

Le langage Python

Table des matières

Chapitre 1. Introduction	9
1.1. Le public visé	9
1.2. Pourquoi Python ?	10
1.3. Caractéristiques du langage	12
1.4. Domaines d'application	14
1.5. Conventions	15
1.6. La documentation Python	15
1.7. Style de programmation	15
Conventions de nommage	15
Conventions d'écriture	16
1.8. Exécution d'une session <i>Python</i> interactive	16
Exécution en ligne de commande	16
Exécution par script (ou module)	19
Les éditeurs Python	20
Les environnements complets	20
Déboguer un programme	23
Exercices	25
Solutions des exercices	25
Chapitre 2. Premiers pas	29
2.1. Pour vérifier votre installation	29
2.2. Aide en ligne, dir() et help()	30
2.3. Commentaires	31
2.4. Indentation	31
2.5. Variables, types	32
2.6. Opérations	35
2.7. Chaînes de caractères	36
2.8. Listes	39
2.9. Dictionnaires	41
2.10. Lecture-écriture sur fichier	42
Le module pickle	43
2.11. Fonctions	44
2.12. Documentation et tests	45
2.12.1. Tests dans le fichier module	46
2.12.2. Tests dans un fichier séparé	49

Exercices	50
Solutions des exercices	52
Chapitre 3. Structures de contrôle	59
3.1. L'instruction while	59
3.2. L'instruction if, else, elif	60
3.3. L'instruction for	61
3.4. Les conditions	62
3.4.1. Valeurs booléennes	62
3.4.2. Opérateurs de comparaisons	63
3.4.3. Comparaisons hétérogènes	64
3.5. Les techniques de boucles	65
3.5.1. La fonction range()	65
3.5.2. Les méthodes dict.keys(), dict.values() et dict.items()	66
3.5.3. La fonction enumerate()	66
3.5.4. La fonction zip()	66
3.5.5. La fonction reversed()	67
3.5.6. La fonction sorted()	67
3.5.7. La fonction map()	67
3.5.8. La fonction filter()	68
3.5.9. Les <i>list comprehension</i>	68
Exercices	69
Solutions des exercices	71
Chapitre 4. Les types intégrés (builtins)	79
4.1. Les variables Python	79
4.2. Les types numériques (int, float, complex)	81
4.3. Itérateurs et générateurs	82
4.3.1. Définition	82
4.3.2. Itérateurs	83
4.3.3. Générateurs	84
4.3.4. Expressions génératrices	85
4.4. Les séquences tuples et list	85
4.4.1. Les tuples	85
4.4.2. Le découpage (<i>slicing</i>)	87
4.4.3. Les listes	88
4.5. Les chaînes de caractères (str, bytes, bytearray)	90
4.5.1. Identification et codage des caractères	90
4.5.2. Comparatif sommaire Python 2 et 3	92
4.5.3. Encodage et décodage	94
4.5.4. Manipulations usuelles	94
4.5.5. Le module string	95
4.5.6. Formatage	96

4.5.7. Autres fonctionnalisés pour le	99
4.6. Les ensembles (set et frozenset)	99
4.7. Les dictionnaires (dict)	100
4.8. Les exceptions (exceptions)	101
Exercices	103
Solutions des exercices	107
 Chapitre 5. Fonctions, scripts et modules	 117
5.1. Fonctions	117
5.1.1. Exemple détaillé	118
5.1.2. Passage d'arguments	119
5.1.3. Les formes lambda	122
5.2. Règles de portée	123
5.2.1. Espace de noms	123
5.2.2. Portée, la règle LGI	124
5.3. Modules et paquetages	125
5.3.1. Modules	125
5.3.2. Paquetages	127
5.3.3. Le chemin de recherche des paquetages et modules	128
5.3.4. Fichiers bytecode	128
5.3.5. Modules standard	128
5.4. Décorateurs	129
5.5. Fonctions intégrées, le module __builtin__	130
5.5.1. Manipulation d'attributs	130
5.5.2. Conversion de type, construction	131
5.5.3. Logique	132
5.5.4. Programmation	132
5.5.5. Itérables	133
5.5.6. Entrées-sorties	135
5.5.7. Classes	137
5.5.8. Mathématiques	137
5.5.9. Divers	137
Exercices	138
Le format IGC (International Gliding Commission)	143
Solutions des exercices	144
 Chapitre 6. Programmation orientée objet	 153
6.1. Une première approche des classes	153
6.1.1. Syntaxe	154
6.1.2. Une classe Complexe	154
6.1.3. Objets Classes	155
6.1.4. Objets Instances	155
6.1.5. Objets Méthodes	156

6.1.6. Quelques remarques	156
6.2. Héritage simple, multiple	157
6.2.1. Héritage ou pas ?	158
6.2.2. Exemple de la classe Polygone TODO : commenter	158
6.2.3. Surcharge	159
6.2.4. Héritage multiple	160
6.3. La classe object	160
6.4. Méthodes spéciales	161
6.5. Variables privées	162
6.6. Les <i>properties</i>	162
6.7. Manipulation dynamique des attributs	164
6.8. Slots	165
Exercices	166
Solutions des exercices	170
Chapitre 7. La bibliothèque standard	177
7.1. Le module os	177
7.1.1. Le module <code>os.path</code>	178
7.1.2. Autres services du module os	179
7.2. Le module sys	179
7.3. Dates et heures TODO	180
7.4. Fichiers	181
7.4.1. Le module glob	181
7.4.2. Le sous module glob.fnmatch	181
7.4.3. Le module shutil	181
7.5. Compression de données TODO	182
7.6. Arguments de la ligne de commande	182
7.7. Mesure des performances	183
7.7.1. Le module time	183
7.7.2. Le module timeit	184
7.7.3. Les modules profile, cProfile et pstats	185
7.7.4. Le module unittest TODO	187
7.8. Calcul scientifique	187
7.8.1. Le module math	187
7.8.2. Le module cmath	189
7.8.3. Le module random	189
7.8.4.	190
7.9. Autres modules	190
Exercices TODO	191
Solutions des exercices	192
Chapitre 8. <i>Python</i> pour le calcul scientifique	197
8.1. NumPy	197

8.1.1.	Méthodes et fonctions de <i>NumPy</i>	197
8.1.2.	Les bases de Numpy	198
8.1.3.	Création de tableaux	199
8.1.4.	Opérations sur les tableaux	200
8.1.5.	Indiçage, découpage, itérations sur les tableaux	201
8.1.6.	Remodelage des tableaux	203
8.1.7.	Empilage de tableaux	204
8.1.8.	Copies et vues	205
8.1.9.	Quelques fonctions plus évoluées	206
8.1.10.	Algèbre linéaire de base	208
8.2.	<i>Scipy</i>	209
8.3.	Calcul formel avec <i>Sage</i>	209
8.4.	C/C++ en Python	209
8.4.1.	Écrire en <i>Python</i> , relire en C++ (binaire)	210
8.4.2.	pyrex, cython	212
8.4.3.	ctypes, swig	216
8.4.4.	boost	216
8.4.5.	shiboken TODO	216
8.4.6.	weave	217
8.5.	Visualisation de données	217
8.5.1.	matplotlib, pylab	217
8.5.2.	SciTools, EasyViz	217
8.5.3.	Vtk	217
8.5.4.	Mayavi	218
8.5.5.	Paraview	218
	Exercices	218
	Solutions des exercices	220
	Index	229
	Bibliographie	233

CHAPITRE 1

Introduction

1.1. Le public visé

Cet ouvrage s'adresse à un public ayant déjà programmé, et curieux d'apprendre *Python*.

Une habitude de la programmation dans un langage procédural ou fonctionnel, comme *Fortran*, *C++* ou *C* ou encore *Scilab*, *Matlab*, etc. est préférable avant d'en aborder la lecture.

Les principes de programmation orientée objet utiles sont suffisamment détaillés dans le chapitre consacré aux classes pour servir de cours d'initiation.

Cet ouvrage a été bâti à partir de notes assemblées pour des cours et des formations *Python* que j'ai assuré à l'Université de Pau, en formation interne, en Master de Mathématiques Appliquées et lors de diverses interventions.

Ces notes ont été largement inspirées des ouvrages ou sites internet cités dans la bibliographie. J'y ai bien sûr rajouté ma touche personnelle au cours des formations, en fonction des difficultés rencontrées par le public.

Étant mathématicien de formation, ma connaissance des concepts informatiques est ancienne et essentiellement expérimentale. Mon approche du langage *Python* pourra sembler naïve aux informaticiens de métier. Elle l'est. Il m'a paru intéressant de m'attarder sur certains concepts contre lesquels j'ai longtemps achoppé, et de faire profiter le lecteur de ce point de vue naïf.

Le point de vue que je tente de développer s'est en tout cas montré efficace pour les tâches de programmation auxquelles j'ai été confronté. Cet ouvrage vise un public d'*utilisateurs* d'informatique, et non pas un public de *concepteurs* d'informatique.

Les puristes me pardonneront donc cette incursion dans leur domaine et les imprécisions ou erreurs qu'ils relèveront.

La version 3 de *Python* est en principe utilisée tout au long de cet ouvrage. La rédaction en a commencé alors que *Python* 2 était seul disponible, aussi, malgré le soin apporté à la rédaction et à la relecture, il est possible que quelques incompatibilités demeurent dans le cours du texte, les exercices proposés ou leur solution. Toute remarque, correction ou suggestion sera la bienvenue par mail à l'adresse pierre@puiseux.name.

Les différences essentielles entre *Python* 2 et *Python* 3 seront soulignées au cours de l'exposé.

1.2. Pourquoi Python ?

Python est un langage de programmation récent (apparu en 1990) et dont une des caractéristiques essentielle est la facilité de lecture et d'écriture. Voici un exemple simple qui illustre bien ceci : supposons que l'on veuille créer une liste des carrés des entiers compris entre -5 et 12, puis la faire afficher à la console. Voici un programme C++ qui effectue cette tâche :

```

1  #include <iostream>
2  int main()
3  {
4      int l[18];
5      for (int i=0; i<18; i++)
6      {
7          int x(i-5);
8          l[i] = x*x;
9      }
10     for (int i=0; i<18; i++)
11         std ::cout << l[i] << ", ";
12     std ::cout << std ::endl;
13     return 0;
14 }
```

Et voici l'équivalent **Python** :

```

1  l = [x*x for x in range(-5,13)]
2  print(l)
```

Ce programme **Python**, outre sa concision, propose une lisibilité optimale. Il est en effet inutile d'être un programmeur chevronné pour comprendre qu'il crée la liste l des x^2 pour x dans l'intervalle $[-5, 13]$ (ou bien $[-5, 13[$?), puis qu'il imprime la liste l .

Mark Lutz, dans [ML] dans nous met en garde :

« You should be warned, though once you start using **Python**, you may never want to go back. »

Pour avoir une idée plus précise de la philosophie qui sous-tend le langage, dans un shell **Python**, importons le module `this` (voir 1.8 page 16). On obtient ce texte :

« *The Zen of Python, by Tim Peters*
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

*Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one – and preferably only one – obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those ! »¹*

On le voit, **Python** est un langage très attractif.

Les exemples, et indications données ici ont été testés sous *Linux*, distribution *Ubuntu* 10.10.

La portabilité de **Python** d'un OS à l'autre est excellente, quelques rares instructions peuvent être spécifique au système utilisé, elles seront signalées.

¹En voici une traduction qui se permet quelques libertés avec l'original

Le Zen de **Python**, par Tim Peters

« Mieux vaut beau que laid.
Mieux vaut explicite qu'implicite.
Mieux vaut simple que complexe.
Mieux vaut complexe que compliqué.
Mieux vaut à plat que imbriqué.
Mieux vaut aéré que dense.
La lisibilité est importante.
Les cas particuliers ne le sont pas assez pour enfreindre les règles.
Bien que la pratique l'emporte sur la pureté.
Les erreurs ne doivent pas être passées sous silence.
Sauf si elles sont explicitement silencieuses.
Face à l'ambiguïté, ne pas tenter de deviner.
Il devrait y avoir un - et de préférence un seul - moyen évident.
Bien que ce moyen ne soit pas évident, sauf à être néerlandais.
Mieux vaut maintenant que jamais.
Bien que jamais soit préférable à immédiatement et sans délai.
Si l'implémentation est difficile à expliquer, c'est une mauvaise idée.
Si l'implémentation est facile à expliquer, c'est peut être une bonne idée.
Les espaces de nommage : une bonne idée qui décoiffe - et si on en abusait ? »

On ne parlera pas de l'installation de **Python** sur les différentes plateformes. Internet regorge de tutoriels, *howto*, etc... dédiés à ce problème. En particulier la page <http://www.python.org/getit/> est très explicite.

Certaines particularités de **Python** 2 seront soulignées dans le texte.

Python est développé depuis 1989 par Guido van Rossum et de nombreux contributeurs. Un historique complet pourra être consulté sur [http://fr.wikipedia.org/wiki/Python_\(langage\)](http://fr.wikipedia.org/wiki/Python_(langage))

Sur le site <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, on trouve un classement des langages de programmation les plus populaires.

Les taux d'utilisation sont calculés en comptant le nombre de recherche dans les moteurs de recherche les plus populaires (Google, Google Blogs, MSN, Yahoo!, Wikipedia et YouTube) pour les 12 derniers mois.

Le titre de « langage de l'année », attribué au langage de programmation ayant le plus progressé au cours de l'année passée, a été décerné en 2010 au langage **Python**.

Fév 2011	Fév 2010	Langage	Ratings	delta
1	1	Java	17.9%	-1.47%
2	2	C	17.1%	+0.29%
3	4	C++	9.8%	+0.83%
4	7	Python	7.0%	+2.72%
5	3	PHP	7.0%	-3.0%
6	6	C#	6.8%	+1.79%
7	5	Visual Basic	4.9%	-2.1%
8	12	Objective C	2.6%	+0.79%
9	10	JavaScript	2.6%	-0.08%
10	8	Perl	1.9%	-1.69%

1.3. Caractéristiques du langage

Les principales caractéristiques de **Python** :

- **Python** évolue rapidement, il en est à ce jour à la version 3.2. Pour un numéro majeur de version donné (version 3 actuellement), les évolutions d'un numéro mineur de version à l'autre (par exemple de 3.0 à 3.1) sont *toujours* des améliorations du langage et des corrections de bogues. Le principe de compatibilité ascendante est généralement respecté : un programme qui fonctionne en **Python** 2.6 fonctionnera au pire aussi bien en version 2.7, et en principe mieux. Il n'en va pas de même pour un changement de version majeure : la compatibilité ascendante n'est plus respectée. Ainsi, les nombreux changements entre les versions 2 et 3 de **Python** risquent de poser quelques problèmes de conversion.
- **Python** est portable, non seulement sur les différentes variantes d'*Unix*, mais aussi sur les OS propriétaires : *MacOS* et les différentes variantes de *Windows*. De plus, les *bytecodes* (fichiers

compilés automatiquement) générés sur un système sont portables tels quels d'un système à l'autre.

- **Python** est un logiciel libre, que l'on peut utiliser sans restriction dans des projets commerciaux. Sa licence d'utilisation (*Python Software Foundation*), est analogue la licence *BSD*, une des plus permissive dans le domaine informatique, qui permet une grande liberté d'exploitation.
- **Python** définit des structures de données usuelles et utiles (listes, dictionnaires, chaînes de caractères) comme faisant partie du langage lui-même.
- **Python** possède de nombreuses extensions et paquetages de grande qualité (*Qt*, *VTK*, *SQL*...).
- **Python** est inter-opérable, extensible (avec *Fortran*, *Java*, *C*, *C++*...).
- **Python** convient aussi bien à des scripts d'une dizaine de lignes qu'à des projets complexes.
- La syntaxe de **Python** est très simple et combinée à des types de données évolués (listes, dictionnaires,...). Les programmes **Python** sont en général très compacts et très lisibles. A fonctionnalités égales, un programme **Python** est toujours plus court qu'un programme *C++* (ou même *Java*) équivalent. Le temps de développement est également beaucoup plus court et la maintenance largement facilitée.
- **Python** gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de comptage de références (proche, mais différent, d'un *garbage collector*). Il n'y a pas de pointeurs explicites en **Python**.
- **Python** est (en option) *multi-threadé*.²
- **Python** est orienté-objet. Il supporte l'héritage multiple et la surcharge des opérateurs. Dans son modèle objets, et en reprenant la terminologie de *C++*, toutes les méthodes sont virtuelles.
- **Python** intègre un système d'exceptions, qui permet d'améliorer considérablement la gestion des erreurs.
- **Python** est dynamique (la fonction `eval()` peut évaluer des chaînes de caractères représentant des expressions ou des instructions **Python**).
- L'utilisation de la fonction `property()` ou des décorateurs permettent de définir des variables privées comme en *C++*.
- **Python** est orthogonal (un petit nombre de concepts suffit à engendrer des constructions très riches).
- **Python** est réflexif (il supporte la métaprogrammation, par exemple les méthodes `setattr()`, `delattr()`, `getattr()` permettent à un objet de se rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution).
- **Python** est introspectif (un grand nombre d'outils de développement, comme le *debugger* ou le *profiler*, sont implantés en **Python** lui-même).
- **Python** est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.

²Wikipedia : un *thread* ou processus léger, également appelé fil d'exécution (autres appellations connues : unité de traitement, unité d'exécution, fil d'instruction, processus allégé), est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les *threads* d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'appel.

- **Python** possède une librairie standard très fournie³, proposant une grande variété de services : chaînes de caractères et expressions régulières, services *Unix* standard (fichiers, pipes, signaux, sockets, threads...), protocoles Internet (*Web*, *News*, *FTP*, *CGI*, *HTML*...), persistance et bases de données, interfaces graphiques.
- *Jython*, est une variante de **Python** écrite en *Java* et génère du bytecode *Java*.
- *cython* est une surcouche de **Python** contenant le langage **Python** augmenté d'extensions, proches du *C*, qui permettent d'optimiser la vitesse d'exécution de boucles, de fonctions ou de classes sensibles.

1.4. Domaines d'application

Les domaines d'application naturels de **Python** incluent entre autres :

- L'apprentissage de la programmation objet.
- Les scripts d'administration système ou d'analyse de fichiers textuels, traitements de fichiers par lot.
- Le prototypage rapide d'applications. L'idée générale est de commencer par écrire une application en **Python**, de la tester (ou de la faire tester par le client pour d'éventuelles modifications du cahier des charges). Trois cas peuvent alors se présenter
- Le calcul scientifique et l'imagerie. **Python** est utilisé comme langage « glue » pour enchaîner les traitements par différents programmes. Par exemple pour une simulation numérique, un mailleur produit un maillage, repris par un code éléments finis pour simulation, dont les résultats sont à leur tour exploités pour visualisation. **Python** ne sert alors pas à écrire les algorithmes, mais à combiner et mettre en oeuvre rapidement des bibliothèques de calcul écrites en langage compilé (*C*, *C++*, *Fortran*, *Ada*,...).
- Tous les développements liés à l'*Internet* et en particulier au *Web* : scripts *CGI*, navigateurs *Web*, moteurs de recherche, agents intelligents, objets distribués etc..
- L'accès aux bases de données (relationnelles).
- La réalisation d'interfaces graphiques utilisateurs. La plupart des bibliothèques de widgets (*Qt4*, *wxWidget*, *GTK*, etc.) sont interfacées pour **Python**.
 - ▷ Les performances sont satisfaisantes, après optimisation éventuelle du code **Python**. On livre alors le produit tel quel au client.
 - ▷ Les performances ne sont pas satisfaisantes, mais l'analyse de l'exécution du programme (à l'aide du profiler de **Python**) montre que l'essentiel du temps d'exécution se passe dans une petite partie du programme. Les fonctions, ou les types de données, correspondants sont alors réécrits en *C* ou en *C++*, sans modification du reste du programme.
 - ▷ Sinon, il est toujours possible de réécrire tout le programme, en utilisant la version **Python** comme un brouillon.

³voir <http://docs.python.org/library/>

1.5. Conventions

Afin d'inciter le lecteur à tester les instructions proposées, et peut-être de piquer sa curiosité, les résultats de certaines instructions ne sont pas données en clair dans le texte, au moins dans les premiers chapitres.

- Les instructions précédées d'un \$ sont à exécuter en mode terminal *Unix*.
- Les instructions précédées de >>> sont à exécuter depuis une session *Python*.
- Les instructions, variables, mots-clé, etc. du langage *Python* sont écrits en chasse fixe.
- Les mots-clé de *Python* sont en chasse fixe et en caractères gras : **class**, **def**, **return**, **if**, etc.

1.6. La documentation Python

La documentation *Python* est très riche et bien organisée. Voici quelques liens indispensables :

- le document de base :
<http://docs.python.org>
- un tutoriel :
<http://docs.python.org/tutorial>
- le manuel de référence :
<http://docs.python.org/library/index.html>
- une bonne entrée en matière pour comprendre les concepts :
<http://diveintopython.org/>
- un résumé complet de la syntaxe sous forme compacte :
<http://rgruet.free.fr/#QuickRef>
<http://www.python.org/doc/QuickRef.html>

1.7. Style de programmation

Conventions de nommage. Dans un programme, une bonne « convention de nommage » peut éviter bien des migraines car si l'on n'y prend garde, rien n'empêche des variables d'écraser des fonctions si elles portent le même nom. Il faut donc utiliser une convention qui minimise les chances de conflit.

Il est important de donner aux différentes entités d'un programme un nom « évocateur » et le moins ambigu possible.

Par exemple une fonction de « tri bulle » s'appellera `triBulle()` plutôt que `tintinAuTibet()`.

Dans la suite, les conventions de nommage adoptées sont les suivantes :

- les noms de variables sont en `minuscule_minuscule` ;
- les noms des constantes de niveau module ou package : `MAJUSCULE_MAJUSCULE` ;
- les noms des fonctions et des méthodes (si possible des verbes) : les mettre en `MinMaj()` ;
- les noms des fichiers et des modules : `minusculeminuscule.py` ;

- les noms de classe : `MajusculeMajuscule` on dit aussi *camel case* ;
- les noms des données (ou attributs) sont des substantifs ou adjectifs, en `min_min` ;
- les attributs que l'on désire signaler comme privés sont préfixés par un underscore `_x`. En **Python** il n'y a pas comme en **C++** d'attribut réellement privé. Il est cependant possible de protéger l'accès aux données de divers manières comme nous le verrons.

Avec ces conventions, un nom de variable, fonction, module... porte en lui même autant de renseignements qu'il se peut, la « casse » est utilisée à bon escient. Par exemple, on est immédiatement informé que `TintinAuTibet` est une classe, `tintinAuTibet` est une méthode ou une fonction, `tintin_au_tibet` est une variable ou une donnée attribut, etc...

Conventions d'écriture. Suivant les recommandations des développeurs **Python** (sur <http://www.python.org/dev/peps/pep-0008/>), la « valeur d'indentation » standard sera de 4 espaces.

On évitera les tabulations au profit des espaces.

Les lignes feront moins de 89 caractères.

1.8. Exécution d'une session **Python** interactive

Un programme (ou des instructions) **Python** peut être exécuté de différentes manières :

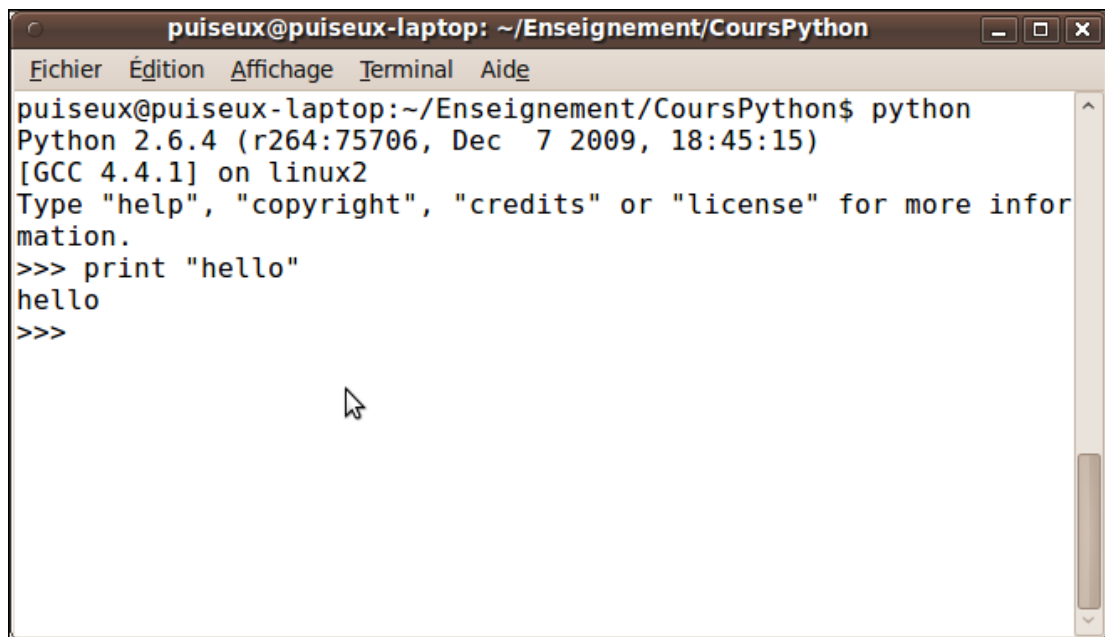
- en entrant directement les instructions après avoir lancé l'interpréteur **Python**. On parle d'exécution en ligne de commande ;
- en écrivant les instructions dans un fichier, appelé fichier module, ou scripts **Python** que l'on exécute après l'avoir sauvegardé.

