.

BOTTOM-UP | APPROCHE INCRÉMENTALE

Ascendant : c'est le premier aspect auquel est confronté le débutant, et cette démarche est incontournable : le potier a d'abord appris comment la glaise réagit à la pression, et dans quelle mesure la forme qu'il lui imprime persiste dans le temps.

C'est pourquoi l'apprenti programmeur est invité à manipuler du code directement sous l'interprète, pour acquérir le FEELING du langage ; ça suppose un minimum de curiosité, qui se manifeste comme une approche expérimentale : chaque expression est soigneusement testée avant d'être incorporée à une expression plus complexe. Et ceci vaut quel que soit le niveau : sous-expression dans une instruction, instruction dans une fonction, fonction dans un module, ou module dans un programme.

Il est évident ici que l'expérience du programmeur joue un rôle considérable, puisqu'il a déjà testé tellement d'expressions qu'il a, sans s'en rendre compte d'ailleurs, constitué un capital de savoirs-faire. Il va donc aller beaucoup plus vite puisqu'il peut *anticiper* l'évaluation de son code, donc coder directement des expressions complexes, voire des fonctions complètes.

Ce qui nous amène au 2ème aspect de la démarche…

TOP-DOWN | APPROCHE CONCEPTUELLE

Concevoir un programme, ça demande un effort d'abstraction, de projection. Le bon vieux principe *diviser pour régner* est ici la clé de la résolution de problèmes. Il faut s'arranger pour ramener le problème à des sous-problèmes plus faciles à résoudre, et récursivement… C'est pour cette raison qu'il m'arrive de coder des fonctions idiotes qui retournent ce qu'on leur a envoyé : faire comme si le problème était déjà résolu permet de conserver l'altitude.

Cette *déstructuration descendante*, c'est la partie *analyse* de la tâche de l'ANALYSTE-PROGRAMMEUR. Et en principe, l'analyse guide le *design* du programme jusque dans ses moindres détails, en partant de la forme originale de l'information, de sa représentation, donc des objets que le programme va manipuler, et des transformations qu'il leur faudra subir pour aboutir à la forme finale, le *résultat* du programme.

Remarquons que c'est une approche *orientée objet*, qui ressemble beaucoup à celle du potier *expérimenté* : si je veux obtenir ceci, il faut que je fasse ça — et quand j'aurai obtenu telle forme, il me sera plus facile de la décomposer pour en obtenir une autre, et une autre encore, jusqu'à ce je puisse recombiner uniquement celles qui m'intéressent : l'anse et le pot.

Voilà donc en définitive ce qu'on appelle le traitement de l'information : tout est *formes*, et l'art de

programmer est en fait l'art de *transformer* en appliquant des outils de décomposition, de sélection (filtrage), de recombinaison : une sorte de jeu de construction où les briques LEGO sont des données qu'on assemble, qu'on agrège, qu'on structure…

MIDDLE-OUT | APPROCHE PRAGMATIQUE

Tout ça, c'est très joli à dire, mais ce n'est pas comme ça que nous fonctionnons réellement : la conception est un cycle d'involutions, pour la démarche analytique, et d'évolutions, pour reprendre de l'altitude et synthétiser.

Si vous reprenez le chapitre précédent à son début, vous verrez que je m'occupe d'abord d'obtenir les données à traiter, le texte (une préoccupation globale) puis que je descends immédiatement très bas, au niveau d'une instruction élémentaire, `texte.split('\n') [0]` qui me donne une ligne, pour remonter avec une idée d'algorithme (boucle sur les lignes), puis redescendre de nouveau très bas pour considérer quoi mettre dans l'index, et encore une fois remonter avec une idée de boucle sur les mots d'une ligne : il me faut, alternativement, considérer 4 niveaux de données, du texte au caractère, jusqu'à la fonction `nettoie()`…

La démarche naturelle serait donc plutôt faite d'oscillations, d'allers/retours : mais remarquez que c'est *toujours* après examen des données que se prennent les décisions algorithmiques, jamais l'inverse.

Cette démarche, dite MIDDLE-OUT, procède à partir d'une idée de niveau intermédiaire, la constitution d'un dictionnaire, pour affiner progressivement les besoins, tout en gardant une « vue d'avion ». C'est pour cette raison que j'ai relégué aux derniers raffinements le nettoyage des mots, ou la présentation du résultat. Et il suffit d'un coup d'œil au code source de ce programme, pour voir que l'idée centrale, celle de `ajoute()`, s'est développée vers l'aval, mais qu'elle nécessitait des préparatifs en amont.

Ainsi, on peut très bien programmer sans savoir comment fonctionne un processeur, mais si on veut vraiment *tout* comprendre, il faudra un jour savoir quel type de signal est présent sur les pattes de la puce dans telle ou telle condition… Ou bien en ferons-nous *abstraction* ?

En pratique, comme on l'a montré dans les pages précédentes, certaines transformations peuvent être programmées directement à partir des types de données primitifs manipulés par des opérations fournies intrinsèquement pas le langage. C'est pourquoi l'apprentissage de la programmation passe généralement par la maîtrise des structures de données et de contrôle.

7.2 STRUCTURES DE DONNÉES

Cela va sans dire : il ne saurait être question de traiter de l'information si on ne pouvait la représenter en machine… Le mot NUMÉRISATION recouvre justement cet aspect de la conversion des données de notre monde analogique sous la forme de nombres :

- les caractères qui nous servent à écrire sont codés par des nombres entiers : par exemple, le `'b'` est représenté par le CODE ASCII 97, et le `'é'` par le CODE POINT 0xC3A9 ;
- l'état d'une ampoule électrique pourrait être simulé par une valeur logique (vrai, faux) selon qu'elle est allumée ou éteinte ;
- une image aura chacun de ses points représenté comme une combinaison RGB de rouge, de vert et de bleu, plus ou moins intense (RGB : red, green, blue) ;
- un son deviendra une série de nombres, mesures successives des variations de pression acoustique tous les 44 millièmes de seconde – du moins, en « qualité CD » ;
- la pollution pourrait se coder comme un vecteur de nombres réels représentant la proportion dans l'atmosphère des gaz les plus toxiques

Comme on le voit ci-dessus, seules les données *élémentaires*, comme les nombres et les caractères ASCII, peuvent être codées comme des *scalaires* (terme technique désignant un codage à échelle de valeurs) ; les autres supposent l'agrégat (ou *séquence*) de plusieurs scalaires, chacun représentant l'un des aspects de la donnée numérisée, comme, par exemple, l'intensité de la *composante* rouge d'un point lumineux.

Il y a donc plusieurs sortes de codage : plusieurs façons différentes de coder la même chose. Le choix du codage dépend, bien sûr, de l'application envisagée, autrement dit du rendement de ce codage pour un traitement spécifique : ainsi, à quoi bon représenter les composantes RGB d'une image en noir et blanc ?

Le problème n'est pas si trivial que ça, dans la mesure où les données sont souvent des *collections* organisées en plusieurs niveaux hiérarchisés ; par exemple, un agenda où chaque année comporte 12 mois, eux-mêmes décomposables en jours, pour lesquels chaque heure pourrait correspondre à un rendez-vous différent…

Nous avons vu 3 types scalaires primitifs :
- le type int, valeurs numériques entières : 3, 789, -15 ;
- le type float, valeurs numériques réelles : 3.0, 7.89, -1.5 ;
- le type bool (booléen), valeurs logiques : True, False ;

Nous avons manipulé 3 types primitifs de structures :
- le type str, ou séquence de caractères : '1 + 2 * 3' ou "aujourd'hui" ;
- le type list : [False, ['3/4 * 2', "crève-cœur"], -17.3] ;
- le type dict, ou dictionnaire : {1 : 'machin', 2 : 'truc', 3 : 'chose'} ;

Nous avons également manipulé des *références*, en particulier :
- des objets de type file, référençant des fichiers sur disque ou en réseau ;
- des fonctions (et des méthodes), dont le nom symbolique réfère à un bout de code défini dans l'environnement de l'interprète ;
- et, de façon transparente, les variables et autres symboles – en fait, il n'y a que des références, partout….

Cette récapitulation sommaire ne veut pas dire qu'il n'y ait pas d'autres types, mais pour le moment, ceux-là sont suffisants pour se débrouiller dans la plupart des cas… Et notez que, comme en LISP, il n'y a pas de différence formelle entre données et code exécutable : c'est juste une question de point de vue, d'interprétation ; d'ailleurs, incidemment, comme en LISP, tous les objets manipulés en PYTHON sont des *références*.

7.3 STRUCTURES DE CONTRÔLE

Comme on l'a dit au début, un programme est une *séquence* d'instructions – en tout cas pour les langages procéduraux (il est vrai que LISP, SQUEAK ou PROLOG se présentent différemment). Admettons que, pour ce qui nous occupe, la séquence soit la forme la plus primitive de programme : une séquence est, en principe, évaluée dans l'ordre où elle se présente.

Et ça, il y a des fois où ça ne nous arrange pas ; il viendra un moment où le programme devra décider tout seul, en fonction de l'état des données, ce qu'il convient de faire, et c'est pour ça que nous avons introduit l'exécution *conditionnelle*, aussi appelée *branchement* (dans le sens de *bifurcation*) : on peut ainsi « sauter » une ou plusieurs instructions de la séquence si une certaine condition n'est pas satisfaite.

Mieux, encore : au lieu de tester explicitement des conditions, il est possible de « sauter » une instruction dans le cas où elle provoquerait une erreur : il suffit d'essayer, et si ça ne marche pas on fait autre chose… c'est la gestion d'exception, qu'on retrouvera en LISP avec catch/throw, et en OBJECTIVE C (ou c++) avec try/catch.

Et puis nous avons vu le cas où certaines parties de la séquence doivent être répétées un certain nombre de fois : c'est l'*itération*.

Ensuite, nous avons appris à manipuler une forme assez spéciale, la *fonction*, qui est un bout de code défini indépendamment, utilisable dans une expression, à condition de lui passer les arguments qui lui permettent de *paramétrer* son comportement. Utiliser une fonction revient donc à « dérouter » temporairement la séquence d'expressions en cours d'évaluation pour en évaluer une autre.

Enfin, les fonctions peuvent elles aussi « sauter » des instructions de leur propre séquence, en évaluant l'instruction return. C'est encore une forme de *déroutement* de la séquence en cours : elle a pour effet de retourner juste après le point d'où la fonction avait été appelée.

Nous avons ainsi vu 6 façons fondamentales de contrôler le déroulement d'un programme :

1. la cascade, ou séquence d'opérations ;

2. l'exécution conditionnelle, introduite par l'instruction if, contrôlée par une expression booléenne ;
3. la gestion d'exceptions[28], introduite par l'instruction try, assortie de son acolyte, except ;
4. l'itération, implicitement exhaustive (avec for) ou explicitement conditionnelle – contrôlée par une expression logique (avec while) ;
5. l'appel de fonction, y compris l'effet sophistiqué de eval() ;
6. le retour au point d'appel par l'instruction return.

Nous ferons, pour le moment, l'impasse sur certaines structures de contrôle : en particulier toutes celles qui relèvent d'une approche *fonctionnelle* de la programmation, aspect qui sera de toute façon étudié en détail dans un cours spécialisé[29].

7.4 OBJETS

Mais il est une autre façon d'aborder la programmation : c'est l'approche par objets, qui permet une autre façon de structurer les données que manipulent les programmes, et du même coup, une autre façon de contrôler le déroulement de ces programmes.

MÉCANISME D'HÉRITAGE

Le point fort des langages à objets, c'est que les types de données sont eux-mêmes des sous-types d'objets plus abstraits. L'intérêt d'une telle architecture hiérarchique est qu'il est alors possible de définir des propriétés, des comportements, à un niveau *abstrait* sachant qu'on dispose d'un mécanisme d'héritage qui permet au sous-type *concret* d'exploiter les modalités de comportement définies plus haut dans la hiérarchie.

Par exemple, toutes les séquences se comportent de la même façon, que ce soit des listes ou des chaînes de caractères ; elles exploitent ainsi en commun un certain nombre de méthodes :

```
[3, 2, 1, 2, 1, 2].count(2)                    # retourne la valeur 3
'œil de bœuf'.count('œ')                       # retourne la valeur 2
```

fonctionnent de manière similaire, retournant le nombre d'occurrences de l'élément donné comme argument à la méthode count(), méthode applicable à toute séquence.

Pour le programmeur, ça veut dire moins de code à écrire, puisqu'il peut « factoriser » des comportements qui seront automatiquement hérités. Ainsi, toutes les séquences ont une méthode pour faire + et une pour faire * ; bien entendu, il y a des choses qu'on peut faire avec des listes, mais pas avec des chaînes (et inversement), ce qui veut dire que certaines méthodes ne doivent être définies qu'au niveau du sous-type concerné.

Un autre avantage n'a pas encore été mis en évidence : c'est le fait que tout objet hérite des propriétés du type dont il est un représentant. On l'a bien compris pour ce qui est des propriétés comportementales (ou dynamiques), mais c'est également vrai pour les propriétés statiques, techniquement appelées *attributs* ; un exemple va éclaircir ce point sur-le-champ.

Considérez les définitions suivantes – pour des raisons historiques, les *types* s'appellent ici des *classes*, ce qui est l'inverse de ce qu'on entend par là en mathématiques ou en philosophie, où les classes sont définies en EXTENSION, alors que les types sont définis en INTENSION.

Le mot-clé class permet de définir une nouvelle CLASSE – autrement dit un nouveau TYPE – et d'y définir des attributs ; dans l'exemple suivant, pour faire simple, je n'en ai défini qu'un à chaque niveau, et je n'ai défini *aucune* méthode (de cet aspect des choses, nous verrons un exemple un peu plus tard). Une expression de la forme class A (B) : … définit la classe A comme héritière de la classe B ; techniquement, on dit que l'objet A est une *instance* de la classe B :

28 mais voir aussi http://oranlooney.com/lbyl-vs-eafp/
29 autrement dit http://foad.iedparis8.net/claroline/courses/E3c0/

```
class Mammifere (object) : prop1 = "j'allaite mes petits ;"

class Carnivore (Mammifere) : prop2 = "je me nourris de chair ;"

class Chien (Carnivore) : prop3 = "j'aboie..."

class Chat (Carnivore) : prop3 = "je miaule..."
```

On voit dans ces quatre lignes la définition de quatre classes, où Chien et Chat sont des *instances* de Carnivore, qui est elle-même une instance de Mammifere (instance de object). Maintenant, nous allons créer encore trois nouveaux objets : deux *instances* de Chien, et une de Chat[30].

```
Milou = Chien()
Gromit = Chien()
Garfield = Chat()
```

Une expression de la forme A = B() crée l'objet A en tant qu'*instance terminale* de la classe B ; par instance terminale, j'entends *individu*, et non plus une classe capable de générer encore d'autres instances… Ceci fait, on fait cracher à ces objets ce qu'ils ont dans le ventre :

```
print 'Milou :', Milou.prop1, Milou.prop2, Milou.prop3
print 'Gromit :', Gromit.prop1, Gromit.prop2, Gromit.prop3
print 'Garfield :', Garfield.prop1, Garfield.prop2, Garfield.prop3
```

Ici, une expression comme Garfield.prop1 dénote la propriété ou attribut prop1 de l'objet Garfield ; or il n'y a aucune propriété nommée prop1 dans l'objet Garfield ; cependant, s'il l'affiche, c'est bien qu'il la tire de quelque part ; et ce quelque part, c'est justement de la classe Mammifere dont il hérite les propriétés par transitivité, puisque la classe Chat, dont il est une instance, hérite elle-même de la classe Carnivore, qui hérite, elle, directement de Mammifere.

Et c'est évidemment le même mécanisme pour les propriétés prop2 et prop3…

On voit donc que, bien qu'il n'y ait *aucun* attribut défini localement dans l'objet Garfield, il hérite tout de même de toute la hiérarchie dont il est issu. Mais le plus intéressant, c'est que si maintenant je définis, directement dans l'objet Garfield, un attribut de même nom que l'un de ceux dont il aurait normalement hérité, la nouvelle définition *inhibe* l'héritage :

```
Garfield.prop1 = "je n'ai pas de petits ;"
Garfield.prop2 = "j'adore les lasagnes ;"
print 'Garfield :', Garfield.prop1, Garfield.prop2, Garfield.prop3
```

[px30-1] entrez ces définitions dans l'interprète PYTHON, ajoutez-y de nouveaux attributs, et faites cracher vous-même à ces objets ce qu'ils ont dans le ventre…

On peut programmer de nouvelles méthodes dans les classes et régler ainsi le comportement générique de chaque objet ; et définir un attribut de même nom dans une instance terminale masquera cette méthode.

Par exemple, supposons que j'ai besoin de manipuler de objets de type STACK, c'est-à-dire *pile* ; de tels objets acceptent 3 opérations fondamentales :

- initialisation : une pile démarre, en principe, à vide ;
- empilement, qui se dit d'habitude *push*, dans les langages qui supportent ce type d'objet ; typiquement Lisp, mais aussi l'assembleur…
- dépilement, qui se dit *pop* dans ces mêmes langages ;
- et accessoirement, on pourrait aussi demander la hauteur de la pile…

La pile est le type fondamental en LISP, où elle est appelée liste, et s'augmente de droite à gauche grâce à cons(), la fonction de construction. Le « dessus » de la pile est donc, par définition, le dernier élément empilé, qui devra donc être traité en premier, selon le principe : le dernier arrivé est le premier servi !

Voici comment définir ce type en PYTHON :

30 par pure convention, les noms de classes sont typographiés avec une capitale initiale

```
class Stack (object) :                          # par convention : initiale capitale pour les noms de classes
   def __init__(self) : self.stack = []         # nom de méthode réservé à cet usage
   def push(self, val) : self.stack.append(val) # méthode spécifique
   def pop(self) : return self.stack.pop()      # méthode spécifique
```

Toute définition de fonction depuis l'intérieur d'une classe est implicitement une définition de *méthode*, et doit avoir au moins un paramètre, nommé conventionnellement self (mais peu importe, en fait) qui sera pourvu automatiquement au moment de l'invocation de la méthode, et qui représente l'objet qui invoque la méthode, obligatoirement une instance de cette classe.

Une instance n'est pas une variable, et ne peut donc prendre directement de valeur ; mais on peut y définir des attributs, ou propriétés, qui se comportent comme des variables propres à l'instance (ou à la classe) : self.stack est un exemple typique d'attribut local à l'instance, créé par déclenchement automatique de la méthode __init__() au moment de l'initialisation.

Quand une méthode nommée __init__() est définie pour une classe, elle est automatiquement invoquée au moment de l'instantiation de cette classe ; ainsi donc, si j'évalue maintenant :

```
p = Stack()                                     # p est maintenant une instance de Stack
```

l'instance p a automatiquement été pourvue d'un attribut stack, initialisé comme une liste vide par la méthode __init__() ; la preuve :

```
>>> p                                          # instance de Stack dans le module __main__
<__main__.Stack instance at 0x5f9b8>
>>> p.stack                                    # valeur de l'attribut stack de p
[]
```

Maintenant, évaluez interactivement ces quelques lignes :

```
for x in 'un deux trois'.split() : p.push(x)

p.stack                                         # → ['un', 'deux', 'trois']
x = p.pop() ; x ; p.stack                       # → 'trois' -- ['un', 'deux']
x = p.pop() ; x ; p.stack                       # → 'deux' -- ['un']
```

pour avoir le FEELING de comment ça marche ; notez qu'on peut aussi (quand on connaît son nom) manipuler directement la définition de l'attribut ; par exemple, évaluer p.stack += ['chose'].

[px30-2] définir la classe Stack et une instance d'icelle, puis tester l'instance ; définir (et tester) une méthode size() sur le même schéma que la méthode pop() ; redéfinir pop() avec un try/except pour que, comme en LISP, cette méthode retourne [] sans provoquer d'erreur quand la pile est déjà vide (tester ce cas de figure)

C'est bien d'avoir un type de donnée qui se comporte comme une pile, mais en y regardant de près, ça ne m'avance pas terriblement par rapport au comportement natif d'une liste, mis à part que le nom des méthodes de cette classe sont conformes à l'usage. Mais là où ça devient intéressant, c'est quand je veux définir une queue : la différence entre une pile et une queue, c'est que dans le premier type, le dernier arrivé est le premier servi, alors que dans le deuxième, c'est le premier arrivé qui est le premier servi ; l'exemple classique est celui de la queue d'imprimante où les JOBS successifs sont traités dans l'ordre de leur soumission.

```
class Queue (Stack) :
   def pop(self) : return self.stack.pop(0)
```

La syntaxe de cette définition spécifie que :

- Queue est comme une pile, y compris pour ce qui est de l'initialisation ;
- sauf que le dépilement se fait à partir du *premier* empilé, pas du dernier.

De fait, Queue est défini comme une sous-classe de Stack, et partage les mêmes ressources, sauf celles qui seraient redéfinies localement, comme c'est le cas ici de la méthode pop() , alors que les méthodes __init__() et push() [ainsi que size(), si définie] sont ici communes aux deux classes.

Essayez-donc ceci :

```
q = Queue() ; q.stack                  # → []
q.push('truc') ; q.stack               # → ['truc']
q.push('machin') ; q.stack             # → ['truc', 'machin']
q.push('chose') ; q.stack              # → ['truc', 'machin', 'chose']
q.size()                               # → 3 # défini dans Stack par l'exercice 22-2
x = q.pop() ; x ; q.stack              # → 'truc' -- ['machin', 'chose']
x = q.pop() ; x ; q.stack              # → 'machin' -- ['chose']
```

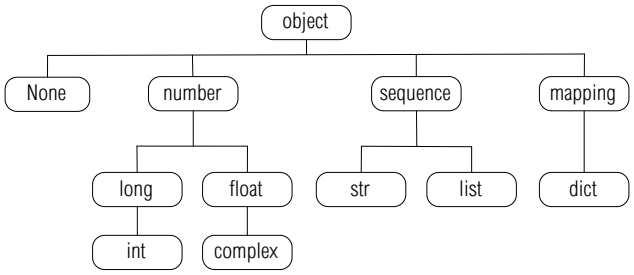
[px30-3] définir la classe Queue puis en tester des instances pour simuler plusieurs queues aux caisses du supermarché le plus proche

Bien entendu, c'est au programmeur qu'il incombe de s'assurer qu'il construit un ensemble cohérent : au niveau des *piles* et des *queues*, ça ne paraît pas bien complexe, mais ça pourrait le devenir pour des applications plus sophistiquées…

L'important, ici, est que l'approche orientée objet me permet de déporter le code générique dans la classe elle-même, donc, par la suite, de manipuler ses instances spécifiques sans avoir à programmer ce comportement à chaque fois : une classe est une sorte d'abstraction qui, une fois codée, permet de s'affranchir des détails comportementaux. Si je fabrique une *pile*, elle démarre à vide, comme il faut… Et je n'ai plus à me soucier de *comment* on dépile une *queue* : je définis une instance de *queue*, et j'empile et dépile à l'envi, comme ça doit naturellement se passer dans une *queue*.

Le mot-clé, ici, c'est *naturellement* : on code le comportement des objets pour qu'il soit aussi intuitif que possible. La programmation est ainsi simplifiée, ce qui permet en contrepartie de concevoir, à moindre effort, des programmes plus complexes : comme manipuler l'icône d'un fichier pour le déplacer effectivement, chose *inimaginable* il y a seulement 25 ans.

La clé de ce comportement, c'est qu'il y a, derrière tout ça, un mécanisme d'héritage qui se débrouille pour retrouver ce dont on a besoin, qu'il soit attribut statique (*propriété*) ou dynamique (*comportement*), quel que soit le niveau où il est défini dans la hiérarchie des classes — souvenez-vous de Garfield : la redéfinition *locale* d'un attribut a pour effet de masquer l'héritage normal.



Et voilà pourquoi '123' + '456' ne produit pas le même résultat que 123 + 456, pas plus, d'ailleurs que les VERBES ne forment leur pluriel comme les NOMS : str, int, VERBES et NOMS sont des types distincts, et ont donc des comportements distincts — et pas que pour la formation du pluriel !

INTERFACES GRAPHIQUES

La manière dont on avait, jusqu'à présent, structuré les programmes graphiques du chapitre 5 est qualifiée de INLINE CODING, et c'est une démarche procédurale, qui s'oppose donc à l'approche structurée sous la forme d'objets : *classes*, *méthodes* et *instances*.

Or le module Tkinter est résolument orienté objet, et ne vous laisse manipuler que des instances des classes Tk, Button, etc… Ce module est un FRAMEWORK, dans le sens où en appelant la méthode loop() d'une instance de Tk, toutes les ressources de Tkinter se mettent à la disposition de votre application.

Je ne suis pas sûr que ça séduise quiconque de prime abord, vu l'aspect rébarbatif du code (ne vous inquiétez pas, on finit par s'y faire) mais il me semblait utile ici de montrer comment programmer (à deux boutons près) le px20-1 avec l'approche objet.

Mais quel est donc l'avantage de cette approche, par rapport à notre façon de faire habituelle ? Réponse : ici, pratiquement, aucun avantage… mais théoriquement, et surtout dans le cas d'applications complexes, on verra très vite l'intérêt de regrouper sous un nom un ensemble de propriétés statiques (les attributs) ou dynamiques (les méthodes), surtout quand on sait qu'une classe est aussi un espace de nommage : on évite

ainsi les conflits entre des objets de même nom — par exemple, un bouton nommé stop qui figurerait dans de multiples fenêtres.

En pratique l'avantage de l'approche objet est de conférer aux données le même statut modulaire que celui d'une fonction : les données d'un objet sont isolées de celles des autres, et les méthodes de ce type d'objet sont « encapsulées » avec. C'est le principe fondamental d'otthogonalité : s'arranger pour minimiser les effets secondaires qui peuvent se produire lors de la mise au point. Ici, notre application n'a qu'une fenêtre, mais pour une réalisation plus complexe, l'approche INLINE CODING va vite devenir intenable.

La définition de la classe Salutations, ci-dessous, correspond aux principes généraux exposés au début du § 7.4 ; remarquez au passage que, dans la définition d'une classe, la définition des méthodes de cette classe est indentée d'un cran.

Les deux premières méthodes, dis_bonjour() et dis_au_revoir() sont telles que je les aurais codées au TOP LEVEL, à ceci près que j'ai dû leur rajouter un paramètre self : c'est ce qui permet au mécanisme d'héritage de suppléer le bon objet, même si la méthode était héritée 17 niveaux plus bas. La raison pour laquelle je les ai placées en haut de la définition, c'est qu'elles doivent être définies avant que la troisième n'essaye de les lier à leur bouton respectif.

La troisième, __init__() est le nom de la méthode invoquée de façon automatique lorsqu'on va instancier cette classe ; cette méthode prend ici deux paramètres :

- le 1er, nommé conventionnellement self, ne correspond pas à un argument explicite ; il est implicitement pourvu, et représente l'instance qui invoque la méthode — dans notre cas, ici, ce sera prog ;
- le 2e, appelé ici W, correspond au *widget maître* : la fenêtre déjà créée, pièce maîtresse de l'interface – instance de Tk, définie ici sous le nom de top, et passée en argument à la création de l'instance prog, avec prog = Salutations(top).

Le reste du code est similaire à ce qu'on aurait écrit en dehors d'une définition de classe, c'est-à-dire que j'ai simplement ajouté self. avant le nom des objets créés pour l'interface, mais aussi pour les méthodes liées aux boutons… L'objectif, c'est qu'un objet issu de cette classe ne connaisse, et donc n'utilise, *que* les méthodes de sa classe.

```
from Tkinter import *

class Salutations :
   def bonjour(self) : print "salut, c'est moi"
   def aurvoir(self) : print 'ciao, a+'
   def __init__(self, W) :
      self.stop = Button(W, text='assez', fg='red', command=W.destroy)
      self.hello = Button(W, text='dis "bonjour"', command=self.bonjour)
      self.good_bye = Button(W, text='dis "au revoir"', command=self.aurvoir)
      self.hello.pack()
      self.good_bye.pack()
      self.stop.pack()

top = Tk()
prog = Salutations(top)
top.mainloop()
```

Du fait que j'ai défini une méthode __init__(), toute instance de Salutations sera *pré-meublée* avec des boutons hello, good_bye, et stop (et n'auront donc pas besoin d'en hériter) : on peut d'ailleurs difficilement imaginer qu'il en soit autrement, puisque ces boutons doivent *impérativement* être présents au moment où on appellera la méthode pack(). Les méthodes, par contre, ne sont jamais recopiées dans l'instance : elles sont obligatoirement héritées.