Exécution en ligne de commande. Lancer la commande `python` depuis une console :

```
$ python
```

```
1 Python 3.1.2 (release31-maint, Sep 17 2010,
   20 :34 :23)
2 [GCC 4.4.5] on linux2
3 Type "help", "copyright", "credits" or "
   license" for more information.
4 >>>
```

Après diverses informations (version, date de *release*, compilateur **C** utilisé pour compiler **Python** lui même, la plateforme), le *prompt* de **Python** apparaît (les trois chevrons `>>>`, par défaut), indiquant que l'interpréteur attend une instruction. Si **Python** attend une instruction sur plusieurs lignes (par exemple si vous avez oublié de fermer une parenthèse avant d'appuyer sur la touche `return` pour valider une instruction), le *prompt* est alors `...` indiquant que **Python** attend une suite. Valider une instruction se fait en appuyant sur la touche `return` ou `enter`.

A screenshot of a terminal window titled "puiseux@puiseux-laptop: ~/Enseignement/CoursPython". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", and "Aide". The terminal content shows the execution of the "python" command, which starts the Python 2.6.4 interpreter. It displays version information and a prompt. The user enters "print 'hello'", and the output "hello" is shown. The prompt returns to ">>>".

```
puiseux@puiseux-laptop: ~/Enseignement/CoursPython$ python
Python 2.6.4 (r264:75706, Dec 7 2009, 18:45:15)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hello"
hello
>>>
```

Fig. 1. *Python* en ligne de commande

La combinaison de touches `Ctrl+d` met fin à une session *Python* sous *Unix*.

La combinaison de touches `Ctrl+z` met fin à une session *Python* sous *Windows* (également sous *Unix*).

Il existe des alternatives à la ligne de commande *Python* brute.

– *ipython*

permet l'exécution d'instructions *Python* en ligne de commande, de manière plus évoluée. C'est une surcouche de l'interpréteur *Python* offrant de nombreuses facilités dont une complétion automatique efficace.

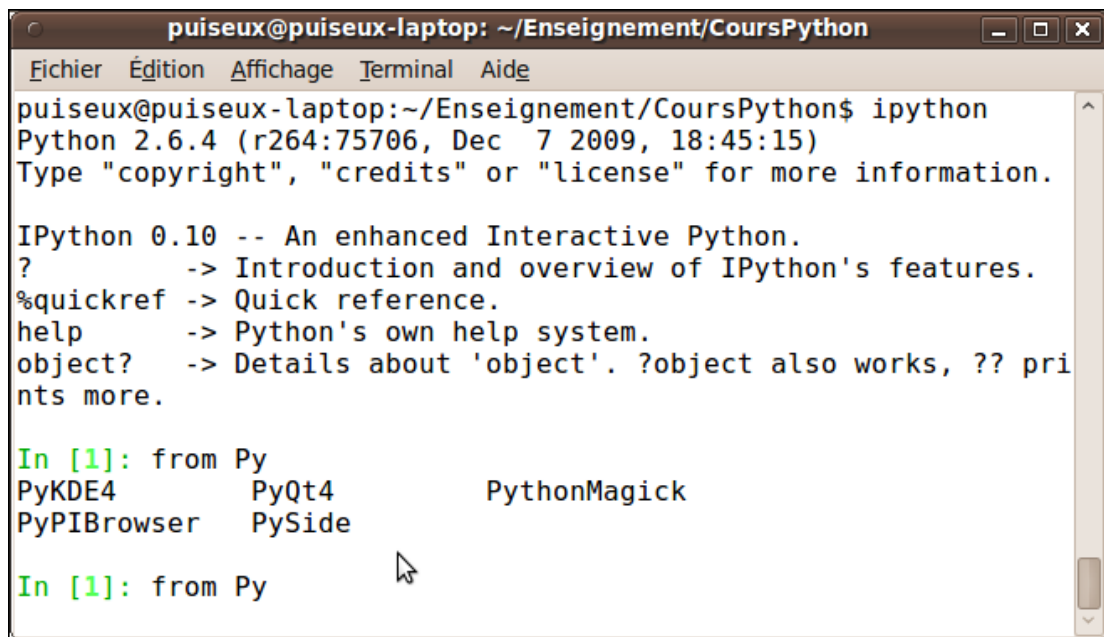
ipython propose des prompts différents de ceux de l'interpréteur de base. Pour les entrées :

In [n] :

et pour les sorties :

Out [n+1] :

où n est le numéro de l'instruction.



```
puiseux@puiseux-laptop: ~/Enseignement/CoursPython
Fichier  Édition  Affichage  Terminal  Aide
puiseux@puiseux-laptop:~/Enseignement/CoursPython$ ipython
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object'. ?object also works, ?? prints more.

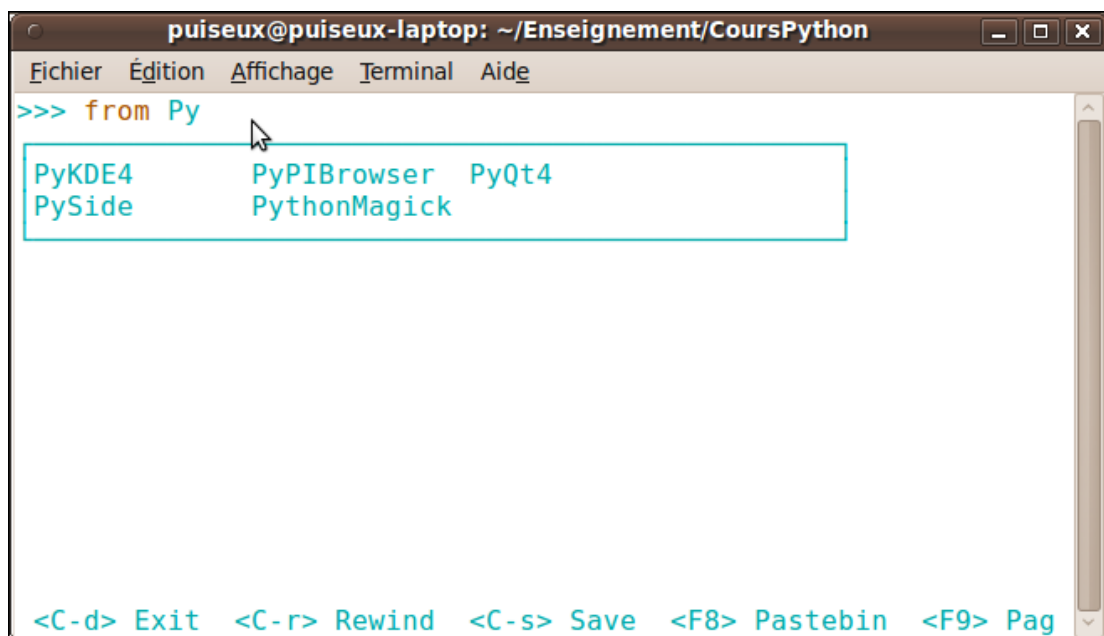
In [1]: from Py
PyKDE4      PyQt4      PythonMagick
PyPIBrowser PySide

In [1]: from Py
```

Fig. 2. ipython

– *bpython*

est analogue à *ipython*, en plus léger, et plus commode. Il propose en plus de *ipython* une coloration syntaxique sur la ligne de commande.



```
puiseux@puiseux-laptop: ~/Enseignement/CoursPython
Fichier  Édition  Affichage  Terminal  Aide
>>> from Py
PyKDE4      PyPIBrowser PyQt4
PySide      PythonMagick

<C-d> Exit <C-r> Rewind <C-s> Save <F8> Pastebin <F9> Pag
```

Fig. 3. bpython

Bien entendu, la ligne de commande *Python* est insuffisante pour produire un programme consistant. *Python* propose différents modes de programmation et d'exécutions.

Outre la ligne de commande, on trouve de nombreux éditeurs et environnements dédiés à *Python*.

Exécution par script (ou module). L'ensemble des instructions et définitions diverses, sauvegardées dans un fichier `tests.py` s'appelle le module `tests`.

On parle de *module* lorsque le fichier contient essentiellement des définitions de constantes, fonctions ou classes et de *script* lorsque le fichier contient essentiellement des instructions directement exécutables. La frontière entre script et module est floue, d'autant qu'un module bien constitué sera toujours exécutable comme un script dans lequel sont testés tous les composants (fonctions, classes, etc.) du module.

Disposant d'un script nommé `tests.py` on peut lancer de plusieurs manières :

- en passant la commande *Unix*
\$ `python tests.py`
- en lançant directement
\$ `tests.py`
à condition qu'il ait été rendu exécutable avec la commande *shell*
\$ `chmod +x tests.py`
et que sa première ligne soit :

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

ce qui permet au *shell* de savoir quel interpréteur il doit utiliser (ici *Python*).

- dans une session *Python*, on peut importer le module ou bien un élément de ce module :

```
1 >>> from tests import uneFonction
2 >>> uneFonction()
```

Un module ou script *Python* (par exemple le fichier `tests.py`) aura la structure suivante :

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 import os
4
5 print ("La variable __name__ vaut : ",__name__)
6
7 def inters(l1,l2) :
8     """Intersection de deux listes"""
9     return list(set(l1)&set(l2))
10
11 if __name__=="__main__" :
12     print (os.listdir('.'))
```

```
13 print (inters([1, 2, 3], [2, 3, 4]))
```

- (1) Indique au shell que l'interpréteur à utiliser pour exécuter ce module est **Python**
- (2) Codage des caractères. Des caractères inconnus sont considérés comme une erreur de syntaxe.⁴
- (3) Importation des modules (et des fonctions, variables, classes) utiles.
- (4)
- (5) Une instruction exécutable
- (6)
- (7) Une fonction.
- (8) La documentation de la fonction.
- (9) Le corps de la fonction.
- (10)
- (11) En **Python**, tout espace de nom est nommé par un champ `__name__`. Lors de l'exécution d'un module (et de toute session **Python**), **Python** crée un espace de noms dont le champ `__name__` a pour valeur `__main__`.
- (12) C'est la partie qui sera exécutée si `tests.py` est lancé comme un script *Unix*.
- (13) Idem

Les éditeurs Python. Ce sont des simples éditeurs de texte avec des facilités spécifiques à **Python** : coloration syntaxique, complétion automatique essentiellement etc.. La plupart des éditeurs de texte courants offrent un mode de coloration **Python** mais ne permettent pas d'exécuter une session **Python**. Sur *Gnu-Linux*, les éditeurs *kate*, *kwrite*, *gedit*, *emacs*, *vi* possèdent tous une coloration syntaxique pour **Python** et un mode de complétion automatique. *Kate* propose en plus une liste des symboles, et un terminal intégré permettant d'exécuter une session **Python**.

Les environnements complets. Ils combinent les avantages de la ligne de commande, de l'éditeur et du débogueur en proposant une console **Python**, un éditeur avec coloration syntaxique des instructions, des facilités de débogage, un mode de complétion plus évolué, la gestion de projets, le travail coopératif, etc.. Ils sont à utiliser pour des développements de projets importants. Sur les environnements *GNU-Linux* on peut citer :

- *idle*, un environnement de programmation **Python**, écrit en **Python** et *tkinter* par Van Rossum lui-même. Il est d'aspect un peu spartiate mais efficace.

⁴L'encodage par défaut des scripts et modules est *utf-8* en **Python3**. Pour utiliser un encodage différent, utiliser cette ligne. Par exemple, la plupart des éditeurs Windows sont en encodage *windows-1252*, vous devez donc insérer en tête de vos fichiers sources : `# -*- coding : windows-1252 -*-`

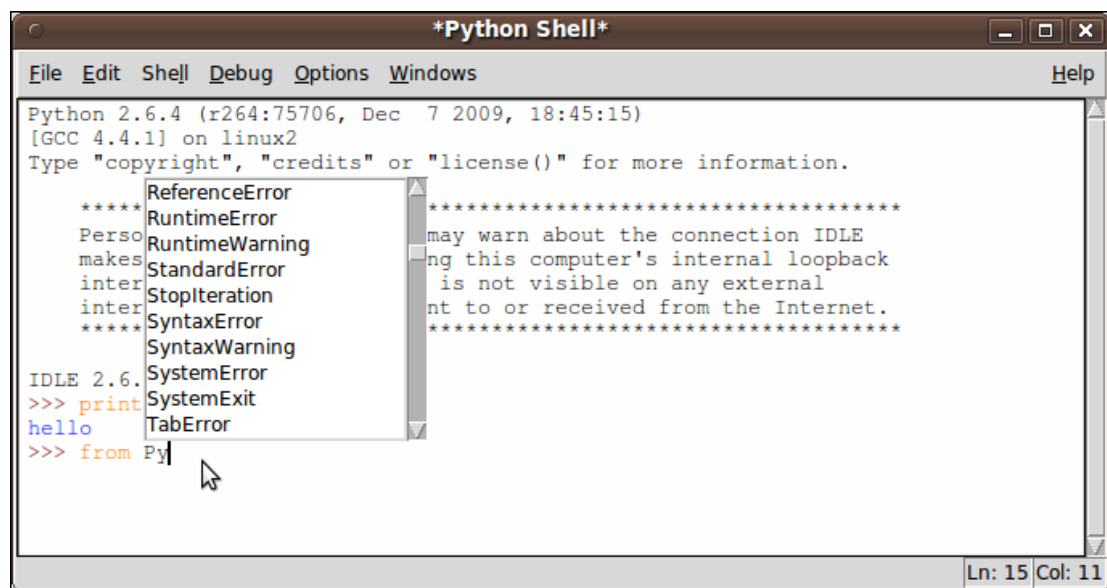


Fig. 4. idle

- *eclipse* + *PyDev* est très complet, permettant de basculer l'éditeur du mode *Python* au mode *C++* ou bien *Java*, *PHP*, ou autre. Adapté pour des gros projets, c'est un environnement très abouti. Il propose également une console *Python* interactive dans ses versions récentes (≥ 3.6).

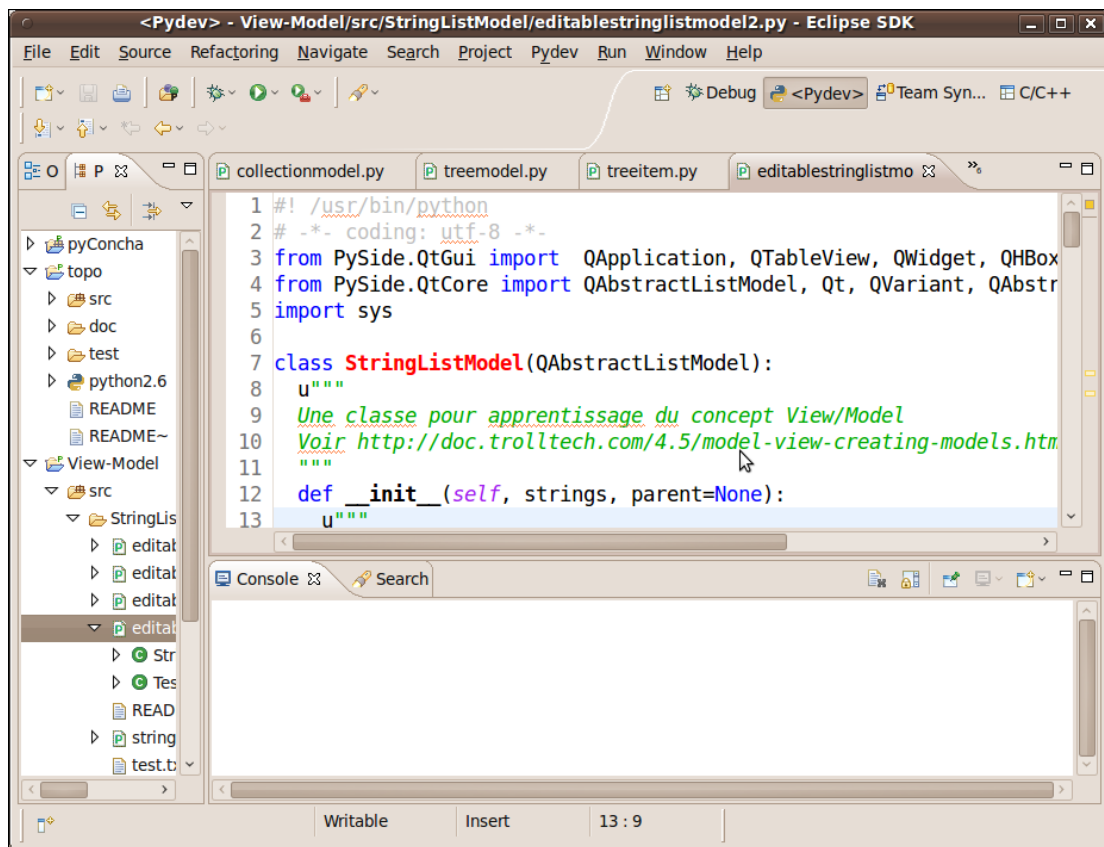


Fig. 5. Pydev

- *Spyder* est l'acronyme de « Scientific PYthon Development EnviRonment », écrit en *Python*, et basé sur *PyQt4*. C'est un environnement très complet et séduisant.

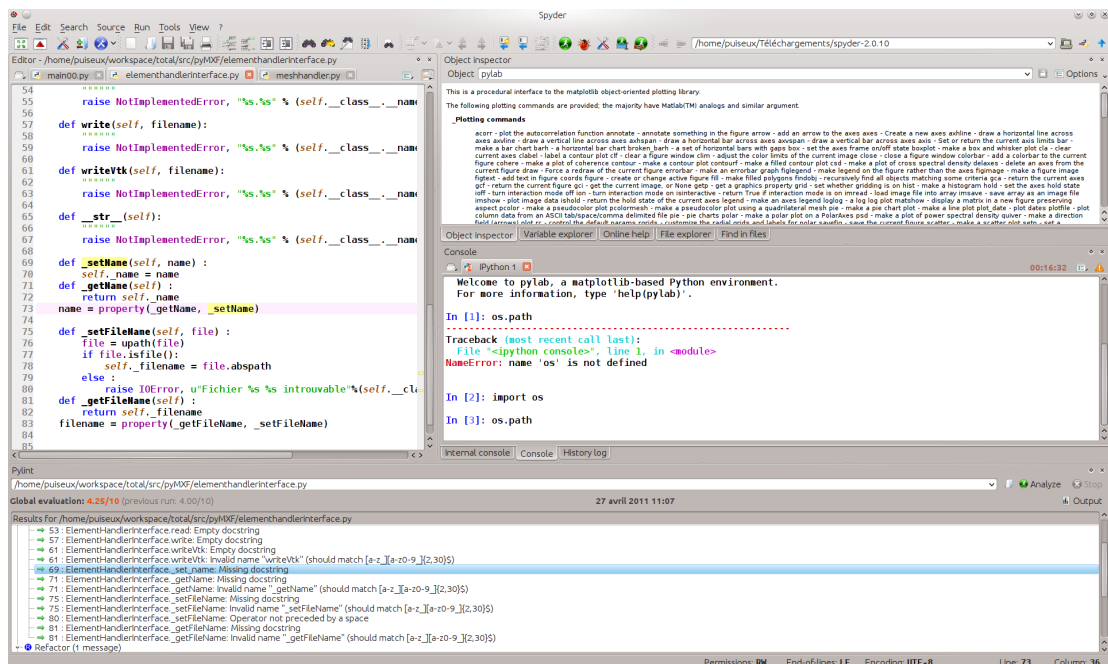


Fig. 6. Spyder

- *eric* est un environnement complet dédié exclusivement à *Python*.
- *spe* est un environnement complet spécialisé en *Python*.

Il en existe de nombreux autres. On en trouvera un comparatif à l'adresse <http://python.developpez.com/outils/Editeurs/>.

Déboguer un programme. Pour déboguer un programme en ligne de commande, on importe le module `pdb` du débogueur *Python*, puis on lance le programme à déboguer par l'instruction `pd.run("tests.py")`.

```
1 >>> import pdb
2 >>> pdb.run("tests.py")
3 (Pdb)
```

On est sous débogueur, le prompt est maintenant (Pdb). Taper ? ou *help* donne accès à la documentation sur les commandes disponibles :

```
1 (Pdb) ?
2 Documented commands (type help <topic>) :
3 =====
4 EOF    bt      cont    enable  jump    pp      run     unt
5 a      c      continue exit    l       q       s       until
6 alias  cl      d       h       list    quit    step    up
```

```

7  args clear  debug  help  n  r  tbreak w
8  b      commands disable ignore next restart u  whatis
9  break condition down j      p  return unalias where
10
11  Miscellaneous help topics :
12  =====
13  exec pdb
14
15  Undocumented commands :
16  =====
17  print retval rv
18  (Pdb) help continue
19  c(ontinue))
20  Continue execution, only stop when a breakpoint is encountered.
21  (pdb)

```

Les commandes de débogage les plus courantes sont :

- `b(reak)` 32 : mettre un point d'arrêt ligne 32
- `n(ext)` : instruction suivante, exécuter les fonctions sans y entrer
- `s(tep)` : avancer d'un pas, entrer dans les fonctions
- `p(rint)` a : écrire la valeur de la variable a
- `c(ontinue)` : continuer jusqu'au prochain point d'arrêt
- `l(ist)` : lister le source autour de la ligne courante

Il est également possible d'utiliser *winpdb*, ou encore d'invoquer *pdb*, directement depuis le *shell*

```
$ pdb toto.py ses arguments
```

ou bien si *winpdb* est installé sur votre système

```
$ winpdb toto.py ses arguments
```

Dans la plupart des environnements de programmation *Python*, le débogueur est intégré et l'utilisation en est plus intuitive qu'en ligne de commande.

Par exemple dans *idle*, pour déboguer un programme :

- (1) se mettre en mode *debug* (menu Debug>Debugger). La fenêtre de débogage apparaît.
- (2) dans le shell *Python* de *idle*, lancer l'instruction à déboguer. le débogueur prend la main et s'arrête à la première instruction. On peut placer ses points d'arrêt, avancer pas à pas, consulter les valeurs des variables, etc..

Exercices

Exercice 1. Exécution d'un script

Exécuter le script `tests.py` :

- en ligne de commande, comme un script Unix ;
- en ligne de commande, comme argument de la commande **Python**.
- dans une session **Python** importez le module `tests` et testez la fonction

Exercice 2. Déboguer un script

Exécuter le script `tests.py` sous débogueur, mettre un point d'arrêt dans la fonction `inters()` et afficher la liste `l1` :

- en ligne de commande
- en utilisant le débogueur de idle

Exercice 3. Documentation.

- (1) Quel chapitre de la documentation un programmeur doit-il garder sous son oreiller (=Pillow) ? Mettez ce lien en signet dans votre navigateur.
- (2) Que signifie « PEP » pour un programmeur **Python** averti ? Trouver un PEP dont le titre est The Zen of Python, où sont décrits, sous forme d'aphorismes, les grands principes qui gouvernent l'évolution du langage.
- (3) Trouvez les principales différences de syntaxe entre **Python 3.1** et **Python 2.6**.
- (4) Trouver la documentation de la fonction `input()`.

Solutions des exercices

Solution 1 Exécution d'un script

- en ligne de commande, comme un script Unix ;

```
$ chmod +x tests.py  
$ tests.py
```
- en ligne de commande, comme argument de la commande **Python** :

```
$ python tests.py
```
- On peut également l'importer dans une session **Python** :

```
1 >>> import sys  
2 >>> sys.path.append("ch1-intro/")  
3 >>> import tests  
4 'La variable __name__ vaut : tests'  
5 >>> print(tests.inters([1,2,3],[1,3,6,9]))  
6 [1, 3]
```

ligne 2 : on ajoute aux chemins de recherche des module, le répertoire contenant `tests.py` avant d'importer `tests`

Solution 2 Exécution d'un script

– en ligne de commande,

```
$ pdb tests.py
> /home/puiseux/Python/ch1-intro/tests.py(3)<module>()
-> import os
(Pdb) b inters
Breakpoint 1 at /home/puiseux/Ellipses/Python-
Ellipse/fromMacOS/Python-Ellipse/ch1-intro/tests.py :7
(Pdb) continue
La variable __name__ vaut : __main__
['tests.py', 'a.out', 'liste.cpp~', 'tests.pyc', 'liste.cpp']
> /home/puiseux/Python/ch1-intro/tests.py(9)inters()
-> return list(set(l1)&set(l2))
(Pdb) print l1
[1, 2, 3]
(Pdb) continue
[2, 3]
The program finished and will be restarted
(Pdb) quit
```

– depuis *idle*. On se positionne d'abord dans le répertoire `ch1-intro/` puis on lance *idle* et on active le débogueur dans menu `Debug>Debugger`