Peut-être que ça vous aiderait à clarifier les choses, de prendre votre éditeur de texte et, dans le code, de remplacer :

- W par window
- self par instance

Mais ce ne sont que des symboles : comprendre ce qu'ils représentent est plus important…

[px30-4] testez le code ci-dessus ; notez que les méthodes bonjour() et aurvoir() utilisent print : c'est donc sur votre *terminal* que vous verrez s'afficher leurs messages.

Et peu importe l'ordre des définitions de boutons : leur placement correspond à l'ordre dans lequel leur méthode pack() est appelée.

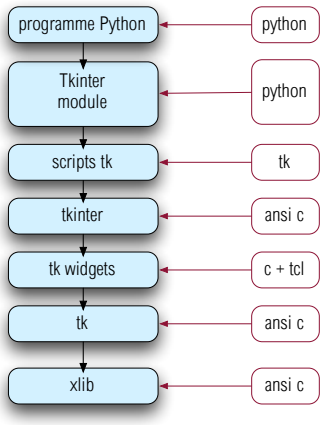
7.5 STRATES LOGICIELLES

L'objectif de cette rapide présentation est de vous permettre de comprendre comment fonctionne le système graphique qui tourne derrière PYTHON, et qui utilise la XLIB à travers des appels à des modules codés en TCL[31] [prononcer “tickle”] et en ANSI-C ; car, pour ceux que ça peut intéresser, PYTHON est interfaçable, dans les 2 sens, avec d'autres langages, de script ou pas…

Votre programme, codé en PYTHON, n'est jamais que la 7e couche qui s'ajoute par dessus le module Tkinter, lui aussi entièrement codé en PYTHON, mais s'appuyant sur Widget.itk, labeledwidget.itk, et scrolledwidget.itk, des scripts codés en TK qui utilisent le module tkinter (en minuscules, notez la différence) codé, lui, en ANSI-C.

Ce tkinter utilise le module de bibliothèque itk, également compilé à partir de sources en ANSI-C et en TCL, et qui s'appuient sur les ressources de tk, compilé à partir de code ANSI-C.

Quant à tk, il exploite les ressources du module de bibliothèque xlib, lui aussi codé en ANSI-C : un chapitre entier du cours 213 sera consacré à cette technique pour interfacer différents langages de scripts et des modules compilés de bibliothèque.



Remarquez que TK est un langage orienté objet, et que c'est lui qui permet à Tkinter d'organiser notre système de WIDGETS (raccourci pour WINDOW GADGETS) de façon hiérarchique ; ce module Tkinter doit être importé avant de pouvoir être exploité, c'est pourquoi la 1ère ligne de nos scripts est toujours : from Tkinter import *.

À partir de là, une expression comme fenêtre = Tk() crée une instance de la classe Tk nommée fenêtre, qui se réalise comme un conteneur : une *fenêtre* qui va servir à contenir tous les autres *widget*s dont je pourrais avoir besoin pour construire mon interface.

Ensuite, selon les besoins, je créerai des objets de type Frame, eux aussi des conteneurs, ou des instances de la classe Canvas, qui est la seule à connaître des méthodes pour dessiner. Et je pourrai entrer du texte dans des instances de la classe Entry, ou en afficher dans des instances de la classe Label. Tout ça, contrôlé au moyen d'instances de la classe Button, liées aux fonctions de l'application elle-même (voir encadré page suivante).

Tous ces objets disposent, bien sûr, de méthodes, génériques ou spécifiques, et sont munis de propriétés : un univers si riche qu'il n'est pas question ici de l'inventorier — pour ce que nous allons en faire, il nous suffira d'entr'ouvrir quelques portes ; je veux dire : *fenêtres*.

Et c'est ce que nous avons vu en détail dans le chapitre 5, et que nous allons revisiter en traversant le chapitre 8, qui prétend montrer comment programmer n'importe quelle interface pour n'importe quelle application…

Bien entendu, ceci n'est jamais qu'une *introduction* à la programmation, et non une formation pour *designer graphique*. Mais vous pourrez toujours approfondir la question en lisant des bouquins spécialisés, dans la mesure où vous maîtrisez le jargon incontournable de ce genre de littérature ; c'est pour cette raison que je donne, autant que possible, le terme technique, et éventuellement, l'original anglo-saxon dont il a été traduit — et parfois même *trahi*.

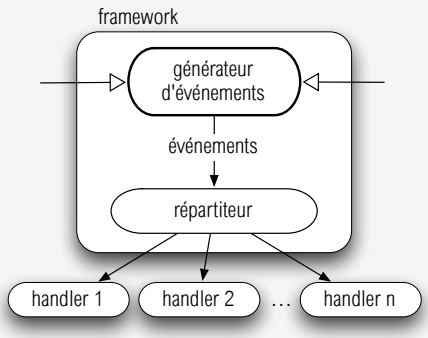
31. demandez-donc man Tk_SetOptions pour en mesurer la complexité…

Avec l'interface Tkinter, un autre concept-clé est à l'œuvre derrière le système de classes et d'instances, celui de la programmation *dirigée par les événements* (EVENT DRIVEN PROGRAMMING), de sorte que des événements extérieurs au programme, comme l'enfoncement d'une touche du clavier ou des mouvements de souris, vont déclencher l'activation de certaines fonctions expressément désignées pour les gérer.

Tout se passe, en fait, comme si votre programme était un PLUG-IN connecté au FRAMEWORK Tkinter, ce qui vous permet de spécifier quels événements vous intéressent, ainsi que ce que vous voulez en faire…

Le concept de FRAMEWORK (charpente) est relativement récent (1985) : il s'agit d'un ensemble de ressources rassemblées dans un programme qui ne fait rien par lui-même (une sorte de bibliothèque), mais qui permet à d'autres programmes de se brancher (PLUG) sur son interface afin d'en exploiter les fonctionnalités.

Dans une telle architecture, ce n'est pas votre PLUG-IN qui contrôle le déroulement global du programme : c'est le FRAMEWORK qui lui délègue ponctuellement la gestion (HANDLING) de l'événement qu'il a intercepté…



Le concept de *framework* est tellement commode qu'il est maintenant employé pour un bon nombre d'autres fonctionnalités du système d'exploitation : par exemple, tous les programmes à frontal graphique se servent du même *framework* pour l'édition de texte ; l'avantage du point de vue du programmeur, c'est qu'il n'a pas à recoder les fonctions de bas niveau ; et du point de vue de l'utilisateur, ça unifie l'interface : si le double clic permet de sélectionner un mot, ce sera pareil partout — tout le monde y trouve son compte !

Mais ceci n'est possible que si le *framework* est orienté objet : car c'est le mécanisme d'héritage qui permet cette transmission des propriétés, donc qui non seulement unifie l'aspect des interfaces, mais aussi unifie les comportements en réaction aux événements gérés par le *framework*.

Cet aspect du système sera étudié en *L2*, mais en attendant, Tkinter permet de se faire une bonne idée de la façon dont fonctionne cet empilement de strates logicielles, tant du point de vue de la conception que du point de vue de l'exploitation. Il permet aussi de mesurer l'intérêt de programmer par objets, et explique pourquoi ce paradigme de programmation connaît actuellement un tel succès dans l'industrie…

NOTES

suppléments sur quelques concepts

python tutorial – classes	http://mandrivalinux.hu/~hron/python24/tut/node11.html
framework	http://visualwikipedia.com/en/Software_framework
event-driven programming	http://eventdrivenpgm.sourceforge.net/
error handling strategies[32]	http://oranlooney.com/lbyl-vs-eafp/

pour plus d'information, voir

TkInter	http://wiki.python.org/moin/TkInter
Tkinter reference	http://infohost.nmt.edu/tcc/help/pubs/tkinter/
thinking in Tkinter	http://www.ferg.org/thinking_in_tkinter/index.html
Tkinter Wiki	http://tkinter.unpythonic.net/wiki/
python interface to Tcl/Tk	http://docs.python.org/library/tkinter.html

32 voir aussi le chapitre 6 de PYTHON IN A NUTSHELL (2003) : http://docstore.mik.ua/orelly/other/python/0596001886_pythonian-chp-6-sect-6.html

RÉCAPITULATION

Considérez ce chapitre comme une sorte de rétrospective culturelle : un bilan des concepts que nous avons apprivoisés jusque-là… De la notion d'itération jusqu'à l'encapsulage de méthodes dans un objet, c'est 60 ans d'histoire d'une technologie encore balbutiante et pourtant tellement complexe qu'on a déjà du mal à en avoir une vision globale !

On peut n'en rien retenir, mais ces quelques pages sont là pour élargir votre cadrage et votre profondeur de champ : qu'on ait réussi à vous présenter les choses simplement ne change rien à la complexité sous-jacente, complexité qu'il vous faudra prendre à bras le corps si vous avez quelque part l'ambition de comprendre comment c'est foutu, et devenir capable de réaliser par vous-même quelque chose qui tient debout.

Ces notions, dont vous avez maintenant une teinture, sont sans doute fondamentales, mais vous aurez bien d'autres occasions de les revisiter, que ce soit en étudiant d'autres langages ou en réalisant des applications plus poussées.

Car ces concepts sont applicables à tous les langages de programmation, même ceux qu'on dit déclaratifs : les structures de données et les structures de contrôle y existent tout pareil, mais de manière totalement implicite, pour en dissimuler la complexité : celle-ci fera l'objet d'une section entière de l'ÉC 312[33], en L2… pour votre culture, encore une fois.

Souvenez-vous qu'un ingénieur, c'est un technicien cultivé : cette culture lui permet, certes, de choisir la solution la plus efficace parmi celle qu'il connaît, mais développe aussi son esprit critique, l'amenant à inventer de nouvelles solutions si aucune de celles qu'il a à sa disposition n'est vraiment bien adaptée pour résoudre son problème particulier[34].

De cette manière de voir les choses découlent les progrès spectaculaires de la recherche et de l'industrie informatique depuis 60 ans, et plus particulièrement depuis 1976, date à laquelle l'approche objet a commencé à se généraliser et les systèmes de fenêtrage à se vulgariser, même si le tout premier, celui de XEROX, est passé à peu près inaperçu auprès du grand public.

Aujourd'hui, l'interface tactile tend à remplacer la souris, conçue par XEROX il y a déjà 35 ans ; alors, pour demain, qu'allez-vous inventer ?

33. types de données abstraits : conception et mise en œuvre
34. à ce propos, lisez-donc l'épilogue à la fin du chapitre 8…

⑧ PROTOTYPAGE D'APPLICATIONS

Ou comment réaliser de véritables prototypes de projets avec le savoir-faire que vous venez d'acquérir. Le prototypage reflète une tendance relativement récente[35] dans l'industrie qui se préoccupe de plus en plus de créer des logiciels centrés sur les besoins réels des utilisateurs, et capables d'évoluer rapidement.

L'accent est mis sur le design du programme, ou comment regarder le problème pour y voir une solution facile et économique. La preuve en est que le code ne dépasse jamais une page-écran, limite au-delà de laquelle je perds mon altitude, en me perdant moi-même…

Qui a dit « *small is beautiful* » ? Je ne sais plus, mais une chose est certaine : quand il s'agit de coder ou maintenir des programmes, plus c'est petit, plus c'est facile.

C'est dans cette optique que nous ferons usage de la technique dite *dirigée par les données*, non seulement parce qu'elle déporte, dans les données, du code parfois encombrant, mais aussi parce que, à moins de recourir au MULTITHREADING, c'est quelquefois la solution la plus élégante pour simuler l'activation parallèle de deux programmes distincts, pas tout à fait indépendants, mais concurrents.

Enfin, on y verra comment mettre en œuvre les techniques fondamentales d'un JIT[36] à travers l'émulation d'un processeur fictif, autrement dit d'une *machine virtuelle* manipulant du BYTE CODE, concept-clé des interprètes modernes, comme JAVA, PYTHON, et bien d'autres… Même si dans cette maquette, le code du *driver* peut être ramené à trois instructions, il servira de base pour comprendre, plus tard, les principes d'un véritable compilateur.

sommaire

35 approximativement, au tournant du millénaire : cf. http://agilemanifesto.org
36 JUST-IN-TIME COMPILER ; cf. http://en.wikipedia.org/wiki/Just-in-time_compilation

Depuis les années 80 fleurissent de-ci de-là des méthodologies dites de « *développement rapide* » : en effet, RAD[37] est devenu un BUZZ WORD très en vogue, mais force est de constater que, globalement, ces méthodologies n'améliorent pas le rendement des équipes de développement de manière *vraiment* significative.

Agile, comme Extreme Programming (XP), ou Joint Application Development (JAD), de même que Lean Software Development (LD), Rapid Application Development (RAD) ou Scrum, toutes se heurtent à un obstacle majeur : aucune de ces méthodologies ne tient compte la personnalité du programmeur, ni des aspects psychologiques de son intégration au sein d'une équipe ; elles ne conviennent donc vraiment qu'à certains cas particuliers — notamment le cas de celui qui a pris la peine de l'élaborer…

Mais dans leur principe, elles ont quelque chose de bon.

1. elles sont « minimalistes » : elles sont réductrices, mais dans le bon sens du terme, autrement dit simplifiantes, tout en préservant la capacité de récupérer ultérieurement la complexité
2. elles donnent priorité à la modélisation du problème à partir de ses données, c'est-à-dire qu'elles en privilégient l'aspect DÉCLARATIF, minimisant ainsi l'aspect IMPÉRATIF de la programmation — elles sont d'ailleurs toutes orientées objet
3. elles privilégient le développement incrémental, visant à produire aussi vite que possible un prototype incomplet, perfectible, mais suffisant pour servir de base à la discussion avec le client, considéré comme faisant partie de l'équipe de développement

En fait, elles correspondent à un besoin du marché : l'industrie est devenue consommatrice de petits programmes dont on a un besoin immédiat, mais qu'on jette après usage, comme un mouchoir en papier.

> L'exemple typique est celui du secteur SPECIAL EFFECTS de l'industrie cinématographique : LUCAS FILMS recrute des programmeurs de post-production, capables de torcher en quelques minutes un bout de code qui simule une boîte de dialogue ou une barre de progression, qu'on utilisera jamais plus, ni pour cette réalisation, ni ailleurs.

Certes, ça ne veut pas dire qu'il n'y a plus de place pour les programmeurs de longue haleine pour des développements sur plusieurs années, mais la tendance (et la préférence) va à la programmation prototypique, celle qui peut très vite démontrer la faisabilité d'un concept, donc donner rapidement au MANAGER une idée du coût économique de la solution, faute de quoi, il ne prendra pas le risque d'y investir ses ressources humaines.

Les prototypes présentés ici exploitent chacun une façon particulière de modéliser les données du problème en vue d'en faciliter le traitement, donc d'accélérer le codage du programme.

- MORPION : structures de données parallèles de niveaux différents
- SEGMENTS : décompose les données pour faciliter leur recomposition à la volée
- PENDU : simulation de processus concurrents coopératifs
- L'ORDINATEUR EN PAPIER : filtrage dynamique indexé par les données

Ils ont tous un point commun : la partie CODE proprement dite est ridiculement réduite, même si ça implique d'élaborer avec soin la structure de données la plus efficace — aucun de ces prototypes ne dépasse une page au total…

Les exercices proposés visent à démontrer les points 1 à 3 ci-dessus : la réduction initiale n'empêche pas les développements ultérieurs, eux-aussi toujours réalisables à moindres frais.

37 cf. http://en.wikipedia.org/wiki/Rapid_application_development

Comme on l'avait annoncé quelques pages plus haut, les langages de scripts en général (et PYTHON en particulier pour ce qui nous concerne) constituent l'environnement idéal, non seulement pour le programmeur débutant, mais aussi pour le programmeur qui débute une application : c'est le prototypage.

Dans un contexte industriel, la phase 0 d'un projet, c'est son cahier des charges, c'est-à-dire les spécifications de l'application telles que stipulées par le client :

- les données d'entrée et de sortie
- ce que doit faire l'application avec ces données
- comment elle doit les présenter, autrement dit l'interface

La phase 1, c'est l'étude de faisabilité (PROOF OF CONCEPT) : une évaluation des besoins, tant en ressources humaines (et en compétences) qu'en ressources logicielles (numérisation des données, pré-traitement, conversions, rédaction du code, tests de mise au point).

La phase 2, c'est la réalisation proprement dite, et le temps requis dépend de la qualité de la phase 1 – auquel il faut bien sûr ajouter des délais pour la finalisation et les modifications de dernière minute. Et comme dans l'industrie le rendement d'une équipe de développement influe directement sur les coûts, la tendance est de favoriser la phase 1, renommée, à juste titre, « prototypage ».

Pour l'apprenti-programmeur, les problèmes se présentent de la même façon, à cette différence près que son inexpérience ajoute un certain flou à la phase 1 : il n'est donc pas question de commencer directement par la phase 2 comme le font hélas beaucoup de débutants…

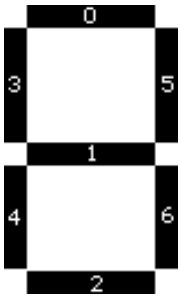
Et si cette profession est appelée “analyste-programmeur”, ce n'est pas par hasard : l'analyse du problème précède obligatoirement sa résolution.

8.1 AFFICHAGE PAR SEGMENTS

Je dois réaliser un afficheur alpha-numérique façon LCD, qui puisse afficher du texte dans une fenêtre, chaque caractère étant dessiné par des segments. Comment vais-je m'y prendre ?

ANALYSE

Dans sa version primitive, la plus simple, un afficheur LCD n'a que 7 segments possibles, comme sur la figure ci-contre :



Pour les repérer, je numérote les segments horizontaux de 0 à 2, et les seg-ments verticaux de 3 à 6 : ceci est donc purement conventionnel – d'ailleurs, je n'ai pas vraiment de raison de commencer par les segments horizontaux, mais bon… À partir de là, je pourrai décrire n'importe quel caractère alpha-numérique en termes des segments qui le constituent. Le "A", par exemple, serait tout sauf le segment n° 2 : 0, 1, 3, 5, 4, et 6.

Maintenant, j'ai idée que je pourrai aisément dessiner chaque segment, parce que je sais que la classe Canvas dispose d'une méthode create_line() qui, étant donnés [les coordonnées de] deux points, sait les joindre par un trait dont on peut spécifier l'épaisseur (et la couleur).

Il me suffira alors d'associer, à chacun de ces segments, le code qui permet de le dessiner à une position absolue dans la matrice du caractère. Je me donne donc des dimensions pour cette matrice, et décide qu'elle fera 70 pixels de large pour 120 pixels de haut : chaque segment aura une épaisseur de 10 pixels, et une longueur de 40 pixels.

Ainsi, le chiffre "8" aura pour hauteur 3 segments horizontaux d'une hauteur effective de 10 pixels chacun, donc 30, plus la longueur de deux segments verticaux, 2 * 40 pixels, soit 110 pixels au total ; quant à la largeur totale, elle sera la somme des épaisseurs de 2 segments verticaux plus la longueur d'un segment horizontal, soit 60 pixels.

La méthode create_line() attend au minimum 4 arguments : les coordonnées du début de la ligne et celles de la fin. C'est donc en connaissance de cause que je décide de représenter chaque segment comme la liste des

arguments de la méthode en question : x_1, y_1 et x_2, y_2.

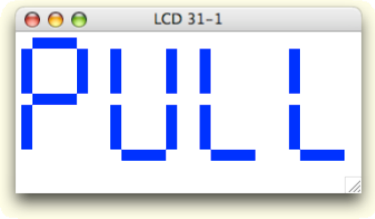
Et ces listes seront elle-mêmes rassemblées dans une liste que j'appellerai segments, de sorte qu'une expression comme segment[n] me retournera la liste des coordonnées nécessaires à create_line() pour le segment n° n.

Ceci fait, il me faudra donner une description de chaque caractère alpha-numérique comme une liste des numéros des segments qui le constituent. Pour représenter ces données, le plus commode serait un objet de type dict (que j'appellerai description), parce que je pourrai alors utiliser naturellement une expression comme description['E'] pour retrouver immédiatement la liste des segments du "E"…

FAISABILITÉ

Je ne sais plus où j'avais écrit : sans données, il n'y aurait pas d'algorithmes. On comprend pourquoi ici : l'analyse que je viens de faire me permet de préciser la représentation des données que le programme devra manipuler. Mais c'est vrai que pendant l'analyse, je n'arrête pas de pondérer l'impact qu'aurait telle ou telle représentation sur la manière dont je pourrais ensuite l'exploiter.

Cependant, il me reste un problème purement algorithmique : puisque mes coordonnées de segment sont absolues, par rapport à la matrice du caractère, elles ne conviennent que pour le tout premier caractère que j'affiche ; ce qui signifie que pour les caractères suivants, je vais devoir *translater* toutes les abscisses des segments que j'affiche.



Sachant que cette matrice a 50 pixels de large, il me paraît raisonnable de décaler le caractère suivant de 80 pixels, ce qui me laisse un espace de 30 pixels entre chaque caractère…

Tous comptes faits, il me faut 5 données :

- segments : liste des abscisses et ordonnées pour chaque segment
- description : dictionnaire de description des caractères
- ep : valeur en pixels de l'épaisseur du tracé d'un segment
- couleur : couleur du tracé d'un segment
- offset : valeur en pixels de la translation
- decal : cumul des translations successives (initialisé à 0)

L'affichage d'un mot quelconque (sous réserve que chacun de ses caractères ait une description) se fera par une double itération :

- pour chaque caractère k, retrouve ses segments dans sa description
- pour chaque segment SG, retrouve les arguments pour reconstruire le texte de l'expression create_line() qui permettra de dessiner ce segment en tenant compte de la position du caractère
- évalue cette chaîne de caractère pour effectuer l'affichage du segment
- actualise la variable decal en prévision du prochain affichage

Notez l'utilisation de la fonction repr() – la converse de eval() – chaque fois que je veux être certain que eval() pourra effectivement évaluer l'expression en question.

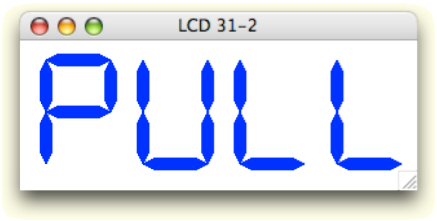
SCRIPT COMPLET

```
# description des sept segments LCD

segments = \
[  # X1, Y1, X2, Y2
   [15, 10, 55, 10],                                    # segment horizontal 0
   [15, 60, 55, 60],                                    # segment horizontal 1
   [15, 110, 55, 110],                                  # segment horizontal 2
   [10, 15, 10, 55],                                    # segment vertical 3
   [10, 65, 10, 105],                                   # segment vertical 4
   [60, 15, 60, 55],                                    # segment vertical 5
   [60, 65, 60, 105],                                   # segment vertical 6
]

# description des lettres en termes de leurs segments

description = {'L' : [3, 4, 2], 'P' : [0, 3, 5, 1, 4], 'U' : [3, 4, 5, 6, 2]} # etc...

# construction des widgets

from Tkinter import *

top = Tk()
canevas = Canvas(top, height=150, width=400)
canevas.pack()

# dessin des segments

offset = 70
ep = 10
couleur = '"blue"'                                      # attention : une chaîne dans une chaîne
decal = 20

for K in 'PULL' :
   les_segments = description [K]                       # les segments pour chaque caractère
   for SG in les_segments :
      S = segments [SG]                                 # les coordonnées pour chaque segment
      X1 = repr(S [0] + decal) ; Y1 = repr(S [1])       # maintenant, translate les abscisses
      X2 = repr(S [2] + decal) ; Y2 = repr(S [3])
      args = '%s, %s, %s, %s' % (X1, Y1, X2, Y2)        # les args pour create_line()
      args += ', width=%s, fill=%s' % (repr(ep), couleur)
      eval('canevas.create_line(%s)' % args)
   decal += offset

top.mainloop()
```

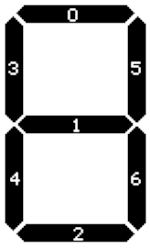
EXTENSIBILITÉ

[px31-1] augmenter le dictionnaire description pour pouvoir afficher les mots DESCRIPTION et SEGMENT ; modifier la taille des segments pour afficher des caractères 20 % plus grands – attention aux dimensions du canevas, qui peuvent devenir insuffisantes pour des mots très longs ; ramasser le code qui affiche le mot pour en faire une fonction à (au moins) un argument : le mot.

[px31-2] redessiner les segments pour produire le véritable effet LCD, tel que :



La méthode à utiliser n'est plus create_line() mais create_polygon() qui attend les coordonnées des sommets du polygone (ici, 6 sommets par polygone), et referme automatiquement le polygone en joignant le dernier point au premier ; voir l'exemple de la fonction fais_x() dans le code du MORPION, à la section 8.3.



[px31-3] les afficheurs LCD récents disposent de segments supplémentaires pour représenter, par exemple, le point, ou l'apostrophe, et autorisent une plus grande latitude pour placer les segments, de sorte qu'on puisse distinguer le « B » du « 8 » ou le « A » du « R » ; étudier la faisabilité de telles modifications au programme, et les réaliser.

8.2 PENDU

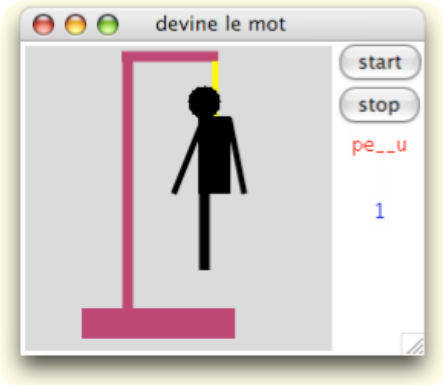
On me demande de réaliser une étude de faisabilité pour un jeu de pendu : la règle est que le joueur a droit à un total de 10 coups pour retrouver toutes les lettres d'un mot à deviner, faute de quoi il sera virtuellement pendu.

ANALYSE DU PROBLÈME

La première question que je me pose, c'est qu'est-ce que je dois représenter, et de quelles données vais-je disposer. Et ici, il y a deux choses : l'interface de l'application, et la donnée des mots connus par le programme, parmi lesquels il choisira au hasard.

Une différence majeure avec l'application précédente (affichage par segments) c'est qu'un tel programme est éminemment interactif :

- chaque fois qu'une nouvelle lettre est proposée, il faut afficher le mot en laissant les “trous” des éléments manquants ;
- le dessin du pendu doit être progressif, en fonction de la validité de la lettre proposée par le joueur.



Le dessin proprement dit ne pose pas vraiment de problème si on utilise la technique qui représente le code du dessin sous la forme d'une liste d'expressions à évaluer. La seule contrainte, c'est que le dessin doit être complété en exactement 10 étapes : je vais donc préparer un gibet (une estrade, une potence en 2 parties, et une corde), et il me restera 6 étapes pour la tête, le corps, et les 4 membres.

Pour les mots, je pourrais avoir préparé un fichier avec des centaines de mots, mais dans le contexte d'un prototype, trois ou quatre mots suffisent à démontrer la faisabilité : je choisis de les rassembler sous la forme d'une liste. Je sais que je dispose d'une méthode de choix aléatoire[38] et que tirer un mot au hasard sans une séquence n'est pas vraiment un problème.

Reste la question de l'affichage des *trous* dans le mot…

- je décide de les afficher comme des '-'
- ainsi, au tout 1er tour, le mot est présenté comme une séquence de tirets, séquence de même longueur que le mot à deviner
- chaque fois qu'une nouvelle lettre me sera proposée, il me suffira, si elle est valide, de la substituer au tiret correspondant
- le jeu est terminé dès que toutes les lettres ont été devinées

Pour la convivialité de l'interface, je prévois :

1. l'affichage du mot à deviner, avec ses tirets
2. l'affichage du nombre d'essais restants
3. l'affichage du mot qu'il fallait deviner
4. un bouton pour arrêter ce jeu idiot
5. un bouton pour recommencer ce jeu fascinant

Les trois premiers objets seront instanciés par des objets de la classe Label, et les deux derniers par des instances de Button.

Le dessin du pendu se fera obligatoirement dans une instance de Canvas que j'appellerai tout simplement c.

Je vais définir 6 variables globales :

- mots, la liste des mots possibles

38 cf. random.choice() à la section 9.1.2 : modules d'usine

- mot, celui qui sera tiré au hasard
- taille, celle du mot – dont j'aurai besoin à plusieurs occasions
- trou, le caractère choisi pour représenter une lettre manquante
- devine, la représentation *interne* du mot à deviner
- P, comme *pendaison* : la liste des actions du bourreau

Mon interface sera composée de quelque sept objets :

- top, le "master *widget*", instance de Tk
- c, l'espace de dessin, instance de Canvas
- LT, comme Label des trous, actualisé à chaque tour
- LM, comme Label du mot ; initialisé avec une chaîne vide, ce n'est qu'à la fin du jeu qu'il sera véritablement *actualisé* avec le mot qu'il fallait deviner
- LP, comme Label de pendaison, affichant le nombre de coups restants
- deux boutons, pour arrêter ou relancer le jeu

FAISABILITÉ

Le problème, ici, et de faire en sorte que la frappe d'une touche de clavier déclenche le HANDLER approprié : je vais donc « sensibiliser » la fenêtre elle-même, en la liant au HANDLER avec la méthode bind(). Appelons ce HANDLER actualise() ; je sais qu'il attend obligatoirement un argument (l'événement) pourvu par le mécanisme de CALLBACK de mainloop(), le gestionnaire d'événements, donc je lui mets un paramètre que j'appellerai E, ev ou event ; je sais que cette variable est elle-même un objet muni d'attributs, et celui qui m'intéresse ici, c'est event.char : le caractère correspondant à la touche qui a été enfoncée.

Maintenant, je dois actualiser la variable globale devine, sauf si
- il ne reste plus rien à pendre – auquel cas c'est perdu
- il ne reste plus de *trous* dans le mot – auquel cas c'est gagné

Il me suffit donc de coder ceci comme une expression booléenne (retournant une valeur logique) pour décider si je dois ou non actualiser devine : autrement dit, s'il reste quelque chose qui n'a pas encore été pendu et qu'il reste des trous dans devine, j'appelle la fonction valide() en lui passant le caractère tapé, le mot original, et la variable devine dans son dernier état.

Comparer la valeur retournée par valide() avec celle de devine me permet de savoir si la lettre proposée est valide :
- si elle l'est, il faut actualiser devine, qui change donc de valeur ;
- si c'est une faute, il faut procéder à l'étape suivante de la pendaison…

Et de toute façon, il faut actualiser l'affichage en appelant la fonction affiche().

Un mot d'explication sur la « mécanique » de la pendaison, exemple typique de DATA-DRIVEN PROGRAMMING : la méthode pop(0), appliquée à une liste, enlève de cette liste l'élément de rang 0, et retourne cet élément comme valeur ; ainsi :
- appliquer eval() à cette valeur dessine cette partie de la pendaison ;
- la liste P, étant amputée d'un élément à chaque fois, finira par être vide, et l'expression P, d'un point de vue logique, sera alors évaluée comme False, d'où le test dans l'expression conditionnelle de actualise() …

La fonction valide() attend donc 3 valeurs :
- C, le caractère à actualiser
- original, le mot original
- devine, ce qui a été deviné jusqu'à présent

Mécaniquement parlant, valide() est très simple : elle scanne l'original et, si elle y rencontre le caractère C, elle le concatène au résultat R, sinon, c'est le caractère situé à la même position, mais dans devine, qui sera concaténé au résultat R, que ce soit une lettre ou non. Lorsque le scan est terminé, ce que la fonction retourne, c'est justement le cumul des concaténations successives…

La fonction affiche() est chargée de la mise à jour de l'interface ; elle ne manipule que des objets globaux de l'interface proprement dite, mise à part la variable locale L qui représente le nombre d'actions restantes pour

la pendaison :

- LT (*label* de tirets) doit être actualisé en fonction de devine ;
- LP (*label* de pendre) dépend de L, doit être converti en texte par la fonction str() ;
- LM (*label* du mot) n'est affiché qu'en fin de pendaison.

SCRIPT COMPLET

```
# une (petite) liste de mots pour tester :
mots = 'pendu xerox clepsydre naims mississippi wagon'.split()

from random import choice
mot = choice(mots)
taille = len(mot)
trou = '-'
devine = trou * taille

def actualise(event) :
   global devine
   if P and trou in devine :
      nouveau = valide(event.char, mot, devine)
      if devine != nouveau : devine = nouveau
      else : eval(P.pop(0))
      affiche()

def valide(C, original, devine, R = '') :
   for x in range(taille) :
      if original [x] == C : R += original [x]
      else : R += devine [x]
   return R

def affiche() :
   L = len(P)
   LT.configure(text = devine)
   LP.configure(text = str(L))
   if not L : LM ['text'] = mot

from Tkinter import *
top = Tk()
top.title("devine le mot")
top.bind("<Key>", actualise)

c = Canvas(top, bg='light grey', height=200, width=200)
c.pack(side = LEFT)

# la pendaison... une liste de dessins
P = [ 'c.create_line(40, 185, 140, 185, width=20, fill="maroon")',
      'c.create_line(70, 185, 70, 10, width=7, fill="maroon")',
      'c.create_line(66, 10, 129, 10, width=7, fill="maroon")',
      'c.create_line(127, 13, 127, 60, width=4, fill="yellow")',
      'c.create_oval(110, 30, 129, 49, width=1, fill="black")',
      'c.create_rectangle(116, 50, 136, 100, fill="black")',
      'c.create_line(120, 50, 100, 100, width=4, fill="black")',
      'c.create_line(136, 50, 146, 100, width=4, fill="black")',
      'c.create_line(120, 100, 120, 150, width=7, fill="black")',
      'c.create_line(133, 100, 133, 150, width=7, fill="black")' ]

Button(top, text='start', command=top.quit).pack(side=TOP)
Button(top, text='stop', command=top.quit).pack(side=TOP)

LT = Label(top, text=devine, fg='red') ; LT.pack()
LM = Label(top, text='', fg='red') ; LM.pack()
LP = Label(top, text=str(len(P)), fg='blue') ; LP.pack()

top.mainloop()
top.destroy()
```

REMARQUES

À propos de la taille des mots, donc de la largeur de la fenêtre pour les mots les plus longs : si vous regardez le code de près, vous constaterez qu'au moment où LT (l'instance de Label qui représente le mot à deviner) appelle la méthode pack(), la variable devine a déjà été définie, et donc que la taille de la fenêtre s'y adaptera automatiquement, quelque soit la longueur du mot.

À propos de la fonction valide(), remarquez que je n'ai pas vraiment besoin de passer des variables globales en argument : ça complique inutilement les choses ; cependant, dans le contexte d'un prototype, je préfère être sûr que l'idiome utilisé pour finaliser l'application n'aura pas de problème de portée des variables, un concept qu'on examinera de plus près lorsqu'on étudiera un langage comme ANSI-C, qui est un peu plus rigide que PYTHON.

Toujours à propos de la fonction valide(), j'ai ajouté un 4e paramètre, appelé R, initialisé avec une chaîne vide…

> Dans un certain nombre de langages, par exemple, OBJECTIVE C ou C++, il est possible de définir des paramètres initialement pourvus de valeurs par défaut ; du même coup, ces paramètres sont optionnels : si on met l'argument correspondant en appelant la fonction, le paramètre accepte la valeur de l'argument ; sinon, il prend par défaut la valeur spécifiée dans la définition de la fonction.

EXTENSIBILITÉ

J'ai déjà fait plus haut quelques remarques sur l'extensibilité de ce prototype : la plus importante était le nombre de mots disponibles, qui peut être facilement augmenté en créant un fichier contenant plein de mots, séparés par des espaces ou des sauts de lignes.

[px32-1] coder une version de ce script qui prend ses mots dans un fichier ; adaptez le code pour qu'il puisse manipuler des glyphes unicode et prendre en compte les caractères accentués, et neutralise la différence majuscule/minuscule.

[px32-2] quand il y très peu de mots à choisir, comme c'est le cas dans le script ci-dessus, il arrive que le même mot soit (par pur hasard) sélectionné plusieurs fois de suite ; adaptez le code pour empêcher que le même mot sorte deux fois consécutivement.

[px32-3] tel qu'il est défini, le bouton start a exactement le même effet que le bouton stop ; modifier le script pour que ce bouton relance le jeu ; astuce : la fonction execfile(), si on lui donne en argument le nom d'un script codé en PYTHON, exécute ce script sans se poser de question.

[px32-4] dans sa version actuelle, ce script comptabilise comme une erreur une lettre non valide, même si elle a déjà été essayée : imaginer un moyen de tenir compte des lettres non valides déjà proposées, et modifier le code pour empêcher qu'elles soient sanctionnées plus d'une fois ; bien entendu, le programme doit être indifférent à la casse.

Toutes ces extensions sont, de préférence, cumulatives : chaque version devrait incoporer les modifications de la précédente ; mais si vous bloquez sur l'une, essayez quand même les suivantes !

8.3 MORPION

Ce jeu se joue à deux joueurs : chaque joueur se voit attribuer l'un des deux symboles “O” ou “X”, et chacun joue tour à tour ; le gagnant est le premier qui réalise une ligne de 3 symboles, horizontalement, verticalement ou diagonalement.

On ne cherche pas ici à élaborer un programme qui émulerait le comportement humain et vous permettrait de jouer contre la machine.

On va juste élaborer une interface qui remplacerait le papier… et voir les problèmes que ça peut poser, en termes de faisabilité d'un prototype d'aspect assez avenant pour convaincre un éventuel client.

ANALYSE

Comme on le voit sur l'image, cette interface comporte simplement :

- neuf cellules permettant à chaque joueur de marquer son symbole
- une instance de Label pour afficher des trucs
- un bouton d'arrêt

La clé d'une telle interface, c'est de créer autant d'instances de Canvas qu'il y a de cellules jouables : en effet, lorsque mainloop(), le gestionnaire d'événements appelle le HANDLER dédié, il lui passe en argument l'événement en question, dont on sait déjà que c'est un objet muni de propriétés, l'une d'elle étant le *widget* qui a déclenché l'événement, autrement dit l'instance de Canvas sur laquelle on vient de cliquer.

DONNÉES DU PROGRAMME

En arrière-plan de cette interface, le programme doit maintenir en miroir sa propre matrice de cellules, qu'on appellera M pour faire simple. La raison en est que le programme ne voit pas l'interface, ne voit pas les symboles figurant dans chaque case. Il existe, il est vrai, des méthodes pour retrouver quels objets ont été dessinés dans une instance de Canvas en un point précis, connaissant ses coordonnées ; mais ce sera toujours plus complexe, plus indirect, donc moins efficace, moins rapide que de tester la valeur d'un élément de la matrice M…

Toutes les cellules de M sont, au départ, initialisées à la valeur False ; et dès qu'un coup y est joué, la cellule prend comme valeur le symbole correspondant au joueur qui a cliqué dessus : “0” ou “X” ; l'avantage est double :

1. marquer une cellule de M, c'est, d'un point de vue logique, basculer sa valeur à vrai, puisque, dans une logique à deux valeurs, ce qui n'est pas False est forcément True ; cette propriété permettra de tester aisément s'il est encore possible de jouer dans une cellule donnée ;
2. marquer une cellule de M, c'est aussi laisser l'empreinte spécifique du joueur qui se l'est ainsi appropriée, et ceci permettra aisément de calculer s'il vient d'aligner 3 empreintes semblables.

RÉCAPITULATION

Côté interface, il me faut 15 objets :

- la constante entière T définit la taille de la grille de jeu : disons 3 pour commencer
- la constante entière H définit la dimension (en pixels) d'une cellule cellule carrée
- morpion, le *widget* maître
- grille, une instance de Frame pour contenir les instances de Canvas
- neuf instances de Canvas, auxquelles je n'ai pas besoin de donner de noms
- info, une instance de Label — pour faire plus convivial
- stop, le bouton d'arrêt : une instance de Button, évidemment

Côté traitement des données, il me suffit de 2 variables :

- la matrice M mémorise l'un des 3 états possibles pour chaque cellule ; c'est une matrice carrée, de dimensions T par T, où T est une constante ;
- la variable logique J définit le joueur, tel que J *vrai* corresponde à “X”, et à “0” sinon ; l'intérêt de définir

le joueur d'un point de vue logique, c'est qu'on alternera très facilement en basculant J à not J.

FAISABILITÉ

HANDLER

Comme on l'a vu dès le début, chaque instance de Canvas est autonome, mais ça ne m'empêche pas de leur affecter le même HANDLER, que j'appellerai ici, tout simplement, joue(). Une telle fonction attend un (et un seul) argument : l'événement qui a déclenché le CALLBACK, et que j'appellerai event.

Et le premier truc que je ferai dans cette fonction, c'est de décortiquer l'événement event pour savoir quel en est le *widget*, autrement dit quelle est l'instance de Canvas qui a déclenché le callback, *widget* que je l'appellerai ici simplement w.

On savait déjà que la nature et la forme des données conditionnait leur traitement, et c'est particulièrement vrai ici, parce que je me suis arrangé pour que chaque instance de Canvas se souvienne, à sa création, de sa position dans la grille, en termes de rangée et de colonne, en définissant dans chaque objet les attributs appropriés.

Ainsi l'accès au *widget* w me permet de manipuler directement l'élément de la matrice M qui lui correspond, toujours en termes de ligne et de colonne, ce qui me donne immédiatement le moyen de savoir si cette cellule est déjà occupée, auquel cas, un return termine le traitement immédiatement : et voilà donc pourquoi il ne se passera rien si on clique sur une case déjà marquée.

Par contre, si la cellule est libre, il y a déjà deux choses que je dois faire inconditionnellement :

- dessiner dans cette cellule le symbole spécifique du joueur : c'est un rôle que je délègue aux fonctions fais_x() et fais_o() ;
- marquer cet élément de la matrice M comme occupé par ce joueur : une petite fonction symbole() me retourne le symbole du joueur courant, que je mets comme valeur dans l'élément approprié de la matrice M.

Maintenant je dois me poser la question de savoir si ce coup était gagnant, travail de détection que j'ai délégué à la fonction gagnant() ; si elle répond vrai, j'affiche un petit compliment dans le Label info ; sinon, la vie continue, mais avec l'autre joueur…

FONCTIONS ANCILLAIRES

Fonctions de dessin, fais_x() et fais_o() : elles n'ont rien de très rusé, à ceci près que pour faire plus « pro », le "X" est dessiné comme deux polygones qui se croisent. Note : ces fonctions devraient être paramétrées pour s'adapter à n'importe quelle taille de cellule.

La fonction symbole() est définie avec le paramètre j, donc attend un argument, le joueur, dont on se souvient que c'est une variable logique ; or il se trouve que, comme dans beaucoup de langages (notamment, ANSI-C), True c'est la même chose que 1, et False c'est la même chose que 0 ; cette fonction retourne donc simplement l'élément de la liste ['O', 'X'] indicé par la valeur de j d'un point de vue numérique.

La fonction gagnant() attend un argument, et j'ai appelé s le paramètre correspondant, qui reçoit donc l'un des deux symboles (caractères) 'O' ou 'X'. Cette fonction est ridiculement bestiale : j'ai bien essayé des solutions plus astucieuses, mais elles demandent toutes de telles acrobaties qu'elles sont en définitive moins efficaces que celle que j'ai retenue.

- Pour chaque ligne, chaque colonne et chaque diagonale, je compte le nombre de fois que la valeur de s apparaît : si j'en trouve T, je retourne *vrai* immédiatement.
- Comme cette fonction est appelée après chaque coup, je ne m'intéresse qu'à l'empreinte du joueur en cours – si c'est l'autre qui aurait dû gagner, je l'aurais forcément su au coup d'avant.
- Mais il reste une dernière chose à vérifier : si personne n'a encore gagné, mais qu'il ne reste plus aucune cellule libre, alors, c'est le match nul — zéro partout, et GAME OVER…

SCRIPT COMPLET

```
J = True                                                    # joueur initial
T = 3                                                       # taille de grille
H = 60                                                      # taille de cellule
M = [T * [False] for x in range(T)]                         # toutes cellules initialement libres

def joue(event) :                                           # handler lié à <Button-1>
   global J
   w = event.widget                                         # le widget actif
   if M[w.R][w.C] :                                         # cellule déjà occupée
      return
   if J :
      fais_x(w)
   else :
      fais_o(w)
   s = symbole(J)                                           # joueur symbolique
   M[w.R][w.C] = s
   if gagnant(s) :
      info['text'] = 'joueur %s gagne' % s
   J = not J

def fais_x(w) :
   w.create_polygon(10, 10, 20, 10, 55, 45, 55, 55, 45, 55, 10, 20, fill = 'red')
   w.create_polygon(10, 55, 10, 45, 45, 10, 55, 10, 55, 20, 20, 55, fill = 'red')

def fais_o(w) : w.create_oval(10, 10, 55, 55, width = 10)

def symbole(j) : return ['O', 'X'][j]                       # j ne peut valoir que 0 ou 1

def gagnant(s) :
   global info
   for x in range(T) :
      if [M[x][0], M[x][1], M[x][2]].count(s) is T :        # ligne x
         return True
      if [M[0][x], M[1][x], M[2][x]].count(s) is T :        # colonne x
         return True
   if [M[0][0], M[1][1], M[2][2]].count(s) is T :           # diagonale \
         return True
   if [M[0][2], M[1][1], M[2][0]].count(s) is T :           # diagonale /
         return True
   if False not in M[0] + M[1] + M[2] :
      info ['text'] = 'GAME OVER'

# ~~~~~~~~~~~~~~

from Tkinter import *

morpion = Tk()
morpion.title('MORPION 1.0')                                # cosmétique
grille = Frame(morpion)

for R in range(T) :
   for C in range(T) :
      Cell = Canvas(grille, bg = 'light grey', width = H, height = H)
      Cell.bind("<Button-1>", joue)
      Cell.grid(row = R, column = C)
      Cell.R, Cell.C = R, C                                 # localisation de chaque cellule

grille.pack()
stop = Button(morpion, text='ASSEZ', command = morpion.destroy)
stop.pack()
info = Label(morpion)
info.pack()

morpion.mainloop()
```

LA MÉTHODE GRID()

Les méthodes pack() et grid() invoquent implicitement ce qu'on appelle des GEOMETRY MANAGERS (que je renonce à traduire). L'avantage de la méthode grid() – où GRID signifie *grille* – c'est qu'elle permet de disposer les *widget*s dans une sorte de grille virtuelle, exactement ce qu'il me fallait pour organiser mes instances de Canvas.

Je m'empresse de signaler qu'il est absolument déconseillé d'invoquer des GEOMETRY MANAGERS différents dans un même programme, parce qu'ils peuvent être amenés à prendre des décisions conflictuelles insolubles et se renvoyer la balle indéfiniment ; je l'ai fait ici parce que ça simplifiait le programme, mais j'ai honte…

LISTES ET RÉFÉRENCES

Un mot à propos de l'étrange syntaxe de l'expression qui me sert à initialiser la matrice M :

```
[3 * [False] for x in range(3)]
```

On avait déjà rencontré ce genre d'expression[39], et on sait que c'est la manière fonctionnelle de calculer la valeur d'une liste de manière itérative.

Mais ici, suivez-moi de très près : si j'évaluais 3 * [0], j'obtiendrais [0, 0, 0], et, de même, 3 * [3 * [0]] me donnerait [[0, 0, 0], [0, 0, 0], [0, 0, 0]], qui ressemble beaucoup à ce dont j'ai besoin, c'est-à-dire une matrice de 3 lignes par 3 colonnes dont tous les éléments sont proprement initialisés à zéro – et soit dit en passant, le résultat aurait été le même avec False au lieu de 0.

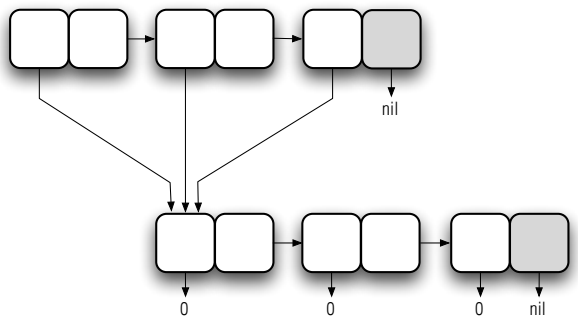
Maintenant, faites l'expérience suivante :

```
L = 3 * [3 * [0]]
L[0][0] = 1
```

et évaluez L ; vous découvrirez ce résultat ahurissant : [[1, 0, 0], [1, 0, 0], [1, 0, 0]].

Ceci s'explique par le fait que l'opérateur * effectue des copies de son opérande, et non pas des *éléments* de cet opérande ; il s'en suit que l'élément [0][0] est en même temps l'élément [1][0] et l'élément [2][0]…

Ce qui ne m'arrange pas du tout, parce que si je modifie une cellule de ma matrice M, il ne faut surtout pas que d'autres cellules (plus exactement, des cellules que je crois être autres) prennent simultanément cette même valeur !



Ce comportement de l'interprète n'est pas une erreur : c'est dû à la façon dont fonctionne la méthode de réplication. En fait, et c'est [presque] totalement invisible, les objets représentant des listes sont des *références*, autrement dit des pointeurs, concept abondamment détaillé dans le cours de LISP (113) et celui de ANSI-C (213).

En LISP, étant donné (setq z '(0 0 0)), une construction comme (list z z z) produirait la structure représentée sur le diagramme ci-dessus ; et en PYTHON, 3 * [3 * [0]] produirait exactement la même chose, directement…

Par contre, l'expression que j'ai utilisée génère une nouvelle liste à chaque tour, donc aucune de ces listes ne peut avoir d'élément en commun.

Pour mettre ceci en évidence, il suffit de regarder quels sont les objets impliqués, grâce à la fonction id() qui retourne effectivement l'adresse de la structure :

39 cf. l'exercice px12-1, par exemple

```
>>> [id(x) for x in 3 * [3 * [0]]]
[497384, 497384, 497384]
>>> [id(x) for x in [3 * [0] for z in range(3)]]
[497864, 497904, 497944]
```

À PROPOS DE L'EXPLOSION EN MULTIPLES PETITES FONCTIONS

Comme dans tous mes programmes vous observerez qu'il y a généralement une fonction principale que je m'efforce de laisser aussi « maigre » que possible pour en préserver l'aspect « tour de contrôle », laissant le soin a des fonctions subalternes de calculer des choses qui encombreraient la compréhension du traitement principal, le HANDLER.

Nous ne sommes pas loin de la mentalité du programmeur LISP, ou même du programmeur ANSI-C, qui développent de manière incrémentale, en ayant testé chaque expression et mis au point chaque petite fonction.

Et comme chacune de ces petites fonctions a un nom, ce nom (s'il est bien choisi) est auto-documentaire, ce qui permet de s'abstraire des détails de *comment* elle est programmée, pour ne plus mémoriser que ce *pourquoi* elle est programmée.

La preuve en est que vous savez ce que ça va donner si vous évaluez l'expression morpion = Tk(), alors que je suis absolument certain que vous ne savez pas de quoi Tk est fait… Moi non plus, d'ailleurs !

Je sais bien que je l'ai déjà écrit quelque part, mais c'est ce sont ces couches successives d'abstractions qui permettent de construire avec aisance des programmes complexes.

EXTENSIBILITÉ

De toute façon, ce jeu est invendable, face à la concurrence, mais c'est une opportunité de développer des techniques de programmation, et peut-être même d'en inventer de nouvelles. Alors rien n'est indispensable, mais rien n'est inutile.

Modification qui ne demande même pas de rajouter une ligne de code…

[px33-1] tel quel, ce programme ne s'arrête pas quand un joueur a gagné ; comment peut-on, à peu de frais, modifier le code pour qu'il n'accuse réception d'aucun nouveau coup une fois que l'un des joueurs a gagné ?

Modification qui requiert un peu d'astuce pour généraliser la fonction gagnant()…

[px33-2] étendre le programme pour jouer sur une grille de, mettons, 5 par 5 – suffirait-il pour cela de changer la valeur de la variable globale T ? Le programme continuerait-il à fonctionner ? Que faudrait-il changer pour améliorer son comportement ? Peut-on coder une solution générale pour n'importe quelle dimension ?

Modification qui requiert une petite formalisation de la stratégie gagnante, mais ne risque pas de vous embarquer dans les arcanes des techniques d'intelligence artificielle…

[px33-3] qu'en coûterait-il d'adapter ce programme pour jouer contre la machine, et non plus à deux joueurs humains ? Réalisez cette adaptation…

8.4 L'ORDINATEUR EN PAPIER

Soit le langage décrit dans L'ORDINATEUR EN PAPIER[40] dont on se contentera ici de rappeler très brièvement les caractéristiques lexicales et syntaxiques : voir l'original, pour plus de précision.

On se propose de réaliser un interprète de ce langage, autrement dit un programme qui lit le code d'un autre programme, puis simule ce que ce processeur fictif ferait pour exécuter ce code.

La mémoire dans laquelle sont chargés programme et données est de 256 *bytes* : on la représentera sous la forme d'un vecteur de 256 caractères, donc indexé de 0 à 255 ; notez que les éléments d'indice 0 à 31 représentent de la *mémoire morte* (ROM) dans laquelle réside, à demeure, un *boot-strap loader* inamovible : on ne peut donc écrire que dans les éléments indicés de 32 à 255, qui correspondent à la RAM, ou *mémoire vive*.

Le code pourra être représenté de façon symbolique, c'est-à-dire avec des *mnémoniques* comme dans l'exemple du *boot-strap loader* : le programme devrait, à terme, pouvoir fonctionner comme un désassembleur aussi bien qu'un assembleur.

Les instructions in et out seront simulées avec raw_input() et print().

Il n'est pas nécessaire de rendre compte de l'aspect *microcodé* du processeur : on se bornera à émuler l'effet de chaque instruction en actualisant la RAM, l'*accumulateur* A et le *compteur ordinal* PC).

mnémonique	opcode	interprétation
add #	20	A ← A + V
add @	60	A ← A + (@)
add * @	E0	A ← A + *(@)
sub #	21	A ← A - V
sub @	61	A ← A - (@)
sub * @	E1	A ← A - *(@)
nand #	22	A ← ¬ [A & V]
nand @	62	A ← ¬ [A & (@)]
nand * @	E2	A ← ¬ [A & *(@)]
load #	00	A ← V
load @	40	A ← (@)
load * @	C0	A ← *(@)
store @	48	(@) ← A
store * @	C8	*(@) ← A
in @	49	(@) ← Entrée
in * @	C9	*(@) ← Entrée
out @	41	Sortie ← (@)
out * @	C1	Sortie ← *(@)
jump @	10	PC ← @
brn @	11	si A < 0 : PC ← @
brz @	12	si A = 0 : PC ← @

[px34-1] programmer cet émulateur de sorte qu'il puisse lire le code machine à partir d'un fichier, et l'exécuter ; résolvez, entre autres exemples, l'énigme des 2 programmes-mystère de la section exercices du document original.

ANALYSE

Le problème ressemble à celui de l'exercice [px18-2] déjà résolu au chapitre 4, et il devrait être possible de lui appliquer une solution similaire… mais il y a quelque chose ici qui nous oblige à changer notre angle d'attaque : ce langage dispose de *modes d'adressage* plus puissants lui permettant d'accéder à n'importe quel élément de la mémoire, qui, du même coup, ne peut plus être saucissonnée en tranches de 4 octets comme précédemment — c'est pourquoi on adoptera la représentation sous forme d'un vecteur d'octets (appelons-le data), de type list.

D'ailleurs, comme spécifié à la section 13.3, chaque instruction est codée sur 2 octets… et il est tout à fait possible que le programme écrive une nouvelle valeur à la place de celle donnée dans le code source original : voilà pourquoi l'approche simpliste du [px18-2] ne convient plus.

Du fait que le programme doit maintenant lire le code à partir d'un fichier, le chargement du programme en mémoire est quelque peu simplifié, puisqu'il s'agit simplement de recopier des valeurs entières dans le vecteur data.

Là où les choses se compliquent, c'est que le document original ne montre pas moins de trois formats de programme…

le format de la section *13.7 — exemple de programme* :

40 É.C iAO, introduction à l'architecture des ordinateurs, chapitre 13 : *l'ordinateur en papier*

```
00 49 20 in 20
02 49 22 in 22
04 40 20 load 20
```

le format de la section *13.8 — exercice* 13.8 :

```
10 3a 00 31 40 32 60 33 48 32 49 33 40 33 22 ff 12 34 41 32 10 44
```

et toujours dans la section *13.8,* le format de l'*exercice* 13.9 :

```
50 49 70
52 40 70
54 48 71
```

Notez qu'il s'agit dans, tous les cas, de codes hexadécimaux… parfois donnés en minuscules et parfois en majuscules : aucune importance, les deux formes sont équivalentes, et de toute façon, la représentation interne des programmes, elle, est binaire.