```
>>> Python 3.2 (r32 :88452, Feb 20 2011, 11 :12 :31)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "copyright", "credits" or "license()" for more informa-
tion.
>>>
[DEBUG ON]
>>> import tests
```

Solution 3 Documentation.

(1) Dans <http://docs.python.org/py3k/> on lit ceci :

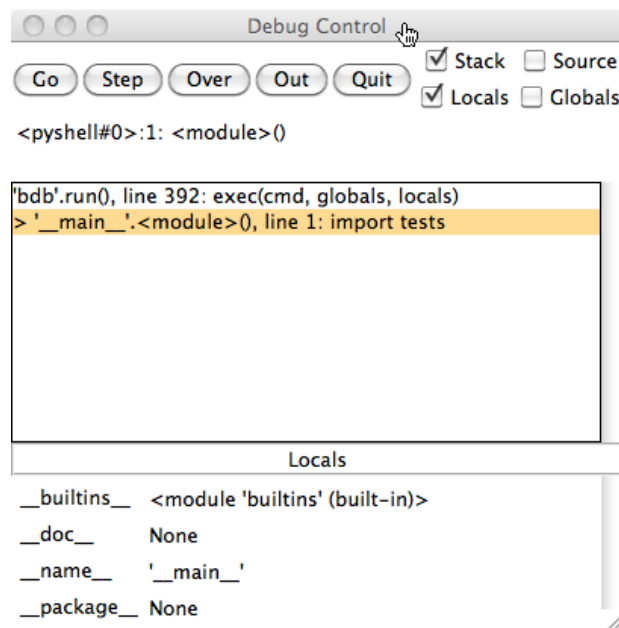
Library Reference

keep this under your *pillow*

Il s'agit du manuel de référence de la bibliothèque standard *Python*.

(2) PEP signifie *Python Enhancement Proposals* (PEPs). Voir sur

<http://www.python.org/dev/peps/>

Fig. 7. Débogueur *idle*

Ce sont des propositions d'amélioration du langage, soumises et discutées par les utilisateurs avertis, et sélectionnées par Guido Van Rossum.

Le PEP 20 a pour titre *The Zen of Python*, on peut le trouver à :

<http://www.python.org/dev/peps/pep-0020/>

(3) Le document

<http://docs.python.org/release/3.1/whatsnew/index.html>

fournit toutes les différences entre les différentes versions de **Python**.

(4) Une recherche sur internet avec les mots clé « python input » renvoie sur la page

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

contenant une documentation complète.

CHAPITRE 2

Premiers pas

Dans ce chapitre, on ne se préoccupe pas d'être exhaustif, ni rigoureux, mais plutôt de faire un tour d'horizon du langage et de tester quelques instructions élémentaires afin de s'imprégner de l'esprit et des constructions *Python*. Dans les chapitres suivants, nous reviendrons plus en détail sur tous les éléments présentés ici.

Ouvrez une session *Python* puis testez, jouez avec les instructions suivantes.

2.1. Pour vérifier votre installation

vous pouvez tester ces instructions :

```
1 >>> import sys
2 >>> sys.argv
3 ['']
```

On importe le module `sys`, et on consulte son attribut `sys.argv` qui contient les arguments de la ligne de commande quand *Python* a été lancé. Ici, la liste d'arguments est vide.

Les lignes de continuation sont nécessaires lorsqu'on saisit une construction sur plusieurs lignes. Comme exemple, dans cette instruction *if* :

```
1 >>> a = 1
2 >>> if a :
3 ...     print ("1 est vrai")
4 ...
```

On peut customiser ses prompts, par exemple les mettre en couleur¹ :

```
1 >>> sys.ps1 = 'moi##> '
2 moi##> sys.ps1 = '\033[95m>>> \033[0m'
3 >>> sys.ps2 = '$$$ '
4 >>> help(
5 $$$ )
```

Pour interrompre une instruction trop longue à s'exécuter taper `Ctrl+c`

¹Dans *Ipython*, ainsi que dans *idle* il semble que `sys.ps1` et `sys.ps2` n'existent plus

```
1 >>> while 1 :
2 ...     print(3.14)
3 <Ctrl+c>
4 Traceback (most recent call last) :
5     [...]
6 KeyboardInterrupt
```

2.2. Aide en ligne, dir() et help()

Une fonction importante est `help()` qui, comme son nom l'indique, donne accès à l'aide en ligne :

```
1 >>> help
2 Type help() for interactive help, or help(object) for help about
   object.
```

`help(obj)` permet de consulter l'aide sur l'objet `obj`.

Pour consulter l'aide sur le module `sys` :

```
1 >>> import sys
2 >>> help(sys)
3 Help on built-in module sys :
4 NAME
5 sys
6 FILE
7 (built-in)
8     [etc ...]
9 warnoptions = []
```

On voit que l'aide est consistante, et décrit en particulier les fonctions du module `sys` que l'on a importé auparavant.

La fonction `dir()` sans paramètre, donne la liste des noms accessibles dans la portée actuelle. Pour savoir ce que représentent ces noms, tapez simplement le nom :

```
1 >>> dir()
2 ['__builtins__', '__doc__', '__name__', '__package__', 'help', 'os',
3  'sys']
4 >>> sys
5 <module 'sys' (built-in)>
```

On y voit en particulier le module `sys` importé, et que c'est un module *builtin*.

Avec un *objet* en paramètre, `dir(objet)` retourne une liste de tous les attributs (méthodes et données) valides de cet objet. Le nombre entier 1 est un objet *Python*, en *Python* tout est objet au sens programmation orientée objet. À ce titre, tout objet possède des attributs. Les attributs d'une variable peuvent être consultés :

```
1 >>> dir(1)
2 ['__abs__', etc..., 'numerator', 'real']
3 >>> a = 1
4 >>> dir(a)
5 ['__abs__', etc..., 'numerator', 'real']
6 >>> a.numerator
7 1
```

D'ailleurs, pour savoir ce que fait réellement la fonction `dir()`, on peut demander de l'aide :

```
1 >>> help(dir)
2 Help on built-in function dir in module builtins :
3 dir(...)
4     dir([object]) -> list of strings
5     [...]
```

2.3. Commentaires

En *Python*, est commentaire (ignoré par l'interpréteur) tout ce qui va du caractère `#` à la fin de la ligne, à condition que le `#` ne fasse pas partie d'une chaîne de caractères.

```
1 >>> # voici le premier commentaire
2 >>> truc = 1 # et voici le deuxième commentaire
3 >>> STRING = "# Ceci nest pas un commentaire \
4             mais une chaîne de caractères \
5             sur trois lignes "
```

2.4. Indentation

Dans les langages comme *C++*, ou *Pascal*, *Java* le corps d'une boucle est déterminé une paire d'accolade `{}` ou bien `begin...end`

En *Python*, c'est l'indentation qui est interprétée pour définir le corps de la boucle.

Si l'on démarre le corps de la boucle avec une indentation de 4 espaces (ce qui est la valeur recommandée), alors toutes les autres instructions du corps de la même boucle devront être indentées de

4 espaces. C'est une erreur de syntaxe d'indenter deux instructions d'une valeur différente dans un même corps de boucle.

Par exemple ce programme confus contiennent au moins trois erreurs de syntaxe :

```

1 >>> L = '€€€€€€€€$$$$$$$$$$$'
2 >>> n= 0
3 >>>     for c in L :
4 ...     print c
5 ...     if c == '€' :
6 ...     print 'encore un euro'
7 ...         n= n+1
8 ...         print ('comptage terminé')

```

il est donc illisible pour l'interpréteur *Python* et illisible pour l'œil humain.

Alors que le programme qui suit est beaucoup plus clair et exempt d'erreur de syntaxe.

```

1 >>> L = '€€€€€€€€$$$$$$$$$$$'
2 >>> n = 0
3 >>> for c in L :
4 ...     if c == '€' :
5 ...         print ('encore_un_euro')
6 ...         n += 1
7 ...     print('comptage_terminé')

```

Python nous *oblige* à présenter les programmes de manière lisible pour lui *et* pour l'œil humain.

Attention de ne pas mélanger les tabulations et les espaces car cela donne lieu à des erreurs de compilation qui peuvent paraître incompréhensibles.

2.5. Variables, types

On peut déclarer des variables dont le type est déterminé à l'affectation. La fonction intégrée `type()` permet de connaître le type (ou la classe) d'une variable :

```

1 >>> a = 2
2 >>> type(a)
3 int
4 >>> x = 3.14
5 >>> type(x)
6 float
7 >>> import math
8 >>> x - math.pi
9 -0.0015926535897929917

```


a est de type entier, x de type réel.

Le module `math` contient la constante `math.pi`.

Une variable *Python* est constituée d'un nom, un type et une valeur.

L'opérateur d'affectation est `'='`. Une instruction comme

```
1 a = 2
2 l = [1, 2, 3]
```

affecte la valeur entière 2 à la variable a. Ce qui signifie que a est un nom donné à l'entier 2, et a pour type `int`. La variable a est donc (`'a'`, `int`, 2)

De même l est un nom donné à la liste [1, 2, 3], de type `list`.

On peut faire une affectation multiple de deux manières différentes :

```
1 >>> x = y = z = 0 # Mettre a zero x, y et z
2 >>> x, y, z = 0, 0, 0 #idem
3 >>> x
4 0
```

Il est possible de déclarer des variables plus élaborées comme des tuples, listes, dictionnaires :

```
1 >>> T = (1, 2, 3)
2 >>> type(T)
3 <class 'tuple'>
4 >>> L = [1, 2, 3]
5 >>> type(L)
6 <class 'list'>
7 >>> D = {}
8 >>> type(D)
9 <class 'dict'>
```

On examinera plus loin ces objets *Python*.

Le simple fait d'entrer au clavier le nom d'une variable affiche son contenu. Une alternative consiste à utiliser la fonction `print()`.²

```
1 >>> a, b = 1, 3.14
2 >>> a, b
3 (1, 3.14)
4 >>> print("a et b valent" , a, 'et', b)
5 a et b valent 1 et 3.14
```

²En *Python* 2 `print` est une *instruction* et utilise la syntaxe `print` x sans parenthèse.

Outre les entiers (`int`) et les réels (`float`), les complexes (`complex`) font également partie du langage de base de *Python* :

```
1 >>> 1j * 1j
2 (-1+0j)
3 >>> 1j * complex(0,1)
4 (-1+0j)
5 >>> 3+1j*3
6 (3+3j)
7 >>> (3+1j)*3
8 (9+3j)
9 >>> (1+2j)/(1+1j)
10 (1.5+0.5j)
11 >>> a = 1.5+0.5j
12 >>> a.real
13 1.5
14 >>> a.imag
15 0.5
16 >>> abs(a)
17 1.5811388300841898
18 >>>
```

Quelques variables sont prédéfinies :

```
1 >>> import math, sys
2 >>> math.pi
3 3.141592653589793
4 >>> sys.float_info[0]
5 1.7976931348623157e+308
6 >>> sys.float_info.min
7 2.2250738585072014e-308
```

La variable `_` (« underscore ») contient la dernière valeur calculée :

```
1 >>> tva = 12.5 / 100
2 >>> prix = 100.50
3 >>> prix * tva
4 12.5625
5 >>> prix + _
6 113.0625
```

2.6. Opérations

Les opérateurs suivants sont définis pour les types `int`, `float`, et `complex` :

`x+y` : addition

`x-y` : soustraction

`x*y` : multiplication

`x/y` : division

`x//y` : division entière (`float` et `int` seulement)

`x%y` : reste de la division entière (`float` et `int` seulement)

`x**y` : élévation à la puissance x^y

Les opérations arithmétiques se comportent de manière conventionnelle, même la division.³

```
1 >>> 2*5.1
2 10.2
3 >>> 0+3.14
4 3.14
5 >>> 3**2
6 9
7 >>> pow(3,2)
8 9
9 >>> 7/3
10 2.3333333333333335
11 >>> 7.0/3.0
12 2.3333333333333335
```

La fonction `divmod(a, b)` retourne un couple de valeurs (quotient, reste), résultat de la *division entière* de a par b . Il s'agit de l'unique couple (q, r) , de $\mathbb{N} \times \mathbb{R}$ vérifiant $a = bq + r$ avec $0 \leq r < b$. Les arguments a et b peuvent être réels ou entiers.

Les opérateurs `//` et `%` donnent séparément le quotient et le reste de la division entière.

```
1 >>> 7//3
2 2
3 >>> 7%3
4 1
```

L'opérateur de comparaison `==` permet de tester l'égalité de deux objets, ici des couples.

³Attention,

- en *Python* 2, la division `/` appliquée à des entiers retourne le résultat de la division *entière* ;
- en *Python* 3, l'opérateur `/` retourne toujours le résultat de la division réelle. L'opérateur `//` retourne le résultat de la division entière.

```
1 >>> divmod(7,3) == (7//3, 7%3)
2 True
```

La division entière peut s'appliquer aux nombres réels (**float**) :

```
1 >>> 7.0//3.2
2 2.0
3 >>> 7.0%3.2
4 0.59999999999999996
```

2.7. Chaînes de caractères

Les *chaînes de caractères* sont le type **str** de **Python**. Une chaîne de caractères est une suite finie de caractères placés

- entre apostrophes (ou *simples quotes*) ' ou
- entre triples apostrophes ''' ou
- entre guillemets (ou *doubles quotes*) " ou
- entre triples guillemets """.

Les chaînes délimitées par des triple apostrophes ou triple guillemets sont appelées *docstring*. Certaines fonctions (en particulier la fonction **help()**), certains modules de documentation (le module **doctest**) utilisent ces docstrings pour extraire la documentation des fonctions, classes, modules, etc..

```
1 >>> 'spam eggs'
2 'spam eggs'
3 >>> 'Une chaîne qui \
4 a une suite'
5 'Une chaîne qui a une suite'
6 >>> "une chaîne entre guillemets"
7 'une chaîne entre guillemets'
8 >>> salut = """Ceci est une chaîne plutôt longue contenant
9 plusieurs
10 lignes de texte
11     Notez que les blancs au début de la ligne et les sauts de ligne
12 sont significatifs."""
13 >>> print (salut)
14 Ceci est une chaîne plutôt longue contenant
15 plusieurs
16 lignes de texte exactement
17     Notez que les blancs au début de la ligne et les sauts de ligne
18 sont significatifs.
```

Placé en fin de ligne, le caractère `'\'` (*antislash*) sert de « carte suite » pour indiquer à l'interpréteur que la ligne courante et la suivante doivent être considérées comme une seule ligne logique. C'est le cas pour une ligne trop longue par exemple.

Pour « échapper » un caractère ayant une signification particulière, (par exemple une apostrophe `'` fermant une chaîne de caractères) on le fait précéder du caractère `'\'` (antislash ou *backslash*). Il perd sa signification particulière, et devient un caractère standard.

```
1 >>> 'n\'est-ce pas'
2 "n'est-ce pas"
3 >>> "n'est-ce pas"
4 "n'est-ce pas"
5 >>> '"Oui," dit-il.'
6 '"Oui," dit-il.'
7 >>> "\"Oui,\" dit-il."
8 '"Oui," dit-il.'
9 >>> '"N\'est-ce pas," répondit-elle.'
10 '"N\'est-ce pas," répondit-elle.'
```

Les chaînes peuvent être concaténées (accolées) avec l'opérateur `'+'`, et répétées avec `'*'` :

```
1 >>> word = 'Help' + 'A'
2 >>> '<' + word*5 + '>'
3 '<HelpAHelpAHelpAHelpAHelpA>'
```

Il n'existe pas de type *caractère* en *Python*, un caractère est simplement une chaîne de longueur un.

```
1 >>> type('a')
2 <class 'str'>
3 >>> type('aaa')
4 <class 'str'>
```

La fonction intégrée `len()`, appliquée à une chaîne de caractères retourne le nombre de caractères de la chaîne :

```
1 >>> len('anticonstitutionnellement')
2 25
```

Les éléments d'une chaîne de caractère `ch` sont indexés (numérotés) de 0 à `len(ch) - 1`. On peut accéder à chaque caractère par l'opérateur *crochet* `[]` :

```
1 >>> ch = 'anticonstitutionnellement'
2 >>> ch[0]
3 'a'
4 >>> ch[len(ch)] #ouuuups
5 Traceback (most recent call last) :
6   [...]
7   ch[len(ch)] #ouuuups
8 IndexError : string index out of range
```

Les éléments d'une chaîne de caractères sont *non modifiables*.

```
1 >>> ch[0] = 'A' #ouuuups
2 Traceback (most recent call last) :
3   [...]
4   ch[0] = 'A' #ouuuups
5 TypeError : 'str' object does not support item assignment
```

On peut adresser les éléments d'une chaîne à partir de la fin, avec des indices négatifs, le dernier élément porte le numéro -1

```
1 >>> ch[-2]
2 'n'
```

On peut sélectionner des *tranches (slices)* d'une chaîne. Attention, le dernier élément n'est pas pris en compte : `ch[1 :3]` retourne `ch[1]`, `ch[2]`

```
1 >>> ch[1 :4]
2 'nti'
```

On peut découper avec des indices négatifs :

```
1 >>> ch[1 :-4]
2 'nticonstitutionnelle'
```

Dans un *slice* (caractérisé par l'opérateur de *slicing* ' : '), si une valeur manque d'un côté ou de l'autre de ' : ', elle désigne *tout* ce qui suit, ou *tout* ce qui précède :

```
1 >>> ch[5 :]
2 'onstitutionnellement'
```

2.8. Listes

Les listes sont représentées par le type `list` en *Python*. C'est une liste de valeurs (éléments) entre crochets et séparés par des virgules. Les éléments d'une même liste n'ont pas nécessairement le même type :

```
1 >>> a = ['Ossau', 2884, 'Mont Blanc', 4807]
2 >>> a
3 ['Ossau', 2884, 'Mont Blanc', 4807]
```

Comme les indices des chaînes, les indices des listes commencent à 0, et les listes peuvent être découpées, concaténées, et ainsi de suite :

```
1 >>> a[0]
2 'Ossau'
3 >>> a[-2]
4 'Mont Blanc'
5 >>> a[1 :-1]
6 [2884, 'Mont Blanc']
7 >>> a[:2] + ['Soum de Rey', 1526]
8 ['Ossau', 2884, 'Soum de Rey', 1526]
9 >>> 3*a[:2]
10 ['Ossau', 2884, 'Ossau', 2884, 'Ossau', 2884]
```

A la différence des chaînes, qui sont non-modifiables, les listes sont modifiables : il est possible de changer les éléments individuels d'une liste :

```
1 >>> a[2] = ''
2 >>> a
3 ['Ossau', 2884, '', 4807]
```

L'affectation dans des tranches est aussi possible, et cela peut même changer la taille de la liste :

```
1 >>> a = [1,2,3]
2 >>> a[0 :2] = [1, 12]
3 >>> a
4 [1, 12, 3]
5 >>> a[0 :2] = []
6 >>> a
7 [3]
8 >>> a[1 :1] = ['pomme', 'tomate']
9 >>> a
10 [3, 'pomme', 'tomate']
11 >>> a[:0] = a
```

```

12 >>> a
13 [3, 'pomme', 'tomate', 3, 'pomme', 'tomate']

```

La fonction intégrée `len()` s'applique aussi aux listes :

```

1 >>> len(a)
2 6

```

Il est possible « d'emboîter » des listes (créer des listes contenant d'autres listes) :

```

1 >>> q = [2, 3]
2 >>> p = [1, q, 4]
3 >>> len(p)
4 >>> p[1][0]
5 2

```

Dans l'exemple précédent, `p[1]` et `q` se réfèrent réellement au *même* objet.

Dans l'exemple suivant, on teste cette assertion, par l'utilisation du mot-clé `is`, puis en changeant une valeur de la liste `q` :

```

1 >>> p[1] is q
2 True
3 >>> q[0] = '000H'
4 >>> p
5 [1, ['000H', 3], 4]

```

La méthode de liste `append()` ajoute en fin de liste *un* élément tel quel. Contrairement à l'opérateur `+` qui ajoute *les* éléments d'une autre *liste*.

```

1 >>> p = [1, 2, 3]
2 >>> p.append('extra')
3 >>> p
4 [1, 2, 3, 'extra']

```

Une chaîne de caractères *n'est pas* une liste, mais on peut la transformer en liste :

```

1 >>> p = [1, 2, 3]
2 >>> p = [1,2,3]+'extra'
3 Traceback (most recent call last) :
4   [...]
5     p = [1,2,3]+'extra'
6   TypeError : can only concatenate list (not "str") to list
7 >>> p = [1,2,3]+list('extra')
8 >>> p

```



```
9 [1, 2, 3, 'x', 't', 'r', 'a']
```

Lorsque l'on tente d'exécuter une instruction illicite, une *exception* est levée, et l'interpréteur affiche des lignes commençant par Traceback, citant l'instruction mise en cause, pour finir par le nom de l'exception (ici **TypeError**) et un message associé. Ces messages sont en général explicites et suffisent à comprendre l'erreur. Ici on a tenté de concaténer une chaîne à une liste, ce qui a déclenché l'exception car on ne peut concaténer qu'une liste à une liste.

On peut supprimer des éléments dans une liste, la vider :

```
1 >>> p = [1,2,3]
2 >>> p[1]=[]
3 >>> p
4 [1, [], 3]
5 >>> p[ :] = [] #ou p = []
```

2.9. Dictionnaires

Comme un dictionnaire de traduction français-anglais par exemple, les dictionnaires *Python* fonctionnent par couple *clé : valeur*

Les opérations principales sur un dictionnaire sont :

– La création :

```
1 >>> alt = {'Rakaposhi' : 7788, 'Moule de Jaout' : 2050}
```

– le stockage d'une valeur à l'aide d'une certaine clé,

```
1 >>> alt['Vignemale'] = 3298
```

– l'extraction de la valeur en donnant la clé,

```
1 >>> alt['Rakaposhi']
2 7788
```

– la suppression de couples (clé : valeur) avec l'instruction *del* ou la méthode *dict.pop()*. Cette dernière supprime *et renvoie* l'entrée correspondant à la clé donnée en argument :

```
1 >>> del alt['Vignemale']
2 >>> alt.pop('Rakaposhi')
3 7788
4 >>> alt
5 {'Moule de Jaout' : 2050}
```

L'accès à la liste des clés, des valeurs et des couples (clé,valeur) est assuré par les méthodes suivantes :

- `keys()` retourne une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre quelconque (pour obtenir une liste triée, appliquer la méthode `sort()` à la liste des clés). L'expression `key in mondict` retourne `True` si la clé `key` est dans le dictionnaire `mondict`⁴;
- `values()` renvoie une liste des valeurs ;
- `items()` retourne la liste des couples (clé, valeur)

Voici un exemple d'utilisation :

```
1 >>> alt = {'Rakaposhi' : 7788, 'Moule de Jaout' : 2050, 'Vignemale' : 3298}
2 >>> alt.keys()
3 dict_keys(['Vignemale', 'Moule de Jaout', 'Rakaposhi'])
4 >>> alt.values()
5 dict_values([3298, 2050, 7788])
6 >>> alt.items()
7 dict_items([('Vignemale', 3298), ('Moule de Jaout', 2050), ('Rakaposhi', 7788)])
8 >>> 'Rakaposhi' in alt
9 True
```

2.10. Lecture-écriture sur fichier

La fonction intégrée `open()` permet d'ouvrir un fichier texte ou binaire, en lecture ou en écriture. Elle renvoie le descripteur de fichier.

```
1 >>> f = open('toto.txt')
```

`f` est un fichier texte, ouvert en mode lecture, ce sont deux valeurs par défaut.

Pour ouvrir un fichier en écriture, on le précise avec le mode `'w'` :

```
1 >>> f = open('toto.txt', 'w')
```

Attention, si le fichier `toto.txt` existe, son contenu est *irréremédiablement* perdu.

Pour ouvrir un fichier binaire en lecture, on écrira :

```
1 >>> f = open('toto', 'b')
```

et en écriture :

⁴*Python* 2 proposait la méthode `has_key()` qui n'existe plus en *Python* 3.

```
1 >>> f = open('toto', 'wb')
```

Lire un fichier texte et transférer tout son contenu dans une variable chaîne de caractères se fait par l'instruction :

```
1 >>> S = f.read()  
2 >>> f.close()
```

On peut préférer une lecture de fichier qui retourne la *liste* des lignes :

```
1 >>> lignes = f.readlines()  
2 >>> f.close()
```

Lorsque le fichier est trop gros pour que son contenu tienne en une variable, on peut itérer sur le fichier et lire les lignes à la demande, l'une après l'autre :

```
1 >>> f = open('toto.txt')  
2 >>> for line in f :  
3 ...     print(line)  
4 >>> f.close()
```

Pour écrire dans un fichier ouvert, on utilise la méthode `write()` du fichier :

```
1 >>> f = open('tutu.txt', 'w')  
2 >>> f.write('qui a peur du grand méchant loup ?\n')
```

La méthode `write()` écrit des *chaînes de caractères* sur le fichier, puisqu'il a été ouvert en mode texte (défaut). Elle n'ajoute pas de fin de ligne (`'\n'`).

Inutile d'essayer d'écrire des entiers ou de réels, il faut les convertir avant en chaîne, avec la fonction `str()` par exemple :

```
1 >>> f.write(str(3.14))  
2 >>> f.close()
```

Le module pickle. Pour lire et écrire les *nombre*s les méthodes `read()` et `write()` que l'on vient de voir renvoie ou écrit une chaîne de caractères. En lecture, elle devra être convertie avec `int()`, `float()`, ...

Pour sauvegarder des types de données plus complexes (listes, dictionnaires, ...), le module standard `pickle` convertit en chaîne de caractères presque n'importe quel objet *Python*. Ce processus s'appelle *pickling*.

Reconstruire l'objet à partir de sa représentation en chaîne de caractères s'appelle *unpickling*.

Entre *pickling* et *unpickling*, la chaîne de caractères représentant l'objet a pu avoir été enregistrée dans un fichier ou des données, ou avoir été envoyée à une machine éloignée via une connexion réseau.

```
1 >>> f = open('pik.txt', 'w')
2 >>> x = [1, 2, 3, 'hello', 1j]
3 >>> pickle.dump(x, f)
```

Pour « unpickler » l'objet, si *f* est un objet fichier ouvert en lecture :

```
1 >>> x = pickle.load(f)
```

pickle est le moyen standard pour enregistrer des objets *Python* et les réutiliser dans d'autres programmes ou dans une future invocation du même programme. On parle de la persistance d'un objet.

2.11. Fonctions

Pour définir une fonction en *Python*, on utilise le mot-clé *def* avec la syntaxe suivante :

```
def <fonction>(<arguments>) :
    <instructions>
    [return [<valeur>]]
```

Une fonction peut retourner explicitement une valeur ou pas, avec l'instruction *return*. Si aucune valeur n'est explicitement retournée, *Python* retourne la valeur *None*

Quelques exemples :

- Une fonction qui retourne le carré de son argument :

```
1 >>> def f(x) :
2 ...     return x**2
```

- Une fonction qui ne retourne rien mais qui inverse *in situ*⁵ l'ordre de la liste passée en argument :

```
1 >>> def reverse (L) :
2 ...     for i in range(len(L)//2) :
3 ...         L[i], L[-i-1] = L[-i-1], L[i]
4 >>> L = [0, 1, 2, 3, 4, 5]
5 >>> reverse(L)
6 >>> L
7 [5, 4, 3, 2, 1, 0]
```

⁵C'est à dire qui modifie la liste elle même

En testant la fonction, on constate que `L` a bien été modifiée.

Ligne 3 : il faut se souvenir que lors d’une affectation, le second membre est d’abord évalué intégralement, puis affecté au premier membre. Dans un langage traditionnel, pour permuter deux éléments `a` et `b`, il faudrait écrire une suite d’instructions comme :

```
w = a
a = b
b = w
```

En **Python**, une seule instruction est suffisante :

```
1 a, b = b, a
```

– Une fonction qui retourne une copie de la liste passée en argument, dont elle inverse l’ordre :

```
1 >>> def reversed(L) :
2 ...     LL = L[ :]
3 ...     reverse(LL)
4 ...     return LL
5 >>> reversed(L)
6 [0, 1, 2, 3, 4, 5]
7 >>> L
8 [5, 4, 3, 2, 1, 0]
```

Un paradigme du développement agile est la *réutilisation* des fonctions déjà écrites (par d’autres ou par vous-même). Dans l’exemple précédent, on applique ce principe, en réutilisant la fonction `reverse(L)`, déjà écrite auparavant.

L’instruction `L = L[:]` effectue une *recopie* du contenu de `L` dans `LL`.

L’instruction `LL = L` aurait affecté un nouveau nom à la liste `L`, sans en créer une nouvelle.

On constate ici que la liste `L` n’a pas été modifiée. Nous reviendrons sur le problème délicat du passage d’arguments dans la section [5.1.2 page 119](#) et des arguments modifiables ou non modifiables dans la section [4.1 page 79](#).

2.12. Documentation et tests

La programmation ne consiste pas uniquement à aligner des lignes de code, il faut également penser à les pérenniser, à les rendre *réutilisable*. La réutilisabilité est une des motivations essentielles à la genèse de la Programmation Orientée Objets.

Une approche de l'écriture du logiciel de haute qualité consiste à écrire des tests pour chaque fonction au début de son développement et à faire tourner ces tests fréquemment durant le processus de développement. Le module `doctest` fournit un outil pour examiner un module et valider les tests immergés dans les chaînes de documentation du programme. La construction d'un test consiste à copier-coller un appel typique et son résultat dans la chaîne de documentation.

Il est également important qu'un code qui fonctionne à l'instant t , continue de fonctionner après une modification. Pour s'en assurer, on peut (ré-)imaginer toute une batterie de tests ou bien prévoir les tests en début de développement.

L'approche `doctest` incite l'utilisateur à écrire *au préalable* un cahier des charges directement dans les chaînes de documentation (*docstring*) du module (ou de la fonction, classe, etc.). Cette manière de procéder porte le nom de *programmation dirigée par la documentation*.

2.12.1. Tests dans le fichier module. Voici exemple simple d'utilisation des *doctests*. Considérons le module `parfait` créé à l'exercice 10 page 50, et appliquons lui les préceptes de la programmation dirigée par la documentation.

Voici le script complet :

```
1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  from doctest import testmod
4
5  def parfait(n) :
6      """
7      Détermine si un nombre est parfait,
8      c'est à dire somme de ses diviseurs propres
9      >>> parfait(100)
10     False
11     >>> parfait(496)
12     True
13     """
14     N=0
15     for i in range(1,n) :
16         if n%i == 0 :
17             N+=i
18     return N==n
19     #return N
20
21 if __name__ == "__main__" :
22     testmod()
```

Lignes 9 à 12 : on demande qu'un appel à la fonction `parfait()` fournisse les résultats indiqués. La *docstring* de la fonction `parfait` est une copie quasiment conforme du cahier des

charges (ne pas oublier les `>>>` en tête des instructions à tester), y compris les résultats attendus.

Ligne 22 : le programme principal, ne fait qu'appeler la méthode `doctest.testmod()` ce qui a pour effet :

- lecture du *docstring* et exécution des instructions commençant par `>>>` ;
- comparaison des résultats avec les résultats attendus ;
- affichage d'un message d'erreur s'il n'y a pas coïncidence.

Puis on exécute le script :

```
$ parfait-doctest.py
```

S'il n'y a aucune réponse, c'est que tout s'est bien déroulé, les deux instructions dans la *docstring* ont donné des résultats conformes aux attentes.

On peut préférer une exécution plus bavarde. Dans ce cas on passe l'option `-v` :

```
$ parfait-doctest.py -v
Trying :
  parfait(100)
Expecting :
  False
ok
Trying :
  parfait(496)
Expecting :
  True
ok
1 items had no tests :
  __main__
1 items passed all tests :
  2 tests in __main__.parfait
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Imaginons que lors d'un développement ultérieur, on modifie la dernière ligne de la fonction `parfait()` en

```
1 return N
```

(au lieu de `return N==n`). On relance le test et on obtient cette fois un message d'erreur, montrant que la compatibilité ascendante n'est pas respectée :

```
$ parfait-doctest.py
*****
```

```

File "./parfait-doctest.py", line 9, in __main__.parfait
Failed example :
  parfait(100)
Expected :
  False
Got :
  117
*****
[...]
*****
1 items had failures :
  2 of 2 in __main__.parfait
***Test Failed*** 2 failures.

```

La fonction `doctest.testmod()` accepte des paramètres.

L'un des plus utiles est `optionflags=doctest.ELLIPSIS` : lorsque les résultats attendus sont longs (plusieurs lignes), ou bien sont dépendants de la plateforme ou de l'environnement, cette option permet d'en remplacer une partie par « ... » (points de suspension ou *ellipsis*) dans la *docstring*. Par exemple imaginons que la fonction `dictionnaire()` suivante lise un fichier et le convertisse en dictionnaire. Le test de la chaîne de documentation donne un exemple d'utilisation dont le résultat s'étend sur plusieurs lignes. On tronque le résultat attendu en remplaçant une partie par « ... » :

```

1 def dictionnaire(infile) :
2     """
3     >>> dictionnaire('aha.txt')
4     {'mont-blanc' :4807, ... , 'aneto' :3400}
5     """
6     #les instructions

```

Que ce soit pour des projets d'envergure ou pour des petites bibliothèques à usage personnel, l'intérêt du module `doctest` est multiple :

- inciter le programmeur (et son éventuel client) à écrire *par avance* un cahier des charges précis ;
- *documenter* le code par les exemples *use case* donnés dans la *docstring* de chaque fonction ;
- lors de modifications ultérieures, permettre des *tests complets*, garantissant la compatibilité ascendante du code ;
- s'assurer que le code reste fidèle à la documentation ;
- il existe des outils qui permettent d'extraire et d'interpréter les *docstrings* d'un module, puis qui *gènèrent automatiquement* (ou presque) une documentation complète.

2.12.2. Tests dans un fichier séparé. La fonction `doctest.testfile()` peut tester des exemples d'utilisation situés dans un fichier texte séparé des sources *Python*. Supposons un fichier `parfait-doc.txt` contenant ces lignes :

```
Le module ``parfait``
=====
Utiliser ``parfait``
-----
Voici un exemple de texte au format reStructuredText.
Importer ``parfait`` dans le module ``parfait`` :
>>> from parfait import parfait
Qui s'utilise ainsi :
>>> parfait(100)
False
```

On peut alors tester les instructions qui s'y trouvent

- soit depuis une session *Python* :

```
1 >>> import doctest
2 >>> doctest.testfile("parfait-doc.txt")
3 TestResults(failed=0, attempted=2)
```

- soit à l'aide de la commande *Unix* :

```
$ python -m doctest -v parfait-doc.txt
```

qui donne le résultat :

```
Trying :
  from parfait import parfait
  Expecting nothing
  ok
  Trying :
  parfait(100)
  Expecting :
  False
  ok
1 items passed all tests :
  2 tests in parfait-doc.txt
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

Exercices

Exercice 4. Editez et exécutez les divers instructions, modules ou programmes indiqués en 1.8

Exercice 5. Documentation.

- (1) Lancez **Python**, et importez le module `sys`. Trouver dans ce module une variable contenant des informations sur les réels en virgule flottante (**float**). Quel est le plus grand nombre en virgule flottante représentable en **Python** ?
- (2) Quel est le contenu du module `matplotlib` ? Dans quel répertoire se trouve ce module ?
- (3) Dans une session **Python** importez le module `sys` et explorez-le pour trouver
 - le numéro de version de votre interpréteur **Python**,
 - la version du compilateur utilisé pour compiler l'interpréteur,
 - en quel langage a été développé **Python**.
- (4) Trouver la documentation de la fonction `input()`.

Exercice 6. Quel est la différence entre les opérateurs `/` et `//` ?

Exercice 7. Nombres complexes.

- (1) Soit $j = e^{i\frac{2\pi}{3}}$. Vérifier numériquement que $1 + j + j^2 = 0$
- (2) Ecrire une fonction qui résout les équations du second degré à coefficients complexes. On devra pouvoir l'utiliser ainsi :

```
1 >>> racines1(1,2,1)
2 (-1.0, -1.0)
3 >>> racines1(1,1,1-1j)
4 ((0+1j), (-1-1j))
```

Exercice 8. Utiliser une docstring, la fonction `len()` et un copier/coller de votre terminal pour déterminer le nombre de caractères visibles actuellement dans votre terminal.

Exercice 9. Utiliser la fonction `range()` (et l'aide en ligne) pour créer la liste `[10, 20, 30, 40]`

Exercice 10. Nombres parfaits

Écrire et tester une fonction qui détermine si un nombre n , passé en argument, est parfait. Un nombre est parfait s'il est égal à la somme de ses diviseurs propres (c'est à dire de ses diviseurs, y compris 1, et à l'exception de lui-même)

Exercice 11. Fichiers :

- (1) Ouvrir un fichier de ce nom en mode texte, et écriture, puis y écrire une chaîne de caractères de votre choix, le nombre entier 1 et le complexe $1 + i$. N'oubliez pas de fermer le fichier.
- (2) Ouvrir le même fichier, cette fois en lecture, et le lire dans une variable `S`. Afficher `S`.

- (3) Reprendre le même exercice en utilisant la fonction `input()`⁶ pour demander à l'utilisateur un nom de fichier.

Exercice 12. Fichiers :

- (1) Utiliser le module `pickle` pour sauvegarder dans un fichier `pick.txt` la liste `[1, 2, 3, ..., 100]`.
- (2) Avec une clé usb, ou via le réseau (e-mail, ftp) transférez ce fichier sur une autre machine, si possible avec un autre système d'exploitation.
- (3) Unpicklez-le dans une session **Python** de la nouvelle machine.

Exercice 13. Lecture d'un fichier

On considère un fichier `montagnes.txt` contenant les lignes suivantes :

```
Arbizon ;2831 ;Pyrenees ;
Aneto ;3350 ;Pyrenees ;
Nanga Parbat ;8125 ;Himalaya ;
Broad Peak ;8047 ;Karakoram ;
Gasherbrum I ;8068 ;Karakoram ;
K2 ;8611 ;Karakoram ;
Araïlle ;2759 ;Pyrenees ;
Anie ;2504 ;Pyrenees ;
```

Ecrire une fonction qui lit le fichier et qui renvoie un dictionnaire `sommet : (altitude,pays)`.

Pour cela :

- (1) Lire le fichier ligne par ligne, et pour chaque ligne,
- (2) la nettoyer de ses caractères espaces et `' ; '` en début et fin de ligne (méthode `str.strip()`)
- (3) la découper en une liste de trois mots `[sommet, altitude, pays]` (méthode `str.split()`)
- (4) créer l'entrée correspondante dans le dictionnaire.

Exercice 14. Fonctions

- (1) Ecrire une fonction qui retourne le nombre de secondes entre deux heures données au format `'hh :mm :ss'`. On devra pouvoir l'utiliser ainsi :

```
1 >>> duree('05 :00 :00', '06 :00 :00')
2 3600
```

- (2) Une fonction qui retourne les deux racines complexes ou réelles d'une équation du second degré $ax^2 + bx + c = 0$ à coefficients réels. La fonction `racine carrée` est accessible dans le module `math` :

```
1 >>> from math import sqrt
```

On désire utiliser la fonction `racines()` ainsi :

⁶en **Python** 2, il faut plutôt utiliser la fonction `raw_input()`

```

1 >>> racines(1, -2, 1)
2 (1.0, 1.0)

```

Solutions des exercices

Solution 4 Faites...

Solution 5 Documentation.

(1) L'importation du module se fait par l'instruction `>>> import sys`

L'instruction `>>> dir(sys)` permet de lister le contenu du module `sys`. On obtient une liste d'objets contenu dans le module :

```
['__displayhook__', '__doc__', ..., 'float_info', ...].
```

La variable qui nous intéresse est `float_info` dont on visualise le contenu :

```

1 >>> print (sys.float_info)
2 sys.floatinfo
3 (max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
4 min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307,
5 dig=15, mant_dig=53, epsilon=2.2204460492503131e-16,
6 radix=2, rounds=1)

```

le plus grand nombre en virgule flottante représentable en *Python* est donc

```
max=1.7976931348623157e+308
```

(2) Les instructions suivantes fournissent les informations sur `matplotlib` :

```

1 >>> import matplotlib
2 >>> dir(matplotlib)
3 >>> matplotlib.__file__

```

(3) Numéro de version

```

1 >>> import sys
2 >>> dir(sys)
3 ['__displayhook__', etc...
4 ... 'version', 'version_info', 'warnoptions']

```

Les informations contenues dans `sys.version` devraient répondre aux questions posées.

```

1 >>> import sys
2 >>> dir(sys)
3 >>> print sys.version
4 2.6.6 (r266 :84292, Sep 15 2010, 15 :52 :39)
5 [GCC 4.4.5]

```

GCC est le compilateur *C* de *gnu*, *Python* a donc été écrit en *C*.

(4) L'instruction

```
1 >>> help(input)
```

donne la documentation de la fonction `input()`.

Une recherche sur internet avec les mots clé « python input » renvoie sur la page

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

contenant une documentation plus complète.

Solution 6 opérateurs / et //

L'opérateur `/` retourne le résultat de la division *réelle* de ses deux opérandes, même si les opérandes sont entiers.

L'opérateur `//` retourne le résultat de la division *entière* de ses deux opérandes, même si les opérandes sont réels.

```
1 >>> 5/6,5//6
2 (0, 0)
3 >>> 5.0/6,5.0//6
4 (0.8333333333333333, 0.0)
5 >>> 7/6,7//6
6 (1, 1)
7 >>> 7.0/6,7.0//6
8 (1.1666666666666667, 1.0)
```

Solution 7 Nombres complexes

(1) Le complexe j

```
1 >>> from cmath import pi, exp
2 >>> j = exp(1j*2*pi/3)
3 >>> j
4 (-0.49999999999999978+0.86602540378443871j)
5 >>> 1 + j + j*j
6 (-1.1102230246251565e-16+3.3306690738754696e-16j)
```

(2) Équation du second degré : le module `math` contient les fonctions utiles `sqrt()`, `sin()`, `cos()` et `atan2()`. Un appel à `math.atan2(y, x)` retourne $\arctan\left(\frac{y}{x}\right)$, qui est l'argument du complexe $x + iy$.

```
1 def racines(a,b,c) :
2     """
3     >>> racines1(1,2,1)
4     (-1.0, -1.0)
```

```

5  >>> racines(2,2,1)
6  ((-0.5+0.5j), (-0.5-0.5j))
7  >>> racines1(1,1,1-1j)
8  ((1.110...e-16+1j), (-1-1j))
9  """
10 from math import sqrt, cos, sin, atan2
11 delta = b*b-4*a*c
12 if type(delta) in (float, int) or delta.imag == 0 :
13     try : delta = delta.real
14     except TypeError : pass
15     if delta<0 :
16         z1 = complex(-b,sqrt(-delta))/(2*a)
17         return z1,z1.conjugate()
18     return (-b+sqrt(delta))/(2*a),(-b-sqrt(delta))/(2*a)
19 else :
20     t = atan2(delta.imag, delta.real)/2
21     rd = sqrt(abs(delta))*complex(cos(t),sin(t))
22     return ((-b+rd)/(2*a), (-b-rd)/(2*a))

```

Solution 8 L'écran contient une session *bash*. Pour obtenir le nombre de caractères, il suffit de copier le contenu de l'écran dans le presse-papier, de le coller dans une variable *Python* et d'appeler la fonction `len()` sur cette variable. Pour coller le contenu du presse-papier dans une variable *Python*, il faut utiliser une chaîne de caractères de type *docstring* qui permet les sauts de ligne. On ouvre la chaîne avec une triple apostrophe, on colle les lignes, puis on referme la chaîne avec une triple apostrophe.

```

1
2 >>> a="""puiseux@puiseux :~$ ls *.*
3 ... Ellipses.tar.gz
4 ... Formations-contenu.pdf
5 ... GDP_PPP_per_capita_world_map_IMF_figures_year_2006(2).png
6 ... maxout.gnuplot
7 ... puiseux@puiseux :~$"""
8 >>> len(a)
9 156

```

Solution 9 Création de liste

```

1 >>> list(range(10,45,10))

```

Solution 10 Nombres parfaits

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  from doctest import testmod
4
5  def parfait(n) :
6      """
7      Détermine si un nombre est parfait,
8      c'est à dire somme de ses diviseurs propres
9      >>> parfait(100)
10     False
11     >>> parfait(496)
12     True
13     """
14     N=0
15     for i in range(1,n) :
16         if n%i == 0 :
17             N+=i
18     return N==n
19     #return N
20
21 if __name__ == "__main__" :
22     testmod()

```

Solution 11 Fichier

```

1  >>> nom = input('Nom fichier ? ')
2  Nom fichier ? toto.tmp
3  >>> nom
4  'toto.tmp'
5  >>> f = open(nom, 'w')
6  >>> f.write('les chaussettes de l\'archiduchesse sont humides')
7  47
8  >>> f.write(str(1))
9  1
10 >>> f.write(str(1+1j))
11 6
12 >>> f.close()
13 >>> open(nom).read()
14 "les chaussettes de l'archiduchesse sont humides1(1+1j)"
15 >>> S = open(nom).read()
16 >>> S
17 "les chaussettes de l'archiduchesse sont humides1(1+1j)"

```

La fonction `write()` de la classe `file` retourne le nombre d'octets écrits. C'est aussi le nombre de caractères (en *ASCII*).

Solution 12 module pickle

(1) Sur la machine d'origine :

```
1 >>> import pickle
2 >>> f = open('pick.txt', 'w')
3 >>> pickle.dump(list(range(100)), f)
4 >>> f.close()
```

(2) ftp ou e-mail ou clé usb... faites.

(3) Sur la machine réceptrice :

```
1 >>> import pickle
2 >>> f = open('pick.txt')
3 >>> x = pickle.load(f)
4 >>> f.close()
5 >>> x
6 [1, 2, 3, ..., 99]
```

Solution 13 Lecture d'un fichier

```
1 def DictMontagnes(file) :
2     """
3     Utilisation :
4     >>> DictMontagnes('montagnes.txt')
5     {'Gasherbrum I' : ('8068', 'Karakoram'), ... 'Araille' : ('2759', 'Pyrenees')}
6     """
7     d = {}
8     for line in open(file).readlines() :
9         line = line.strip('\n; ')
10        if not line : continue
11        m, a, p = line.split(';')
12        d[m] = (a,p)
13    return d
14 import doctest
15 doctest.testmod(optionflags=doctest.ELLIPSIS)
```

Solution 14 Fonctions

(1) Durée un script complet avec documentation et doctests :

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import doctest
4
5  def duree(h1,h2) :
6      """
7      >>> duree('05 :00 :00', '06 :05 :30')
8      3930
9      """
10     h1 = [int(w) for w in h1.split(' :')]
11     h2 = [int(w) for w in h2.split(' :')]
12     duree = 3600*(h2[0]-h1[0])+60*(h2[1]-h1[1])+h2[2]-h1[2]
13     return duree
14
15 if __name__ == '__main__' :
16     doctest.testmod()

```

(2) Racines d'un polynôme du deuxième degré, version sommaire, sans vérification

```

1  def racines(a,b,c) :
2      """
3      >>> racines(1,2,1)
4      (-1.0, -1.0)
5      >>> racines(1,4,2)
6      (-0.5857864376269049, -3.414213562373095)
7      """
8      from math import sqrt
9      delta = b*b-4*a*c
10     return (-b+sqrt(delta))/(2*a), (-b-sqrt(delta))/(2*a)

```

Une version plus élaborée, complète, avec coefficients réels, qui autorise les racines complexes. L'explication de la ligne `doctest.testmod(optionflags=doctest.ELLIPSIS)` est donnée en [2.12 page 45](#).

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import doctest
4
5  def racines(a,b,c) :
6      """
7      >>> racines(1,2,1)
8      (-1.0, -1.0)

```

```
9 >>> racines(1,1,2)
10 ((-0.5+1.32287565553...j), (-0.5-1.32287565553...j))
11 """
12 from math import sqrt
13 delta = b*b-4*a*c
14 if delta<0 :
15     z1 = complex(-b,sqrt(-delta))/(2*a)
16     return z1,z1.conjugate()
17 return (-b+sqrt(delta))/(2*a),(-b-sqrt(delta))/(2*a)
18
19 if __name__ == '__main__' :
20     doctest.testmod(optionflags=doctest.ELLIPSIS)
```

CHAPITRE 3

Structures de contrôle

3.1. L'instruction while

Dans l'exemple qui suit, nous écrivons une sous-séquence de la suite de Fibonacci

$$u_0 = 0, u_1 = 1, \text{ et } \forall n \geq 1, u_{n+1} = u_n + u_{n-1}$$

```
1 >>> a, b = 0, 1
2 >>> while b < 10 :
3 ...     print b
4 ...     a, b = b, a + b
```

La première ligne contient une affectation multiple : les variables `a` et `b` prennent simultanément les nouvelles valeurs 0 et 1.

Les expressions en partie droite sont d'abord toutes évaluées avant toute affectation.

La boucle `while` s'exécute tant que la condition (ici : `b<10`) reste vraie. En *Python*,

- toute valeur entière différente de zéro est vraie ;
- n'importe quoi avec une longueur différente de zéro est vrai ;
- zéro est faux ;
- les séquences (liste, dictionnaire, ensemble, tuple) vides sont fausses.

Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs de comparaison standard sont : `<`, `>`, `==`, `<=`, `>=` et `!=`¹.

Le corps de la boucle est « indenté » : l'indentation qui a valeur de syntaxe, est le moyen par lequel *Python* regroupe les instructions.

La fonction `print()` écrit la valeur de la ou des expressions qui lui sont données. Elle accepte plusieurs expressions et chaînes. Une virgule finale empêche le retour chariot après l'affichage :

```
1 >>> a, b = 0, 1
2 >>> while b < 1000 :
3 ...     print (b,)
4 ...     a, b = b, a + b
```

¹En *Python* 2 les deux opérateurs de comparaison `<>` et `!=` produisent le même résultat. En *Python* 3, l'opérateur `<>` n'existe plus.

```
5 ...
```

La forme générale d'une boucle **while** est le suivant :

```
while <condition> :
    <instructions>
else :
    <instructions> #si 'break' n'a pas été atteint
```

Par exemple :

```
1 >>> while 1 :
2 ...     a = input('Entrez un nombre pair ')
3 ...     if int(a)%2 == 0 : break
4 ...     else : continue
5 ...
6 Entrez un nombre pair 1
7 Entrez un nombre pair 2
```

Dans les instructions de la clause **while**, on peut trouver des tests comme :

```
if <condition> : break #sortie de boucle
if <condition> : continue #retour début de boucle
```

Ces instructions ont le comportement suivant :

break : permet la sortie de boucle toutes affaires cessantes, sans passer dans le **else**.

continue : renvoie l'exécution en début de boucle, à l'itération suivante.

pass : est l'instruction vide.

else de boucle : est exécuté si et seulement si la boucle se termine sans passer par le **break**

3.2. L'instruction if, else, elif

La forme générale d'un **if** est la suivante :

```
if <condition> :
    <instructions>
elif <condition> :
    <instructions>
else :
    <instructions>
```

La clause **elif** et son bloc d'instructions peut être répété à l'envie.

Le bloc de la clause **else** est exécuté si aucun des blocs **elif** n'a été exécuté, ce qui revient à dire qu'aucune des conditions rencontrées dans les clauses **elif** n'a été satisfaite. Par exemple

```
1 >>> a, b = 1, 2
2 >>> if a < b :
3 ...     print (a, '<', b)
4 ...     elif a=b :
5 ...     print (a, '=', b)
6 ...     else :
7 ...     print (a, '>', b)
8 ...     w = b
9 ...
```

Il peut y avoir plusieurs clauses *elif*, mais une seule clause *else*

La syntaxe suivante peut être parfois commode :

```
1 >>> a = 1 if condition else 2
```

affecte 1 à a si la condition est vraie, 2 sinon.