DIFFÉRENCES

Le 13.7 liste en même temps l'adresse de l'instruction, le code machine et le code assembleur symbolique[41] pour chaque ligne impaire, alors que les lignes paires ne contiennent qu'une adresse et une valeur : format somme toute, assez complexe à interpréter, surtout quand on arrive à l'offset 0x18, car ici, le 1E — contrairement à ce qu'on attend dans cette colonne — n'est pas un code d'opération (il n'est d'ailleurs pas défini) mais une indication[42] que l'intervalle d'adresses, de 0x18 à 0x1E, est initialisé avec rien que des zéros.

Pour le 13.8, on nous donne l'adresse où charger le code, lequel est ramené à une séquence d'octets, ce qui est, semble-t-il le format le plus simple qu'on puisse manipuler.

Le 13.9 liste l'adresse et le code machine correspondant ; plus simple que le 1er, mais quand même plus complexe que le 2e… de surcroît, ce format pose un autre problème du fait que la présentation n'en est pas uniforme : le code source est le plus souvent listé par blocs de deux octets, mais de temps à autre, il n'y en a plus qu'un par ligne :

```
6C 41 71
6E 10 6E
70 00
71 00
```

Il est vrai que dans ce cas particulier, le nombre d'octets isolés est toujours pair, mais il ne va pas être simple de lire les données d'un tel fichier en gardant cette forme pour le code source.

PRÉPARATION DES DONNÉES

Nous allons donc commencer par *normaliser* tous les programmes au format 13.8, ce qui implique que la toute première valeur indiquée dans le fichier soit l'adresse de chargement du programme, suivie des d'octets consécutifs à charger à partir de cette adresse ; il nous faudra donc manuellement conformer le programme du boot-strap loader, ainsi que celui de l'exercice 13.9.

Nous pourrons alors écrire une fonction PYTHON capable de lire du code machine au format 13.8, et en interner les valeurs dans le vecteur data ; nous aurons au moins 3 programmes distincts pour tester son bon fonctionnement — et sans doute même plus, puisque vous avez déjà pris la peine de résoudre les exercices 13.5 et 13.6.

D'ailleurs, avant même d'arriver à ce stade, nous pourrions déjà nous contenter de recopier manuellement les données du fichier PDF pour les coller dans l'interprète sous forme de texte ; par exemple, pour le programme-mystère 13.8 :

```
>>> code = '10 3A 00 31 40 32 60 33 48 32 49 33 40 33 22 FF 12 34 41 32 10 44'.split()
>>> code = [eval('0x' + x) for x in code]
```

et hop, une petite ligne pour tester que c'est bien la même chose que l'original :

41 sans parler des commentaires, d'ailleurs non délimités formellement…
42 hélas non documentée, dans le but malicieux de servir de piège dans l'exercice 13.4

```
>>> for c in code : print '%02X' % c,
...
10 3A 00 31 40 32 60 33 48 32 49 33 40 33 22 FF 12 34 41 32 10 44
```

sans oublier qu'il faut mémoriser quelque part l'adresse origine de ce segment de programme, et garder les deux ensemble… par exemple dans une table d'associations (type 'dict') :

```
segment_table = {}
segment_table['13.8'] = {'offset': 0x30, 'code': code}
```

ce qui permet d'y mettre également le tout premier segment, celui du boot-strap loader, après l'avoir normalisé, sachant que l'adresse origine de ce segment est, par définition, 0x00 :

```
>>> code = '49 20 49 22 40 20 48 21 C9 21 40 22 21 01 12 1F \
            48 22 40 21 20 01 10 06 00 00 00 00 00 00 00 10'.split()
>>> code = [eval('0x' + x) for x in code]
```

Tiens, tiens, tiens : ça fait deux fois que j'applique la même transformation à un texte source de programme ; ça vaudrait le coût d'en faire une petite fonction… à moins de l'intégrer dans la procédure de lecture du fichier… En attendant :

```
>>> segment_table['boot-strap loader'] = {'offset' : 0, 'code' : code}
```

Pendant qu'on y est, procédons de même pour interner le code du programme-mystère 13.9 :

```
>>> code = '49 70 40 70 48 71 48 72 00 5E 48 8D 10 74 40 71 48 72 40 70 48 71 00 6C 48 8D 10 74
            41 71 10 6E 00 00 00 00 00 00 48 73 40 71 12 8C 21 01 41 71 40 73 60 72 48 73 10 78
            40 73 48 71 10 00'.split()
>>> code = [eval('0x' + x) for x in code]
>>> segment_table['13.9'] = {'offset' : 0x50, 'code' : code}
```

Mine de rien, nous sommes en train de nous faire la main pour la définition de la procédure de lecture de fichiers : nous venons de faire 3 fois la même chose, pour 3 programmes différents, il est donc probable que notre façon de procéder soit une assez bonne généralisation de cette procédure — ce qui n'est pas vraiment inattendu, du fait de notre normalisation.

Nous avons maintenant des données dans une table nommée segment_table, dont chaque élément est une information double : l'adresse origine du segment[43] et la séquence de valeurs qui représente le code à interpréter pour ce segment.

Ce qui signifie que nous pouvons déjà tester un prototype d'émulateur (ou interprète) sur tous ces segments de programmes.

CONCEPTION DE L'INTERPRÈTE

On peut se demander pourquoi avoir décidé d'une représentation aussi complexe des données, alors qu'il suffisait d'avoir la séquence des octets du code machine (BYTE CODE) pour mettre en place les principes de l'interprétation du code en question…

L'important c'est qu'on ait défini la structure des données auxquelles accèdera l'émulateur après chargement, sachant que ce chargement se fait théoriquement par l'exécution du tout premier segment, donc qu'il était nécessaire, en toute rigueur, d'intégrer la notion de segment à cette structure de données.

En fait, cette représentation n'est probablement pas encore assez complexe pour accommoder des programmes comportant eux-mêmes plusieurs segments, ni permettre d'accommoder du code symbolique comme suggéré par le format 13.7, mais pour l'instant, on peut quand même faire avec.

Le principe fondamental de l'émulation, comme on l'a vu au chapitre 4, c'est de disposer d'un procédé capable de convertir une instruction de code machine en une expression du langage souche : ici, PYTHON.

Or le code machine ne sait manipuler que des registres, et il n'y en a que deux, l'accumulateur A et le compteur ordinal PC : à première vue, il semblerait judicieux de les représenter par les variables A et PC.

43 ou offset (décalage) par rapport au début de la mémoire

Prenons l'exemple du load immédiat, opcode 0x00, utilisé dans le 13.9 à l'offset 0x66 : traduite en PYTHON, l'instruction 00 6C deviendrait simplement A = 0x6C… à un détail près : l'incrémentation automatique du registre PC est microcodée, et PC doit être incrémenté de 2, le nombre d'octets consommés par l'instruction. L'émulateur devra donc effectuer :

OFFSET	DATA	INSTRUCTION	ÉMULATION
66	00 6C	load #6C	A = 0x6C ; PC += 2

> En fait, dans ce langage, toutes les instructions consomment 2 octets, il nous paraît donc judicieux de déléguer au séquenceur (i.e. la boucle d'émulation) le rôle d'incrémenter le compteur ordinal PC ; nous ferons donc désormais comme si c'était *déjà* automatisé.

Autre exemple, celui de l'instruction suivante, qui code le transfert de la valeur de A vers la mémoire, que nous avons appelée (pour le moment) data :

OFFSET	DATA	INSTRUCTION	ÉMULATION
68	48 8D	store 8D	data[0x8D] = A

Et juste après (offset 0x6A), nous tombons sur une instruction de branchement inconditionnel, jump, qui a pour effet de modifier explicitement la valeur du registre PC ; avec brn et brz, ce sont les seules instructions dont l'exécution altère directement le registre PC ; il faudra y faire attention : l'émulation du séquenceur ne doit pas, indirectement, contrarier cet effet…

OFFSET	DATA	INSTRUCTION	ÉMULATION
6A	10 74	jump 74	PC = 0x74

Il devrait en être de même pour les instructions de branchement conditionnel ; ainsi, le code machine à l'offset 0x7A sera simulé par un if et un prédicat sur la valeur de l'accumulateur A :

OFFSET	DATA	INSTRUCTION	ÉMULATION
7A	12 8C	brz 8C	if not A : PC = 0x8C

Le document original ne contient pas d'exemple de l'instruction brn mais on devine qu'elle sera émulable de façon tout à fait similaire, sauf que le test devra être de la forme if A < 0 : …

Il existe une autre instruction load, qui implique une valeur dont l'adresse est directement donnée par l'opérande de l'opcode 0x40, et correspond fonctionnellement au store (opcode 0x48) qu'on vient de voir ; un exemple se trouve à l'offset 4 du BOOT-STRAP LOADER :

OFFSET	DATA	INSTRUCTION	ÉMULATION
04	40 20	load 20	A = data[0x20]

Ce processeur est également capable d'accéder à la mémoire de manière indirecte, c'est-à-dire que l'adresse où doit s'effectuer la lecture, ou l'écriture, est donnée *indirectement* par la valeur située à l'adresse spécifiée par l'opérande de l'opcode 0xC0, pour load *, ou de l'opcode 0xC8, pour store * ; il n'y a malheureusement aucune occurrence de l'utilisation de ces instructions dans le document original, mais la table de la section 13.3 nous en dit assez pour que nous puissions l'interpréter convenablement :

DATA	INSTRUCTION	ÉMULATION
C0 X	load *X	A = data[data[X]]
C8 X	store *X	data[data[X]] = A

En pratique, nous avons déjà utilisé ce mode d'adressage dans nos programmes ; par exemple à la section 4.3 où AF[FA['le']] utilisait comme clé d'accès dans AF la valeur extraite par FA['le'] ; ou encore à la section 4.4, où op[m[a][0]] se servait du 1er élément de la liste extraite de la table m pour retrouver, dans op, l'opérateur à appliquer…

Notez que, pour la première fois, j'ai été amené à coder l'instruction PYTHON en mettant un X pour représenter l'inconnu, là où, jusqu'à présent, j'avais vraiment une valeur donnée dans l'exemple : ce X me permet de formaliser l'expression PYTHON, autrement dit de décrire de façon générique la forme de l'expression qui deviendra effectivement interprétable lorsque j'aurais substitué à X la valeur spécifiée dans le code machine que je suis sur le point d'interpréter.

REPRÉSENTATION FORMELLE

Plutôt que X, je vais utiliser le symbole _ : pour la lisibilité, d'une part, mais aussi pour éviter tout risque de conflit avec une vraie variable du programme ; ce qui fait que les instructions déjà décortiquées plus haut vont maintenant prendre une forme totalement générique :

DATA	INSTRUCTION	ÉMULATION
00 …	load #…	A = _
40 …	load …	A = data[_]
C0 …	load *…	A = data[data[_]]
48 …	store …	data[_] = A
C8 …	store *…	data[data[_]] = A
10 …	jump …	PC = _
11 …	brn …	if A < 0 : PC = _
12 …	brz …	if not A : PC = _

Et de la même façon, je pourrai donner une description formelle similaire pour les opérations arithmétiques, l'addition et la soustraction :

DATA	INSTRUCTION	ÉMULATION
20 …	add #…	A += _
60 …	add …	A += data[_]
E0 …	add *…	A += data[data[_]]
21	sub #…	A -= _
61	sub …	A -= data[_]
E1	sub *…	A -= data[data[_]]

OPÉRATIONS D'ENTRÉES/SORTIES

Restent deux types d'opérations que nous n'avons pas encore décortiquées : celles qui ont à voir avec les entrées/sorties, et les opérations logiques binaires (BITWISE LOGICAL OPERATIONS).

Les entrées/sorties seront simulées avec la fonction intrinsèque raw_input()[44] et l'instruction print ; le code machine sera donc interprété par les expressions ci-après…

DATA	INSTRUCTION	ÉMULATION
49 …	in …	data[_] = input("byte ? ")
C9 …	in *…	data[data[_]] = input("byte ? ")
41 …	out …	print data[_]
C1 …	out *…	print data[data[_]]

44 déjà présentée à la section 4.3 pour la fonction traduis()

OPÉRATIONS LOGIQUES BINAIRES

Logique, ici, s'oppose à arithmétique ; ces opérations sont dites « binaires » parce qu'elles manipulent directement la représentation interne (en binaire, codé en complément à 2) des valeurs numériques entières :

OPÉRATION	NÉGATION		CONJONCTION				DISJONCTION			
EXPRESSION	~0	~1	0 & 0	1 & 0	0 & 1	1 & 1	0 | 0	1 | 0	0 | 1	1 | 1
VALEUR	1	0	0	0	0	1	0	1	1	1

L'opérateur ~ fonctionne comme un inverseur, et l'opérateur & peut se comprendre comme une *multiplication* bit à bit... Pour plus de détails, voir le manuel de référence du langage.

À la section 3.4, UNICODE, nous avions vu que la différence entre majuscule et minuscule dépendait de la valeur du bit 5, autrement dit le 6e en partant du bit le plus à droite de la représentation binaire du caractère : si ce bit est 0, c'est une majuscule, alors que si c'est 1, c'est une minuscule. Par exemple le caractère ASCII 'A', est représenté par le code 65_{10}, ou 41_{16}, ou 01000001_2 — alors que son homologue minuscule 'a' est représenté par 97_{10}, ou 61_{16}, autrement dit 01100001_2 :

caractère	base 10	base 16	base 2
A	65	41	01000001
a	97	61	01100001

```
>>> chr(int('01000001', 2)) # conversion base 2 → base 10 → caractère
'A'
>>> chr(int('01100001', 2)) # conversion base 2 → base 10 → caractère
'a'
```

Voici donc comment convertir de *minuscule* en *majuscule* avec un and binaire :

```
>>> chr(ord('a') & ~0x20) # 0b01100001 & 0b11011111
'A'
```

Cette expression peut paraître complexe au premier abord, mais elle ne fait qu'opérer une conjonction bit à bit de deux nombres, dont le premier est la valeur de ord('a'), c'est-à-dire son code ASCII[45], et le second est un nombre où tous les bits sont à 1, sauf celui qu'il nous intéresse de basculer à 0, le 5e précisément.

> Notez qu'ici, pour des raisons pratiques, il est plus commode de partir d'un nombre où tous les bits sont à 0, sauf le 5e, et d'en faire la négation bit-à-bit avec ~, ce qui a pour effet d'inverser chaque bit : le résultat du & avec ~ 0b00100000 est alors reconverti sous forme de caractère avec la fonction chr() pour produire un A majuscule.

Grâce à nos opérateurs logiques, nous pouvons donc changer la valeur d'un, ou plusieurs bits, en opérant directement sur la représentation interne binaire du caractère — ou d'ailleurs, de n'importe quel nombre entier ; par exemple, étant donné la représentation du caractère '9', en masquer les bits de poids fort avec 0x0F effectue une conversion en sa valeur numérique :

```
>>> ord('9') & 0x0F
9
```

Quant à l'opérateur de conjonction, il permet de forcer un bit à 1 sans toucher aux autres ; partant d'un nombre quelconque et lui appliquant une conjonction avec le nombre 1, je peux garantir que le résultat est impair sans aucun test préalable :

```
>>> 4 | 1 # 0b100 | 0b001
5
```

45 voir la présentation des fonctions ord() et chr() à la section 2.4 : *filtrage*

Le même genre d'opération me convertirait du texte en minuscule par simple application d'un masque 0x20, c'est-à-dire 0b00100000, sur chacun de ses caractères[46] :

```
>>> for x in 'SNCF' : print chr(ord(x) | 0x20),
...
s n c f
```

Ces opérations sont extrêmement utiles dès lors qu'on manipule des données implicitement binaires, comme, par exemple, les droits d'accès à un fichier, notés 'rwxr-xr-x-' : si les 'r', 'w' et 'x' apparaissent, c'est qu'ils correspondent à un 1, sinon c'est que le bit sous-jacent est à 0, et c'est justement ça que manipule la commande chmod.

Pour revenir à notre émulateur, voici donc l'interprétation des codes 0x22, 0x62 et 0xE2 :

DATA	INSTRUCTION	ÉMULATION
22 …	nand #…	A = ~(A & _)
62 …	nand …	A = ~(A & data[_])
E2 …	nand *…	A = ~(A & data[data[_]])

En fait, dans le support original, il est précisé que « & représente le ET logique » ; on s'en serait douté, mais il n'est pas spécifié s'il s'agit de logique binaire ou de logique booléenne… à vous de décider, et de mettre en œuvre les opérateurs convenables !

Comme dans tous les cas d'ambiguïté du cahier des charges, vous avez le choix entre deux options : faire le malin ou faire l'imbécile ; comme elles ont toutes deux des avantages aussi bien que des inconvénients, je me garderai bien de vous influencer.

TRADUCTION DYNAMIQUE

L'ensemble de ces représentations formelles est donc la clé de l'émulation des 21 codes d'opérations de ce langage, par conversion en 21 expressions PYTHON spécifiques, à la seule condition de pouvoir substituer une valeur pertinente au symbole '_'.

En fait, c'est la technique utilisée dans un JIT — *just in time compiler* — qui fera l'objet d'un cours plus poussé en L3. Pour un un langage compilé en BYTE CODE, comme JAVA ou PYTHON, le JIT, c'est le module qui effectue la traduction dynamique ; autrement dit, qui convertit à la volée le BYTE CODE en instructions directement compréhensibles par le processeur.

Et nous sommes justement sur le point d'en étudier la faisabilité…

46 c'est d'ailleurs comme ça qu'est codée la méthode lower(), ce qui explique pourquoi elle ne vaut pas pour l'utf-8

FAISABILITÉ

Pour effectuer cette conversion du code machine en PYTHON, je vais me fabriquer une table d'association qui met en correspondance les codes d'opération et leur équivalent PYTHON ; elle est directement construite à partir de nos tables précédentes, et présentée dans le même ordre qu'à la section 13.3 du document original :

```
opcode_table = \
{                                                          # d'après la table 13.3
   0x20: ['add #',                                         'A += _'],
   0x60: ['add',      'A += data[_]'],
   0xE0: ['add *',    'A += data[data[_]]'],
   0x21: ['sub #',    'A -= _'],
   0x61: ['sub',      'A -= data[_]'],
   0xE1: ['sub *',    'A -= data[data[_]]'],
   0x22: ['nand #',   'A = ~(A & _)'],
   0x62: ['nand',     'A = ~(A & data[_])'],
   0xE2: ['nand *',   'A = ~(A & data[data[_]])'],
   0x00: ['load #',   'A = _'],
   0x40: ['load',     'A = data[_]'],
   0xC0: ['load *',   'A = data[data[_]]'],
   0x48: ['store',    'data[_] = A'],
   0xC8: ['store *',  'data[data[_]] = A'],
   0x49: ['in',       'data[_] = input("val ? ")'],
   0xC9: ['in *',     'data[data[_]] = input("val ? ")'],
   0x41: ['out',      'print data[_]'],
   0xC1: ['out *',    'print data[data[_]]'],
   0x10: ['jump',     'PC = _'],
   0x11: ['brn',      'if A < 0 : PC = _'],
   0x12: ['brz',      'if not A : PC = _'],
}
```

La clé d'accès est le code d'opération, et la valeur est une liste de 2 éléments : le mnémonique de l'opcode et l'instruction PYTHON proprement dite. Pour le moment, seule cette dernière nous intéresse, et elle pose deux problèmes :

1. *comment remplacer le symbole '_' par quelque chose d'évaluable*
2. *comment exécuter l'instruction*

Prenons l'exemple de l'exercice 13.9, et supposons que le code machine soit chargé en mémoire, donc que data[0x50 : 0x52] retourne la liste [49 70], c'est-à-dire les deux premiers octets de code…

- la clé 0x49, dans opcode_table, nous extrait : ['in', 'data[_] = input("VAL ? ")']
  dont nous ne garderons que le 2e élément : 'data[_] = input("VAL ? ")'
- invoquer la méthode replace('_', '0x70') retourne alors 'data[0x70] = input("VAL ? ")'
- et exec('data[0x70] = input("VAL ? ")') exécute l'instruction, modifiant ainsi la valeur de l'élément d'indice 0x70 du vecteur data

En pratique, pour démarrer du code machine, il faut que le registre PC soit initialisé avec l'adresse origine, 0x50 ; à partir de là, la procédure ci-dessus peut être reformulée en PYTHON :

```
op, arg = data[PC:PC+2]
PC += 2
exec(opcode_table[op][1].replace('_', str(arg)))
```

Notez la conversion de l'opérande arg en chaîne, pour pouvoir l'insérer à la place du '_'.

Itérer ces 3 instructions constitue donc le cœur du séquenceur : notez que le registre PC est incrémenté aussitôt après le décodage, ce qui garantit un séquencement correct dans tous les cas, y compris ceux de branchement, puisque l'instruction à exécuter aura le dernier mot.

À ce stade, réaliser le programme revient à définir, en 5 lignes, une fonction run() qui prendrait comme argument l'adresse origine du programme à émuler, et tournerait indéfiniment en itérant les trois instructions que nous avions listées plus haut :

```
def run(PC) :
   while True :
      op, arg = data[PC:PC+2]
      PC += 2
      exec(opcode_table[op][1].replace('_', str(arg)))
```

Et ceci satisfait pleinement le cahier des charges pour l'exercice [px34-1]…

Mais le problème, c'est que, mises à part les instructions out, nous n'aurons aucun feed-back du déroulement de l'exécution — sachant que, par ailleurs, aucun des programmes simulés n'a prévu de s'arrêter[47] — à avorter obligatoirement par ^C !

D'une part, les contraintes de la mise au point nous imposent une approche un peu plus conviviale, d'autrepart, ce serait sympathique de voir ce que fait le code machine, où en sont les registres, ou même quel est l'état de la mémoire. Car en définitive, comment savoir si le code machine ne contient pas lui-même un bug ? Le fait que le code ne fait pas ce qu'on attend dépend de la qualité de l'émulation, mais aussi de la qualité du code émulé. On va donc aménager le séquenceur pour qu'il nous affiche l'instruction en cours d'exécution, ainsi que les valeurs des registres :

```
def run(PC, A = 0) :
   while True :
      print 'PC: %03i | A: %03i |' % (PC, A),
      op, arg = data[PC:PC+2]
      PC += 2
      acode = opcode_table[op][0]
      xpress = opcode_table[op][1].replace('_', str(arg))
      print '%s %i\t| %s \t' % (acode, arg, xpress),
      step()
      exec(xpress)
```

Maintenant, la toute première instruction de la boucle while affichera la valeur courante du compteur ordinal et de l'accumulateur ; la variable acode représente le mnémonique qui serait utilisé dans le langage d'assemblage (ASSEMBLY CODE), et xpress a comme valeur le code PYTHON sur le point d'être exécuté par la fonction exec() de la dernière ligne.

Quant à la fonction step(), ce n'est, pour le moment, qu'un embryon de ce que devrait être un véritable STEPPER, mais nous pourrons l'augmenter à loisir.

```
def step() :
   k = raw_input ('? ')                          # command processing
   while k.isdigit() :                           # memory dump
      print k, ':', data[int(k)]
      k = raw_input ('? ')
```

Telle qu'elle est codée, elle ne sait interagir qu'avec l'appui sur ↵ qui retourne immédiatement à run(), ou l'entrée d'un nombre entier, interprété comme une adresse, dont la valeur est alors affichée — et grâce au while, on peut en faire plus d'une d'affilée…

Voici ce que ça donne, pour le 13.9, jusqu'à l'instruction out à l'offset 126_{10}, c'est-à-dire 0x7E ; le ? en fin de ligne est l'invite affichée par step() :

```
PC: 080 | A: 000 | in   112     | data[112] = input("val : ")
val : 3
PC: 082 | A: 000 | load 112     | A = data[112]        ?
PC: 084 | A: 003 | store 113    | data[113] = A        ?
PC: 086 | A: 003 | store 114    | data[114] = A        ?
PC: 088 | A: 003 | load # 94    | A = 94               ?
PC: 090 | A: 094 | store 141    | data[141] = A        ?
PC: 092 | A: 094 | jump 116     | PC = 116             ?
PC: 116 | A: 094 | load # 0     | A = 0                ?
PC: 118 | A: 000 | store 115    | data[115] = A        ?
PC: 120 | A: 000 | load 113     | A = data[113]        ?
PC: 122 | A: 003 | brz  140     | if not A : PC = 140  ?
PC: 124 | A: 003 | sub # 1      | A -= 1               ?
PC: 126 | A: 002 | out  113     | print "\n", data[113]
3
```

Il n'en coûterait pas grand'chose de réaliser l'affichage en hexadécimal, en remplaçant les %i par des %x

47 il n'y a pas d'instruction stop, mais le programme pourrait faire un jump à l'adresse 0 pour se mettre en attente…

dans les instructions print de la boucle du séquenceur — et peut-être aussi str(arg) par str(hex(arg)), pour une cohérence globale :

```
PC: 50 | A: 00 | in 70  | data[0x70] = input("val ? ")
val ? 3
PC: 52 | A: 00 | load 70        | A = data[70]         ?
PC: 54 | A: 03 | store 71       | data[71] = A         ?
PC: 56 | A: 03 | store 72       | data[72] = A         ?
PC: 58 | A: 03 | load # 5E      | A = 5E               ?
PC: 5A | A: 5E | store 8D       | data[8D] = A         ?
PC: 5C | A: 5E | jump 74        | PC = 74              ?
PC: 74 | A: 5E | load # 00      | A = 0                ?
PC: 76 | A: 00 | store 73       | data[73] = A         ?
PC: 78 | A: 00 | load 71        | A = data[71]         ?
PC: 7A | A: 03 | brz 8C         | if not A : PC = 8C   ?
PC: 7C | A: 03 | sub # 01       | A -= 1               ?
PC: 7E | A: 02 | out 71         | print "\n", data[71]
3
```

Bien entendu, ces 15 lignes de code ne comprennent pas la lecture des données, que nous allons définir avec une petite fonction, dès que nous aurons adopté un format pour le fichier. Pour rester simple, conservons comme modèle celui du 13.8, et fixons-en le format source. Pour être sûr de ne pas faire d'erreur en recopiant les données du document original, je propose de garder la forme hexadécimale *non préfixée*, telle qu'on l'a manipulé au début :

```
offset 30
code 10 3A 00 31 40 32 60 33 48 32 49 33 40 33 22 FF 12 34 41 32 10 44
```

Bien entendu, on peut présenter le code autrement, par exemple avec deux octets par ligne, comme pour le 13.9, mais les deux mots-clé, offset et code, devront être présents ; on pourrait s'en passer, mais leur présence est, pour le programme, une indication de quoi faire avec les données — cf. EXTENSIBILITÉ, ci-après ; la lecture du fichier est relativement simple :

```
def read_hexcode(seg, dict) :                          # un nom, un dict vide
   f = open('paper ' + seg + '.hexcode')
   bytes = f.read().split()
   if bytes[0] == 'offset' :
      dict['offset'] = eval('0x' + bytes[1])
   if bytes[2] == 'code' :
      dict['code'] = [eval('0x' + x) for x in bytes[3:]]
   return dict
```

Et la valeur qu'il retourne (de type 'dict') va servir à augmenter la table segment_table (qui doit déjà exister), prenant éventuellement la place d'un segment de même nom dont une ancienne version pourrie aurait été chargée précédemment :

```
seg = '18.3'
segment_table[seg] = read_hexcode(seg, {})
```

Notez que je ne passe à read_hexcode() que le nom du segment, mais que, chez moi, le fichier s'appelle véritablement 'paper 18.3.hexcode', nom reconstitué implicitement par la fonction. Pensez à modifier ce détail si vous décidez d'adopter d'autres conventions que les miennes…

L'émulateur ne peut être lancé avant de charger le code en mémoire : je vais donc devoir le copier, octet par octet, dans le vecteur data, tâche que je vais déléguer à la fonction load_image() :

```
def load_image(seg_name) :                             # segment name
   offset = segment_offset(seg_name)
   code = segment_code(seg_name)
   for x in range(len(code)) :
      data[offset + x] = code[x]
```

Celle-ci nécessite deux petites fonctions, segment_offset() et segment_code(), qui font l'interface entre les données et le programme : ce n'est pas pour le plaisir de compliquer les choses, c'est pour garantir qu'il n'y ait qu'une manière d'accéder aux données : si plus tard je décide de changer la structure des données, adapter le programme entier à cette nouvelle structure ne requiert que la redéfinition de ces fonctions d'interface :

```
def segment_offset(name) : return segment_table[name]['offset']

def segment_code(name) : return segment_table[name]['code']
```

Lancer le programme, c'est charger le code à émuler, puis appeler le séquenceur :

```
def simul(new) :                                    # un nom de nouveau segment
   for seg in ['boot-strap loader', new] :
      segment_table[seg] = read_hexcode(seg, {})
      load_image(seg)
   run(segment_offset(new))
```

Remarquez que, pour me conformer au cahier des charges, j'installe obligatoirement, comme premier segment, le 'boot-strap loader', avant de charger le véritable segment à émuler.

J'ai ajouté, en fin de script, ce qu'il faut pour le rendre autonome ; il faut donc l'appeler en lui passant, comme argument, le nom du fichier contenant le code à émuler.

La fonction read_hexcode() s'attend à ce que le fichier soit de la forme :

```
offset
   30
code
   10 3A
   00 31
   40 32
   60 33
   48 32
   49 33
   40 33
   22 FF
   12 34
   41 32
   10 44
```

L'exemple ci-dessus est celui du 13.8 : les sauts de ligne n'ont pas d'importance, les tabulations non plus, puisque, de toute façon, split() les confond avec les espaces ; le fichier peut donc être formaté comme on préfère, l'important étant qu'on n'y introduise pas d'erreur.

Pour vous faciliter la tâche, les fichiers de code machine peuvent être téléchargés directement à partir du site de l'IED :

code de l'exercice 13.8 :
http://foad.iedparis8.net/claroline/courses/E464/document/python-code/paper%2013.8.hexcode

code de l'exercice 13.9 :
http://foad.iedparis8.net/claroline/courses/E464/document/python-code/paper%2013.9.hexcode

code du boot-strap loader :
http://foad.iedparis8.net/claroline/courses/E464/document/python-code/paper%20boot-strap%20loader.hexcode

SCRIPT COMPLET

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

data = [0] * 256                                          # data bytes used as memory
segment_table = {}

opcode_table = \
{
   0x00: ['load #',                                    'A = _'],
   0x10: ['jump',    'PC = _'],
   0x11: ['brn',     'if A < 0 : PC = _'],
   0x12: ['brz',     'if not A : PC = _'],
   0x20: ['add #',   'A += _'],
   0x21: ['sub #',   'A -= _'],
   0x22: ['nand #',  'A = ~(A & _)'],
   0x40: ['load',    'A = data[_]'],
   0x41: ['out',     'print data[_]'],
   0x48: ['store',   'data[_] = A'],
   0x49: ['in',      'data[_] = input("val : ")'],
   0x60: ['add',     'A += data[_]'],
   0x61: ['sub',     'A -= data[_]'],
   0x62: ['nand',    'A = ~(A & data[_])'],
   0xC0: ['load *',  'A = data[data[_]]'],
   0xC1: ['out *',   'print data[data[_]]'],
   0xC8: ['store *', 'data[data[_]] = A'],
   0xC9: ['in *',    'data[data[_]] = input("val ? ")'],
   0xE0: ['add *',   'A += data[data[_]]'],
   0xE1: ['sub *',   'A -= data[data[_]]'],
   0xE2: ['nand *',  'A = ~(A & data[data[_]])']
}

def simul(new) :
   for seg in ['boot-strap loader', new] :
      segment_table[seg] = read_hexcode(seg, {})
      load_image(seg)
   run(segment_offset(new))

def read_hexcode(segment, new) :
   f = open('paper %s.hexcode' % segment)                 # file name
   text = f.read().split()
   if text[0] == 'offset' : new['offset'] = eval('0x' + text[1])
   if text[2] == 'code' : new['code'] = [eval('0x' + x) for x in text[3:]]
   return new

def load_image(name) :                                    # segment name
   offset = segment_offset(name)
   code = segment_code(name)
   for x in range(len(code)) : data[offset + x] = code[x]

def run(PC, A = 0) :
   while True :
      print 'PC: %03i | A: %03i |' % (PC, A),
      op, arg = data[PC:PC+2]
      PC += 2
      acode = opcode_table[op][0]
      xpress = opcode_table[op][1].replace('_', str(arg))
      print '%s %i\t| %s \t' % (acode, arg, xpress),
      step()
      exec(xpress)

def step() :
   k = raw_input ('? ')                                   # command processing
   while k.isdigit() :                                    # memory dump
      print k, ':', data[int(k)]
      k = raw_input ('? ')

def segment_offset(name) : return segment_table[name]['offset']

def segment_code(name) : return segment_table[name]['code']

from sys import argv
if len(argv) is 1 : exit('manque le nom du fichier à émuler')
else : simul(argv[1])
```

EXTENSIBILITÉ

Un problème sournois se pose du fait qu'on émule un processeur 8-bits avec un langage qui, en interne, représente les valeurs entières sur 32 ou 64 bits — j'ai ainsi vu, au cours de mes expériences, des valeurs en mémoire dépasser 255 : voyez-vous un moyen de contraindre le programme à travailler exclusivement sur 8 bits ?

Cet émulateur est largement perfectible : pour le moment, il est codé comme si on ne pouvait se tromper dans le nom du fichier code machine, et comme s'il n'y avait jamais d'erreurs dans le code à émuler…

[px34-2] insérer des try ~ except pour détourner les erreurs d'ouverture de fichier, et pour piéger les opcodes non-définis, erreur typique d'un programme qui branche sur une donnée.

Le STEPPER existant est terriblement rudimentaire : en utilisant le programme, ne serait-ce que pour sa mise au point, on aimerait pouvoir afficher une plage de mémoire en spécifiant deux valeurs entières (au lieu d'une), ou encore toute la mémoire sous la forme de 16 lignes de 16 bytes, ou même une commande a, et une pc, pour forcer la valeur des registres…

[px34-3] développez l'idée du STEPPER en lui ajoutant la capacité de traiter d'autres commandes que vous jugez utiles.

La capacité de l'émulateur à désassembler le code machine est à peine esquissée : on devrait pouvoir afficher le code machine de manière symbolique, de sorte que les codes d'opération soient remplacés par leur mnémonique, et les adresses par une étiquette symbolique, comme store valeur, ou jump encore ; par exemple, pour le 13.8 :

```
30  10 3A             jump  start
32  00 31             load  #31
34  40 32      next:  load  32
36  60 33             add   data
38  48 32             store val
3A  49 33      start: in    data
3C  40 33             load  data
3E  22 FF             nand  #FF
40  12 34             brz   next
42  41 32             out   val
44  10 44      end:   jump  end
```

[px34-4] ajouter au programme la capacité d'afficher le code machine de manière symbolique.

Remarque : aucune de ces extensions n'exige un travail démesuré ; celui-ci, par exemple, se réalise en moins de 20 lignes de code… à condition de rajouter les étiquettes symboliques dans la table du segment, comme ceci :

```
segment_table['13.8'] = \
{  'offset' : 48,
   'code' : [16, 58, ..., 16, 68],
   0x3A : 'start',
   0x32 : 'val',
   0x33 : 'data',
   0x34 : 'next',
   0x44 : 'end' }
```

La fonction d'affichage n'a plus qu'à vérifier, avant le print d'un opérande, si cette adresse est définie dans la table, et la remplacer, le cas échéant, par le nom correspondant. Évidemment, ce serait mieux de pouvoir conserver ça dans le fichier source, et voilà pourquoi j'ai prévu un format à mots-clé aisément augmentable : OFFSET, CODE, LABELS, … — cf. read_hexcode().

[px34-5] exploiter les données existantes pour construire un assembleur symbolique (capable de lire un texte source de programme ne contenant que des symboles mmémoniques ou d'adresses) dont la sortie serait le code machine exécutable par notre émulateur.

ÉPILOGUE

Ce qu'on a appris ici est tout à fait rudimentaire – et j'écris ceci en étant pleinement conscient de l'effort que tout ça vous a quand même demandé. Mais l'important, c'est que vous devriez avoir maintenant des bases solides pour aborder les concepts mis en œuvre dans les autres langages de programmation.

Des bases solides ? Quand on y regarde de près, il n'y en a qu'une : c'est l'application du principe diviser pour régner, qui implique (1) que les données peuvent être manipulées, déstructurées puis restructurées, et (2) que ces manipulations se font plus facilement en déléguant à des fonctions ancillaires des tâches spécifiques. Ainsi, le dénominateur commun à tous les langages, c'est qu'ils permettent de représenter les données à traiter ; ensuite, c'est la manière de les représenter qui est déterminante, voire contraignante, pour la façon de les traiter : il incombe donc au programmeur de choisir avec soin ses représentations initiales.

En fait, les concepts ne diffèrent guère d'un langage à l'autre, et le principal obstacle à surmonter pour l'apprentissage d'un nouveau langage, c'est la syntaxe de ses représentations, que ce soient des données ou des programmes.

Certains, comme LISP, vous paraîtront plus simples, alors qu'un langage de bas niveau comme C se montrera incroyablement primitif ; d'autres, comme PROLOG, vous obligeront à exploser votre mode de raisonnement ; SQUEAK ou OBJECTIVE C vous demanderont de penser en termes de transmission de messages : au lieu d'écrire un expression comme `2 + 3 + 4` vous enverrez à l'objet `2` un message composé du sélecteur de méthode “`+`” et des objets `3` et `4`.

En définitive, vous vous rendrez compte que ce qui s'exprime difficilement dans un langage donné peut se formuler aisément dans un autre : car en principe, chaque langage a été créé pour représenter et résoudre plus facilement certains types de problèmes, et c'est cette culture-là que vous êtes en train d'acquérir – culture qui vous permettra, à l'avenir, de choisir l'outil le plus approprié pour accomplir une tâche spécifique.

Et même, ceci vous incitera peut-être à créer votre propre outil, autrement dit votre propre langage : vous ne serez certainement pas le premier, ni probablement le dernier…

Mais le plus important n'est pas le langage : la première qualité d'un programmeur, c'est sa curiosité, et il se trouve que la curiosité, ça va de pair avec l'imagination. C'est pour cette raison qu'il ne faut pas craindre de considérer l'informatique comme une discipline expérimentale : au début de l'histoire, personne n'aurait imaginé qu'un « mulot » pourrait enfoncer des boutons virtuels ou déplacer des objets *iconiques* sur un simulacre de bureau.

Et ces choses-là, il faut bien que quelqu'un les invente… Lisez ce que dit le Dr. Dobb à propos de Xerox PARC (http://www.ddj.com/184404385) et vous découvrirez qu'il y a déjà plus de 30 ans, quelqu'un s'est dit : et si on mettait ensemble les plus bricolos d'entre nous pour voir ?

C'est comme ça qu'ont été inventés les concepts d'écran PAPER-WHITE et d'interface graphique, d'où la métaphore des fenêtres et celles des SCROLL-THUMBS (ou ascenseurs), le concept d'icône représentant des objets, le concept de POINTING DEVICE (souris, tablette graphique), mais aussi la technologie de l'imprimante LASER, le langage POSTSCRIPT, ou encore le réseau ETHERNET…

Et notre bon Dr. Dobb a cette formule rafraîchissante :

> *it is because the best way to understand the present is to study (and invent) multiple futures*.

Tout un *programme*, n'est-ce pas ?

⑨ ANNEXES

Le texte de ce chapitre est d'abord une compilation d'articles publiés sur le forum, mais aussi des fragments de cours qu'il a fallu déporter : des pages culturelles, qui disent des choses qu'il faut connaître pour comprendre le reste, même si on en retient rien d'autre que le principe.

C'est le cas des modules, par exemple… On peut très bien importer le module turtle sans se poser la question de ce qui se passe au moment d'une telle opération, comme on peut avoir envie de comprendre pourquoi des fois on utilise la syntaxe from <module> et des fois pas…

On revisitera les questions souvent posées à propos de l'indentation en PYTHON, et on mettra au point quelques détails à propos de la programmation en SHELL bash.

La dernière section, PROGRAMMATION NUMÉRIQUE, est le passage obligé pour ceux qui veulent faire les graphes proposés dans le cours de mathématiques : si j'y suggère de convertir le code ANSI-C en PYTHON, c'est parce que c'est ce qu'il y a de plus facile pour tirer profit de l'expérience acquise au long des chapitres 7 et 9 du cours d'introduction ; l'alternative, pour ceux qui en auraient le temps, serait d'apprivoiser un autre système graphique utilisable directement en ANSI-C, donc de conserver tels quels les programmes ANSI-C déjà codés.

Il existe une 3e route qui permettrait de préserver le code ANSI-C en le compilant sous la forme d'un module PYTHON, donc de persister avec Tkinter ; petit problème : c'est une manipulation que nous n'apprendrons à faire que dans l'avant-dernier chapitre de l'ÉC 213 – donc vers la fin du 2e semestre…

sommaire

9.1 MODULES

Nous avions déjà manipulé le module turtle aux chapitres 0 et 2, puis le module sys au chapitre 3 : le moment est venu de clarifier le concept ; c'est pourquoi je vais vous proposer de créer votre propre module PYTHON… en PYTHON. Il y a d'autres manière de créer des modules, mais c'est quelque chose qu'on apprendra plutôt vers la fin du cours 213, avec le langage ANSI-C.

En attendant, cette section discute de ce qu'il est utile de savoir pour manipuler ces extensions en toute sérénité. Encore une fois, il n'y a rien à apprendre, et on pourrait, à la rigueur, se contenter de connaître la syntaxe de l'import d'un module pour l'utiliser immédiatement ; d'ailleurs, n'est-ce pas ce que nous avons fait jusqu'à présent ?

9.1.1 MODULES MAISON

Ce que je vais proposer ici est un peu vaniteux : comment faire de plusieurs un module… C'est sûr que ça n'en vaut pas vraiment la peine, vu son utilité, mais après tout, ça ne change rien à la technique, alors celui-là ou un autre…

Un module est un fichier contenant du code dont on pourrait avoir besoin dans plusieurs scripts différents ; dans cette optique, il paraît judicieux de le conserver dans un fichier à part, sachant qu'il est aisé de l'importer dans un nouveau programme, où il se comporte alors comme s'il avait été littéralement inclus dans le programme : les variables et les fonctions définies dans le module sont directement disponibles.

CRÉATION D'UN MODULE

Pour faire de plusieurs un module, mettez la définition de pluriel() dans un fichier nommé plusieurs.py, et c'est tout ; vous pouvez, bien sûr, y mettre des commentaires sur comment ça marche et quels bugs n'ont pas encore été corrigés : c'est certainement très utile, mais ce n'est pas nécessaire pour faire un module… Voilà donc le contenu de plusieurs.py :

```
def pluriel(mot) : return mot + 's'
```

Maintenant, il ne reste plus qu'à coder le script qui va utiliser ce module ; il doit contenir, en pratique, tout ce que contenait le script autonome du [px11-3], hormis la définition de la fonction modularisée – ainsi que les définitions de variables qui allaient avec, s'il y en avait. Donc, pour faire simple, voici le contenu du script essaye :

```
#! /usr/bin/env python
import plusieurs
import sys
if len (sys.argv) > 1 : print plusieurs.pluriel(sys.argv[1])
else : exit('argument manquant : mot au singulier')
```

Le simple fait de rencontrer une instruction import va obliger l'interprète à compiler le module, ce qui va générer un fichier de même nom, mais affublé du suffixe pyc (PYTHON compilé) ; désormais, chaque fois qu'on importera ce module, c'est la version compilée qui sera chargée directement. Ceci veut dire que si le code du module nécessite des retouches et des mises au point, il est nécessaire de détruire manuellement la version compilée, pour forcer PYTHON à la recompiler.

Remarquons qu'à ce stade il n'est pas obligatoire de faire de essaye un script autonome ; il est toujours possible de l'utiliser en tant qu'argument de la commande PYTHON :

```
python essaye le dernier mot banal
```

De ces six mots entrés sur la ligne de commande, PYTHON enlève le tout premier, 'python', pour recréer les conditions dans lesquelles le script essaye serait lancé de façon autonome : la variable sys.argv ne contiendra donc que ['essaye', 'le', 'dernier', 'mot', 'banal'].

ESPACES DE NOMMAGE

On avait remarqué qu'après le chargement d'un module, les noms qu'il définit sont accessibles à condition de les préfixer du nom du module : ici, plusieurs.pluriel() ou sys.argv. La raison en est que pour chaque module

est créé un espace de nommage, et que pour accéder à un nom de cet espace, il faut le préfixer du nom de l'espace (les même mécanismes existent en CLISP).

Mais, comme on l'a vu avec module turtle, il y a une autre syntaxe d'importation, qui importe directement les noms du module dans l'espace courant : il n'y a donc plus besoin de préfixer quoi que ce soit…

Mais il faut être prudent car, dans ce cas, il se peut qu'un nom importé remplace un nom défini dans votre programme, ou l'inverse : ça dépend du point où se fait effectivement l'importation. Si le module est importé tout en haut du script, il est possible de modifier par mégarde une définition de fonction ou de variable. Si, au contraire, vos définitions précèdent l'importation, il est possible que soit celles du module qui supplantent les vôtres…

C'est pour ça qu'il existe encore une 3e syntaxe, qui permet de n'importer que les noms que vous voulez rendre accessibles sans préfixe :

```
from plusieurs import pluriel
```

En fait, je ne sais pas ce qu'il y a dans le module plusieurs, mais ce dont je suis sûr, c'est que je n'aurais jamais besoin d'utiliser directement les variables qui listent les exceptions utilisées par la fonction pluriel().

Et pour le cas où le nom pluriel lui-même entrerait en conflit avec une de mes définitions locales, il est encore prévu une 4e syntaxe :

```
from plusieurs import pluriel as multiple
```

qui définit implicitement un synonyme[48] de la fonction pluriel(), ce qui veut dire qu'en évaluant multiple('trou'), j'obtiendrai, tout pareil, la valeur 'trous'… En fait, ça revient exactement au même que si j'avais codé :

```
import plusieurs
multiple = plusieurs.pluriel
```

sauf que c'est automatique…

Remarquons, au passage, que les symboles globaux d'un module ne sont accessibles qu'en lecture seule, et qu'il n'est pas possible de redéfinir ces symboles à partir du script importateur : du fait que le module est compilé, les liaisons sont figées dans ce contexte.

Une fonction du module appelée par une autre fonction du même module ne peut donc être remplacée par une fonction du script importateur ; de même, les variables globales auront forcément la dernière valeur attribuée dans le module.

En fait, les fonctions du module peuvent toujours changer les valeurs des globales du module ; donc, si c'est nécessaire, l'importateur peut changer ces valeurs… à la seule condition de passer par une fonction définie à cet effet dans le module, et de lui passer les nouvelles valeurs en arguments.

MISE AU POINT DU MODULE

Pour faciliter la mise au point du module (et du même coup, documenter son utilisation), il est possible d'y inclure un bout de code conditionnel qui ne sera évaluée que si le module est utilisé directement. En effet, PYTHON attribue le nom '__main__' au programme principal, et ce nom est la valeur de la variable __name__ ; le prédicat __name__ == '__main__', évalué dans un script, ne sera donc vrai que si c'est le script maître, pas s'il est exécuté en tant que module…

En pratique, ça reviendrait à prendre les 3 lignes rajoutées pour faire du script un programme autonome, mais à les soumettre à une évaluation conditionnelle :

```
if __name__ == '__main__' :
   import sys
   if len (sys.argv) > 1 : print plusieurs.pluriel(sys.argv[1])
   else : exit('argument manquant : mot au singulier')
```

48 cf. chapitre 1

Lorsque le script est compilé en tant que module, il se débarrasse du code qui ne sera jamais évalué : c'est comme si ces lignes n'existaient pas…

9.1.2 MODULES D'USINE

Pour des raisons d'économie, PYTHON ne charge en standard qu'une (très) faible partie de ses ressources, vous laissant le soin de charger les modules spécialisés dont vous pourriez avoir besoin. Par exemple, essayez ceci :

```
help('modules')
```

qui vous affiche une liste des 380 modules que vous pourriez charger…

MODULE RANDOM

Par exemple, à la dernière ligne de la liste, je vois le module random, qui définit des fonctions utilisées pour des calculs statistiques. Essayez :

```
import random
dir(random)
```

qui vous retourne la liste de tout les objets importés depuis le module random… Ainsi, l'expression random.random() retourne un nombre réel pseudo-aléatoire entre 0.0 et 1.0. Essayez :

```
random.random() ; random.random() ; random.random()
```

et vous verrez ce que je veux dire… Si vous avez besoin d'un entier entre 5 et 8, essayez plutôt ceci :

```
random.randint(5, 8)
```

Et une expression comme random.choice('azertyuiop') vous retournera un élément pris au hasard dans la séquence donnée en argument.

Mais notez que ce mode d'importation vous oblige à *préfixer* le nom de la fonction par le nom de son module, ce qui élimine les risques de collision avec les noms déjà existants dans votre espace de travail.

Une autre façon d'importer les fonctionnalités d'un module, c'est d'utiliser l'instruction :

```
from random import *
```

où, à la différence du mode précédent, les nouveaux symboles importés supplantent ceux déjà définis avec le même nom dans votre environnement, il n'est donc plus nécessaire de les préfixer… Il n'est d'ailleurs pas non plus nécessaire de les importer tous :

```
from random import choice                          # importe le nom choice, une référence
choice('azertyuiop')                               # choisis un élément de la séquence
choice('pique cœur carreau trèfle'.split())        # comme par hasard...
```

MODULE OS

Autre exemple, le module os (i.e. *operating system*) qui contient *tout* ce qu'il faut pour effectuer, par programme, les opérations qu'on pourrait commander depuis le SHELL :

```
from os import getcwd, listdir, chdir, chmod
```

- getcwd() fait pareil que la commande SHELL pwd, sauf que c'est une fonction, donc que je peux mémoriser la valeur qu'elle retourne pour y revenir plus tard ;
- listdir() est l'équivalent de ls, sauf que là encore, la valeur retournée est une liste de noms de fichier, qu'on peut éplucher à loisir ;
- chdir() est comme cd, à ceci près qu'elle ne sait pas interpréter le symbole '~' qui veut dire *dossier personnel de l'utilisateur* ; et attention, elle ne retourne rien ;
- chmod() accepte un nom de fichier et un nombre qui représente le mode d'accès pour ce fichier ; en bash, les arguments sont inversés, pour pouvoir spécifier plusieurs fichiers en une seule commande…

Ne vous laissez pas désemparer par les noms de fonction qui ne ressemblent pas à ceux des commandes bash : ces noms sont ceux de la bibliothèque ANSI-C standard, et ce sont ceux que vous aurez à manipuler si vous voulez faire ce genre de chose en ANSI-C ; ce sont donc les noms des commandes bash qui se démarquent des originaux, et non l'inverse : la raison en est que quand on veut aller vite pour faire des trucs à la console, on préfère taper cd que chdir.

Voici, pour l'exemple, un programme idiot qui marque sans discernement tous les fichiers '.py' comme s'ils étaient véritablement exécutables :

```
ici = getcwd()
fichiers = listdir(ici)
for f in fichiers :                          # scanne la liste
   if f.endswith('.py') :                    # ceux qui finissent en '.py'
      chmod(f, 0744)                         # met-les moi en exécutable

chdir('..')                                  # remonte dans l'arborescence
print listdir(getcwd())                      # listdir s'applique à la valeur de getcwd()
chdir(ici)                                   # retour à la case départ
```

Dernière remarque : la navigation avec chdir() n'affecte que le point de vue de PYTHON, le processus en cours, mais ne peut modifier le point de vue qu'avait le SHELL avant de lancer ce processus : quand on arrête PYTHON, on se retrouve là où on était avant. Ce mécanisme est expliqué en détails dans le cours de « système », en 2e année.

Encore une fonction que vous retrouverez telle quelle en ANSI-C : system() prend une chaîne de caractères en argument, et invoque le SHELL en lui passant cette chaîne…

```
from os import system
exit_code = system('ls -alF')
```

La valeur retournée par system() est le code numérique qu'aurait retourné la commande si on l'avait exécutée à partir du SHELL, code qu'on mettrait en évidence en demandant echo $? immédiatement après ; en pratique, c'est 0 s'il y a 0 erreur, toute autre valeur dénotant une anomalie dans le déroulement du processus invoqué par la commande.

Si la chaîne passée en argument comporte plusieurs lignes, tout se passe donc comme si on exécutait un SHELL-SCRIPT. Encore un exemple de programme idiot, en supposant, bien sûr, que le script zou.py existe :

```
from os import system
prog = 'zou.py'
prog = 'chmod u+x ' + prog + '\n' + prog
exit_code = system(prog)
```

où le code de retour sera celui du programme zou.py, la dernière commande. Ici, le caractère '\n' dans la chaîne prog est un alinéa (ou *linefeed*) qui fait que prog comporte bien deux lignes distinctes.

Comme on l'a vu plus haut, la distribution PYTHON 2.5 comprend 380 modules[49] standard, et il en existe plein d'autres, plus ou moins spécialisés : la page http://pypi.python.org/pypi en recense plus de 14000, ce qui veut dire qu'il y a peut-être déjà une solution au problème que vous avez l'intention de résoudre par programme.

49. cf. http://docs.python.org/

9.2 INDENTATION

Nulle part on ne trouve de règles explicites pour l'indentation, un peu comme si ça allait de soi… Ces règles sont effectivement évidentes pour le programmeur qui se frotte d'abord à l'interprète en direct avant de penser son code en tant que script – et c'est d'ailleurs l'approche que je préconise tout au long du cours ; et pas seulement pour cette raison.

MODE INTERACTIF, MODE SCRIPT

Un script PYTHON peut être évalué en mode *interactif* (ou *conversationnel*, ligne par ligne, directement sous l'interprète en tant que *shell*), ou en mode *script ;* les deux modes doivent se soumettre aux mêmes règles de « bonne » formation des expressions ; la différence est que dans le premier cas l'interprète est obligé de prendre des décisions immédiates, « à la volée » : toute faute de syntaxe empêche l'évaluation de l'expression ; alors que dans le deuxième cas, la totalité du script est analysée *avant* évaluation, ce qui permet d'assouplir certaines règles, en particulier celles de l'indentation.

Il y aurait, semble-t-il, un troisième cas, celui ou un script est importé (en tant que *module*) par un autre, mais, en fait, ça nous ramène implicitement au deuxième cas, avec cette différence qu'après vérification, le code-source est compilé et envoyé dans un fichier de même nom + le suffixe « pyc » (PYTHON *compilé*) ou « pyo » (PYTHON *compilé* et *optimisé*).

DU SIMPLE AU COMPLEXE

On distinguera les instructions simples des instructions complexes (*compound statements*) ; les premières sont obligatoirement contenues dans une seule ligne, en principe terminée par un code d'alinéa (0x0A dans l'environnement unix), faute de quoi la ligne n'est tout simplement pas « envoyée » ; on peut aussi en regrouper plusieurs sur une ligne en les séparant par un « ; »

```
a = 1
b = 2
c = a + b

a = 1 ; b = 2 ; c = a + b
```

les secondes sont reconnaissables au fait qu'elles contiennent obligatoirement un « : » dans l'instruction d'origine, et doivent être suivies d'au moins une instruction ; elles peuvent s'épanouir sur plusieurs lignes : typiquement, les définitions de *fonction* ou de *classe*, mais aussi les structures de *contrôle*, comme with, for, while, if, else, try, except, dont il y a de nombreux exemples dans le support de cours.