Elle est un raccourci pour :

```
1 >>> if condition :
2 ...     a = 1
3 ... else :
4 ...     a = 2
```

3.3. L'instruction for

La forme générale du *for* est la suivante :

```
for <element> in <iterateur> :
    <instructions>
else :
    <instructions>
```

element parcourt l'itérateur qui peut être une liste, un tuple, un fichier, ou tout autre *objet itérable*.

Un *itérable* est un objet *Python* sur lequel est défini un itérateur. Pour chaque élément de cet itérateur, les instructions du corps de la boucle sont exécutées.

La clause *else* fonctionne comme dans le *while* : si aucun *break* n'a été rencontré, c'est à dire si tous les éléments de l'itérateur ont été visités, alors les instructions de la clause *else* sont exécutées.

```
1 >>> for i in range(3) :  
2 ...     print (i)  
3 0  
4 1  
5 2
```

Un *itérateur* est un objet qui représente un flux de données et qui possède une méthode `next()` permettant de passer d'un élément du flux au suivant.

Vis à vis de l'instruction :

```
1 for x in <itérateur> : <instruction>
```

un itérateur se comporte exactement comme la liste des objets que doit parcourir la variable `x`.

Derrière, se cache un appel répété à la méthode `next()` de l'itérateur, qui retourne les items successif du flux. Lorsqu'il n'y a plus de données disponibles, la méthode `next()` lève l'exception **StopIteration**. À ce point, tous les items du flux ont été passés en revue, et tout appel ultérieur à `next()` lève l'exception **StopIteration**.

De sorte que la construction :

```
1 for x in <itérateur> : <instruction>
```

peut être vue comme un raccourci pour la boucle :

```
1 while 1 :  
2     try :  
3         x = <itérateur>.next()  
4         <instruction>  
5     except StopIteration :  
6         break
```

L'utilisation typique d'itérateur est le parcours d'un flux de données (séquence, dictionnaire, ensemble, fichier...) par une boucle `for`, à l'aide du mot-clé `in`.

3.4. Les conditions

3.4.1. Valeurs booléennes. Tout objet possède une valeur logique `True` ou `False` qui peut être utilisée dans un test `if` ou `while`.

Les valeurs fausses (`False`) sont : `None`, `False`, zéro, une séquence ou un dictionnaire vide (`''`, `[]`, `()`, `{}`). Une instance d'une classe définie par l'utilisateur est `False`, si une des méthodes spéciales `__len__()` ou `__bool__()` est définie, et renvoie `0` ou `False`.

Les autres objets ont la valeur vraie (`True`).

3.4.2. Opérateurs de comparaisons.

Les opérateurs de comparaison standard sont :

< : strictement inférieur
<= : inférieur ou égal
> : strictement supérieur
>= : supérieur ou égal
== : égale
!= : différent
is : objet identique
is not : objet non identique

On peut rajouter à cette liste les deux opérateurs **in** et **not in** qui vérifient si une valeur apparaît (ou non) dans une séquence.

Les opérateurs **is** et **is not** vérifient si deux objets a et b sont réellement le même objet (ont la même adresse mémoire `id(a) == id(b)`)

```
1 >>> a = [1,2,3]
2 >>> b = a
3 >>> a is b
4 True
5 >>> 1 in a
6 True
7 >>> b = a[ : ]
8 >>> b is a
9 False
10 >>> b == a
11 True
```

Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle de tous les opérateurs numériques. Les comparaisons peuvent être enchaînées. Par exemple, `a < b == c` teste si a est strictement inférieur à b et de plus si b est égal à c.

Les opérateurs **and** et **or** ont une priorité inférieure à celle des opérateurs de comparaison. Entre eux, **not** a la plus haute priorité, et **or** la plus faible, de sorte que

A **and not** B **or** C est équivalent à (A **and** (**not** B)) **or** C.

Les opérateurs **and** et **or** sont dits *court-circuit* : leurs arguments sont évalués de gauche à droite, et l'évaluation s'arrête dès que le résultat est trouvé.

Par exemple, si A et C sont vrais mais que B est faux, l'instruction

```
1 >>> A and B and C
```

n'évalue pas l'expression C.

Il est possible d'affecter le résultat d'une comparaison ou une autre expression booléenne à une variable :

```
1 >>> A, B, C = None, 'Mont Blanc', 4807
2 >>> non_null = A or B or C
3 >>> non_null
4 True
```

Notez qu'en *Python*, au contraire du *C*, les affectations ne peuvent pas être effectuées à l'intérieur des expressions. Il est probable que cela déplaît aux programmeurs *C*, mais cela évite une classe de problèmes qu'on rencontre dans les programmes *C* : écrire `=` dans une expression alors qu'il fallait `==`.

3.4.3. Comparaisons hétérogènes. Les objets de type séquence (*list*, *tuple*, *str*, *bytes*) peuvent être comparés à d'autres objets appartenant au même type de séquence.

Ce type de pratique est en général déconseillé, sauf cas très particulier.

La comparaison utilise l'ordre lexicographique : les deux premiers éléments sont d'abord comparés, et s'ils diffèrent cela détermine le résultat de la comparaison. S'ils sont égaux, les deux éléments suivants sont comparés, et ainsi de suite, jusqu'à ce que l'une des deux séquences soit épuisée.

En termes mathématiques la relation d'ordre est définie par : deux séquences $a = (a_i)_{0 \leq i \leq m}$ et $b = (b_i)_{0 \leq i \leq n}$ vérifient $a \leq b$ si $\forall i \in \{0, 1, \dots, \min(m, n)\}, a_i \leq b_i$.

Si deux éléments à comparer sont eux-mêmes des séquences du même type, la comparaison lexicographique est reconsidérée récursivement. Si la comparaison de tous les éléments de deux séquences les donne égaux, les séquences sont considérées comme égales. Si une séquence est une sous-séquence initiale de l'autre, la séquence la plus courte est la plus petite (inférieure). L'ordonnement lexicographique pour les chaînes utilise l'ordonnement *ASCII* pour les caractères. Quelques exemples de comparaisons de séquences du même type :

```
1 >>> (1, 2, 3) < (1, 2, 4)
2 >>> [1, 2, 3] < [1, 2, 4]
3 >>> ABC < C < Pascal < Python
4 >>> (1, 2, 3, 4) < (1, 2, 4)
5 >>> (1, 2) < (1, 2, -1)
6 >>> (1, 2, 3) == (1.0, 2.0, 3.0)
7 >>> (1, 2, (aa, ab)) < (1, 2, (abc, a), 4)
```

Notez que la comparaison d'objets de types différents est licite mais dangereuse. Le résultat est déterministe mais arbitraire : les types sont triés selon leur nom. Ainsi une liste (*list*) est toujours inférieure à une chaîne (*str*), une chaîne est toujours inférieure à un tuple (*tuple*), etc.

Les types numériques mélangés sont comparés en fonction de leur valeur numérique, ainsi 0 est égal à 0.0, etc.

3.5. Les techniques de boucles

Certaines techniques de boucles sont particulières à *Python* et très utiles. On en présente ici quelques unes.

3.5.1. La fonction `range()`. Permet de parcourir une liste d'entiers régulièrement espacés. Elle génère un itérateur sur une progression arithmétique.

Syntaxe :
`range([a,] b [, s])`

Les paramètres `a`, `b` et `s` doivent être entiers (positifs ou négatifs), `s` doit être non nul. Si `s` est nul, l'exception **`ValueError`** est levée. Renvoie un *itérateur* sur la liste² des entiers $n_i = a + is$ où $0 \leq i \leq \text{int}(\frac{b-a}{s})$.

Pour obtenir une liste plutôt qu'un itérateur, il faut le demander explicitement :

```
1 >>> range(0, 100, 10)
2 range(0, 100, 10)
3 >>> list(range(0, 100, 10))
4 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

L'intérêt de renvoyer un itérateur, plutôt que la liste intégrale, est évident en terme d'encombrement mémoire. Les éléments de l'itérateur sont calculés à la demande et la liste n'est jamais intégralement construite, sauf si, comme c'est le cas ci-dessus, on le demande explicitement.

Voici une utilisation typique de la fonction `range()` avec une boucle **`for`**

```
1 >>> for x in range(3) :
2 ...     print(x, end = ', ')
3 0, 1, 2,
```

²En *Python* 2 la fonction `range()` renvoie une *liste* de valeurs.

3.5.2. Les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Dans l'ordre où elles sont citées, ces méthodes renvoient un itérateur sur les clés, sur les valeurs ou bien sur les items (clé,valeur) du dictionnaire. On peut ensuite boucler sur ces itérateurs.

```
1 >>> prefectures = {'PA' : 'Pau', 'HP' : 'Tarbes', 'PO' : 'Perpignan'}
2 >>> for dpt, pref in prefectures.items() :
3 ...     print (dpt,pref)
4 ...
5 HP Tarbes
6 PO Perpignan
7 PA Pau
```

3.5.3. La fonction `enumerate()`. Lorsqu'on boucle sur une séquence, on peut obtenir simultanément l'indice et la valeur d'un élément en utilisant la fonction `enumerate()`.

Syntaxe :

`enumerate(L)`

L doit être un itérable (séquence, ...). Renvoie un itérateur sur les couples (index,valeur) de l'itérable L

```
1 >>> a = ['tic', 'tac', 'toe']
2 >>> for i, val in enumerate(a) :
3 ...     print (i, val)
```

Cette construction est préférable (plus rapide) à :

```
1 >>> for i in range(len(a)) :
2 ...     print (i,a[i])
```

3.5.4. La fonction `zip()`. Pour boucler sur deux séquences, ou plus, en même temps, les éléments peuvent être appariés avec la fonction `zip()`.

Syntaxe :

`zip(i1, i2, ...)`

i1, i2, ... sont des itérables. Renvoie un itérateur qui parcourt la liste de tuples (v1,v2,...) constitués d'un élément de chaque argument. Les arguments i1,i2, ... ne sont pas nécessairement de même longueur, la longueur du résultat est `min(len(i1), len(i2), ...)`

```
1 >>> questions = ['nom', 'caracteristique', 'sport favori', 'autre']
2 >>> answers = ['charma', 'mutant', 'escalade']
3 >>> for q, a in zip(questions, answers) :
4 ...     print ('quel est votre', q, '?', a)
5 ...
6 quel est votre nom ? charma
7 quel est votre caracteristique ? mutant
8 quel est votre sport favori ? escalade
```

3.5.5. La fonction `reversed()`. Pour boucler à l'envers sur une séquence, spécifiez d'abord la séquence à l'endroit, ensuite appelez la fonction `reversed()`.

```
1 >>> for i in reversed(range(1,10,2)) :
2 ...     print i
```

3.5.6. La fonction `sorted()`. Pour boucler sur une séquence comme si elle était triée, utilisez la fonction `sorted()` qui retourne une nouvelle séquence triée tout en laissant la source inchangée.

```
1 >>> phrase = ['La', 'liste', 'suivante', 'énumère', 'les',
2              'plantes', 'couramment', 'consommées',
3              'comme', 'légumes']
4 >>> for word in sorted(phrase) :
5 ...     print word
```

3.5.7. La fonction `map()`. Permet de créer une liste en appliquant une fonction sur toutes les valeurs renvoyées par un itérateur.

Syntaxe :

`map(f, it)`

`it` est un itérable (séquence, dictionnaire, fichier, ...). `f` est une fonction qui prend en argument les items de `it`. Retourne l'itérateur sur les valeurs `f(i)` pour `i` parcourant `it`.

```
1 >>> def f(x) : return 2*x
2 >>> map(f, range(5))
3 <map object at 0x9ff9aec>
4 >>> list(_)
5 [0, 2, 4, 6, 8]
```

`map()` renvoie un objet `map` que l'on peut ensuite transformer en liste, c'est l'objet de l'instruction `list(_)` la variable `'_'` désigne le dernier objet calculé.

3.5.8. La fonction `filter()`. Permet la création de liste après application d'une fonction filtrante.

Syntaxe :

```
filter(f, It)
```

`f` est une fonction filtre qui doit renvoyer `True` ou `False`, et `It` un itérateur.

Renvoie un itérateur constitué de tous les item `x` de `It` qui vérifie `f(x) == True` est passé en argument à la fonction filtrante.

```
1 >>> def f(x) : return x<3
2 >>> filter(f, range(10))
3 <filter object at 0x8669ecc>
4 >>> list(_)
5 [0, 1, 2]
```

La première ligne définit la fonction `f()` qui renvoie la valeur (`True` ou `False`) de l'expression booléenne `x<3`. L'application de la fonction `filter()` avec comme argument le filtre `f()` et la liste (en fait l'itérateur) `range(10)` renvoie un objet de type `filter` qu'il suffit de transformer en liste.

Les programmeurs venant de langages plus traditionnels, comme le `C++` ou le `C`, sont souvent surpris par ces facilités offertes par *Python*, et « oublient » de les utiliser... du moins dans un premier temps.

3.5.9. Les *list comprehension*. Les *list comprehension*³ fournissent une façon concise de créer des listes sans avoir recours à `map()`, `filter()` et/ou `lambda`. Chaque *list comprehension* consiste en une expression suivie d'une clause `for`, puis zéro ou plus clauses `for` ou `if`. Le résultat sera une liste résultant de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Si l'expression s'évalue en un tuple, elle doit être mise entre parenthèses. Voici quelques exemples d'utilisation :

```
1 >>> entiers = range(5)
2 >>> reels=[float(i) for i in entiers]
```

³On trouve diverses traductions de *list comprehension* dans la littérature.

- « compréhension de liste » est un contresens
- « listes compréhensives » également

J'adopterai ici « listes en compréhension », par opposition aux « listes en extension », lorsque tous les éléments de la liste sont cités explicitement.

La liste reels résultante est analogue à la liste entiers, dans laquelle chaque élément a été transformé en (**float**)

Testez les instructions suivantes :

```

1 >>> vec,vec1 = [2, 4, 6],[8,9,12]
2 >>> [3*x for x in vec]
3 [6, 12, 18]
4 >>> [3*x for x in vec if x > 3]
5 [12, 18]
6 >>> [v[0]*v[1] for v in zip(vec,vec1)]
7 [16, 36, 72]
8 >>> [3*x for x in vec if x < 2]
9 []
10 >>> [{x : x**2} for x in vec]
11 [{2 : 4}, {4 : 16}, {6 : 36}]
12 >>> [[x,x**2] for x in vec]
13 [[2, 4], [4, 16], [6, 36]]
14 >>> [x, x**2 for x in vec] # erreur
15 SyntaxError : invalid syntax
16 >>> [(x, x**2) for x in vec]
17 [(2, 4), (4, 16), (6, 36)]
18 >>> [x*y for x in vec for y in vec1]
19 [16, 18, 24, 32, 36, 48, 48, 54, 72]
20 >>> [vec[i]*vec1[i] for i in range(len(vec))]
21 [16, 36, 72]
```

Exercices

Exercice 15. Déterminez à l'aide d'une boucle **while** le plus petit réel strictement positif de la forme 2^{-n} , $n \in \mathbb{N}$

Exercice 16. Dans \mathbb{C} les racines n -ième de 1 sont les n nombres complexes $(e^{2ik\frac{\pi}{n}})_{0 \leq k < n}$. Vérifier numériquement (modules *cmath*) que

$$\sum_{0 \leq k < n} e^{2ik\frac{\pi}{n}} = 0$$

$$\prod_{0 \leq k < n} e^{2ik\frac{\pi}{n}} = 1$$

Exercice 17. Développement limité de $\log(1-x)$, $x < 1$

On rappelle que le développement en série entière de $\log(1-x)$ a pour rayon de convergence $r = 1$ et

$$\log(1-x) = -\sum_{k \geq 1} \frac{x^k}{k}$$

- (1) Calculer la somme des 100 premiers termes de la série, pour $x = 0.5$. Comparer avec $\log(\frac{1}{2})$
- (2) Calculer la somme des termes de la série jusqu'à ce que la somme soit (numériquement) stationnaire. Combien de termes faut-il ?

Exercice 18. (Exercice à faire en **Python** 2 seulement. En **Python** 3 donne lieu à une boucle infinie car il n'y a pas de plus grand entier)

Utiliser une boucle **while**, la fonction **type()** pour déterminer la valeur, M , du plus grand entier, en **Python**. Montrer que $\exists n \in \mathbb{N} : M + 1 = 2^n$. Calculer M^2

Exercice 19. Trouver $n \in \mathbb{N}$ tel que $1.0 \times 2^{-n} = 0.0$ et $1.0 \times 2^{-n+1} \neq 0.0$

Exercice 20. Produit scalaire

- (1) On représente les vecteurs de \mathbb{R}^n par des listes **Python** de réels. Écrire une boucle qui calcule le produit scalaire $(X, Y) = \sum_{1 \leq i \leq n} x_i y_i$.
- (2) Écrire la même boucle en utilisant les list compréhension. On peut écrire le produit scalaire sans voir apparaître un seul indice.

Exercice 21. Produit matrice vecteur.

La liste $X=[1, 1, 1]$ représente le vecteur $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ et la liste de liste $A=[[1, 2, 3], [0, 1, 0], [1, -1, 0]]$ représente la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \end{pmatrix}$.

- (1) Écrire une boucle qui calcule le produit matriciel $Y = A.X$ et le stocke dans une liste Y .
- (2) Écrire la même boucle en utilisant les list compréhension. On peut écrire le produit matrice-vecteur sans voir apparaître un seul indice.

Exercice 22. Matrices. On décide de représenter une matrice $A \in \mathbb{R}^{m,n}$ par une liste A de m listes à n éléments. De sorte qu'un élément $a_{i,j}$ est stocké en $A[i][j]$

Pour deux matrices $A \in \mathbb{R}^{m,n}$ et $B \in \mathbb{R}^{n,p}$, le produit $A \times B$ est défini par $A \times B \in \mathbb{R}^{m,p}$ et, pour $1 \leq i \leq m, 1 \leq j \leq p$

$$c_{i,j} = \sum_{1 \leq k \leq n} a_{i,k} b_{k,j}$$

- (1) Écrire une fonction `zeros(m, n)` qui renvoie une liste de m listes de taille n remplies de 0. Cette liste de listes représente la matrice nulle de $0_{m,n} \in \mathbb{R}^{m,n}$.

- (2) Ecrire une fonction `mult(A, B)` qui calcule de manière traditionnelle, par trois boucles imbriquées, le produit de deux matrices A et B aux dimensions que l'on supposera compatibles.
- (3) Ecrire une fonction `transposed(B)` qui retourne la matrice transposée de son argument B .
- (4) Écrire une fonction `dot(X, Y)` qui retourne le produit scalaire de deux vecteurs (**list**) de même dimension.
- (5) En remarquant que le terme $c_{i,j}$ du produit $A \times B$ est le produit scalaire de la i -ème ligne de A avec la j -ème colonne de B , et en utilisant la fonction `dot()`, et la matrice transposée de B écrire le produit $A \times B$ en utilisant deux list comprehension imbriquées.

Exercice 23. Algorithme de Horner.

Pour calculer la valeur d'un polynôme en un point x , l'algorithme de Horner est le plus économique. Le principe est d'écrire le polynôme $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ sous la forme

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + x(a_n))))))$$

Écrire l'algorithme puis la fonction correspondante `horner(A, x)`, qui retourne la valeur de $p(x)$, où A est une liste contenant les coefficients a_i

Solutions des exercices

Solution 15 Le plus petit réel

```
1 >>> a, n = 2.0, 0
2 >>> while a > 0.0 :
3 ...     a, n = a/2, n+1
4 ...
5 >>> n
6 1076
```

Solution 16 Racines n -ième de l'unité :

Solution « à la C » :

```
1 from math import sin, cos
2 n = 200
3 S, P = 0.0, 1.0
4 for k in range(n) :
5     S += complex(cos(2*k*pi/n), sin(2*k*pi/n))
6     P *= complex(cos(2*k*pi/n), sin(2*k*pi/n))
7 print(S, P)
```

Solution « à la **Python** »⁴ :

⁴en **Python** 2, la fonction `reduce()` est une fonction intégrée qu'il est donc inutile d'importer

```

1 from cmath import exp, pi
2 from operator import mul
3 from functools import reduce
4 n = 200
5 L = [exp(2j*k*pi/n) for k in range(n)]
6 S = sum(L)
7 P = reduce(mul, L)
8 print(S, P)

```

Solution 17 (1) Les 100 premiers termes

```

1 >>> x = 0.5
2 >>> -sum([x**k/k for k in range(1,101)])
3 -0.6931471805599451
4 >>> import math
5 >>> math.log(0.5)
6 -0.6931471805599453

```

L'algorithme de Horner (3.5.9 page 76) permet de calculer cette somme de manière moins gourmande en ressources et plus précise :

```

1 >>> -horner([0]+[1/n for n in range(1,101)],0.5)
2 -0.6931471805599453

```

(2) La série est numériquement stationnaire TODO

```

1 def myLog0(x) :
2     """
3     >>> x = 0.5
4     >>> myLog0(x)
5     (-0.693147180559945..., 48)
6     """
7     p = 0
8     xi = 1.0
9     A = (1.0/n for n in range(1,10000))
10    for (i,a) in enumerate(A) :
11        xi *= x
12        axi = a*xi
13        q = p + axi
14        if p == q :
15            return -q,i
16        p = q
17    return (-p,i)

```


Solution 18 Le plus grand entier

```
1 >>> m = 2
2 >>> while type(m) == int :
3 ...     m *= 2
4 ...
5 >>> m
6 2147483648L
```

Solution 19 $2^{-n} = 0$ et $2^{-n+1} \neq 0$

```
1 >>> a, b, n = 1.0, 0.5, 1
2 >>> while a != 0 :
3 ...     a, b, n = a*b, b/2, n+1
4 ...     else :
5 ...         print(n)
6 >>> n
7 47
```

Solution 20 Produit scalaire.

On déclare deux vecteurs test :

```
1 >>> X, Y = [0, 1, 1], [2, -1, 1]
```

Les programmeurs familiers du C écriront sans doute :

```
1 >>> s = 0
2 >>> for i in range(len(X)) :
3 ...     s += X[i]*Y[i]
4 >>> print (s)
5 0
```

On peut appairer X et Y en utilisant la fonction `zip()` :

```
1 >>> s = 0
2 >>> for x,y in zip(X,Y) :
3 ...     s += x*y
```

On peut aussi éviter la boucle `for` et la variable `s` en utilisant la fonction intégrée `sum()` et une *list comprehension* :

```
1 >>> sum([x*y for x,y in zip(X,Y)])
```

On peut également, (c'est plutôt plus économique en mémoire) se dispenser de créer explicitement la liste en utilisant une *expression génératrice* (voir 4.3.4 page 85) :

```
1 >>> sum(x*y for x,y in zip(X,Y))
```

Cette instruction s'adapte à des vecteurs de taille quelconque.

On peut également en faire une fonction que l'on utilisera plus loin 21 page 70 :

```
1 def dot(X,Y) :
2     return sum(x*y for x,y in zip(X,Y))
```

Solution 21 Produit matrice vecteur.

On construit la matrice et le vecteur

```
1 >>> A=[[1, 2, 3], [0, 1, 0], [1, -1, 0]]
2 >>> X=[1, 1, 1]
```

(1) Une boucle classique :

```
1 >>> n = len(X)
2 >>> Y = [0, 0, 0]
3 >>> for i in range(n) :
4     for j in range(n) :
5         Y[i] = Y[i] + A[i][j]*X[j]
6 >>> Y
7 [6, 1, 0]
```

(2) En utilisant les *list comprehension*. Pour un i fixé, la boucle sur j qui calcule $y[i]$ peut s'écrire :

```
1 >>> Y[i] = sum([A[i][j]*X[j] for j in range(n)])
```

En observant que $Y[i]$ est en fait le produit scalaire de la ligne $A[i]$ avec le vecteur X , et en utilisant l'exercice 3.5.9 page précédente, on obtient :

```
1 >>> Y[i] = sum([a*x for a,x in zip(A[i],X)])
```

Il faut ensuite boucler sur i pour affecter $Y[i]$. Or, le parcours des $A[i]$ *for i in range(n)* équivaut au parcours des L *for L in A*. On intègre cette construction dans une *list comprehension*

```
1 >>> Y = [sum([a*x for a,x in zip(L,X)]) for L in A]
```

Finalement, si l'on utilise la fonction `dot()` de l'exercice 20 page 70 dont la solution est en 3.5.9 page 73, on obtient l'instruction la plus lisible, exprimant que l'élément Y_i est le produit scalaire de X et de la i -ème ligne de A , c'est à dire

```
1 >>> Y = [dot(X,L) for L in A]
```

Solution 22 Produit matrice matrice :

(1) Fonction `zeros(m,n)` :

```
1 def zeros(m,n) :
2     return [[0 for j in range(n)] for i in range(m)]
```

Attention : on peut aussi penser à écrire `return m*[n*[0]]` qui retourne effectivement une liste de listes avec les bonnes dimensions *mais* les m listes sont identiques, autrement dit il n'y a qu'une liste, et si on modifie une ligne de la matrice, toutes les lignes sont modifiées. Par exemple :

```
1 >>> C = 3*[2*[0]]
2 >>> C
3 [[0, 0], [0, 0], [0, 0]]
4 >>> C[0][0]=1
5 >>> C
6 [[1, 0], [1, 0], [1, 0]]
```

(2) Produit usuel :

```
1 def mult0(A,B) :
2     m,n,p = len(A),len(A[0]),len(B[0])
3     C = zeros(m,p)
4     for i in range(m) :
5         for j in range(p) :
6             C[i][j] = sum([A[i][k]*B[k][j] for k in range(n)])
```

On a remplacé la boucle la plus imbriquée (boucle sur k) par une somme sur une *list comprehension*.

(3) Transposée

```
1 def transposed(B) :
2     n,p = len(B),len(B[0])
3     return [[B[k][j] for k in range(n)] for j in range(p)]
```

ou bien :

```
1 return [[L[j] for L in B] for j in range(p)]
```

(4) Produit scalaire

```
1 def dot(X,Y) :
2     sum([x*y for x,y in zip(X, Y)])
```

(5) Produit.

L'élément $c_{i,j} = \sum_{1 \leq k \leq n} a_{i,k} b_{k,j}$ est le *produit scalaire* de la ligne i de A avec la colonne j de B . Donc calculer la matrice C consiste à faire

[[L.C pour les colonnes C de B] et pour les lignes L de A]

La i -ème ligne de A est $1A = A[i]$.

La j -ème colonne de B , est la j -ème ligne de la matrice transposée Bt .

Pour i et j fixés, $c_{i,j}$ vaut donc $\text{dot}(A[i], Bt[j])$.