```
def pair (x) :
   return not x % 2
```

SÉMANTIQUE FORMELLE

PYTHON est l'un des rares langages formels à attribuer une sémantique à la *forme* globale du programme : cette forme est régie par des règles strictes d'indentation ; l'indentation permet de s'affranchir des *caractères délimiteurs* auxquels les autres langages sont obligés de recourir pour délimiter les blocs de code de façon non ambigüe. Par exemple, les {…} en perl :

```
my $count = 1 ;
while (my $line = <STDIN>)
{
   print "$count\t$line" ;
   $count++ ;
}
```

En PYTHON, toute instruction, simple ou complexe, commence au niveau d'indentation par défaut, niveau 0, c'est-à-dire en colonne 1 ; ce serait une faute de commencer plus loin dans la ligne ; d'ailleurs, en mode *interactif*, personne n'aurait l'idée de « décaler » une instruction de façon gratuite, et il doit en être de même dans un *script*.

Une instruction complexe introduit implicitement un *bloc* de code : plusieurs instructions à évaluer en séquence, séquence *assujettie* à cette instruction complexe ;

- dans ce cas, le bloc doit être indenté, c'est-à-dire que les lignes de la séquence qui constitue le bloc doivent être toutes indentées d'un *cran* vers la droite, et toutes exactement de la même manière ; le *cran* en question est autodéfini par le tout premier *cran* du bloc ; il peut être constitué de tabulations ou d'espaces, mais la mixture des deux est déconseillée ; après tout, s'il suffit de l'espace pour marquer un décalage, pourquoi faire plus compliqué ?
- toute contravention est une faute de syntaxe ;
- lorsqu'une instruction complexe n'introduit qu'une seule autre instruction, il est possible de la mettre immédiatement à la suite du « : », sans passer à la ligne, donc sans indenter quoi que ce soit ; en principe, il est possible de ramener un bloc à une seule ligne en séparant par « ; » les instructions qui le constituent, mais, en pratique, ça nuit à l'intelligibilité du code, et c'est donc considéré comme une *mauvaise* pratique ; en tout cas, on ne peut *pas* ramener 2 blocs à une seule ligne : il ne peut *jamais* y avoir plus d'une instruction complexe par ligne ;
- quand, en mode *interactif*, l'interprète rencontre un « : » dans la ligne, il passe délibérément en mode *continuation*, indiqué par un « ... » en colonne 1 ; à ce stade, on peut *terminer* le bloc avec un simple alinéa, mais si on veut vraiment rédiger un bloc, on doit indenter (au moins 1 espace) pour exprimer que cette ligne appartient à la séquence du bloc ; et *pareil* pour les suivantes, si besoin...

Attention, *bug* : ce mode de *continuation* sera invoqué même s'il y a *déjà* une instruction après le « : » et pourtant, malgré les apparences, ce bloc ne *peut pas* être continué – la ligne suivante *doit* revenir à l'un des niveaux précédents…

Et voilà pourquoi il *doit* y avoir une ligne *vide* après *chaque* définition de classe dans l'exercice [px30-1] : il faut revenir au top-level, colonne 1, *niveau zéro*, après chaque définition complète.

```
class Mammifere : prop1 = "j'allaite mes petits ;"

class Carnivore (Mammifere) : prop2 = "je me nourris de carne ;"

class Chien (Carnivore) : prop3 = "j'aboie..."

class Chat (Carnivore) : prop3 = "je miaule..."
```

Et c'est aussi pour ça que toute définition de fonction doit être séparée du reste du code par une ligne *vide*.

Ce n'est pas vrai pour une définition de méthode (voyez la définition de la classe Stack, juste avant l'exercice [px30-2]), parce qu'une telle définition étant forcément un bloc de niveau 2 dans une classe, il suffit de revenir au niveau 2 pour démarrer un nouveau bloc, de la même façon qu'on indente de multiples blocs dans une définition de fonction – voyez la définition de explose() à la section 2.4 {itération sur une séquence}.

```
class Stack :
   def __init__(self) : self.stack = []
   def push(self, val) : self.stack.append(val)
   def pop(self) : return self.stack.pop()
   def size(self) : return len(self.stack)
```

NIVEAUX D'INDENTATION

Un bloc indenté peut lui-même contenir d'autres blocs indentés ;

- un bloc se termine lorsque la ligne suivante est revenue à l'indentation précédente ;
- ainsi, dans tous les cas de changement d'indentation (mis à part celui d'un nouveau bloc) l'interprète s'attend à *revenir* à l'une des indentations précédentes, sinon, c'est une erreur de syntaxe ;
- la raison en est que le niveau d'indentation est géré sous la forme d'une pile, ou plus précisément, d'une liste d'associations, mémorisant, pour chaque niveau, la séquence de caractères qui constitue l'indentation ; ce n'est qu'en rencontrant un « : », que l'interprète empile un nouveau niveau ; et si l'indentation change, c'est *forcément* un retour à un niveau *antérieur* : l'indentation *doit* correspondre à l'un des niveaux précédents, faute de quoi, il ne sait pas où raccrocher les wagons...

De l'application de ces principes en mode *interactif*, il vient que la rédaction de *scripts* devrait théoriquement respecter les mêmes règles ; tout bloc commencé en colonne 1 doit être terminé par une ligne vide – et, en un sens, cette ligne *fait partie* du bloc ; alors qu'un bloc commencé avec une indentation *autre* que la 1ère colonne est implicitement terminé lorsque la *ligne suivante* revient à ce même niveau d'indentation.

SCRIPTS

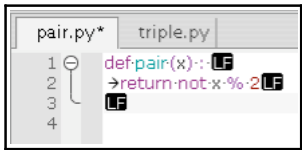
Tout programme qui satisfait ces règles formelles en mode *interactif* passera sans problème en mode *script*. L'inverse n'est pas vrai : un programme *peut* violer les règles en profitant du fait qu'en mode *script* l'interprétation est plus intelligente qu'en mode *interactif* : il est ainsi possible de « couper » une définition avec des lignes *vides* (donc ne contenant que 0x0A) tandis qu'en mode *interactif*, chacune de ces lignes *blanches* devrait être indentée strictement comme la précédente, sous peine de casser immédiatement le bloc en cours de définition.

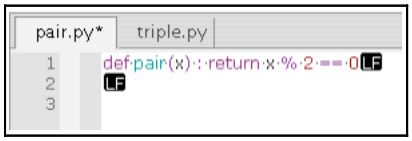
Maintenant, les programmeurs qui estiment pouvoir *contourner* l'expérience interactive doivent, d'une façon ou d'une autre, la *reconstituer* tout en rédigeant leur programme : c'est bien ainsi que procèdent les vétérans, mais ce n'est pas gratuit – ça suppose assez d'expérience pour imaginer, et donc *mentalement émuler*, la réaction de l'interprète à ce qu'ils sont en train de coder, et ceci dans les moindres détails, c'est-à-dire à une tabulation près !

Ainsi, nombreux sont ceux qui arment l'option indentation automatique de leur éditeur de texte : c'est bien commode, puisque cette option indente chaque nouvelle ligne comme la précédente. Et quand il s'agit d'aligner, par exemple, des blocs for, c'est *spontanément* qu'on corrige l'indentation pour mettre en évidence que la ligne suivante ne fait plus partie du bloc… sauf si c'est la *dernière* ligne d'une définition : on garde l'indentation, et on passe à la ligne suivante pour coder autre chose ; et vue d'en haut, la ligne qui précède a l'air *blanche* – mais on a déjà oublié qu'elle n'est pas *vide* !

EXEMPLES

Voici comment la définition précédente de la fonction pair() pourrait apparaître dans un éditeur de programme, sachant que sa deuxième ligne commence par une tabulation :

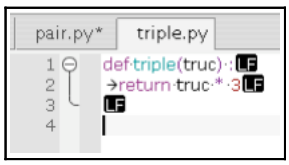




Encore que, peut-être bien que ce serait plus simple de la rédiger sous la forme d'un *one-liner* – comme ça, plus besoin de se poser inutilement des problèmes d'indentation…

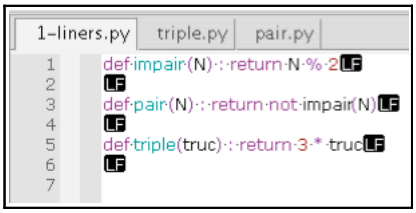
Puisqu'on en parle, il y a d'ailleurs une autre bonne raison de préférer les *one-liners*, et c'est une évidence quand on travaille en interactif : tant que la ligne n'a pas été envoyée, il est possible de la corriger ; après, si on s'aperçoit qu'on a oublié, par exemple, le mot return, c'est trop tard… Sauf que si c'est un *one-liner*, l'utilisation de la touche curseur-haut réaffiche la définition complète, et permet de la corriger. Oh, c'est aussi, bien sûr, possible avec une définition à lignes multiples, mais ça demande un peu plus d'attention…

Un exemple similaire serait celui de la fonction triple() de la section {définition de fonction}, ici aussi, définie sur deux lignes, la 2ème étant tabulée…



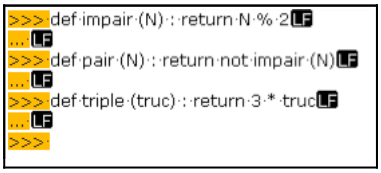
Que pour indenter, on mette des *tabs* ou *espaces* n'a guère d'importance, et c'est comme on veut, même si les statistiques montrent que les programmeurs PYTHON préfèrent utiliser des espaces.

Mon conseil : si possible, faire des *one-liners* ; comme ça, pas de question inutile… Au passage, remarquez que l'alinéa (codé 0x0A) symbolisé par LF, c'est justement ce qu'on obtient en pressant ↵



En d'autres termes, ce qu'il doit y avoir dans le fichier, c'est *exactement* ce qu'on aurait tapé sous l'interprète, y compris les indentations et les alinéas…

D'ailleurs, comparez avec les définitions interactives ci-contre, sachant que c'est l'interprète qui met les >>> et les … (notez l'espace qu'il ajoute en plus), et moi le reste… C'est, *caractère* pour *caractère*, rigoureusement la même chose que dans le fichier !



Il est ainsi tout à fait possible de *développer* une application en mode *interactif*, puis de recopier le texte de la session dans un éditeur de programme ; on y fera alors une *recherche* globale suivie d'une *suppression* de tout ce que l'interprète a lui-même affiché, pour ne garder que le texte réellement entré.

Le script ainsi obtenu pourrait être immédiatement réinjecté dans l'interprète, et devrait alors produire exactement les mêmes effets que précédemment.

Après, si vous vous permettez d'ajouter des lignes vides *gratuites*, des commentaires, ou quoi que ce soit, à vous de savoir ce que vous faites, mais les explications de ces quelques pages devraient vous suffire pour prédire comment ces adjonctions seront interprétées…

9.3 SHELL

Votre premier contact avec le système s'est sans doute produit lorsque vous vous êtes identifié pour y accéder, et vous êtes probablement passé par la fenêtre… je veux parler de la fenêtre de *login*, bien sûr. En effet, la plupart des systèmes modernes sont munis d'une interface graphique, qui occulte derrière des métaphores confortables certains aspects techniques de l'environnement et du système d'exploitation.

L'objectif de ce document est de lever le voile sur l'environnement fondamental, et d'expliquer en partie les concepts qui contribuent à donner à l'environnement sa cohérence. S'il est vrai que la plupart des manipulations d'objets informatiques peuvent se faire à travers des fenêtres, il faut tout de même savoir que derrière ces fenêtres tourne un monde logiciel géré par des programmes qui ont demandé des années d'élaboration et de réglages.

Nous n'irons pas jusqu'à en faire l'historique : il suffit de savoir qu'autrefois, exécuter un programme se faisait en chargeant d'abord ce programme manuellement dans la mémoire de la machine, puis en réglant un registre du processeur sur l'adresse en mémoire où commençait le programme avant d'appuyer sur le bouton « RUN » qui lançait effectivement l'exécution – et encore, je simplifie.

Cette interface rudimentaire a très vite été remplacée par un programme capable de charger d'autres programmes, et dont l'exécution était contrôlée au moyen d'un TÉLÉTYPE, sorte de machine à écrire envoyant des commandes au système d'exploitation.

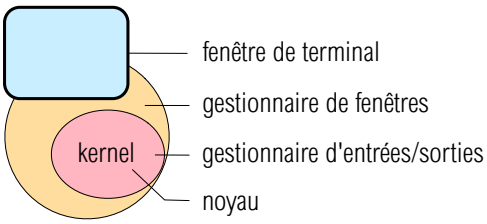
Et c'est resté l'interface de base : un programme qui attend qu'on lui dise ce qu'on veut faire, et qui le fait si on lui en donne l'ordre d'une manière compréhensible — même si l'antique TÉLÉTYPE a finalement été remplacé par le concept plus moderne de TERMINAL vidéo, aussi appelé CONSOLE (ou Konsole pour certains systèmes) qui fait l'écho, sur un écran, des commandes tapées au clavier, et en affiche le résultat.

De la même façon, presser [cliquer sur] un bouton dans une fenêtre déclenche une action pré-programmée, dont les résultats vont peut-être s'afficher dans la fenêtre, ou dans une autre… tout dépend du programme.

Il faut garder à l'esprit que la *fenêtre* et le *bouton* ne sont que l'interface entre l'utilisateur et le système, et que pour certaines opérations, aucune interface n'a encore été programmée, par conséquent, il vous faudra taper vous-même les commandes requises dans ce qui remplace le terminal dans les environnements modernes multi-fenêtrés, et qu'on appelle traditionnellement la fenêtre de terminal, ou console, selon les cultures.

9.3.1 INTERFACES

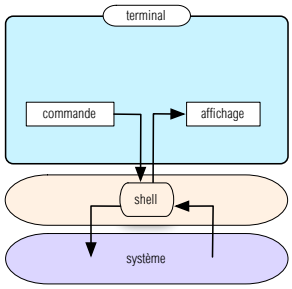
Le schéma ci-contre vous donne une idée sommaire de l'organisation du système d'exploitation et des autres composants. Au plus profond, le kernel, ou *noyau* du système est essentiellement un programme qui tourne sur lui-même en écoutant ce qui se passe dans le monde extérieur en gérant des organes périphériques d'entrée et de sortie.



Au-dessus de lui, le gestionnaire de fenêtres en ouvre au moins une au démarrage du système : la fenêtre de login, qui permet à un utilisateur de s'identifier pour pouvoir accéder à ce système – bien entendu, pour des raisons de sécurité, si vous ne pouvez pas *montrer patte blanche*, vous êtes considéré comme un intrus et l'accès vous est refusé.

Une fois l'accès autorisé, cette fenêtre se referme derrière vous, mais vous pouvez maintenant ouvrir une fenêtre de terminal (de console) pour dialoguer [presque] directement avec le système, et exécuter de nouveaux programmes.

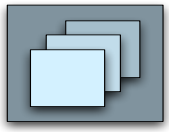
En fait ce dialogue se fait au travers d'un SHELL, un programme (encore) qui sert d'interface logicielle entre l'utilisateur et le système, protégé par cette coquille. Le SHELL est un *interprète* qui exécute à la volée des programmes explicitement appelés comme des commandes tapées au clavier. On en verra de nombreux autres exemples, mais une commande comme pwd provoque l'exécution d'un petit programme dont la mission est d'afficher votre *position* (on y reviendra, mais essayez-donc cette commande dans une fenêtre de terminal, ou console).



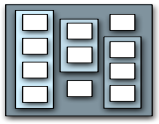
Position ? Encore une métaphore… Disons approximativement que les programmes et les données résident sous la forme de fichiers sur un support magnétique, le disque, et que comme il y en beaucoup, on les organise en dossiers (techniquement, *directories*, autrement dit, *répertoires*). Position signifie donc ici, le dossier dans lequel on travaille.

On pourrait imaginer qu'une simple pile de dossiers devrait suffire, mais comme il y a *vraiment* beaucoup de fichiers, on fait des dossiers dans les dossiers.

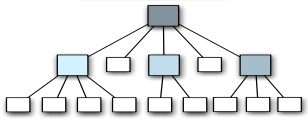
Bien entendu, ceci n'est qu'une représentation abstraite de la structure des données sur le disque. On pourrait visualiser cette structure comme plein de petites boîtes (les *dossiers*) dans une grande boîte (le disque).



Et comme, dans n'importe quel dossier, on peut créer autant de sous-dossiers qu'on veut, ça finit par rassembler à des casiers à tiroirs.



Une telle structure peut aussi être représentée sous la forme d'un graphe branchant appelé *arborescence*, ou *arbre* ; dans le graphe ci-contre, le nœud-origine, tout en haut, est appelée racine ou root, et les branches expriment la filiation d'un nœud par rapport à un autre.



Remarquez que seules les boîtes colorées ont des branches : ce sont des dossiers, ou directories, alors que les boîtes claires (aussi appelées *feuilles* de l'arbre) sont des fichiers contenant des données. En fait, un dossier est un fichier qui répertorie des fichiers.

Dans cette représentation hiérarchisée, tout élément, fichier ou dossier, a un ascendant unique : il n'y a donc jamais, partant du sommet, qu'un seul chemin qui mène à un élément donné ; à tout moment de l'utilisation du terminal, le programmeur se situe quelque part dans cette arborescence, dans ce qu'on appelle le *dossier de travail* (WORKING DIRECTORY), et c'est exactement ce que répond la commande pwd : le chemin, partant du sommet, pour parvenir en ce point.

- Dans une telle organisation hiérarchique, chaque nœud (ou point de ramification) porte un nom. Dans un système comme unix, un nom peut être fait de n'importe quels caractères (chiffres, lettres, etc.), et les minuscules ne sont pas assimilées aux majuscules correspondantes : le fichier “f”, par exemple, est distinct du fichier “F”.

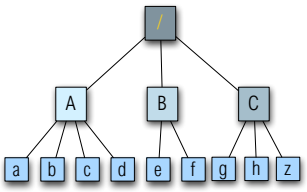
- Chaque dossier peut répertorier (on dira volontiers *contenir*) des fichiers et des dossiers, mais un dossier ne peut répertorier deux objets de même nom. En revanche, rien ne s'oppose à ce que deux dossiers différents contiennent des dossiers ou fichiers de même nom ; c'est d'ailleurs ce qui arriverait si on recopiait une branche entière dans une autre…

9.3.2 CHEMIN D'ACCÈS À UN FICHIER : MKDIR, CD

On peut naviguer, se déplacer d'un dossier à un autre avec la commande cd, abrégé de *change directory* : cd .. pour dire « *remonte au niveau supérieur* », et cd truc pour « *descend dans le dossier truc* ».

- La première commande, cd .., est forcément non-ambiguë, puisque dans une organisation arborescente, il n'y a jamais qu'un supérieur hiérarchique. Par convention, le double point représente le dossier contenant le dossier de travail, lui-même représenté par le simple point, dont en verra

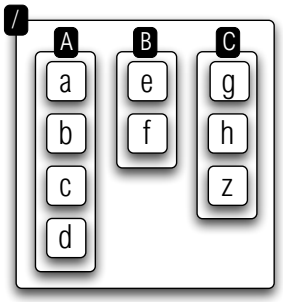
l'intérêt plus tard : cd . est syntaxiquement possible, mais n'est pas vraiment d'une grande utilité…
- La deuxième commande, cd truc, ne peut être exécutée que si le dossier “truc” est directement visible depuis la position courante (de la même façon, dans une fenêtre représentant les dossiers par des icônes, on ne peut double-cliquer, pour les ouvrir, que les dossiers visibles).



Imaginons une structure arborescente comme ci-contre, où chaque “boîte” représente un dossier, ou répertoire (ou *directory*). On peut créer une telle structure en utilisant la commande mkdir (*make directory*) suivie du nom du directory (*dossier*) qu'on veut créer. Par exemple, depuis le directory B, les commandes mkdir e et mkdir f ont fabriqué les deux sous-dossiers e et f.

Note : nous utiliserons désormais indifféremment le terme technique *directory* (ou son équivalent français, *répertoire*), ou son équivalent [grand public] non technique, *dossier*, sachant qu'ils désignent exactement la même chose.

Remarquez que le nœud à l'origine de l'arborescence (tout en haut sur le diagramme) porte un nom un peu spécial : on le désigne par le symbole “/”. En fait, il ne porte pas vraiment de nom : ce symbole “/” représente (de façon absolue) le nœud-racine de l'arborescence, et permet de spécifier de façon non-ambiguë un *chemin* dans cette arborescence. Un chemin *absolu* est la spécification d'une *séquence* de dossiers à visiter, partant de la racine, pour parvenir en un autre point de l'arbre : par exemple, “/A/b” indique une certaine route, alors que “/B/f” en indique une autre. Notez au passage que chaque nom est toujours séparé du précédent par un symbole “/”.



Mais pourquoi se déplacer ? Parce que les commandes sont exécutées là où on les commande : si je veux manipuler les données qui se trouvent dans le directory truc, il faut que je le fasse depuis le directory truc, ou alors il faut une petite gymnastique qu'on décrira plus tard {cf. copie de fichiers : cp}.

La commande cd s'utilise en spécifiant où on veut se déplacer : par exemple, en partant de la racine, je pourrais commander cd A, ou cd B, ou cd C – il n'y a pas d'autre directory dans lequel je puisse “descendre” directement. Arrivé dans A, je pourrais alors commander cd a ou cd b… Toujours en partant de la racine, on pourrait aussi spécifier directement un chemin complexe en commandant cd C/z ou cd B/e…

La commande pwd, vue plus haut, aurait dû s'appeler vousêtesici (ou vei) : elle est l'abrégé de *print working directory* (en français, *affiche dossier de travail*). Ainsi, à tout instant, pwd permet de savoir où on se situe dans la hiérarchie.

Une fois arrivé dans le dossier e du dossier B, la commande cd .. me ferait remonter au niveau supérieur, c'est-à-dire dans le dossier B. Ou bien je pourrais commander directement cd ../f, ce qui aura pour effet de (1) remonter d'un cran, puis (2) de descendre dans le dossier f du B. Et de la même façon, j'aurais pu commander cd ../../C/h, ce qui me ferait remonter de deux niveaux avant de redescendre dans le dossier h du dossier C. Mais pourquoi “..” fait-il remonter d'un cran ? Parce que, par design, tout dossier sait qui le contient sans qu'il soit besoin de le nommer : une spécification telle que “..” est toujours valide et désigne le *contenant* (c'est dû à la conception même du système de fichiers, et nous y reviendrons).

Je sais bien que tout ceci n'a pas l'air pas trop *fun*, mais c'est juste pour expliquer : on verra plus tard que les spécifications de fichiers peuvent servir à bien d'autres choses que la navigation. L'important ici c'est de comprendre que n'importe quel dossier ou fichier de l'arborescence peut être désigné de façon univoque, que ce soit pour s'y déplacer ou pour le manipuler — avec comme conséquence qu'un fichier peut être manipulé à distance, de n'importe quel point de l'arborescence.

La raison d'être d'un tel mécanisme de commandes est que toutes les opérations sont programmables, et que les applications munies d'une interface graphique utilisent en fait ces commandes pour naviguer dans la hiérarchie et en manipuler les objets, dossiers ou fichiers.

Pour récapituler :
- mkdir abc crée un directory nommé abc, là où la commande est exécutée

- cd abc permet alors de “descendre” dans le directory abc
- pwd affiche maintenant le chemin complet, depuis la racine, pour parvenir en ce point abc

terminologie : ici, remarquons que certaines commandes *doivent* être suivies de quelque chose, et d'autres non. Ce *quelque chose* est appelé *argument* de la commande : en effet, les commandes sont des *fonctions*, au sens mathématique du terme, c'est-à-dire qu'elles effectuent une *opération*, ont un *effet*, et retournent une *valeur* qui est fonction de l'argument qu'on leur a donné, qui lui-même dépend du contexte d'exécution. En ce sens, la fonction pwd n'est pas une exception : elle n'a simplement pas besoin d'argument parce qu'elle opère directement à partir du contexte courant (en fait, elle peut effectivement prendre un argument qui modifie son fonctionnement, mais n'anticipons pas). Et puisqu'on parle d'*opérer*, remarquez qu'il y a la même relation entre un *opérateur* et ses *opérandes* qu'entre une *fonction* et ses *arguments*…

valeur, effet : dans le cas des commandes SHELL, la *valeur* retournée par la fonction n'apparaît jamais, par contre, son *effet* est souvent évident, mais pas toujours. Les commandes mkdir ou cd, par exemple, n'ont pas d'effet directement visible, puisque si tout se passe bien, elles opèrent silencieusement ; mais en cas d'erreur, elles afficheraient un bref message pour expliquer pourquoi elles ont failli, et ce serait là leur seul effet directement visible. En principe, une fonction retourne *toujours* une valeur mais n'a pas forcément d'effet (on y reviendra).

9.3.3 LES COMMANDES INDISPENSABLES

Dire d'une commande qu'elle est *indispensable* dépend en grande partie de qui fait quoi avec le SHELL : cependant, il y en a une petite poignée qui se distingue du lot…

LISTE DES FICHIERS : LS

L'arborescence de fichiers est ce qu'on appelle une structure de données. Et puisque cette structure organise des fichiers, on a souvent besoin de lister le contenu d'un dossier. Tellement souvent que les interfaces graphiques listent le contenu du dossier courant sans même qu'on ait à leur demander : si on est venu là, c'est bien pour y faire quelque chose, donc autant savoir où est-ce qu'on met les pieds. La commande SHELL pour lister le contenu d'un dossier s'appelle ls (abrégé de *list*) et peut s'utiliser sans argument aucun ; dans ce cas elle affiche simplement la liste des noms de fichiers et dossiers répertoriés.

On peut aussi lui dire d'afficher selon d'autres points de vue. Ainsi, comme dans l'exemple ci-dessous, ajouter l'option l (c'est-à-dire, *elle*) en argument affichera les mêmes noms, mais en format *long*, c'est-à-dire que ça va faire apparaître, pour chaque fichier, un certain nombre de propriétés qui ne seraient pas apparentes en format *normal*. Voici, par exemple, comment m'apparaît le directory /usr/share/X11 quand listé avec les options a, l et F :

```
ls -alF /usr/share/X11
total 92
drwxr-xr-x   4 root root  4096 2009-10-28 21:58 ./
drwxr-xr-x 303 root root 12288 2009-11-23 10:57 ../
drwxr-xr-x  62 root root  4096 2009-10-28 21:56 locale/
lrwxrwxrwx   1 root root    16 2009-11-04 05:33 rgb.txt -> /etc/X11/rgb.txt
-rw-r--r--   1 root root 41481 2009-07-28 00:11 XErrorDB
drwxr-xr-x  10 root root  4096 2009-10-28 21:55 xkb/
-rw-r--r--   1 root root  8982 2009-07-28 00:11 XKeysymDB
-rw-r--r--   1 root root  8305 2009-07-28 18:26 xman.help
```

Chaque ligne correspond à un fichier, et chaque colonne représente une propriété.

1. une séquence de caractères qui décrit ce qu'on appelle le *mode d'accès* au fichier
2. le nombre de *liens* – sans intérêt pour le moment
3. le nom de l'utilisateur auquel ce fichier appartient
4. le nom du groupe qui aurait aussi un accès privilégié à ce fichier
5. la taille du fichier (en octets)
6. la date de dernière modification
7. le nom du fichier

Ici, trois remarques :

- un dossier est un fichier – un fichier particulier qui peut répertorier d'autres fichiers ;
- en haut de la liste apparaissent deux dossiers très spéciaux : le premier, désigné par un point, est une

référence au dossier lui-même (on en verra l'intérêt plus tard) ; le deuxième est une référence vers le dossier conteneur, ce qui éclaire un peu ce qu'on a vu plus haut à propos de la commande cd .. – remarquons que de la même façon, cd . serait aussi une commande valide, encore que sans intérêt aucun ;
- root est le super-utilisateur, celui qui a tous les droits pour installer puis administrer le système ; de toute évidence, ce dossier a été créé en 2009 au moment de l'installation du système et a été protégé contre les manipulations intempestives d'utilisateurs susceptibles de nuire au bon fonctionnement de l'ensemble…

à propos du *mode* d'accès :
- le tout premier caractère est ici un "d", un "l", ou un "-" ; le "d" dénote un *directory*, donc potentiellement un argument pour la commande cd : de façon redondante (du fait de l'option F), toutes les lignes commençant par un "d" se terminent aussi par un "/", une manière très visuelle de repérer rapidement les dossiers dans une telle liste… le "l" caractérise un lien, une référence à un fichier situé quelque part ailleurs ; quant au "-", il dénote un fichier normal, ici un texte probablement, que je n'ai d'ailleurs jamais lu ;
- suivent immédiatement trois séquences de "rwx" (sauf que certains "w" sont remplacés par un "-") : ce sont les permissions d'accès, ou privilèges de lecture, READ, écriture, WRITE, et exécution – qui, pour un dossier, s'interprète comme le droit d'aller voir dedans ; ces trois séquences rendent compte respectivement des droits d'accès du propriétaire du fichier, des autres membres de son groupe, et enfin des autres utilisateurs non membres, comme moi ; remarquons que moi, je n'ai pas le droit de modifier – ni d'ailleurs d'exécuter, si ça avait un sens – les trois fichiers texte (lien non compris), ce qui signifie que ① ils ne sont pas du code exécutable, et ② je ne peux pas les détruire – détruire étant une façon radicale de modifier.

La commande ls dispose de plein d'options pour configurer l'affichage, le trier… Elle offre aussi la possibilité de spécifier les fichiers qu'on veut lister, à l'exclusion des autres : ici, on aurait pu spécifier ls -d ?i*, pour obtenir la liste de tous les fichiers ou dossiers dont le nom comporte un *i* en 2e position ; "?" et "*" sont ce qu'on appelle des *wildcards* (ou *jokers* en français – et certains les appellent *des méta-caractères*) :

- ? représente n'importe quel caractère, en un seul exemplaire
- * représente n'importe quelle séquence de n'importe quels caractères

On peut ainsi filtrer une liste pour ne sélectionner que les noms des fichiers intéressants…

HISTORIQUE DES COMMANDES

L'éditeur de ligne, incorporé au SHELL, permet de rappeler l'historique des commandes ; les touches de curseur haut et bas retrouvent les dernières commandes, et les touches de curseur gauche et droit permettent de se placer précisément à l'endroit qu'on veut corriger.

Pour ceux qui utilisent le shell bash, l'historique des commandes est conservé dans un fichier nommé ".bash_history" ; dans la liste des fichiers, tous ceux dont le nom commence par un "." sont invisibles, sauf si on utilise l'option a, c'est-à-dire *all*, de la commande ls (voir ci-dessus).

> Le reste de cette section passe en revue un certain nombre de commandes dont nous pouvons nous dispenser, pour le moment, de connaître les détails, d'autant su'on peut, à tout moment, consulter le manuel en ligne avec man. Mais ne perdez pas de vue que, tôt ou tard, on aura besoin faire des manipulations dans le terminal : il sera toujours temps, alors, de reprendre ce chapitre, et d'en assimiler le contenu, au fur et à mesure des besoins.

COPIE DE FICHIERS : CP

La commande cp (abrégé pour *copy*) prend au moins deux arguments : la *source* et la *destination* (il faut bien sûr avoir le droit d'accès en *lecture* pour la source et le droit d'accès en *écriture* pour la destination). On peut copier plusieurs fichiers en en donnant la liste ou en les spécifiant avec des *wildcards*. De plus, je peux utiliser la spécification ".", vue plus haut, pour dire que c'est là où je me trouve (pwd) que je veux copier. On comprend maintenant l'intérêt d'accéder à des fichiers en spécifiant leur chemin d'accès : je vais ainsi pouvoir copier chez moi un fichier qui se trouve ailleurs, dans une tout autre branche, par exemple celle d'un

autre utilisateur qui m'autorise à regarder le code source de son programme.

DÉPLACEMENT DE FICHIERS : MV

La commande mv (abrégé pour *move*) prend au moins deux arguments : la *source* et la *destination* (il faut bien sûr avoir le droit d'accès en *lecture* pour la source, et le droit d'accès en *écriture* pour la destination). Cette commande accepte les mêmes spécifications de fichiers que cp, mais déplace littéralement les fichiers, sans avoir à les recopier avant d'effacer les originaux. De surcroît, si cette fonction est utilisée sans spécification de déplacement, elle a pour effet de *renommer* les fichiers : mv truc machin changera le nom de truc en machin…

DESTRUCTION DE FICHIERS : RM

La commande rm (abrégé pour *remove*) accepte une liste de fichiers (y compris par *wildcards*) et les retire (définitivement) du système de fichier : cette opération est irrémédiable, et on comprend pourquoi le système se protège en retirant le droit d'accès en écriture aux crétins qui farfouillent.

DESTRUCTION DE DOSSIERS : RMDIR

La commande rmdir (abrégé pour *remove directory*) n'est applicable que si le dossier est vide ; si, par exemple, le dossier w contient un dossier (vide) v, il faudra commander dans l'ordre rmdir w/v w, pour que v soit détruit avant que la commande n'essaye de détruire w…

DOCUMENTATION

Toutes ces commandes sont documentées à la fois de façon interne et externe : *interne* parce qu'une erreur de syntaxe provoque un message qui rappelle quels sont les arguments attendus, *externe* parce que *toutes* les commandes figurent dans un *manuel* en ligne qu'on peut consulter directement grâce à la commande man. Par exemple, man ls affiche toutes les options acceptées par cette commande, et man rmdir vous explique comment détruire un dossier *même* s'il n'est pas vide.

Et pour les plus distraits, man man vous rappelle, bien sûr, comment utiliser man… On peut aussi consulter la documentation en ligne sur le WEB : <http://linux.die.net/man/>.

9.3.4 LES COMMANDES UTILES

env, set, printenv : dans tous les systèmes unix, chaque processus a son propre jeu de variables d'environnement (lorsqu'un processus est créé, il hérite par défaut de l'environnement du processus-père), parmi lesquelles des informations comme le nom de l'utilisateur, ou le type de machine… Un processus peut se servir de ces valeurs pour ajuster son comportement (c'est le cas notamment du compilateur gcc). La commande set est la plus générale (elle donne la liste de toutes les variables et de leur valeur) alors que env n'affiche que celles qui sont exportées (voir man env, man printenv, et man bash). Exemples de variables :

- PATH : liste des chemins d'accès aux commandes utilisables à partir du SHELL (détaillée plus loin)
- PS1 : invite pour chaque ligne de commande ; par exemple, on redéfinira PS1='\W : ' pour que le directory courant soit affiché à chaque nouvelle ligne de commande (voir man bash).

Quelques commandes :

- echo : pour afficher une valeur de l'environnement ; par exemple echo $PATH
- alias : pour définir des synonymes, par exemple : alias l='ls -alF' (voir man bash).
- chmod : redéfinit le mode d'accès à un fichier (ou une liste de) ; par exemple chmod u+x machin modifie l'accès à machin pour qu'il soit exécutable (si ça a un sens) ; le mode d'accès est la première information listée par ls en format long.

Notion de script exécutable : on peut mettre une commande (ou une *séquence* de commandes) dans un fichier, et le considérer comme une nouvelle commande ; il suffit d'en changer le mode d'accès. Par exemple : si j'ai mis dans le fichier *machin* la commande chmod a+x $1, je peux le rendre exécutable avec chmod +x machin, et l'utiliser désormais pour rendre exécutable un script *truc* en tapant machin truc… Remarquons que $1 représente le 2e mot de la ligne de commande (on y reviendra).

Bien entendu, vous pouvez utiliser man pour une documentation plus complète sur ces commandes.

9.3.5 LE SHELL ET SES EXTENSIONS

Comme on l'a vu plus haut, le *SHELL* est un programme lancé automatiquement à l'ouverture de la fenêtre de terminal, ou de la console ; "SHELL" est ici un nom générique : bash est le SHELL par défaut, telles que sont configurées beaucoup de machines unix. Et comme la plupart des programmes, il contient des définitions de variables et de fonctions :

- les variables constituent ce qu'on appelle l'environnement ;
- les fonctions sont appelées « commandes ».

Les variables du SHELL servent à contrôler certains aspects de son comportement ; par exemple, la variable PS1 (abrégé de *prompt string 1*) définit l'*invite de commande* qui apparaît au début de chaque nouvelle ligne de commande ; la variable PATH définit une liste de "dossiers" dans lesquels bash cherche le programme dont on a tapé le nom en tant que commande, autrement dit comme premier mot de la ligne de commande. Ces variables peuvent être listées de deux manières :

- la commande set affiche le *nom* et la *valeur* de toutes les variables
- la commande echo suivie du *nom* d'une variable (précédé du symbole $) en affiche directement la valeur – le $ est ici un opérateur d'évaluation : echo PATH afficherait bêtement le texte "PATH", alors que echo $PATH affiche la *valeur* de la variable PATH.

Les commandes sont en fait de deux sortes :

- les commandes *internes* sont des fonctions définies dans le SHELL proprement dit, et ne dépendent donc pas d'une localisation par la variable PATH ; par exemple : cd, pwd, set ou echo sont des commandes internes ;
- les commandes *externes* sont des programmes exécutables autonomes, résidant généralement dans le dossier usr (*unix system resources*) ; on peut donc les reprogrammer indépendamment, ou en ajouter de nouvelles.

Le terme « commande » ne doit pas nous faire perdre de vue qu'on est dans un contexte *opérationnel*, donc que les commandes sont interprétées comme des *opérateurs* – autrement dit s'appliquent à des *opérandes*, et de ce fait constituent des *expressions* qui prennent alors une valeur dans ce contexte ; voilà pourquoi echo PATH ne fait pas la même chose que echo $PATH.

La plupart des distributions sont livrées avec plusieurs programmes de SHELL, comme sh, le SHELL original, tcsh, syntaxiquement plus proche de ANSI-C, ou zsh, pour les amateurs. Même si votre terminal a été ouvert avec le bash comme SHELL par défaut, vous pouvez, à tout moment, invoquer tcsh ou zsh, en les appelant par leur nom ; à partir de ce moment, c'est ce SHELL qui interprétera toutes les commandes entrées au clavier… Pour ce qui est de l'interprétation des scripts, voir plus loin : {*spécifier l'interprète*}.

ENTRÉES/SORTIES : LES OPÉRATEURS DE REDIRECTION

Comme on l'a vu plus haut, une commande comme set affiche quelque chose sur l'écran du terminal, de la console : ce quelque chose est appelé (de l'anglais *output*) la *sortie* de set. L'intéressant, ici, c'est que cette sortie peut être redirigée : une expression comme

```
set > variables
```

n'affiche rien sur l'écran, parce que le texte qu'elle aurait affiché a été redirigé, et constitue maintenant le contenu du fichier variables… La preuve en est que si maintenant on commande :

```
cat variables
```

ce que cat affichera sur l'écran (le contenu du fichier variables) est exactement ce qu'aurait affiché la commande set si elle n'avait pas été redirigée. Et en commandant :

```
cat variables > encore
```

on demande en fait à cat de recopier dans le fichier encore ce qu'il a lu dans le fichier variables, réalisant ainsi une copie de ce fichier.

Remarquons que de genre de redirection crée automatiquement un nouveau fichier, c'est-à-dire que même s'il existait déjà un fichier de ce nom, la nouvelle information qu'on y écrit supplante ce qu'il y avait avant.

Mais il est possible de contourner ce comportement :

```
set > variables
env >> variables
```

parce que la redirection par >> ajoute les nouvelles données à la fin du fichier existant…

On peut aussi rediriger l'entrée avec l'opérateur <, ce qui force le programme à prendre ses données d'entrée dans un fichier. Par exemple, le programme date s'attend à recevoir une spécification de format telle que :

```
date "+Date : %d/%m/%y%nHeure : %H:%M:%S"
```

Supposons que ce texte, "+Date : %d/%m/%y%nHeure : %H:%M:%S" soit justement le contenu d'un fichier appelé mon-format. La commande :

```
date < mon-format
```

lira ce fichier et s'exécutera exactement comme si on avait tapé tout sur la ligne de commande.

Autre exemple typique, celui de l'utilisation de la commande tr pour convertir des données textuelles à la volée :

```
tr -d '\r' < ms-windows.txt > unix.texte
```

qui supprime, dans un *texte* en provenance d'une machine M$W, tous les caractères de retour-chariot superflus, ne gardant que les sauts de lignes attendus dans un fichier au standard U.

L'OPÉRATEUR DE RECONDUCTION

Quant au pipeline |, ou opérateur de reconduction, il permet de faire l'économie de fichiers temporaires intermédiaires : il effectue une capture de la sortie d'un programme et la reconduit en tant que *données d'entrée* d'un second programme ; un exemple banal serait :

```
ls -alF | less
```

quand l'affichage (en mode long) du programme ls serait tellement volumineux qu'il prendrait plus d'un écran ; en le reconduisant en entrée du programme more ou less, celui-ci va automatiquement le segmenter en autant d'écrans que nécessaire, attendant une intervention manuelle (barre d'espace, curseur, …) pour passer de l'un à l'autre.

Pour plus ample documentation, consulter le bash reference manual.[50]

Les implications pratiques de ce comportement du SHELL seront étudiées plus en détail quand il s'agira d'écrire des programmes utilisables comme des commandes du SHELL, typiquement des programmes en langage C {cf. cours éléments de systèmes d'exploitation}.

SCRIPTS

Cependant, il est possible d'écrire des programmes en d'autres langages que ANSI-C : le SHELL est lui même un interprète de commandes, donc un langage. Si un programme est défini comme une séquence de commandes, alors ceci est un programme :

```
date +"%H:%M:%S"
pwd
```

Et si je mets ce texte dans un fichier appelé zou, je peux l'exécuter en disant que je veux l'interpréter, par exemple, avec le SHELL sh – ou d'ailleurs n'importe quel autre :

```
sh zou
```

Mieux encore : si je marque ce fichier comme exécutable – avec la commande chmod vue plus haut, je peux directement taper son nom sur la ligne de commande et obtenir son exécution :

50 http://www.gnu.org/software/bash/manual/bashref.html

```
chmod u+x zou
zou
```

Un tel fichier est appelé « script », impliquant que le *code source* doit être interprété – par opposition aux commandes en langage machine, qui, elles, sont déjà traduites par un *compilateur* sous une forme directement exécutable par le processeur de la machine. Il existe des dizaines de langages de script, à commencer par les différents programmes de SHELL (SH, TCSH, BASH, ZSH, …), mais aussi AWK, BISTRO, C#, JAVASCRIPT, MATLAB, PERL, PHP, PYTHON, RUBY, ou encore SMALLTALK, pour ne citer que les dix plus connus (google scripting languages).

Ce qui caractérise les langages de script, c'est leur haut niveau de granularité ; dans le SHELL bash, par exemple, la plupart des opérations disponibles sont en fait des programmes complets, déjà éprouvés et rodés ; et ceci présente un avantage considérable : un tel langage est conçu pour une utilisation interactive, ce qui facilite et accélère sa mise au point – on peut vérifier instantanément le bon fonctionnement du programme à chaque pas de sa réalisation ; le programme est donc à la fois plus *facile* à écrire et plus *rapide* à mettre au point.

Par contraste avec les langages de programmation de bas niveau (comme le langage ANSI-C) où le programmeur doit s'occuper de la gestion de la mémoire et des variables en même temps qu'il doit programmer tous les détails du traitement de l'information, l'inconvénient des langages de script n'apparaît qu'en termes de *souplesse* et de *performance* : effectivement, les programmes *compilés* sont, d'une manière générale, plus flexibles, plus rapides à l'exécution et moins gourmands en mémoire ; cependant, dans la programmation par scripts avec les machines modernes, compte tenu du coût du matériel et de celui de la main-d'œuvre, l'inconvénient du *run-time* est largement compensé par l'économie du *write-time*…

PORTABILITÉ DES SCRIPTS

Dans un environnement unix, le suffixe – ou extension – d'un fichier n'est PAS un critère pour déterminer son type : c'est simplement une partie du nom de ce fichier – sauf si le programme qui le manipule y voit, lui, une intention particulière…

Il n'y a donc rien, en *apparence*, qui distingue un script d'un exécutable compilé ou d'une poésie en alexandrins, à moins de regarder dedans : il existe une commande, file, à laquelle on donne comme argument un nom de fichier, et qui dit ce que ça peut être au vu de la signature (interne) du fichier.

Si on sait que le fichier x est un script, on peut demander à l'exécuter en invoquant explicitement le nom de l'interprète pour lequel ce script a été écrit ; par exemple :

```
bash x
```

Cependant, comme on l'a vu précédemment, un script peut être marqué *exécutable*, comme le sont d'ailleurs les programmes compilés, avec cette différence qu'un script est par défaut, dans ce cas, interprété par le SHELL actif ; le problème, c'est qu'il existe différents SHELLS – qui présentent des différences de syntaxe – ce qui fait qu'il se peut qu'un script particulier ne fonctionne pas comme prévu si l'environnement n'est pas celui pour lequel il a été conçu…

SPÉCIFIER L'INTERPRÈTE

Pour contourner ce problème, il est convenu que si la toute première ligne du script commence par les caractères "#!", le SHELL actif appelle automatiquement l'interprète dont l'adresse *doit* être donnée sur cette même ligne. Par exemple, voici un script qui ne tourne qu'avec *tcsh*, parce que la syntaxe du if n'est pas la même qu'en *bash* :

```
#! /bin/tcsh
gcc -Wall $1.c -o $1
if ($? != 0) exit
echo executable \"$1\" ok
```

Maintenant, le problème est que tcsh ne réside pas forcément dans /bin sur tous les systèmes : certains le mettent dans /usr/local/bin, par exemple. Pour contourner ce deuxième problème, une commande spéciale est installée (en principe) dans /usr/bin ; elle s'appelle env, et son boulot est de « scanner » la variable d'environnement PATH pour trouver et invoquer l'interprète dont le nom figure sur la première ligne du script, ligne rédigée comme suit :

```
#! /usr/bin/env tcsh
```

Mais ce serait trop simple, car *rien* n'oblige *personne* à placer env dans /usr/bin ; la preuve en est que dans l'environnement MANDRIVA, par exemple, c'est dans /bin que env est installé, mais pour le savoir, il faut utiliser la commande which :

```
which env
```

qui affichera /bin – du coup, la première ligne du script ci-dessus devrait donc être :

```
#! /bin/env tcsh
```

Et puisqu'on parle de syntaxe, voici comment la même chose se dirait avec le SHELL bash :

```
#! /bin/env bash
gcc -Wall $1.c -o $1
[[ $? != 0 ]] && exit
echo executable \"$1\" ok
```

Remarquez qu'il faut *impérativement* laisser un espace après le 2e crochet ouvrant [, sous peine d'erreur, alors qu'il n'y en a pas besoin après la parenthèse ouvrante en tcsh.

Il y a, quelque part, un fichier qui recèle la liste des *SHELLS* installés dans votre système ; il s'appelle SHELLS, et se trouve dans le dossier de configuration, nommé /etc :

```
cat /etc/shells
```

serait donc la commande pour afficher son contenu à l'écran.

NB : ce problème a fait l'objet de tellement de discussions que nous avons résolu de développer une section dédiée, la 9.4 ci-après… maintenant, il faudrait tout récrire pour consolider la réflexion ! Cependant, un SHELL est un langage de programmation, et en tant que tel, il exigerait un cours de 200 pages à lui tout seul ; on peut toujours télécharger ADVANCED BASH-SCRIPTING GUIDE <http://www.tldp.org/LDP/abs/abs-guide.pdf>, en gardant à l'esprit que c'est quand même un volume de 758 pages…

CHEMINS D'ACCÈS

La variable PATH, dont vous pourriez voir la valeur en demandant

```
echo $PATH
```

est une liste d'adresses, de *chemins absolus* vers les répertoires qui contiennent les programmes qu'on veut pouvoir invoquer directement depuis le SHELL ; par exemple, chez moi, sa valeur est :

```
/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin
```

Ici, le symbole ':' sert à séparer les éléments de la liste, ce qui signifie que, pour trouver une commande, le SHELL (en fait, le programme env) cherchera successivement dans les six dossiers :

```
/usr/local/bin
/bin
/usr/bin
/sbin
/usr/sbin
/usr/local/bin
```

jusqu'à ce qu'il trouve le programme invoqué – et s'il ne le trouve pas, il affichera l'infâme message "*command not found*".

L'ordre des éléments de cette liste est intentionnel : ne le modifiez que si vous êtes sûr de ce que vous faites. Si vous tenez à rajouter un nouveau chemin, par exemple, vers le directory courant, rajoutez-le à la fin de la liste :

```
PATH=$PATH:./
# → /usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin:./
```

Remarquez la forme du chemin que j'ai rajouté ; cette forme, le point, qui veut dire « ici même », m'aurait de toute façon permis d'invoquer une commande strictement locale, par exemple un script (marqué *exécutable*) que je suis en train de tester :

```
./teste-moi
```

Encore que, comme on l'a vu plus haut, je pourrais l'exécuter tout pareil (sans même avoir à le marquer *exécutable*, ni à spécifier quoi que ce soit sur sa première ligne) en invoquant directement l'interprète :

```
bash teste-moi
```

NOTES

Il existe en fait plusieurs centaines de commandes, dont il n'est pas question de faire ici le catalogue :

- certaines, comme cd ou pwd, sont dites intrinsèques, parce que prédéfinies dans le langage de SHELL lui-même ;
- mais la plupart des commandes sont extrinsèques : ce sont, en fait, des « extensions » du SHELL, sous la forme de programmes autonomes qui résident sur le disque dans un espace réservé – espace défini en terme de *chemins d'accès* par la variable d'environnement path : vous pouvez ainsi rajouter les commandes que vous voulez…

On se contentera donc d'un bref survol de quelques commandes : simple aperçu vu d'avion des ressources d'un système *à la unix*.

LE TEXTE

- gedit : *gnome editor* – l'éditeur de texte dans l'environnement GNOME
- ed : *edit* – un éditeur de ligne adapté pour du texte
- vi : *visual interface* (to ed) – l'éditeur de texte fourni en standard
- emacs : *editor macros* – éditeur (en évolution depuis 1970) avec interprète LISP intégré
- gcc : *gnu compiler collection* – convertit du code-source en code exécutable

LES FICHIERS

- grep : *global regular expression print*
- locate : *localise*, recherche un fichier sur le disque (voir aussi find)
- which : *quel*, recherche un fichier dans la liste $PATH
- df : *disk free* – espace libre sur disque

LES PROCESSUS

- ps : process status – liste le n° d'identification des processus en cours
- kill : stoppe un processus

LE RÉSEAU

- netstat : NETWORK STATUS
- ping : renvoie l'écho d'une autre machine du réseau
- ssh : SECURE SHELL LOGIN – permet la connexion à une autre machine

Pour plus ample documentation, consulter le man de chaque commande. Vous pouvez également utiliser les ressources du www pour collecter les documents qui vous intéressent ; à titre indicatif :

référence des commandes linux (PDF, 1527 pages)
http://fuerteventura-losaloes-apartamentos.com/Linux%20Command%20.pdf

description de 687 commandes, par ordre alphabétique (HTML)
http://www.oreillynet.com/linux/cmd/

man pages au format HTML
http://linux.die.net/man/

9.4 PROGRAMMATION SHELL

Cette section n'a pas l'ambition de vous apprendre à programmer en SHELL : il faudrait, pour ça, un manuel entier… Nous ne ferons ici que reprendre quelques problèmes qui se posent au débutant, en particulier, les problèmes qui ont été soumis sur le forum.

Le premier est celui de la création d'une archive en vue de soumettre, en un seul paquet bien ficelé, une série de 3 exercices dont vous avez conservé la solution dans les fichiers suivants :

```
~/python sessions/px99-1.txt
~/python sessions/px99-2.txt
~/python sessions/px99-3.txt
```

autrement dit, il y a, dans votre dossier personnel, un dossier nommé 'python sessions' qui contient les fichiers qu'il vous intéresse d'emballer proprement.

Si vous deviez le faire manuellement, il suffirait de commander :

```
zip 99 *
  adding: px99-1.txt (deflated 49%)
  adding: px99-2.txt (deflated 52%)
  adding: px99-3.txt (deflated 62%)
```

qui compresse puis ramasse tous ces fichiers dans une archive nommée 99.zip.

Un problème se poserait s'il y avait, dans ce dossier, d'autres fichiers que les trois qui nous intéressent : le symbole * s'expanse automatiquement[51] en une liste de tous les fichiers qui résident là, donc ramasse tout ce qui traîne ; dans ce cas, on peut contraindre l'expansion :

```
zip 99 px99-?.txt
```

garantit de ne s'occuper que des fichiers dont le nom commence par px99- suivi d'un caractère quelconque et finissant par .txt : c'est à ça que sert le ? ; une autre manière de contraindre la liste expansée serait d'utiliser une spécification comme px99-[1-3].txt : le caractère possible en cette position du nom doit alors appartenir à l'ensemble des chiffres de 1 à 3, ce qui exclut le fichier px99-4.txt, de même que le px99-a.txt. Tout se passe donc comme si on avait tapé :

```
zip 99 px99-1.txt px99-2.txt px99-3.txt
```

Mais quand je dis « *emballé proprement* », le « proprement » signifie que le nom des fichiers et le nom de l'archive sont conformes aux spécifications du client : les fichiers doivent être renommés pour comporter votre patronyme, et l'archive elle-même doit être nommée d'après votre nom et le numéro de la série, 99.

Cette dernière contrainte n'est pas une difficulté, puisqu'il suffit de bien nommer l'archive :

```
zip "machin 99" px99-?.txt
```

Notez les quotes qui permettent de contruire un nom comportant un blanc ; une autre manière de faire serait de préfixer l'espace avec le caractère \ d'échappement :

```
zip machin\ 99 px99-?.txt
```

La première contrainte suppose qu'on utilise la commande mv pour renommer chaque fichier ; à la main, ça donnerait quelque chose comme :

```
mv px99-1.txt "machin px99-1.txt" # ou la variante avec \
```

mais même pour 3 fichiers, ça rique d'être un peu pénible, donc risque d'erreur ; mieux vaut le faire par programme ; voici un ONE-LINER qu'on taperait directement dans le terminal :

```
for x in px99-[1-3].txt ; do mv $x "machin $x" ; done
```

51 comme mentionné à la section 0.3 : liste des fichiers

Notez la syntaxe de cette moulinette, que je reformule comme si c'était un script :

```
for x in px99-[1-3].txt
do
   mv $x "machin $x"
done
```

Ici, la variable x prendra, à chaque tour, une valeur différente dans la liste expansée à partir du modèle px99-[1-3].txt, autrement dit, comme si j'avais tapé :

```
for x in px99-1.txt px99-2.txt px99-3.txt...
```

Le mot-clé do n'est là que pour permettre à l'interprète BASH de repérer où commence la liste d'instructions qui se termine par le mot-clé done, même s'il n'y a qu'une instruction…

Le cœur de la moulinette est donc la commande mv, qui prend 2 arguments :
- le fichier à renommer, dont le nom est la valeur de x, c'est-à-dire $x
- le nouveau nom, une chaîne de caractères, également fabriqué à partir $x

Subtilité : une 'chaîne' n'est pas, comme en PYTHON, la même chose qu'une "chaîne" :

'machin $x' ≠ "machin $x"

dans le 1er cas, il s'agit d'une véritable constante, alors que dans le 2e, il s'agit d'une *forme* dans laquelle $x se verra substituer sa véritable valeur[52] ; une autre manière de faire serait de concaténer (implicitement) la constante et la variable : 'machin '$x sera équivalent à "machin $x" ; utiliser la constante 'machin $x' dans une telle boucle aura pour conséquence que chaque fichier sera renommé 'machin $x', et comme mv remplace le fichier s'il existe déjà, il n'y aura plus, en définitive qu'un seul fichier, nommé 'machin $x'.

L'intérêt du SHELL, c'est que chaque commande est en fait une instruction isolée, donc que rien n'empêche d'en assembler plusieurs en séquence dans un script pour réaliser, par programme, ce qu'on sait faire manuellement :

1. renommer les fichiers comme il se doit
2. les ramasser en une archive

Supposons que le script s'appelle, par exemple, emballe ; il devrait commencer par la ligne magique :

```
#! /usr/bin/env bash
```

Ce n'est pas strictement obligatoire dans le cadre d'un système où BASH est le SHELL par défaut, le script étant alors de toute façon passé à BASH pour interprétation ; mais le préciser dans le fichier permet de porter le script sur des systèmes où SH est le SHELL par défaut, car SH et BASH ne sont pas rigoureusement interchangeables[53].

Pour la portabilité du script, définissons le nom à ajouter comme une variable ; notez que sa valeur comporte un blanc à la fin :

```
auteur='machin-'
```

On peut maintenant procéder au renommage des fichiers ; ici, on suppose que le script est appelé avec une spécification de fichiers :

```
emballe 78
```

de sorte qu'il sache qu'il faut collecter tous les fichiers dont le nom est de la forme px78-?.txt ; ce 78 est un argument de la commande emballe ; et vu de l'intérieur du script, cet argument est représenté par $1 ; en effet, tout script a, par défaut autant de paramètres que de mots tapés sur la ligne de commande, $0 étant le 1er mot (c'est-à-dire le nom du script), $2 le 3e mot, s'il y en a un, et ainsi de suite…

52 ce serait l'équivalent du résultat de l'opérateur PYTHON %, dans l'expression 'machin %s' % x
53 cf. le support de cours de l'ÉC 412, éléments de systèmes d'exploitation

La boucle for sera donc de la forme for x in px$1-?.txt, suivi de do, des instructions, puis done

```
for x in px$1-?.txt
do
   echo renaming $x as $auteur$x                    # inutile mais rassurant
   mv $x $auteur$x
done
```

À ce stade, nous allons raffiner la procédure en déplaçant, dans un nouveau dossier, le lot de fichiers à compresser ; en prenant le soin de nommer ce dossier comme il faut, en fonction du nom de l'auteur et du n° de série :

```
new=$auteur$1
mkdir $new
ls -l $new*.txt                                     # inutile : juste pour bien voir ce qu'on obtient
echo moving all $new*.s* into $new                  # inutile mais rassurant
mv $new*.txt $new
```

Si tout s'est bien passé, il y a maintenant un nouveau dossier nommé truc 78 ne contenant que les fichiers à archiver, qu'on traite d'un seul coup d'un seul :