```
1 def mult(A,B) :
2     Bt = transposed(B)
3     return [[dot(ligne,colonne) for colonne in Bt] for ligne in
                A]
```

On peut ensuite tester avec les deux matrices :

```
1 >>> A = [[1,2,3],[0,1,0],[1,-1,0],[1,2,3]]
2 >>> B = [[0,1],[1,0],[1,1]]
```

Solution 23 Algorithme de Horner

```
p = an
pour i = n-1, n-2, ..., 0 faire
    p = p*x + ai
```

```
1 def horner(A,x) :
2     """
3     >>> A = [-3, -5, -1, 1]
4     >>> horner(A,3), horner(A,-1)
5     (0, 0)
6     >>> horner([0]+[1.0/n for n in range(1,101)],0.5)
7     0.693147180559945...
8     """
9     p = 0
10    for a in reversed(A) :
11        p = p*x + a
```

Il est inutile d'appeler les coefficients de A par leur numéro $A[i]$. La variable a parcourt A dans le bon ordre, l'indice est inutilisé.

CHAPITRE 4

Les types intégrés (builtins)

4.1. Les variables Python

Une variable *Python* est un triplé (nom, type, valeur).

En *Python* 3, les noms de variables doivent obéir à quelques règles simples :

un nom de variable est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre.

Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc... sont interdits, à l'exception du caractère _ (souligné ou *underscore*).

La casse est significative (les caractères majuscules et minuscules sont distingués). Ainsi les noms Chabal, chabal, CHABAL désignent des variables distinctes.

En *Python* 3, les noms de variables peuvent contenir des lettres accentuées.

Le tableau 4.1 donne la liste des 33 mots réservés ou mots-clé du langage *Python* 3¹. Ces mots clé ne peuvent bien entendu pas être utilisés comme noms de variables.

<i>False</i>	<i>class</i>	<i>finally</i>	<i>is</i>	<i>return</i>
<i>None</i>	<i>continue</i>	<i>for</i>	<i>lambda</i>	<i>try</i>
<i>True</i>	<i>def</i>	<i>from</i>	<i>nonlocal</i>	<i>while</i>
<i>and</i>	<i>del</i>	<i>global</i>	<i>not</i>	<i>with</i>
<i>as</i>	<i>elif</i>	<i>if</i>	<i>or</i>	<i>yield</i>
<i>assert</i>	<i>else</i>	<i>import</i>	<i>pass</i>	
<i>break</i>	<i>except</i>	<i>in</i>	<i>raise</i>	

Tab. 1. Liste des mots clé de *Python*3

La valeur d'une variable est d'un type donné, ce type peut être *modifiable* (se dit *mutable* en anglais) ou *non modifiable* (on trouve aussi parfois le qualificatif *immuable* ou en anglais *immutable*).

Les types de base non modifiables sont *int*, *float*, *tuple*, *str*, *bytes*

Les types de base modifiables sont *list*, *dict*, *bytearray*

¹En *Python* 2 *False*, *None*, *nonlocal* et *True*, ne sont pas des mots-clé, tandis que *exec* et *print* sont des mots-clé.

La *valeur* d'une variable de type modifiable peut être changée en gardant la même adresse mémoire, donnée par la fonction `id()`. Un exemple d'utilisation de la fonction `id()` :

```
1 >>> id(1)
2 137163584
3 >>> i = 1
4 >>> id(i)
5 137163584
```

L'instruction `id(1)` crée la valeur 1 de type `int`, et renvoie l'adresse de cette valeur, puis libère les ressources à cette adresse, puisqu'aucune référence ne pointe plus dessus.

L'instruction `i = 1` crée à nouveau la valeur 1 et lui associe le nom `i`. L'adresse de `i` est l'adresse de la valeur 1. Elle a été recréé à la même adresse que la valeur 1 précédente, c'est peut-être un hasard.

Le seul moyen de changer la valeur d'une variable d'un type non modifiable est de recréer la nouvelle valeur et de l'associer au nom de la variable par l'instruction d'affectation `nom = valeur`. L'adresse mémoire de la nouvelle valeur est modifiée.

```
1 >>> t = (0,1)
2 >>> id(t)
3 3071847852
```

`id(t)` est l'identificateur de la liste `t` est en quelque sorte son adresse mémoire, il est unique au cours d'une session *Python* et ne sera pas modifié.

```
1 >>> t[1] = 7
2 Traceback (most recent call last) :
3 ...
4     t[1] = 7
5 TypeError : 'tuple' object does not support item assignment
```

Les tuples sont non modifiables. Pour modifier un tuple, il faut en créer une copie modifiée et l'affecter à `t`. Son adresse n'est plus la même.

```
1 >>> t = (0,7)
2 >>> id(t)
3 3070871436
```

La même suite d'instructions avec une liste ne produit pas d'erreur, et la liste ne change pas d'adresse :

Diverses opérations et manipulations mathématiques sur les types numériques sont possibles avec les modules :

- `numbers` classes numériques de base, opérations de bas niveau ;
- `math` fonctions et constantes mathématiques ;
- `cmath` fonctions et constantes mathématiques pour les complexes ;
- `decimal` nombres décimaux, virgule fixe ;
- `fractions` nombres rationnels ;
- `random` nombres pseudo-aléatoires ;

4.3. Itérateurs et générateurs

4.3.1. Définition. En première approche, il n’y a pas grand inconvénient à songer à « séquence » (list ou tuple, ...) comme synonyme de « itérable » ou même « itérateur ».

Le glossaire² de la documentation *Python* fournit les explications suivantes :

itérable : un objet capable de retourner ses items un par un. Les séquences (`list`, `str`, `tuple`) sont itérables, certains types comme `dict`, `file` sont itérables, ainsi que tous les types que vous pouvez définir possédant une méthode `__iter__()` ou `__getitem__()`. Les itérables peuvent être utilisés dans les boucles `for` mais aussi partout où une séquence est nécessaire (fonctions `zip()`, `map()`, etc.). Lorsqu’un objet itérable est passé en argument à la fonction intégrée `iter()`, elle retourne un *itérateur* sur cet objet. Cet itérateur permet de passer en revue les items de l’objet. En général il n’est pas nécessaire d’utiliser la fonction `iter()` ni de traiter directement avec l’itérateur. L’instruction `for` le fait directement, en créant une variable itérateur temporaire non nommée pour la durée de la boucle.

itérateur : un objet représentant un flux de données. Un appel répété à la méthode `next()` de l’itérateur, retourne les items successifs items du flux. Lorsqu’il n’y a plus d’item disponible, l’itérateur lève l’exception `StopIteration`. L’itérateur est alors épuisé. Tout appel ultérieur à sa méthode `next()` lève l’exception `StopIteration` à nouveau. Les itérateurs doivent avoir une méthode `__iter__()` qui retourne l’objet itérateur lui-même. Ainsi tout itérateur est un itérable et peut, *presque* partout, être utilisé à la place de l’itérable qu’il représente. Une exception toutefois : lorsque l’on effectuer plusieurs cycles complets d’itérations sur un même itérateur, à la fin du premier cycle et les cycles suivants, il apparaîtra comme un conteneur vide. Les objets itérables intégrés (`list`, `dict`, `str`, etc.) recréent un nouvel itérateur pour chaque boucle `for` dans laquelle ils sont utilisés.

générateur : une fonction qui retourne un itérateur. C’est une fonction ordinaire, sauf qu’elle contient une instruction `yield` produisant une suite de valeurs à destination d’une boucle `for` ou bien de la fonction `next()`. Chaque `yield` suspend temporairement l’exécution

²<http://docs.python.org/py3k/glossary.html>

du générateur, mémorise l'endroit où il a été interrompu, et la valeur des variables locales au moment de l'interruption. Ces valeurs sauvegardées sont ensuite réutilisées à la prochaine invocation du générateur.

4.3.2. Itérateurs. Nous avons vu que la plupart des conteneurs peuvent être parcourus en utilisant une instruction **for**

```
1 for element in [1, 2, 3] :
2     print element
3 for element in (1, 2, 3) :
4     print element
5 for key in {'one' :1, 'two' :2} :
6     print key
7 for char in "123" :
8     print char
9 for line in open("myfile.txt") :
10    print line
```

Ce style d'accès est clair, concis et pratique. L'utilisation d'itérateurs imprègne *Python* et l'unifie.

Derrière cette simplicité se cache la suite d'opération suivante :

- l'instruction **for** appelle **iter()** sur l'objet conteneur.
- Cette fonction renvoie un objet itérateur définissant une méthode **next()** qui accède les éléments dans le conteneur, un à la fois.
- Lorsqu'il n'y a plus d'éléments, **iter()** lève une exception **StopIteration** qui dit à la boucle **for** de se terminer.

Exemple d'itérateur :

```
1 >>> s = 'abc'
2 >>> it = iter(s)
3 >>> it
4 <iterator object at 0x00A1DB50>
5 >>> it.next()
6 'a'
7 >>> it.next()
8 'b'
9 >>> it.next()
10 'c'
11 >>> it.next()
12 Traceback (most recent call last) :
13 File "<pyshell#6>", line 1, in -toplevel
14 it.next()
15 StopIteration
```

Pour ajouter un comportement d'itérateur à une classe, il suffit de définir une méthode `__iter__()` qui renvoie un objet ayant une méthode `next()`. Si la classe définit `next()`, alors `__iter__()` peut se limiter à renvoyer `self`.

Voici un exemple d'itérateur personnalisé :

```
1 class Reverse :
2     """Iterator for looping over a sequence backwards"""
3     def __init__(self, data) :
4         self.data = data
5         self.index = len(data)
6     def __iter__(self) :
7         return self
8     def next(self) :
9         if self.index == 0 :
10            raise StopIteration
11            self.index = self.index - 1
12            return self.data[self.index]
13
14 >>> for char in Reverse('spam') :
15 ...     print char
16 ...
17 m
18 a
19 p
20 s
```

4.3.3. Générateurs. Les générateurs sont un outil simple et puissant pour créer des itérateurs. Ce sont des fonctions ordinaires dont la particularité est d'utiliser l'instruction `yield` chaque fois qu'ils veulent renvoyer une donnée. Lorsque `next()` est appelé, le générateur reprend là où il s'était interrompu (il mémorise les valeurs des données et quelle instruction a été exécutée en dernier). L'exemple suivant montre à quel point il est trivial de créer un générateur :

```
1 def reverse(data) :
2     for index in range(len(data)-1, -1, -1) :
3         yield data[index]
4 >>> for char in reverse('zeus a ete a suez') :
5 ...     print char,
6 ...
7 zeus a ete a suez
```

Tout ce qui peut être fait avec des générateurs peut aussi être fait avec des itérateurs basés sur des classes comme décrit dans la section précédente. Ce qui rend les générateurs si compacts est le fait que les méthodes (`__iter__()`) et (`next()`) sont créées automatiquement.

Un autre point clé est que les variables locales et l'état de l'exécution sont sauvés automatiquement entre les appels. Cela rend la fonction facile à écrire et beaucoup plus claire qu'une approche utilisant les variables d'instance comme `self.index` et `self.data`. En plus de la création de méthodes et la sauvegarde de l'état automatiques, lorsque les générateurs terminent ils lèvent automatiquement l'exception **StopIteration**. Ensemble, ces particularités facilitent la création d'itérateurs sans plus d'effort que l'écriture d'une fonction ordinaire.

4.3.4. Expressions génératrices. Certains générateurs simples peuvent être codés succinctement comme des expressions qui utilisent une syntaxe similaire à celle des *list comprehensions* mais avec des parenthèses au lieu de crochets. Ces expressions sont destinées aux situations où le générateur est immédiatement utilisé par une fonction englobante. Les expressions génératrices sont plus compactes mais moins souples que les définitions de générateurs complètes. Elles ont tendance à être plus économes en mémoire que les list comprehensions équivalentes, mais beaucoup plus voraces en temps d'exécution. Exemples :

```
1 >>> sum(i*i for i in range(10)) # sum of squares
2 285
3 >>> xvec = [10, 20, 30]
4 >>> yvec = [7, 5, 3]
5 >>> sum(x*y for x,y in zip(xvec, yvec)) # dot product
6 260
7 >>> from math import pi, sin
8 >>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))
9 >>> unique_words = set(word for line in page for word in line.
    split())
10 >>> valedictorian = max((student.gpa, student.name) for student in
    graduates)
11 >>> data = 'golf'
12 >>> list(data[i] for i in range(len(data)-1, -1, -1))
13 ['f', 'l', 'o', 'g']
```

Pour des raisons pédagogiques de simplicité, on parle des *fonctions* `range()`, `sorted()`, `reversed()`, `enumerate()`, `zip()`, `map()`, `filter()` etc. qui sont en réalité des *classes*. Mais dans le contexte présent, elles s'utilisent avec une syntaxe de fonction, qui n'est autre que la syntaxe du constructeur de la classe.

4.4. Les séquences tuples et list

4.4.1. Les tuples. Un *tuple* est un ensemble de valeurs séparées par des virgules, par exemple :

```
1 >>> t = 12345, 54321, 'salut !'
2 >>> t
3 (12345, 54321, 'salut !')
4 >>> t[0]
5 12345
```

Les tuples peuvent être imbriqués :

```
1 >>> t, (1, 2, 3, 4, 5)
2 ((12345, 54321, 'salut !'), (1, 2, 3, 4, 5))
```

Les tuples sont toujours entre parenthèses. Les tuples, comme les chaînes, sont non-modifiables : il est impossible d'affecter individuellement une valeur aux éléments d'un tuple.

```
1 >>> t = (1, 2, 3)
2 >>> t[0] = 5
3 Traceback (most recent call last) :
4   [...]
5     t[0]=5
6 TypeError : 'tuple' object does not support item assignment
```

Un problème particulier consiste à créer des tuples à 0 ou 1 élément : la syntaxe admet quelques subtilités pour y parvenir. Les tuples vides sont construits grâce à des parenthèses vides. Un tuple à un seul élément est construit en faisant suivre une valeur d'une virgule (il ne suffit pas de mettre une valeur seule entre parenthèses). Par exemple :

```
1 >>> empty = ()
2 >>> singleton = 'salut', ##
3 >>> len(empty)
4 0
5 >>> len(singleton)
6 1
7 >>> singleton
8 ('salut',)
```

L'instruction `t = (12345, 54321, 'salut')` est un exemple « d'emballage » en tuple (*tuple packing*) : les valeurs 12345, 54321 et 'salut' sont emballées ensemble dans un tuple. L'opération inverse (« déballage » de tuple -*tuple unpacking*-) est aussi possible :

```
1 >>> x, y, z = t
```

Le déballage d'un tuple nécessite que la liste des variables à gauche ait un nombre d'éléments égal à la longueur du tuple. Notez que des affectations multiples ne sont en réalité qu'une combinaison d'emballage et déballage de tuples !

4.4.2. Le découpage (*slicing*). Les séquences (chaînes de caractères, listes, tuples) peuvent être adressés de différentes manières.

Il est important de rappeler que si *L* est une séquence, l'instruction *y = L* a pour effet de désigner *y* comme un *alias* de *L*. C'est à dire que *y* et *L* ont même adresse mémoire, le résultat du test *y is L* est *True*.

Pour créer une *recopie* de *L* dans *y*, il faut écrire *y = L[:]*.

La manière la plus simple de référencer un élément de la séquence *L* est de spécifier son index entre crochets, le premier élément d'une séquence est en position 0 : *L[0]*.

On peut également préciser une *tranche* de séquence : *L[m : M]* qui désigne une *copie* de la séquence *L[m]*, *L[m+1]* . . . , *L[M-1]*

```
1 >>> L = 'HelpA'
2 >>> L[4]
3 'A'
4 >>> L[0 : 2]
5 'He'
6 >>> L[2 : 4]
7 'lp'
```

Les index dans les tranches ont des valeurs par défaut. La valeur par défaut du premier index est 0, celle du second index est *len(L)*. Ainsi, *L[: 3]* est une copie de la séquence *L[0]*, *L[1]*, *L[2]*. De même *L[3 :]* est une copie de la séquence *L[3]*, *L[4]*, . . . , *L[len(L)-1]*

```
1 >>> L[ : 2]
2 'He'
3 >>> L[2 : ]
4 'lpA'
```

Voici un invariant utile des opérations de découpage : *L[: i] + L[i :] == L*

```
1 >>> L[ : 2] + L[2 : ]
2 'HelpA'
3 >>> L[ : 3] + L[3 : ]
4 'HelpA'
```

Dans une opération de découpage *L[m : M]*, un index qui est trop grand est remplacé par la taille de la séquence, un index de fin inférieur à l'indice de début retourne une séquence vide.

```

1 >>> L[1 :100]
2 'elpA'
3 >>> L[10 :]
4 ''
5 >>> L[2 :1]
6 ''

```

Les index peuvent être des nombres négatifs pour compter à partir de la droite. L'indice -1 désigne le dernier élément de la séquence, -2 l'avant dernier, etc. Le découpage également est autorisé avec des index négatifs.

Les index i et $i - \text{len}(L)$ désignent le *même* élément de la séquence L

```

1 >>> [L[i] is L[i-len(L)] for i in range(len(L))]
2 [True, True, True, True, True]
3 >>> L[-2]
4 'p'
5 >>> L[-2 :]
6 'pA'
7 >>> L[ :-2]
8 'Hel'

```

4.4.3. Les listes. Le type `list` possède d'autres méthodes que celles déjà rencontrées. Voici toutes les méthodes des objets de type `list` :

- `append(x)` : équivalent à `a.insert(len(a), x)`
- `extend(L)` : rallonge la liste en ajoutant à la fin tous les éléments de la liste donnée ; équivaut à `a[len(a) :] = L`.
- `insert(i,x)` : insère un élément à une position donnée. Le premier argument est l'indice de l'élément avant lequel il faut insérer, donc `a.insert(0, x)` insère au début de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.
- `remove(x)` : enlève le premier élément de la liste dont la valeur est `x`. Si cet élément n'existe pas, l'exception `ValueError` est émise.
- `pop([i])` : enlève l'élément présent à la position donnée dans la liste, et le renvoie. Si aucun indice n'est spécifié, `a.pop()` renvoie le dernier élément de la liste. L'élément est aussi supprimé de la liste.
- `index(x)` : retourne l'indice dans la liste du premier élément dont la valeur est `x`. Il y a erreur si cet élément n'existe pas.
- `count(x)` : renvoie le nombre d'occurrences de `x` dans la liste.
- `sort()` : trie les éléments à l'intérieur de la liste.
- `reverse()` : inverse l'ordre des éléments à l'intérieur de la liste.

Voici un exemple qui utilise toutes les méthodes des listes :


```
1 >>> a = [66.6, 333, 333, 1, 1234.5]
2 >>> print a.count(333), a.count(372.)
3 (2, 0)
4 >>> a.insert(2, -1)
5 >>> a.append(333)
6 >>> a.index(333)
7 1
8 >>> a.remove(333)
9 >>> a.pop(1)
10 -1
11 >>> a
12 [66.599999999999994, 333, 1, 1234.5, 333]
13 >>> a.reverse()
14 >>> a
15 [333, 1234.5, 1, 333, 66.599999999999994]
16 >>> a.sort()
17 >>> a
18 [1, 66.599999999999994, 333, 333, 1234.5]
```

Utiliser les listes comme des piles. Les méthodes des listes rendent très facile l'utilisation d'une liste comme une pile, où le dernier élément ajouté est le premier élément récupéré (pile *LIFO*, pour *last-in, first-out*). Pour ajouter un élément au sommet de la pile, utilisez la méthode `append()`. Pour récupérer un élément du sommet de la pile, utilisez `pop()` sans indice explicite.

```
1 >>> pile = [3, 4, 5]
2 >>> pile.append(6)
3 >>> pile.append(7)
4 >>> pile.pop()
5 7
6 >>> pile.pop()
7 6
8 >>> pile.pop()
9 5
10 >>> pile
11 [3, 4]
```

Utiliser les listes comme des files d'attente. Vous pouvez aussi utiliser facilement une liste comme une file d'attente, où le premier élément ajouté est le premier élément retiré (*FIFO*, pour *first-in, first-out*). Pour ajouter un élément à la fin de la file, utiliser `append()`. Pour récupérer un élément du début de la file, utilisez `pop(0)` avec 0 pour indice.

Par exemple :

```
1 >>> file = ["Gaetan", "Gudule", "Gerard"]
2 >>> file.append("Gaston") # Gaston arrive
3 >>> file.append("Gontran") # Gontran arrive
4 >>> file.pop(0)
5 'Gaetan'
6 >>> file.pop(0)
7 'Gudule'
8 >>> file
9 ['Gerard', 'Gaston', 'Gontran']
```

L'instruction *del*. L'instruction **del** permet la suppression d'un élément de liste lorsque l'on dispose de son indice au lieu de sa valeur. **del** peut aussi être utilisé pour enlever des tranches dans une liste (ce que l'on a fait précédemment par remplacement de la tranche par une liste vide). Par exemple :

```
1 >>> a=[-1, 1, 66.6, 333, 333, 1234.5]
2 >>> del a[0]
3 >>> del a[2 :4]
```

del peut aussi être utilisé pour supprimer des variables complètes :

```
>>> dela
```

4.5. Les chaînes de caractères (str, bytes, bytearray)

Les chaînes de caractères font partie des séquences *Python*.

Unicode, *utf-8*, *ASCII*, *Windows-1252*, *ISO 8859-1*, *Latin-1*, etc... tous ces sigles, symboles, abréviations sont un vrai dédale pour le programmeur. Un peu de tri est nécessaire pour s'y retrouver.

4.5.1. Identification et codage des caractères.

Identification Unicode. Le Consortium *Unicode* a développé *Unicode*³ qui est une norme visant à ordonner et attribuer à (presque) tout caractère de (presque) tous les langages un nom et un identifiant numérique unique, appelé *point de code*. L'identifiant numérique est de la forme U+xxxxxx ou xxxxxx est un nombre hexadécimal. Ces points de codes sont regroupés en 17 plans *Unicode* de $16^4 = 65536$ points de code chacun. Seuls les plans 0 et 1 sont actuellement utilisés pour un usage courant. On y trouve pêle-mêle : caractères usuels, idéogrammes, symboles mathématiques, flèches, symboles et formes géométriques diverses, *smileys*, pièces de Mah-Jong, dominos, cartes à jouer etc.. Prévu pour coder jusqu'à 1114112 points de code, on recense actuellement 245000 points de code utilisés.

³<http://fr.wikipedia.org/wiki/Unicode>

Le code *Unicode* de chaque caractère est un simple identifiant. Il est totalement indépendant de son codage logiciel (sous forme d'octets) qui reste à discrétion des plateformes, et des logiciels. *Unicode* ne se mêle pas de codage informatique, contrairement à ce que suggère son nom. *Unicode* n'est que descriptif et taxinomique.

Le codage informatique des caractères est l'objet du paragraphe suivant.

Le codage logiciel. Les différentes normes *ASCII*, *ISO 8859-1*, *Windows-1252*, *utf-8*, etc... gèrent problème du codage de ces caractères en machine.

(1) *ASCII* : historiquement *ASCII*⁴ est apparu le premier dans les années 60, il propose le codage sur un unique octet, de $2^7 = 128$ caractères de base⁵. Sur les 8 bits que compte un octet, un bit est requis pour d'autres considérations techniques. Les caractères accentués et les caractères particuliers à chaque langue sont absents de ce mode de codage.

(2) *ISO 8859* et ses variantes : le huitième bit a été ensuite utilisé pour représenter les caractères accentués. Seuls sont disponibles $2^8 = 256$ caractères, ce qui est insuffisant pour couvrir les caractères utiles de tous les langages candidats. Pour régler ce problème, sont apparus les *pages de code* encore largement utilisées de nos jours. Chacune de ces pages de code détourne l'utilisation de ce fameux huitième bit pour le spécialiser dans la représentation d'un jeu de caractères particulier.⁶

La norme *ISO-8859* ou *Latin-1* décline plusieurs de ces pages de code. Pour ce qui concerne l'Europe Occidentale, la page de code *ISO 8859-1* est la référence. Les pages de code *Windows-1252*, *MacRoman*, *Adobe Latin x* en sont des variantes créées par *Microsoft*, *Apple*, *Adobe* etc...

ISO-8859-2 ou *Latin-2* prend en charge certaines langues d'Europe centrale, *ISO-8859-3* ou *Latin-3* prend en charge les langues d'Europe du sud et l'espéranto, *ISO 8859-11* prend en charge le thaï etc..

La page de code *ISO 8859-15* est une évolution de *ISO 8859-1* incluant les sigles monétaires et les ligatures comme 'œ'.

Toutes ces pages de code incluent et sont compatibles avec le codage *ASCII*.

(3) *utf-8*⁷ : le codage *utf-8* est une spécification de codage pour les points de code *Unicode* de U+0000 à U+D7FF et de U+E000 à U+10FFFF et seulement ceux-là. Les caractères sont codés sur un nombre d'octets variable. Pour chaque caractère, les premiers bits indiquent sur combien d'octets est codé le caractère. Plus précisément :⁸

⁴*ASCII* : American Standard Code for Information Interchange

⁵Les caractères *ASCII* affichables sont les suivants : !"#\$%&'()*+,-./ 0123456789 :;<=>? @ABCDEFGHIJKLMNO PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

⁶Le huitième bit de la page de code *ISO-8859-1* permet la représentation, en sus des caractères *ASCII*, des caractères suivants :

¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ

⁷*utf-8* pour UCS Transformation Format 8 bits ou UCS signifie Universal Character Set

⁸<http://fr.wikipedia.org/wiki/UTF-8>

« Les caractères ayant une valeur scalaire de 0 à 127 (point de code attribué de U+0000 à U+007F) sont codés sur un seul octet dont le bit de poids fort est nul.

Les points de code de valeur scalaire supérieure à 127 sont codés sur plusieurs octets. Les bits de poids fort du premier octet forment une suite de 1 de longueur égale au nombre d'octets utilisés pour coder le caractère, les octets suivants ayant 10 comme bits de poids fort. » Cette règle est imagée dans le tableau suivant :

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

Tab. 2. Codage *Utf-8*, nombre d'octets

Tant que l'on en reste aux caractères *ASCII*, les choses sont assez simples. Pour les caractères accentués ou les alphabets exotiques, la version 2 de *Python* est limitée.

C'est d'ailleurs une des raisons qui ont amené au développement de la version 3 de *Python*. Le traitement des chaînes de caractères a radicalement changé entre les séries 2.x et 3.x de *Python*.

Les chaînes de caractères *Python* sont des séquences non modifiables (*immutable*).

Le type chaîne de caractères est le type *str*. Toutes les chaînes de caractères *Python* sont des séquences de caractères *Unicode*.

4.5.2. Comparatif sommaire Python 2 et 3.

Python 3. Pour créer une chaîne de caractères, il suffit de l'inclure entre apostrophes (*quote* ') ou guillemets (*double quote* "). On peut aussi les inclure entre triple apostrophes ou triple guillemet. La chaîne suivante est obtenue sur un clavier français, en maintenant la touche *Alt-Gr* enfoncée et en tapant certaines lettres du clavier.