```
zip -r $new $new
```

Et pour faire le ménage, nous pourrions maintenant détruire le dossier dont il existe désormais une copie compressée :

```
rm -rf $new                                         # optionnel
```

Le script emballe contient donc les lignes suivantes :

```
#! /usr/bin/env bash
auteur='machin-'
for x in px$1-?.txt
do
   echo renaming $x as $auteur$x                    # inutile mais rassurant
   mv $x $auteur$x
done
new=$auteur$1
mkdir $new
ls -l $new*.txt                                     # inutile : juste pour bien voir ce qu'on obtient
echo moving all $new*.s* into $new                  # inutile mais rassurant
mv $new*.txt $new
zip -r $new $new
rm -rf $new                                         # optionnel
```

9.4.1 OPTIONS D'EXÉCUTION

Chaque interprète définit des options qui contrôlent son exécution ; pour leur documentation, commander man <nom de l'interprète> ; pour PYTHON, l'option -i est discutée page suivante ; pour BASH, ces deux-ci me paraissent les plus utiles.

- l'option -n — NO EXECUTION : lors de la mise au point, permet de vérifier la syntaxe du code ; aucune instruction n'est effectivement évaluée ;
- l'option -x — EXCHANGE :affiche, juste avant leur exécution, une trace des commandes élémentaires ainsi que la valeur de leurs arguments – après les diverses expansions et évaluations, y compris la substitution de commandes par l'opérateur `` ou son homologue $() ; donc *.* devient une liste de fichiers, $1 est remplacé par sa valeur, etc…

Il est possible d'utiliser la commande set pour figer temporairement ces options, autrement dit, jusqu'au prochain exit ou logout, pendant la phase de mise au point ; voir, par exemple, le site : <http://www.delorie.com/gnu/docs/bash/bashref_58.html>

9.4.2 SCRIPT EXÉCUTABLE

Complément au § iLP:9.3.5, *spécifier l'interprète* ; quel que soit le langage de script, il y a trois règles à respecter pour qu'un script soit utilisable – en tant que commande autonome – de la même façon que le serait zip ou ls :

1. le script doit commencer par les deux octets '#!' – la « signature » interne d'un script exécutable ; à la suite, sur la même ligne, il faut qu'il y ait, au minimum, le chemin d'accès à l'interprète de ce langage ; on peut y ajouter des options pour l'exécution ;
2. le script doit être en mode d'accès « exécutable » pour l'utilisateur ou son groupe.
3. le script doit se trouver quelque part dans l'un des dossiers répertoriés par la variable d'environnement PATH ;

Déroger aux règles 1 et 2 provoquera forcément une erreur, assortie d'un message plus ou moins éloquent : *bad interpreter*, *command not found*, *access denied*…

Déroger à la règle 3 est possible à condition de préciser le chemin d'accès au script qu'on veut exécuter ; par exemple, s'il se trouve dans le dossier courant :

```
./essaye
```

Mais si on a pas mal de scripts et qu'on les utilise souvent, la solution la moins encombrante consiste à placer ces scripts dans un dossier, qu'il suffit alors d'ajouter à la valeur de PATH :

```
$ mkdir ~/scripts                          # crée un dossier spécial chez moi
$ PATH=$PATH:$HOME/scripts                 # rajoute-le aux chemins répertoriés
$ mv essaye ~/scripts                      # et mets-y ce script
```

Problème : modifier ainsi la valeur de PATH ne vaut que pour la session en cours… la prochaine fois qu'on lancera le SHELL, la variable PATH sera réinitialisée d'après la valeur spécifiée dans le script .bashrc ou son cousin .bash_profile, qui sont exécutés de manière automatique[54].

Solution : il suffit d'y altérer la valeur donnée à PATH pour que la modification persiste.

Cependant, tout script, même non conforme à ces règles, est quand même parfaitement exécutable, à condition de spécifier – de l'extérieur – l'interprète qui doit l'exécuter ; par exemple, si, dans le dossier de travail, le fichier essaye, marqué par défaut rw-rw-rw-, contient simplement cette ligne :

```
print("hé, mais ce script aussi, tourne comme il faut !\n")
```

on peut aussi bien l'exécuter avec :

```
$ ruby essaye
hé, mais ce script aussi, tourne comme il faut !
$ perl essaye
hé, mais ce script aussi, tourne comme il faut !
```

D'ailleurs, même pour un script autonome conforme aux 3 principes ci-dessus, il reste toujours possible de le lancer en spécifiant l'interprète de l'extérieur ; cette solution permet de spécifier des paramètres qui modifieraient le comportement de l'interprète ; exemple, un script PYTHON nommé zut, normalement utilisé comme :

```
zut a b c
```

pourrait aussi être lancé par la commande :

```
python -i zut a b c
```

qui force PYTHON à passer en mode interactif une fois l'exécution terminée, ce qui nous permet d'examiner les valeurs des variables ou de faire des tests complémentaires ; notons que ceci ne contrarie absolument pas le mécanisme de passage d'arguments : zut les verra de la même façon que s'il avait été lancé en autonome.

54 http://how-linux-works.org.ua/8760final/lib0143.html

RÉCAPITULATION

Ce chapitre, fait de bric et de broc, n'a jamais été qu'un fourre tout commode pour noter en dur un certain nombre de précisions qui ont fait, sur le forum, l'objet de demandes d'éclaircissements ; alors, évidemment, il n'est pas aisé de les intégrer au cours proprement dit sans risquer de l'alourdir, donc d'en gêner la lescture ; considérez-les donc comme une mémoire vestigiale du forum, mémoire qu'il aurait été dommage de perdre…

Après, bien sûr, il faudrait faire l'effort de présenter tout ça avec un souci de cohérence globale et de continuité pour mieux montrer comment ça s'intégre avec les connaissance dispensées par le cours : et en dépit des apparences, c'est ce que j'ai essayé de faire ici, même si le résultat de mon effort semble sauter du coq à l'âne !

INDEX

Ce répertoire est minimaliste : il omet les opérateurs les plus fréquents, comme in ou for, de même que les instructions du genre if ou pass ; ne sont pas indexés non plus des méthodes comme split() ou pack(), tellement utilisées qu'il serait superflu de les référencer…

GLOSSAIRE

La plupart de ces termes sont expliqués en détail quelque part sur des pages WEB, peut-être même avec trop de détails ; c'est pourquoi ce petit glossaire est volontairement limité à une définition sommaire des termes que vous pourriez rencontrer sur nos supports de cours, où ils sont d'ailleurs probablement mieux définis qu'ici…

agrégat : structure de données hétérogènes ; s'oppose à scalaire et à vecteur ; voir {cours ELI 113, introduction aux structures de données, point de vue de la machine}

a-list : voir liste d'associations

analogique : continu ; s'oppose à discret, mais aussi à numérique, donc à digital.

API : application programming interface ; désigne les données et fonctions qui constitue l'interface entre deux programmes ; généralement sous la forme de code-source, cette interface fournit les points d'entrée vers les service d'un programme utilisé de façon ancillaire : système d'exploitation, bibliothèque de fonctions, etc <http://en.wikipedia.org/wiki/Application_programming_interface>

argument : dénote l'opérande d'une fonction.

arité : capacité d'accueil — unaire, binaire, ternaire… En Prolog, l'arité caractérise le nombre d'arguments d'un prédicat.

ascii : standard de codage sur 7 ou 8 bits pour les caractères alpha-numériques et la ponctuation.

atome : élément lexical théoriquement non fissible ; en Lisp et Scheme, qui n'ont que 2 types de données, atome s'oppose à liste, avec cette particularité que la liste vide est en fait l'atome nil.

binaire : qui peut prendre deux valeurs ; c'est le cas de la logique booléenne.

bit : littéralement petit bout ; aussi abréviation de bi]nary dig[it ; au sens informatique, c'est l'atome d'information, représentée en interne comme une tension électrique interprétée, suivant l'intensité, comme un 1 ou un 0 ; comme un bit peut avoir 2 valeurs, il en suit qu'une séquence de n bits peut représenter 2^n valeurs.

byte : vecteur (ou mot) de 8 bits, littéralement une bouchée.

C : langage de programmation polyvalent, créé en 1972.

cache : type de mémoire à accès rapide (de l'ordre de la pico-seconde) servant d'intermédiaire entre les registres du processeur et la mémoire vive ; corruption du français *cachée* (l'anglais ignore les accents).

cahier des charges : ensemble des spécifications pour un projet de programme, émanant généralement du client, donc présenté de manière informelle.

callback : se dit du mode d'invocation d'un handler par un gestionnaire d'événement, event manager ; par extension, dénote le handler lui-même.

car : valeur de la fonction car() en Lisp ou Scheme (*contents of address register*).

cdr : [prononcé ˈkʌdərl or ˈkʊdərl]valeur de la fonction cdr() en Lisp ou Scheme (*contents of decrement register*).

caractère : scalaire dont la représentation interne se compose d'un ou de plusieurs octets selon le codage numérique dépendant du standard utilisé : ASCII, *unicode*, etc.

chaîne : autre nom pour séquence, en particulier séquence de caractères.

char : [se prononce lkɛrl] abrégé pour character ; voir caractère.

classe[55] : objet virtuel muni de propriétés, et capable de générer de nouveaux objets : cette filiation permet un mécanisme d'héritage grâce auquel les propriétés de l'original sont partagées par tous les descendants.

client : dans l'architecture client/serveur, dénote la partie qui utilise le service.

code : convention de représentation ; dans l'activité de l'analyste/programmeur le code est le texte du programme, par opposition à tous les autres textes qui décrivent le programme de manière non-formelle.

55 au sens informatique (et par abus de langage), une classe est, comme un type, définie en intension ; ce n'est pas vrai en mathématiques ou en philosophie, où une classe se définit en extension

collection : structure non-séquentielle de données hétérogènes ; par exemple, la classe bag dans les langages à objets.

commande : terme exclusivement utilisé dans le contexte d'un SHELL, dénotant une expression évaluable (instruction exécutable).

compilateur : programme de traduction d'un langage en un autre, généralement plus primitif (souvent le langage-machine).

constante : se dit d'une valeur immuable, par exemple celle du nombre Pi ; par abus de langage, se dit aussi du symbole qui représente cette valeur : par exemple, la constante Pi.

data-driven programming : voir programmation dirigée par les données.

déclaratif : se dit d'un langage qui, au lieu de structures de contrôle, utilise des données structurées, interprétées par un moteur d'évaluation.

digital : numérique, de l'anglais digit (chiffre) emprunté au latin digit (doigt).

directory : littéralement, annuaire ou répertoire ; équivalent à folder et dossier

double : type scalaire utilisé pour représenter un nombre réel en double précision ; équivalent au type float en PYTHON.

entier : type scalaire représentant une valeur entière ; nombre entier, signé ou non.

EPROM : acronyme anglo-saxon pour erasable programmable read-only memory

évaluation : mécanisme qui calcule la valeur d'une expression ; un littéral est sa propre valeur, une variable a la valeur avec laquelle on l'a défini, et une expression non-atomique est évaluée selon les règles syntaxiques en vigueur dans le langage.

event driven programming : voir {programmation dirigée par les événements}

event manager : voir {gestionnaire d'événements}

fichier : structure de donnée représentant un conteneur de données, généralement sur disque, et souvent statique[56] ;par extension, dénote le conteneur lui-même, ou son nom, ou encore son contenu.

flash memory : voir mémoire flash

float : type scalaire représentant un nombre réel en virgule flottante ; en C, il est implicitement en simple précision ; en PYTHON, il est en double précision.

flux : (stream) le canal de communication avec un fichier.

foncteur : terme obscur dont le sens le plus commun est « fonction qui accepte parmi ses arguments un symbole de fonction à appliquer aux autres arguments ». Dans les langages fonctionnels, la famille map regroupe toute une population de foncteurs, mais il y en a d'autres, comme apply, reduce, ou zip.

fonction : une abstraction[57] — un objet f qu'on peut appliquer à une valeur x dont il se sert pour effectuer un calcul dénoté f(x) ; de l'extérieur de la fonction, cette valeur x est appelée argument ; de l'intérieur, cette valeur est représentée par une variable qu'on appelle paramètre de la fonction. Une fonction retourne généralement une valeur, de sorte qu'on puisse écrire quelque chose comme y = f(x) {voir aussi procédure}. Une fonction définie sans aucun paramètre n'acceptera pas d'argument : dans ce cas, le résultat du calcul qu'elle effectue dépend de son contexte local[58], et du contexte global.

fonctionnel : se dit d'un langage de programmation où la structure de contrôle la plus évidente est l'application d'une fonction au résultat de l'application de fonctions.

framework : environnement logiciel extensible, adaptable aux besoins d'une application spécifique ; voir {plug-in}

gestionnaire d'événements : boucle programmée pour écouter les interruptions en provenance d'organes périphériques.

giga : souvent abrégé en **G** ; pour les informaticiens, c'est 2^{30}, c'est-à-dire 1024 mégas, et ce en dépit de la normalisation de 1998 ; pour les autres ce n'est qu'un milliard.

glyphe : représentation graphique d'un caractère ou d'un idéogramme.

handler : fonction paramétrée pour gérer un type d'événement spécifique.

56 les fichiers (ou plutôt *flux*) unix standard stdin, stdout et stderr sont dynamiques, et représentent respectivement le clavier, et l'afficheur

57 une fonction peut être anonyme : les langages fonctionnels disposent d'une forme appelée lambda-expression qui se comporte exactement comme une fonction mais n'a pas de nom symbolique

58 certains langages (LISP, ICON, PYTHON, SCHEME) disposent de générateurs, fonctions qui auto-génèrent une valeur, en fonction de leur contexte local

hash table : structure non-séquentielle de données organisées sous la forme de couples clé + valeur, telle qu'on accède directement à la valeur quand on en connaît la clé ; il n'y a pas de contrainte de type sur la valeur, qui peut donc, elle aussi, être une hash table.[59] En PYTHON, ce type de donnée s'appelle dictionnaire, ce qui évoque fort justement la manière dont on l'utilise.

hertz : mesure de la périodicité (cadence) par seconde ; le plus souvent abrégé en Hz.

hétérogène : se dit d'un ensemble d'éléments disparates (de types différents).

homogène : se dit d'un ensemble d'éléments de même nature (de même type).

impératif : ou procédural ; s'oppose à déclaratif ; se dit des langages de programmation où la structure de contrôle la plus évidente est la séquence d'instructions impératives (injonctives), style recette de cuisine ; ce qui n'empêche pas, s'ils disposent de fonctions[60], de les utiliser dans le style fonctionnel.

index : valeur entière positive[61] utilisée pour accéder, dans une séquence, à l'élément de ce rang ; dans la plupart des langages, les index commencent à 0, ce qui permet d'accéder à l'élément de rang 0, c'est-à-dire le premier élément. C'est aussi un doigt qui montre où se trouve la donnée.

indice : voir index

instance : représentant d'une classe ou d'un type — objet généré par réification (réalisation).

instruction : une expression évaluable (donc bien formée) dans un langage impératif.

int : dans de nombreux langages, type scalaire d'une valeur entière.

integer : signifie entier, en latin et en anglais (se prononce ˌɪntədʒərl).

interprétation : tout est interprété, en contexte, y compris la tension électrique d'un bit[62] ; le langage-machine, par exemple, suppose l'interprète adéquat (ou processeur), faute de quoi, rien n'a de sens.

interprète : au sens le plus courant, programme capable d'évaluer du code-source en le traduisant à la volée en une série d'appels de fonctions (ou procédures) de haut niveau.

interpréteur : traduction malheureuse de l'anglais interpreter.

kilo : souvent abrégé en **K** ; pour les informaticiens, c'est 2^{10}, c'est-à-dire 1024, et ce en dépit de la normalisation de 1998 ; pour les autres, c'est 10^3.

langage : au sens informatique, un jargon cryptique avec une syntaxe généralement absconse ; sauf Lisp[63]

Lisp : langage de programmation fonctionnel.

liste : séquence de données hétérogènes, qui peut prendre un nombre quelconque d'éléments quelconques, y compris d'autres listes (éventuellement elle-même), et peut aussi être totalement vide. Les listes sont mutables, en ce sens qu'on peut en modifier un élément, en insérer de nouveaux, ou en enlever. Cet aspect dynamique en fait un type de donnée très apprécié pour toutes sortes de représentations.

liste d'associations : en Lisp ou Scheme, données structurées sous la forme d'une liste de sous-listes dont le premier élément sert de clé pour accéder à la valeur, c'est-à-dire le reste de la sous-liste ; le principe de son utilisation est le même que celui des tables de hachage mais la mise en œuvre est différente (voir hash table).

littéral : se dit d'une valeur qu'il faut prendre littéralement, sans évaluation : au pied de la lettre[64]

long long int : type scalaire entier, aussi large que le bus de données (64 bits).

matrice : par atavisme mathématique, dénote une table à 2 dimensions ; par exemple le résultat du produit d'un vecteur par un vecteur transposé.

méga : souvent abrégé en **M** ; pour les informaticiens, c'est 2^{20}, et ce en dépit de la normalisation de 1998 ; pour les autres c'est 10^6.

mémoire : dispositif capable de mémoriser de l'information à plus ou moins long terme ; grossièrement, on distinguera la mémoire à long-terme, relativement non-volatile, telle que mise en œuvre par un disque [magnétique ou optique] ou par une PROM, et la mémoire à court-terme, volatile, mise en œuvre par des composants électroniques qui oublient tout dès qu'on leur supprime leur source d'énergie : ceci comprend la

59 peut-on traduire ça par table de hachage ?
60 le langage-machine, ou langage du processeur, est typiquement impératif, et ne dispose pas de fonctions
61 certains langages permettent d'utiliser des index négatifs, exprimant ainsi la possibilité d'indexer une séquence en partant de la fin (en fait, la taille de l'objet indexé est implicitement ajoutée à l'index)
62 et son équivalent magnétique, l'orientation d'un cristal d'oxyde ferrique
63 même SCHEME
64 si tant est que les lettres ont des pieds…

mémoire vive, mais aussi la mémoire dite cache, et les registres du processeur.

mémoire flash : type d'EEPROM inventé en 1984 ; utilisée dans des sticks de mémoire non-volatile, électriquement effaçable et reprogrammable, connue sous différents noms, selon le fabricant et la technologie employée — USB drive, SmartMedia, etc. En 2007, la capacité maximum en était limitée à 128 GB.

méthode : dans le contexte des langages à objets, c'est la fonction assortie au sélecteur du message, spécifique d'une classe d'objets ; en PYTHON, le sélecteur est le nom de la méthode, et s'utilise en composant <objet>.<sélecteur()>…

microcode : le tout-premier niveau de programmation d'un processeur CISC[65] est fait, en grande partie, de micro-instructions[66] résidentes en ROM, PROM ou EPROM, et chargées[67] au démarrage.

MIME : [multipurpose internet mail extension] ensemble de descripteurs caractérisant les données d'un message électronique.

MIME-type : caractérisation d'un fichier en fonction de son contenu ou du suffixe apposé à son nom, en vue de transfert internet ; cf. <http://www.webmaster-toolkit.com/mime-types.shtml>

nibble : type scalaire dénotant une séquence de 4 bits ; autrement dit, la moitié d'un byte. Utilisé sur les cartes graphiques travaillant sur 12 bits.

nil : abrégé du latin nihil, c'est-à-dire rien… équivalent au NULL de ANSI-C.

null char : caractère codé 0, représentant la chaîne vide (exprimé en C par '\0' ou "").

null : dénote l'opérateur de négation en logique booléenne[68]

NULL : valeur interprétée comme indéfinie, selon le contexte[69] ; en ANSI-C, c'est un pointeur qui ne pointe nulle part : il est, entre autres, utilisé, comme nil, pour terminer une séquence.

numérisation : se dit d'un processus de conversion d'analogique en numérique ; par exemple :

numérisation d'un document = reconnaissance des caractères imprimés + codage des caractères ainsi reconnus.

objet : au sens large, ça peut être n'importe quoi ; au sens strict, dans les langages à objets, ça dénote une structure de données, collection de propriétés spécifiques, tant comportementales que purement attributives : si un objet est une classe, il permet de générer de nouvelles instances qui héritent automatiquement de ses propriétés (qui deviennent du même coup génériques).

octet : vecteur de 8 bits ; voir byte.

opérande : valeur à laquelle s'applique un opérateur.

opérateur : abstraction représentant une opération, dont l'évaluation (effectuation) retourne généralement une valeur ; les opérateurs peuvent être hiérarchisés du point de vue de leur priorité,[70] ce qui peut influencer l'évaluation. Les opérateurs peuvent être, selon les langages, exprimés sous forme préfixée, infixée ou postfixée.

opération : évaluation d'une expression pour produire une valeur, ou un effet, ou même les deux à la fois.

paramètre : symbole de variable représentant, dans la définition d'une procédure ou d'une fonction, la valeur passée en argument.

pile : séquence dont seul le dernier élément empilé (autrement dit le seul élément visible) est accessible,[71] c'est-à-dire qu'on ne peut accéder à l'élément suivant qu'en ayant dépilé le précédent ; équivalent français de l'anglais stack, c'est la converse d'une queue.

plug-in : programme constituant une extension d'un framework.

pointeur : valeur entière positive typée, utilisée pour l'adressage en mémoire ; le type (c'est-à-dire la mesure de l'espace occupé par la donnée) est pris en compte pour les calculs arithmétiques usuels : addition, soustraction, de sorte que le pointeur est utilisable comme un index pour ce type de donnée.

prédicat : par atavisme *logico-mathématique*, dénote une expression qui retourne une valeur logique (vrai ou faux) ; dans beaucoup de langages, certaines expressions, comme la valeur 0 ou la séquence vide, peuvent être considérées d'un point

65 opposé à RISC
66 qui elles-mêmes gèrent des connexions de constituants (au niveau matériel), autrement dit des réglages de l'unité arithmétique et logique ; comme tous les autres programmes, le microcode peut contenir des bugs…
67 la puce dite BIOS de la carte-mère (généralement une EPROM, donc *flashable*), appariée au processeur, est programmée pour corriger ces bugs au démarrage
68 pour les anglo-saxons, noté comme un ~ au dessus d'une variable
69 définie, en fait, comme un 0
70 neutralisable en parenthésant les expressions à opérateurs multiples
71 LIFO — last in, first out

de vue logique, et sont alors équivalentes à faux, ce qui entraîne que toute autre expression a pour valeur logique vrai[72]

procédure : une séquence d'instructions ou fonction qui ne retourne pas de valeur.

processeur : machine logique.

programmation dirigée par les données : technique où le code est généré dynamiquement (donc en tant que donnée d'un programme), puis évalué ; c'est la technique utilisée par la plupart des interprètes, mais aussi, par exemple, par les filtres anti-spam.

programmation dirigée par les événements : technique où les fonctionnalités d'un programmes sont invoquées indirectement par des événements déclencheurs[73], gérés par un event manager qui se charge d'appeler le handler défini pour cet événement.

programme : texte évoquant un truc à faire et comment le faire ; par extension, le code exécutable correspondant.

Prolog : langage de programmation déclaratif qui met en avant l'objectif par rapport aux moyens d'y parvenir ; un peu comme si on programmait avec des if à l'envers : là où Lisp dit : « appuyer sur le bouton pour récupérer votre monnaie », Prolog dit : « pour récupérer votre monnaie, appuyer sur le bouton ». En C, la même chose se dirait : « vous n'aviez qu'à faire l'appoint ».

PROM : acronyme anglo-saxon pour programmable read-only memory.

queue : structure séquentielle de données, qui se gère sur le principe du premier arrivé, premier servi[74] ; c'est la converse d'une pile.

récursion : peut être croisée ; voir récursivité…

récursivité : voir récursion…

regression [tests] : tests de régression ; ensemble de tests associés à une fonction pour s'assurer qu'une modification ne compromet pas son bon comportement.

répertoire : fichier représentant une liste de fichiers ; synonymes : dossier, directory.

ROM : acronyme anglo-saxon pour read-only memory

scalaire : structure de données élémentaire destinée à représenter une valeur qui appartient à une échelle de valeurs, autrement dit, à une séquence implicite ; typiquement, un nombre ou un caractère.

script : dénote un texte de programme destiné à être interprété à la volée.

sélecteur : dans le contexte des langages à objets, c'est la partie du message qui s'apparie à une méthode compatible avec le destinataire du message.

séquence : structure de données ordonnées (organisée de façon séquentielle) ; dans beaucoup de langages, on peut accéder directement à un élément de la séquence d'après son rang, au moyen d'un index ; pour d'autres la séquence est implicitement considérée comme une pile.

serveur : dans l'architecture client/serveur, dénote la partie qui fournit le service.

shell : interprète d'instructions (= commandes) ligne par ligne, ou sous forme de scripts.

sprite : Ispraɪtl a computer graphic that may be moved on-screen and otherwise manipulated as a single entity.

string : voir chaîne

struct : en C, agrégat hétérogène, regroupant en séquence de multiples déclarations.

structure : organisation, arrangement ; C : voir struct.

structure de contrôle : procédé programmatique de routage d'une séquence d'instructions ; dans un programme : structure des évaluations successives (ou flot), que certains s'efforcent de représenter par des diagrammes à peu près inintelligibles au delà d'une certaine complexité, et sans utilité en deçà.

stub : [*angl* → coupon] fonction qui ne fait rien.

symbole : une abstraction représentant quelque chose d'autre : ainsi, pour nous autres, le symbole 4 représente la quantité iiii[75]

syntaxe : ensemble des règles de construction d'expressions bien formées ; dépend largement du langage.

72 en LISP et SCHEME, seule la liste vide, nil, évalue à faux

73 provenant le plus souvent de périphériques, comme la souris ou le clavier

74 FIFO - first in, first out

75 remarquons que iiii est en soi un symbole (une représentation symbolique) où le nombre de i s'interprète comme une quantité

table : séquence de données homogènes, de taille fixe ; dans le jargon des bases de données, une table est en fait un vecteur ; au sens large, dans d'autres contextes, une table peut être aussi bien un vecteur qu'une matrice, et il n'y a pas de limite théorique aux nombre de dimensions d'une table, bien qu'en pratique ça devienne un peu délicat au delà de la quatrième dimension.

table driven programming : technique qui exploite une table associative[76] pour savoir quoi évaluer comme code selon les données[77]

tableau : traduction malheureuse de l'anglais array qui dénote une série (arrangement) d'objets de même nature ; ce terme est utilisé exclusivement par les universitaires francophones, dans le sens large de table.

téra : souvent abrégé en **T** ; pour les informaticiens, 2^{40}, soit 1024 gigas, et ce en dépit de la normalisation de 1998 ; les autres ne s'en servent pas.

tuple : tuplet, séquence de données hétérogènes, non-mutable (immuable), de taille fixe ; s'apparente à la liste, qui, elle, est mutable ; dans une définition une fonction, la liste de paramètres est un tuple.

type : caractérisation[78] (ou nature) d'un objet en termes de sa représentation interne et externe, ainsi que des opérations avec lesquelles il est compatible[79] ; il est des langages qui ne typent que les valeurs, alors que d'autres typent aussi les variables, ce qui peut être vu comme un avantage (conversion automatique) ou comme un inconvénient (inflexibilité). Certains langages opposent type à classe pour exprimer que le premier est primitif, donc non re-programmable.

unicode : standard moderne de codage sur 8, 16, 24 ou 32 bits, pour les idéogrammes, les caractères alpha-numériques et la ponctuation.

URL : [uniform resource locator] adresse précédée d'un protocole d'accès réseau, comme ftp:// ou encore http://

UTF : unicode transformation format ; cf. <http://unicode.org/faq/utf_bom.html>

variable : une abstraction ; un symbole défini pour représenter une valeur, laquelle est susceptible de varier[80]

vecteur : séquence de taille fixe de données de même nature — ou de même type, ou homogènes, c'est comme on veut ; voir cours ELI 113, introduction aux structures de données, point de vue de la machine.

76 JSCRIPT : associative array ; LISP : a-list ; PYTHON : dictionary

77 en C, peut s'approximer en utilisant l'opérateur switch comme structure de contrôle

78 définie de façon *intensionnelle*

79 seul le langage machine est absolument non-typé, ce qui veut dire qu'on peut faire ce qu'on veut avec ce qu'on veut, mais c'est pas facile

80 les langages fonctionnels *vrais* manipulent les variables comme des références, ce qui permet à une variable d'avoir, comme valeur, une variable (éventuellement elle-même)

GNU Free Documentation License
Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual

cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section entitled "Acknowledgements" or "Dedications", preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute

it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

TABLE DES MATIÈRES