```
1 >>> a = "€œ' '↓@~¿×÷j"
2 >>> b = 'Python²'
```

Pour éviter la plongée dans les arcanes de la représentation des chaînes de caractères exotiques, il est sage de se limiter aux caractères accentués, et d'utiliser le codage *utf-8* pour vos fichiers sources *Python*.

Le type *bytes*⁹ ressemble au type *str* avec quelques différences :

- une variable *bytes* ne peut contenir que des caractères *ASCII* ;

⁹Le type *bytes* n'existe pas en *Python2*

- une variable **bytes** se déclare comme une chaîne de caractères *ASCII*, précédée du caractère **'b'**.
- l'accès à un élément d'une variable **bytes** retourne le code *ASCII* du caractère, un entier dans l'intervalle [0, 255] ;

```
1 >>> a = b'avd'
2 >>> a[1]
3 118
```

Pour transformer un **bytes** en **str**, il faut utiliser la méthode `decode()` :

```
1 >>> b = b'azerty'
2 >>> print (b)
3 b'azerty'
4 >>> b.decode()
5 'azerty'
```

Le type `bytearray` est analogue au type **bytes** mais modifiable. Il possède la plupart des méthodes propres aux séquences modifiables (listes, dictionnaires, ...).

Python 2. En **Python2** lorsque l'on doit manipuler des chaînes de caractères contenant des caractères non-*ASCII*, il est conseillé d'utiliser le type **unicode**, obtenu en faisant précéder la chaîne par la lettre **u**.

```
1 >>> x = u'€éï'
2 >>> x
3 >>> print x, type(x)
4 >>> v = '$ebèç'
5 >>> v
6 >>> print v, type(v)
7 >>> u + v #ouuups !
8 >>> v = u'ebèç'
9 >>> u + v #yes !
```

Il est impossible de concaténer une chaîne **str** avec une chaîne **unicode**. On a donc tout intérêt à utiliser systématiquement le type **unicode**, donc à faire précéder toutes les chaînes constantes par le caractère **u**.

En résumé. Si l'on vient de **Python2** et que l'on veut passer à **Python3**, on peut retenir les principes suivants, qui bien que pas tout à fait exacts, éviteront de nombreuses erreurs :

- le type **unicode** de **Python2** devient le type **str** de **Python3** (standard) ;
- le type **str** de **Python2** devient le type **byte** de **Python3**.

4.5.3. Encodage et décodage. Illustrons ce qui concerne l'encodage par un exemple. En *Python* la méthode `str.encode(format)` permet d'encoder une chaîne suivant une page de code donné.

```

1 >>> lc = ['a', 'é', '€']
2 >>> for c in lc :
3 ...     ec = c.encode('utf8')
4 ...     ec, len(ec), ec[0]
5 (b'a', 1, 97)
6 (b'\xc3\xa9', 2, 195)
7 (b'\xe2\x82\xac', 3, 226)

```

On constate qu'en *utf-8*, le 'a' est codé sur un octet, le 'é' sur 2 octets et le '€' sur 3 octets

```

1 >>> codes=['utf8', 'Windows-1252', 'ISO 8859-15', 'MacRoman']
2 >>> for code in codes :
3 ...     c = '€'.encode(code)
4 ...     print ('{ :13s} : { :15s}, { :d} octets'.format(code, c, len
5 ... (c)))
6 utf8          : b'\xe2\x82\xac', 3 octets
7 Windows-1252  : b'\x80' , 1 octets
8 ISO 8859-15   : b'\xa4' , 1 octets
9 MacRoman      : b'\xdb' , 1 octets

```

Les différents codages *utf-8*, *Windows-1252*, *ISO 8859-15* et *MacRoman* n'ont pas la même représentation du caractère '€'.

4.5.4. Manipulations usuelles. L'aide en ligne sur la classe `str` est très complète.

Puisque les chaînes de caractères sont immuables, chacune des méthodes de `str` que l'on s'attend à voir modifier l'instance appelante, renvoie en réalité une *copie* modifiée, l'appelant *ne peut pas* être modifié in-situ.

Les méthodes les plus courantes sont :

- `capitalize()` retourne une copie de la chaîne avec la première lettre en majuscule.
- `count(sub, start=0, end=-1)` retourne le nombre d'occurrences (sans recouvrement) de la chaîne `sub` dans l'intervalle `start : end`
- `find(sub, start=0, end=-1)` retourne l'indice du premier caractère de la chaîne `sub` si elle figure dans l'intervalle `start : end`. Retourne -1 si elle n'a pas été trouvée.
- `format()` voir [4.5.6 page 96](#)
- `index(sub, start=0, end=-1)` comme `find()` mais lève l'exception `ValueError` si `sub` n'est pas trouvée.

- `join(iterable)` concatène les éléments de `iterable`, le séparateur entre les éléments étant l'instance appelante. Si un des éléments de `iterable` n'est pas une chaîne, l'exception **TypeError** est levée. Exemple :

```
1 >>> ' * '.join(['a', 'b', 'c'])
2 'a * b * c'
```

- `lower()` retourne la chaîne en lettres minuscules.
- `upper()` retourne la chaîne en lettres majuscules.
- `strip([chars])` supprime en début et en fin de chaîne *toutes* les occurrences de *tous* les caractères de `chars`. Si `chars` est omis, les espaces sont supprimés.

```
1 >>> ' , ; ; ; ; Ossau! ///'.strip(' , ; ! / ')
2 'Ossau'
```

- `replace(old, new[, count])` retourne une copie de la chaîne dans laquelle au plus `count` occurrences de `old` ont été remplacées par `new`. Si `count` est omis toutes les occurrences sont remplacées.
- `split([sep[, maxsplit]])` retourne la liste des mots de la chaîne, utilisant `sep` comme séparateur. Si `maxsplit` est donné, la liste aura une longueur d'au plus `maxsplit+1`. Si `sep` est omis, tout espace sera séparateur. (Voir la doc pour quelques subtilités)
- `strip(chars=' \t\n\r\x0b\x0c')` retourne une chaîne de caractères dans laquelle ont été supprimés *tous* les caractères spécifiés par l'argument `chars`, en début et fin de chaîne. Par défaut, `chars` contient tous les espaces (voir `string.whitespace`). Par exemple :

```
1 >>> '% : ; \Lionel Terray; //' .strip('; / :% ')
2 'Lionel Terray'
```

4.5.5. Le module string. Ce module propose quelques constantes permettant de connaître la nature d'un caractère ainsi que quelques fonctions, équivalentes aux méthodes de la classe **str**. Les instructions suivantes listent le contenu de certaines de ces constantes. Le résultat devrait être clair :

```
1 >>> import string
2 >>> keys = dir(string)
3 >>> for key in keys :
4 ...     key, getattr(string, key)
5 [...]
6 ('ascii_letters', '
   abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')
7 ('ascii_lowercase', 'abcdefghijklmnopqrstuvwxyz')
8 ('ascii_uppercase', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
9 ('capwords', <function capwords at 0xb71b17ec>)
10 ('digits', '0123456789')
```

```

11 ('hexdigits', '0123456789abcdefABCDEF')
12 ('octdigits', '01234567')
13 ('punctuation', '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~')
14 ('whitespace', ' \t\n\r\x0b\x0c')

```

La fonction `string.capwords()` prend une chaîne en argument et retourne cette chaîne capitalisée (la première lettre de chaque mot a été mise en majuscule).

```

1 >>> string.capwords('mont blanc')
2 'Mont Blanc'

```

4.5.6. Formatage. Jusqu'ici nous avons utilisé deux manières pour afficher des valeurs : en utilisant la fonction `print()`¹⁰ ou bien en donnant simplement le nom de la variable à imprimer.

```

1 >>> a = 12
2 >>> print(a)
3 12
4 >>> a

```

Pour formater plus finement les sorties, on dispose trois méthodes :

- (1) utiliser les méthodes adaptées de la classe `str` (recommandé);
- (2) utiliser l'opérateur de formatage % :
 `format%variables`
 comme pour la fonction `sprintf()` du langage C;
- (3) Manipulez soi-même les chaînes de caractères. Pour convertir n'importe quelle valeur en chaîne de caractères il suffit de la passer à la fonction `repr()`, ou `str()`.

La méthode `format()` de la classe `str`. La méthode `format()` de la classe `str` fonctionne par « champs de remplacement » (*replacement fields*) placés entre accolades `{}`.

On peut accéder aux arguments par position. La chaîne à formater contient des champs `{0}`, `{1}`, `{2}` etc., qui seront remplacés par les valeurs (formatées) des arguments passés à la méthode `str.format()` dans l'ordre indiqué :

```

1 >>> sommet, massif = 'Golden Peak', 'Karakoram'
2 >>> "Le {0} est dans le massif du {1}".format(sommet, massif)
3 'Le Golden Peak est dans le massif du Karakoram'
4 >>> "Le {} est dans le massif du {}".format(sommet, massif)
5 'Le Golden Peak est dans le massif du Karakoram'
6 >>> "Dans le {1} se trouve le {0}".format(sommet, massif)
7 'Dans le Karakoram se trouve le Golden Peak'

```

¹⁰En `Python 2` il s'agit d'une instruction `print`

On peut accéder aux attributs des arguments par leur nom :

```
1 >>> c = 3-5j
2 >>> 'Re{0} = {0.real}'.format(c)
3 'Re(3-5j) = 3.0'
```

Le formatage des arguments peut être plus précis, en utilisant le principe des formats *C* . Par exemple :

```
1 >>> import math
2 >>> "pi :{0 :+12.3f}; {0 :12.5E}".format(math.pi)
3 'pi : +3.142; 3.14159E+00'
```

Il est possible d'aligner les sorties à droite ou à gauche, avec ou sans caractère de remplissage à l'aide des spécificateurs <, ^, > :

```
1 >>> import math
2 >>> "pi :{0 :<12.3f};{0 :*>12.5g}".format(math.pi)
3 'pi :3.142 ;*****3.1416'
```

Extrait et traduit de la documentation *Python* , voici quelques spécifications de format utilisées.

La forme générale d'un spécificateur de format est la suivante (tous les spécificateurs sont facultatifs) :

format_spec ::= [[fill]align][sign][#][0][width][,][.precision][type]

fill : est le caractère de remplissage, autre que ' '

align : est le caractère d'alignement

< : aligné à gauche,
> : aligné à droite,
^ : centré)

sign : pour les types numériques, un des caractères +, -, ,,

+ : permet d'obtenir un affichage signé
- : l'affichage est signé seulement s'il est négatif, c'est le comportement par défaut,
: (espace) l'affichage est signé s'il est négatif, et précédé d'un espace s'il est positif
, : remplace le point décimal par une virgule

width : est un entier indiquant la largeur minimale du champ

precision : est un entier donnant le nombre de décimales pour les réels flottants

type : est un des caractères de l'ensemble {b, c, d, e, E, f, F, g, G, n, o, s, x, X, %}

– type chaînes de caractères :

s ou rien : format chaîne de caractères, c'est le type par défaut ;

– types entiers :

b : format binaire (base 2) ;

- c** : convertit l'entier en sa représentation *Unicode* ;
- d** : format décimal (base 10) ;
- o** : format octal (base 8) ;
- x, X** : format hexadécimal (base 16), utilisant des lettres minuscules ou majuscule ;
- types float. Les entiers peuvent utiliser ces spécificateurs de formats
- e, E** : notation exponentielle ou scientifique (1.0e+2 par exemple) utilisant e ou E pour précéder l'exposant
- f, F** : affiche le réel en virgule fixe, nan et inf pour « *not a number* » et l'infini
- g, G** : format général. Pour une précision donnée, $p \geq 1$, arrondit le nombre à p chiffres significatifs et formate le résultat en virgule fixe ou bien notation scientifique suivant sa valeur.
- n** : nombre. Analogue à 'g', mais en tenant compte des particularités locales (point ou virgule décimale, séparation des milliers)
- %** : pourcentage. Multiplie le nombre par 100 and et l'affiche en virgule fixe, suivi du caractère '%'
- None** : analogue à 'g'

Les fonctions `str()` et `repr()`. La fonction `str()` renvoie des représentations faciles à lire par les humains, alors que `repr()` renvoie des représentations lisibles par l'interpréteur *Python*. De nombreuses valeurs, comme les nombres ou les structures (listes, dictionnaires), ont la même représentation par les deux fonctions. En revanche, les chaînes de caractères et les nombres à virgule flottante, ont deux représentations distinctes.

Quelques exemples :

```

1 >>> s = 'Salut, tout le monde.'
2 >>> str(s), repr(s)
3 ('Salut, tout le monde.', "'Salut, tout le monde.'")
4 >>> s
5 'Salut, tout le monde.'
6 >>> str(0.1)
7 '0.1'
```

```

1 >>> x,y = 3.25, 150
2 >>> print ('x={ :f} tandis que y={ :f}'.format(x,y))
3 x=3.250000 tandis que y=150.000000
4 >>> print ('x=' + str(x) + ' tandis que y=' + str(y))
5 x=3.25 tandis que y=150
```

Utilisation de l'opérateur %. Pour formater les sorties, les habitués du C et du C++ préféreront sans doute utiliser l'opérateur % plutôt que la méthode `format()` de la classe `str`. On en décrit ici quelques éléments.

La forme usuelle est donnée par ce premier exemple qui crée une chaîne contenant `math.pi` sur 5 positions, avec 3 chiffres après la virgule :

```
1 >>> import math
2 >>> 'pi vaut : %5.3f' % math.pi
3 'pi vaut : 3.142'
```

Le `%5.3f` est un *descripteur de format*. S'il y a plus d'un descripteur de format dans la chaîne de caractères, vous devez passer un tuple comme opérande de droite, comme dans cet exemple :

```
1 >>> sommet = 'Mont Blanc'
2 >>> altitude = 4807
3 >>> 'Le %s culmine à %d m' % (sommet, altitude)
4 'Le Mont Blanc culmine à 4807 m'
```

La plupart des formats fonctionnent exactement comme en C et exigent que vous passiez le type approprié¹¹.

Ce mode de formatage n'est pas documenté dans la version 3.2 de *Python*, mais seulement dans les versions 2.x.

4.5.7. Autres fonctionnalisés pour le . Dans <http://docs.python.org/release/3.1.3/library/stdtypes.html>, on trouvera les méthodes de la classe `str`

- `chr(i)` retourne le caractère dont le code *Unicode* est l'entier `i`. Par exemple, `chr(97)` retourne `'a'`. L'argument `i` doit être compris dans l'intervalle `[0, 1114111]`.
- `ord(c)` l'inverse de la fonction `chr()`. Retourne le code *Unicode* (*Unicode code point* dans la terminologie *Unicode*) de l'argument, qui doit être un caractère *Unicode*.
- `str(c, code)` pour transformer une chaîne de caractères `c`, codée au format `code`, en une chaîne de caractères *Python* codée suivant le codage par défaut, il faut utiliser ce constructeur de la classe `str`.

4.6. Les ensembles (set et frozenset)

Python comporte un type de données pour représenter des ensembles. Un `set` est une collection (non ordonnée) sans éléments dupliqués. Les emplois basiques sont le test d'appartenance et l'élimination des entrées dupliquées. Les objets ensembles supportent les opérations mathématiques comme

- l'union avec l'opérateur `'|'`
- l'intersection avec l'opérateur `'&'`
- la différence avec l'opérateur `'-'` et
- la différence symétrique (définie par $A \Delta B = A \cup B - A \cap B$) avec l'opérateur `'^'`.

¹¹On trouvera une description plus complète de ces formats à l'adresse <http://docs.python.org/library/stdtypes.html>

Les instructions suivantes en donnent une démonstration succincte :

```

1 >>> a = set('abracadabra')
2 >>> b = set('alacazam')
3 >>> a
4 set(['a', 'r', 'b', 'c', 'd'])
5 >>> a-b
6 set(['r', 'b', 'd'])
7 >>> a|b
8 set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
9 >>> a&b
10 set(['a', 'c'])
11 >>> a^b
12 set(['b', 'd', 'm', 'l', 'r', 'z'])
13 >>> 'z' in a
14 True

```

Python propose également le type **frozenset** qui est analogue au type **set** mais non modifiable.

4.7. Les dictionnaires (dict)

Le type dictionnaire (**dict**) est intégré à **Python**.

Cette section précise et complète ce qui a été vu à la section 2.9 page 41.

Comme dans un dictionnaire ordinaire où les éléments sont des couples (mot : explication), les éléments des dictionnaires **Python** sont des couples (clé : valeur). Les clés sont les index, qui peuvent être de n'importe quel type *non-modifiable* et chaque clé est *unique* dans un même dictionnaire.

Les chaînes et les nombres peuvent toujours être des clés.

Un couple d'accolades crée un dictionnaire vide : {}.

Les instructions suivantes créent toutes le même dictionnaire {'un' :1, 'deux' :2} :

```

1 >>> {'un' :1, 'deux' :2}
2 >>> dict({'un' :1, 'deux' :2})
3 >>> dict(un=1, deux=2)
4 >>> dict(zip(['un', 'deux'], (1, 2)))
5 >>> dict(['un', 1], ['deux', 2])

```

Si **d** est un dictionnaire, les opérations supportées par **d** sont les suivantes :

- **len(d)** retourne le nombre d'items de **d** ;
- **d[key]** retourne la valeur associée à la clé **key**. Si **key** n'est pas une clé de **d**, l'exception **KeyError** est levée ;

- `d[key] = valeur` affecte valeur à la clé `key`, avec création ou remplacement de la valeur suivant que la clé existe déjà ou pas. L'ancienne valeur est alors perdue ;
- `del d[key]` supprime l'item `d[key]` s'il existe, lève l'exception **KeyError** sinon ;
- `d.pop(key[, default])` si la clé `key` existe dans `d`, elle est supprimée et sa valeur retournée. Sinon, `default` est retourné. Si `default` n'est pas donné, et la clé `key` n'est pas dans `d`, alors l'exception **KeyError** est levée.
- `key in d` retourne `True` si `d` contient la clé `key`, `False` sinon ;
- `d.clear()` supprime tous les items de `d` ;
- `d.copy()` retourne une copie de `d`.
- `d.keys()` retourne une vue des clé du dictionnaire. En **Python** 2, retourne une liste des clés de `d`. La différence entre une vue des clés et une liste des clés est que la vue est dynamique, elle est mise à jour au fur et à mesure de l'évolution de `d`, tandis que la liste des clés est un instantané de l'état des clés au moment de sa création.
- `d.items()` retourne une *vue* des items du dictionnaires (c'est à dire des couples (clé, valeur))
- `d.values()` retourne une *vue* des valeurs du dictionnaire (triée dans le même ordre que la vue des clés)

Exemple d'utilisation des vues :

```
1 >>> d = {'un' :1, 'deux' :2}
2 >>> keys = d.keys()
3 >>> keys
dict_keys(['un', 'deux'])
4 >>> d['infini'] = 'beaucoup'
5 >>> keys
dict_keys(['un', 'deux', 'infini'])
```

4.8. Les exceptions (exceptions)

Lorsqu'un erreur d'exécution survient, **Python** lève une *exception*. Le programme stoppe et **Python** affiche la pile d'appels, et l'exception.

Il est possible d'écrire des programmes qui prennent en charge des exceptions spécifiques. L'exemple suivant, interroge l'utilisateur jusqu'à ce qu'un entier valide ait été saisi, et lui permet d'interrompre le programme en utilisant `Ctrl-C` ou une autre combinaison de touches reconnue par le système d'exploitation (il faut savoir qu'une interruption produite par l'utilisateur est signalée en levant l'exception `KeyboardInterrupt`).

```
1 while 1 :
2     try :
3         x = int(input(u"Veuillez entrer un nombre : "))
4         break
5     except ValueError :
```

```
6 print u"Aïe ! Ce n'était pas un nombre valide. Essayez encore  
   ..."
```

Fonctionnement :

La clause **try** : les instructions entre les mots-clés **try** et **except** est exécutée.

- S'il ne se produit pas d'exception, la clause **except** est ignorée, et l'exécution du **try** est terminée.
- Si une exception se produit, le reste de la clause **try** est ignoré. Puis si son type correspond à l'exception donnée après le mot-clé **except**, la clause **except** est exécutée, puis l'exécution reprend après l'instruction **try**.
- Si une exception se produit qui ne correspond pas à l'exception donnée dans la clause **except**, elle est renvoyée aux instructions **try** extérieures.

L'instruction **raise** vous permet de lever vous même une exception.

Par exemple :

```
1 def get_age() :  
2     age = input('Please enter your age : ')  
3     if age < 0 :  
4         raise ValueError, '%s is not a valid age' % age  
5     return age
```

L'instruction **raise** prend deux arguments : le type de l'exception et une chaîne de caractère qui décrit l'exception.

ValueError est une exception standard.

La liste complète des exceptions :

```
>>> help('exceptions')
```

Les exceptions les plus courantes :

- Accéder à une clé non-existante d'un dictionnaire déclenche une exception **KeyError**
- Chercher une valeur non-existante dans une liste déclenche une exception **ValueError**.
- Appeler une méthode non-existante déclenche une exception **AttributeError**.
- Référencer une variable non-existante déclenche une exception **NameError**.
- Mélanger les types de données sans conversion déclenche une exception **TypeError**.

Exercices

Exercice 24. Écrire une expression qui vaut *True* si l'entier *n* est pair et *False* dans le cas contraire.

Exercice 25. Consultez la liste des répertoires dans lesquels l'interpréteur recherche les modules. Combien y a-t-il de répertoires distincts ?

Exercice 26. Builtins

Depuis une session **Python**

- (1) déterminer le type de l'objet `__builtins__`
- (2) Affecter à la variable *L* une liste des clés de `__builtins__`
- (3) que vaut `L[42]` ?
- (4) Trouver l'aide sur la fonction `sum`

Exercice 27. Formatage.

Écrire une table des carrés et des cubes présentée comme suit :

0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Exercice 28. Transformer la chaîne de caractère `"3.14"` en un flottant.

Exercice 29. Chaînes

Soit la chaîne de caractères

`a='1.0 3.14 7 8.4 0.0'`

Écrire une instruction unique utilisant les fonctions et méthode `float()` et `split()`, qui construit la liste de réels

`[1.0, 3.14, 7.0, 8.4, 0.0]`

à partir de *a*.

Exercice 30. Chaînes

Soit la chaîne de caractères `c="(1.0, 2.0)(3, 4)\n"`. Trouver une suite d'instructions permettant d'en extraire les deux nombres complexes $x=1+2j$ et $y=3+4j$. On pourra par exemple :

- (1) Nettoyer `C` de son `'\n'` → `C='(1.0, 2.0)(3, 4)'`
- (2) Remplacer `','` par `','` → `C='(1.0,2.0)(3,4)'`
- (3) Scinder (`=split()`) la chaîne en deux → `C=['(1.0,2.0)', '(3,4)']`
- (4) A l'aide de la fonction `eval()` et d'un déballage d'arguments, instancier le complexe $1+2j$ à partir de `'(1.0,2.0)'`

Exercice 31. Chaînes, module `string`

- (1) Récupérer le fichier `HUMOUR.txt` de `chap2-premiersPas`, lire son contenu et supprimez tous les signes de ponctuation, ainsi que les `'-'`.
- (2) Enregistrer le résultat dans un fichier `HUMOUR-propre.txt`

Exercice 32. Que fait la suite d'instructions :

```

1 >>> F = open('toto.csv')
2 >>> [[[g.strip() for g in f.strip().split(';')] \
3         for f in e.strip().split('\n')] \
4         for e in F.readlines()[1 :]]
5 >>> F.close()

```

Peut-on améliorer ?

Exercice 33. Formatage

On donne une liste de d'articles et la liste des prix correspondants. Écrire une instruction qui imprime la liste des (prix : article). Par exemple

```

1 >>> articles = ['mousqueton', 'reverso', 'crochet',
2                 'baudrier', 'chausson', 'magnesie']
3 >>> prix = [5, 25, 5, 80, 120, 2]

```

produira l'affichage

```

1 mousqueton : €5
2 reverso : €25
3 crochet : €5
4 baudrier : €80
5 chausson : €120
6 magnesie : €2

```

Exercice 34. Chaînes, encodage

- (1) Écrire la chaîne de caractères `'il vous en coûtera 45€'` dans deux fichiers texte : l'un codée en `utf-8`, l'autre en `ISO 8859-15`.

- (2) Écrire un script `utf2win.py` qui lit un fichier texte encodé en utf-8, et qui le sauvegarde avec l'encodage Windows-1252. On suppose connu l'encodage du fichier lu (en principe c'est l'encodage par défaut de votre système). Pour déterminer l'encodage d'un fichier texte, il n'existe pas de méthode absolument sûre. La commande Unix `file` permet dans certains cas, (mais pas toujours) de le déterminer :

```
$ echo 'éé€'> toto
$ file toto
toto : UTF-8 Unicode text
```

Exercice 35. Lecture-écriture de fichiers

- lire le contenu du fichier `humour.txt`, écrire son contenu à l'écran en majuscule.
- demander à l'utilisateur un nom de fichier, le lire et recopier son contenu, avec fer à droite, dans le fichier `HUMOUR.txt`.
- Créer une liste de votre choix, picklez la dans un fichier `liste.pic`, unpicklez-la dans une variable `zoe`

Exercice 36. Lecture

- (1) Lors de la lecture d'un fichier, le programme lit la chaîne de caractères `s='1;2; 3; 4; 5'`.
- (2) Quelle instruction unique permet d'en faire la liste d'entiers `l=[1, 2, 3, 4, 5]` ?
- (3) Idem, avec `s='1;2; 3; 4; 5;;'`
- (4) Idem, avec `s='1;2; 3; -4; 5;;'` et on ne garde dans `l` que les entiers positifs.

Exercice 37. Lecture sur fichier

- (1) Ecrire un script qui crée un dictionnaire, `d`, dont les clés et les éléments sont lus sur le fichier Octave¹² suivant :

```
# Created by Octave 2.9.12
# name : z
# type : complex scalar
(1,2)
```

- (2) Améliorer le script précédent pour qu'il prenne en compte les enregistrements dans le même fichier Octave, du type suivant :

```
# name : chaine
# type : string
hello
```

- (3) Même question pour des variables matrices :

```
# name : M
# type : matrix
# dimensions : 2 4
1 2 7 6
2 3 8 7
```

¹²Octave est un logiciel libre de calcul scientifique très robuste, analogue à Scilab, Matlab

Les clés sont les noms des variables et les éléments sont les valeurs de ces variables. On testera les instructions programmées sur le fichier `vars.oct`.

Exercice 38. Lecture sur fichier csv

- (1) Une expérience fournit un fichier texte constitué d'une ligne de titre, suivie de lignes, chaque ligne constituée de 10 nombres réels séparés par des virgules. Écrire une fonction qui lit ce fichier et qui retourne la liste des lignes du fichier, chaque ligne étant convertie en une liste de 10 nombres réels.

```
#ux, uy, uz, tx, ty, tz, wx, wy, wz, d
1.0, 2.0, 3.0, -1.0, 0.0, 1.0, 1.0, 0.5, 0.4, 7.2
etc...
```

- (2) Même chose, mais des enregistrements peuvent être vides, comme celui-ci :

```
, , -1.0, 0.0, 1.0, 1.0, 0.5, , 7.2
```

- (3) Même chose, en ne lisant que les colonnes `wx`, `wy`, `wz`

Exercice 39. Instanciation dynamique

Reprendre le script de l'exercice précédent, et modifiez le afin que les entrées du dictionnaire créé soit instanciées comme variables dans une session **Python**. La fonction `execfile()` permet d'exécuter un script **Python** depuis une session **Python**.

Exercice 40. Dictionnaires

La commande Unix :

```
$ ls -l >list.txt
```

permet de lister tous les fichiers du répertoire courant et de sauvegarder le résultat dans le fichier `list.txt`.

- (1) Écrire l'instruction qui permet de lancer cette commande Unix depuis **Python** ;
 (2) relire le fichier `list.txt` pour créer un dictionnaire dont les clés sont les noms des fichiers du répertoire courant, et dont les champs sont la taille de ces fichiers.

Exercice 41. Ensembles

- (1) Soit `L` une liste. Supprimez, à l'aide d'une seule instruction simple, tous les doublons de `L`. Calculer le nombre d'éléments distincts de `L`.
 (2) Reprendre l'exercice précédent où cette fois `L` est une liste de chaînes de caractères ASCII (pas de lettres accentuées). On considère qu'une chaîne « double » avec une autre si elle sont identiques sans tenir compte de la casse (par exemple `'Everest'` et `'eVereSt'` sont identiques). Supprimez tous les doublons de `L`.

Exercice 42. Richesse lexicale

Écrire un script **Python** nommé `rilex.py` qui calcule la richesse lexicale d'un texte contenu dans un fichier dont le nom est passé en argument du script. La richesse lexicale d'un texte est définie comme le quotient entre le nombre de mots différents et le nombre total de mots du texte. Dans

cet exercice, on définit la notion de "mot" comme toute séquence de taille supérieure ou égale à quatre, formée exclusivement de caractères alphabétiques (on ne distinguera pas les majuscules des minuscules).

Exercice 43. Exceptions

Testez les instructions suivantes, quel est le nom de l'exception ?

- une division par zéro :

```
>>> 2/0
```
- un indice hors tableau :

```
>>> a=[]
>>> a[2]
```
- assigner une valeur à un item dans un tuple :

```
>>> tuple=(1, 2, 3, 4)
>>> tuple[3]=45
```

Exercice 44. Exceptions

En utilisant le mécanisme des exceptions, écrire une fonction `isFile(fichier)` qui renvoie `True` si le fichier existe, `False` sinon

Exercice 45. Exceptions

On dispose d'une liste `L` contenant des objets de type varié.

Par exemple `L=[1, '3.14', [], (2, 5), 'hello', {'d' : 3, 2.145}]`. La fonction suivante extrait de `L` tous les réels pour alimenter et retourner une nouvelle liste `P`, mais on oublie simplement le `'3.14'`

```
1 def listeDeReels(L) :
2     P=[]
3     for l in L :
4         if type(l) in (int, float, long) :
5             P.append(float(l))
6     return P
```

- Réécrire la fonction pour prendre en compte le cas des nombres sous forme de chaînes de caractères.
- Utiliser le mécanisme des exceptions pour obtenir le même résultat.

Solutions des exercices

Solution 24 Expression booléenne.

```
1 >>> n%2==0
```

Solution 26 Builtins

```
1 >>> type(__builtins__)
2 <type 'dict'>
```

```
1 >>> L=dir(__builtins__)
```

```
1 >>> L[42]
2 'values'
```

```
1 Help on built-in function sum in module __builtin__ :
2 sum(...)
3     sum(sequence[, start]) -> value
4     Returns the sum of a sequence of numbers (NOT strings) plus the
5     value
6     of parameter 'start' (which defaults to 0). When the sequence
    is
    empty, returns start.
```

Solution 27 Formatage, table des carrés et des cubes

```
1 >>> for x in range(1,11) :
2     ... print ('%2d %3d %4d' % (x, x*x, x*x*x))
```

(Notez qu'un espace entre chaque colonne a été ajouté à cause de la façon dont `print` fonctionne : elle ajoute toujours des espaces entre ses arguments.)

Solution 28 Transformer une chaîne de caractère en un flottant.

```
1 >>> float('3.14')
```

Solution 29 Lire des réels dans une chaîne de caractères

```
1 >>> [float(m) for m in a.split()]
```

Solution 30 Lecture de complexes dans une chaîne de caractères

Une solution compacte :

```
1 >>> C="(1.0, 2.0) (3, 4)\n"
2 >>> [complex(*eval(c)) for c in C.replace(', ', ',').split()]
3 [(1+2j), (3+4j)]
```

l'instruction `complex(*eval(c))` dans laquelle `c` est la chaîne `'(3, 4)'`, se déroule ainsi :

- `eval(c)` renvoie le couple d'entiers (3, 4)
- `complex(*(3,4))` équivaut à `complex(3,4)` par déballage d'arguments, et renvoie le complexe (3+4j)

Solution 31 TODO**Solution 32** La suite d'instruction donnée dans l'énoncé :

ligne 1 : ouvre un fichier .csv (en lecture par défaut),

lignes 2 à 4 : lit les lignes `e`, du fichier sauf la première `F.readlines()[1 :]`. Chaque ligne `e`, est nettoyée par `strip()`, et transformée en liste `f` par `split('\n')`. Comme la ligne `e` ne contient en principe qu'un seul `\n`, cette étape est inutile, car `f` est de longueur 1.

L'unique élément de la liste `f` est ensuite nettoyé, puis transformé en liste par découpage au point-virgule, chaque élément de la liste est nettoyé. On obtient une liste de listes de listes hideuse ressemblant à

```
1 [
2   [['K2', '8611', 'Pakistan,Chine', 'Karakoram', '']],
3   [['Pumori', '7161', '', '', '']],
4   [['Shishapangma', '8013', '', '', '']],
5   ...,
6   [['Mont Woodall', '246', 'USA', 'Mississippi', '']]
7 ]
```

Il est probable que l'auteur de ces lignes voulait une liste lisible, comme celle qui suit.

```
1 [['K2', '8611', 'Pakistan,Chine', 'Karakoram', ''],
2  ['Pumori', '7161', '', '', ''],
3  ['Shishapangma', '8013', '', '', ''],
4  ...,
5  ['Mont Woodall', '246', 'USA', 'Mississippi', '']]
```

Une solution pour l'obtenir consiste à décomposer la tâche en écrivant :

```
1 >>> A = open('toto.csv').readlines()[1 :]
2 >>> B = [s.strip('\n ;') for s in A]
3 >>> C = [s.split(';') for s in B if s]
```

L'assemblage de la liste `C` ne retient que les enregistrements non vides grâce au `if` `s`. Si l'on tient à écrire une seule instruction on obtient le même résultat avec :

```
1 >>> C = [r.split(';') for r in [r.strip('\n ;') \
2   for r in open('toto.csv').readlines()[1 :] if r] if r]
```

dont la lisibilité est déplorable !

Solution 33 Formatage

```

1 >>> for a,p in zip(articles,prix) :
2     print ('{0 :10s} : {1 :3d} €'.format(a,p))

```

Solution 34 Chaînes, encodage

- (1) Il suffit d'encoder la chaîne en *utf-8* avec la méthode `str.encode(format)` puis de l'écrire en binaire dans le fichier `xxx.utf8`. Répéter la même opération avec le format *ISO 8859-15* et un fichier `xxx.iso15`.

```

1 >>> a = 'il vous en coûtera €45'
2 >>> f = open('xxx.iso15', 'wb')
3 >>> f.write(a.encode('ISO 8859-15'))
4 22
5 >>> f.close()
6 >>> f = open('xxx.utf8', 'wb')
7 >>> f.write(a.encode('utf-8'))
8 25

```

La méthode `write()` retourne le nombre d'octets écrits. On remarque qu'il n'est pas le même suivant l'encodage choisi. Les caractères 'û' et '€' sont codés sur 1 octet en *ISO 8859-15* et sur 2 et 3 octets en *utf-8*. D'où la différence de $25 - 22 = 3$ octets constatée ici.

Les instructions précédentes peuvent être écrites de manière plus simple. En effet la fonction builtins `open()` propose l'interface :

```

1 open(file, mode='r', buffering=-1, encoding=None, errors=None,
2       newline=None, closefd=True)

```

l'argument `encoding` permet de demander l'encodage voulu lors de l'écriture. On n'a plus besoin de coder explicitement la chaîne de caractères, et le fichier n'a pas à être ouvert en écriture binaire (mode `'wb'`) mais en écriture standard (mode `'w'`). Les instructions se simplifient :

```

1 >>> a = 'il vous en coûtera €45'
2 >>> f = open('xxx.iso15', 'w', encoding='ISO 8859-15')
3 >>> f.write(a)
4 22
5 >>> f.close()
6 >>> f = open('xxx.utf-8', 'wb', encoding='utf8')
7 >>> f.write(a)
8 25

```

- (2) En utilisant la méthode décrite ci-dessus, un script nu pourrait s'écrire :

```

1 import sys
2 infile,outfile = sys.argv[0], sys.argv[1]
3 s = open(infile).read()
4 open(outfile,'w', encoding='Windows-1252').write(s)

```

Si on en fait un « vrai » script *Python*, y compris précautions, atermoiements et diverses mises en garde, on obtient

```

1 #!/usr/bin/python3.1
2 # -*- coding : utf-8 -*-
3 import sys
4
5 def convertFile(infile, outfile, format='utf8') :
6     try :
7         s = open(infile, encoding='utf8').read()
8     except IOError :
9         print ("Impossible d'ouvrir le fichier {}".format(infile)
10              )
11         return False
12     try :
13         open(outfile,'w',encoding=format).write(s)
14     except IOError :
15         print ("Impossible d'écrire dans le fichier {}".format(
16             outfile))
17         return False
18     return True
19
20 if __name__ == '__main__' :
21     if len(sys.argv) != 3 :
22         print('')
23         utf2win.py, pour convertir un fichier utf8 en Windows-1252.
24
25         Nombre d'arguments ({{}}) invalide. Deux arguments étaient
26         attendus.
27
28         Usage : $ python3.1 utf2win infile outfile
29         ''.format(len(sys.argv)-1))
30         sys.exit()
31
32 # print (convertFile(sys.argv[1], sys.argv[2], 'ISO 8859-15'))
33 print (convertFile(sys.argv[1], sys.argv[2], 'Windows-1252')
34       )

```

Solution 35 TODO

Solution 36 Lecture

```
1 l = [int(mot) for mot in s.split(';') ]
```

```
1 l = [int(mot) for mot in s.strip(';').split(';')]
```

```
1 l = [int(mot) for mot in s.strip(';').split(';') if int(mot)>0]
```

Solution 37 Lecture

Ce dictionnaire devra donc avoir la structure suivante :

```
1 dv = {'M' : [[1, 2, 7, 6], [2, 3, 8, 7]];
2       'z' : (1+2j);
3       'chaine' : 'hello'}
```

Voici un script qui effectue les tâches demandées :

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 import sys,os
4
5 dv={}
6 L = [l.strip() for l in open('vars.oct') if l.strip()]
7 for l in L :
8     if l[0] == '#' : #entete
9         if l.find("# name :")>=0 :
10             nbliq = 0
11             nom = l.split()[-1]
12             dv[nom] = [] #creation de la cle du dico
13         elif l.find("# type :")>=0 :
14             #typ=l[8:].strip()
15             typ = l.split(' ')[1].strip()
16         elif l.find('# dimensions :')>=0 :
17             [m,n]=[int(w) for w in l[13:].strip().split()]
18             print m,n
19         elif l.find('# length :')>=0 :
20             le=int(l[10:].strip())
21             #dv[nom].append(le)
22         else :continue
23     else :
24         #fin entete, debut donnees
25         if typ=='complex scalar' :
26             xx = [float(r) for r in l.strip('()\n ').split(',')]
```



```

27     z = complex(*xx)#conversion en complexe
28     dv[nom] = z#ajout dico
29     elif typ=='string' :
30         dv[nom].append(l.strip())#ajout dico, direct
31     elif l.strip() and typ=='matrix' :# and nbliq<m :
32         ligne=[float(a) for a in l.split()]
33         dv[nom].append(ligne)
34
35 for key, value in dv.iteritems() :
36     instruction = key+'='+str(value)
37     exec(instruction)
38     print "%s=%s"%(key,eval(key))

```

Solution 38 TODO**Solution 39** Instanciation dynamique

Le script suivant effectue les tâches demandées :

```

1
2 for key, value in dv.iteritems() :
3     instruction = key+'='+str(value)
4     exec(instruction)
5     print "%s=%s"%(key,eval(key))

```

Solution 40 Le script *Python* suivant répond à la question :

```

1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3
4 import os,sys
5
6 os.system('ls -l>list.txt')
7 d={}
8 F=open('list.txt').readlines()[1 :]
9 for f in F :
10     f=f.strip().split()
11     (l,nom)=int(f[4]),f[7].strip()
12     d[nom]=l
13 print(d)

```

Solution 41 Ensembles

- (1) Un ensemble peut être vu comme une liste sans doublon. Il suffit donc de transformer la liste en ensemble et de compter le nombre d'éléments :

```

1 >>> L = list('Anti-Constitutionnellement')
2 >>> set(L)
3 {'A', 'C', 'e', 'i', 'm', '-', 'l', 'o', 'n', 's', 'u', 't'}
4 >>> len(set(L))
5 12

```

- (2) Écrire une liste en compréhension en appliquant la méthode `lower()` à tous les caractères est une solution.

```

1 >>> set([l.lower() for l in L])
2 {'a', 'c', 'e', 'i', 'm', '-', 'l', 'o', 'n', 's', 'u', 't'}

```

Solution 42 Richesse lexicale

```

1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 import sys
4 try : f=sys.argv[1]
5 except IndexError :
6     print(u"Donnez le nom du fichier")
7     exit()
8
9 try : text = open(f).read()
10 except IOError :
11     print (u"Problème de lecture sur le fichier %s"%f)
12     exit()
13 text = unicode(text,"utf-8")
14 text = text.replace('\'', ' ')
15 words = [w.strip('.,;!?()').lower() for w in text.split() if len(w) > 3]
16
17 #print u', '.join(sorted(words))
18
19 print len(words), len(set(words)), " => ", float(len(set(words)))/len(
    words)

```

Pour appeler ce script, il faut lui passer le nom du fichier à analyser. Par exemple :

```
$ python rlex.py fichier_a_analyser
```

- Lignes 4 à 7 : s'il n'y a pas d'argument sur la ligne de commande, alors `sys.argv[1]` n'existe pas et l'exception **`IndexError`** est levée. La clause **`except`** est alors exécutée (lignes 6 et 7)
- Lignes 8 à 12 : si la lecture (clause **`try`**) lève une exception **`IOError`**, alors la clause **`except`** est exécutée (lignes 11 et 12)
- Ligne 13 : le texte lu est encodé en utf-8.
- Ligne 14 : les apostrophes sont supprimées, remplacées par une espace.
- Ligne 15 : le texte est découpé en mots de plus de 4 caractères, puis les caractères `.,!?` en début et fin de mot sont supprimés, et enfin les mots sont convertis en minuscule. Les mots sont mis dans la liste `words`.
- Ligne 19 : le nombre de mots est obtenu par `len(words)`, et le nombre de mots distincts par `len(set(words))` puisque les doublons sont éliminés dans un ensemble (`set`). On obtient la richesse lexicale en prenant soin de faire une division réelle et non pas entière (laquelle aurait pour résultat 0).

Solution 44 Exceptions

```

1 def isfile(fichier) :
2     try :
3         f=open(fichier,'r')
4         f.close()
5         return True
6     except IOError :
7         return False

```

Solution 45 Exceptions

Une liste de réels.

Sans le mécanisme des exceptions

```

1 def isNumber(s) :
2     """Détermine si une chaîne de caractères est un nombre"""
3     for c in s :
4         if not c in string.digits+'.-' : return False
5     return True
6
7 def listeDeReels(L) :
8     P=[]
9     for l in L :
10        if type(l) in (int, float) :
11            P.append(float(l))
12        elif type(l) is str and isNumber(l) :
13            P.append(float(l))
14    return P

```

Avec le mécanisme des exceptions

```
1 def listeDeReels1(L) :  
2     P=[]  
3     for l in L :  
4         try : P.append(float(l))  
5         except (ValueError, TypeError) as err :  
6             print (type(err).__name__, ' : ', l, ' impossible à  
                    convertir en réel')
```

Appel des fonctions

```
1  
2 if __name__ == '__main__' :  
3     L=[1, '3.14', [], (2,5), 'hello', {'d' :3}, 2.145]  
4     print (isNumber('3.14'))  
5     print (listeDeReels(L))  
6     print (listeDeReels1(L))
```

CHAPITRE 5

Fonctions, scripts et modules

Jusqu'ici, nous avons rencontré et utilisé de nombreuses fonctions. Dans ce chapitre, nous étudions plus en détail quelques spécificités *Python* de la programmation fonctionnelle.

5.1. Fonctions

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

```
def <nom de fonction> (<liste de parametres>) :  
    ["""<la documentation>"""]  
    <instructions>  
    [return [<valeur de retour>]]
```

Le mot-clé *def* annonce la définition d'une fonction. Il doit être suivi par le nom de la fonction et une liste entre parenthèses de paramètres formels suivit de deux-points ' : '.

En *Python* les fonctions sont des objets (au sens de la programmation orientée objets). Les fonctions sont de type *function*, qui hérite du type *object* :

```
1 >>> def f(x) :pass  
2 ...  
3 >>> type(f)  
4 <type 'function'>  
5 >>> isinstance(f, object)  
6 True
```

À ce titre les fonctions possèdent des attributs dits *spéciaux* dont les plus courants sont :

- `__doc__` la docstring ou `None` ;
- `__name__` le nom de la fonction ;
- `__module__` le nom du module définissant la fonction, ou `None` ;

5.1.1. Exemple détaillé. Dans un fichier `fibonacci.py`, nous pouvons créer une fonction qui écrit la série de Fibonacci jusqu'à une limite quelconque :

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import doctest
4  def fib(n) :
5      """
6      Retourne une liste contenant
7      la série de Fibonacci jusqu'à n
8      >>> fib(100)
9
10     """
11     resultat = []
12     a, b = 0, 1
13     while b < n :
14         resultat.append(b)
15         a, b = b, a+b
16     return resultat
17 if __name__ == '__main__' :
18     doctest.testmod()

```

Les instructions qui forment le corps de la fonction commencent sur la ligne suivant la déclaration, indentée par des espaces. Une chaîne de documentation de la fonction, ou *docstring* peut être utilisée pour expliquer le comportement de la fonction.

L'exécution d'une fonction génère une nouvelle *table de symboles*, utilisée pour les variables locales de la fonction.

Les vrais paramètres (arguments) d'un appel de fonction sont introduits dans la table de symboles locale de la fonction appelée quand elle est appelée. Les arguments sont passés en utilisant un *passage par affectation*.

Un fonction est un objet *Python* comme un autre. Il est possible de renommer une fonction :

```

1  >>> from fibonacci import fib
2  >>> fib
3  <function object at 10042ed0>
4  >>> f = fib
5  >>> f(100)
6  1 1 2 3 5 8 13 21 34 55 89

```

La fonction `fib()` retourne la liste des nombres de Fibonacci inférieurs à `n`.

Une fonction retourne toujours une valeur, éventuellement la valeur `None`

```
1 >>> def f(x) : x[1]=3.14
2 >>> print(f([1,2,3]))
3 None
```

La première instruction d'une fonction, ou d'un module, devrait être la chaîne de documentation, ou *docstring*. Certains outils utilisent les *docstrings* pour générer automatiquement de la documentation papier, ou pour permettre à l'utilisateur de naviguer interactivement dans le code. Un *docstring* peut s'étendre sur plusieurs lignes.

Les *docstrings* sont également utilisées par la fonction `help()` comme le montre l'exemple suivant.

```
1 >>> def carre(x) :
2 ...     """Retourne le carré de son argument."""
3 ...     return x*x
4 ...
5 >>> help(carre)
6 Help on function carre in module __main__ :
7 carre(x)
8     Retourne le carré de de son argument
```

5.1.2. Passage d'arguments. Les arguments sont passés aux fonctions *par affectation*, ce qui signifie que les arguments sont simplement affectés à des noms locaux (à la fonction).

Pratiquement, ce mode de passage par affectation a les conséquences suivantes :

- les arguments *non modifiables* miment le comportement du passage par valeur de C, mais *ne sont pas copiés* (gain de temps) ;
- les arguments *modifiables* se comportent comme des arguments passés par référence en C et ne sont pas copiés non plus.

Attention aux effets de bord, l'affectation crée une référence sur la valeur affectée (sur l'argument) et non pas une copie profonde. Par exemple :

```
1 >>> def modif(liste) :
2 ...     liste[0] = 3.14
3 >>> l = [1,2,3]
4 >>> modif(l)
5 >>> l
6 [3.14, 2, 3]
```

La séquence d'instructions suivante peut être déroutante et mérite quelques explications :

```

1 >>> def f(x) :
2 ...     x = []
3 >>> l = list(range(3))
4 >>> l
5 [0, 1, 2]
6 >>> f(l)
7 >>> l
8 [0, 1, 2]
9 >>> def g(x) :
10 ...     x[ :] = []
11 >>> g(l)
12 >>> l
13 []

```

La fonction `f()`, avec l'instruction `x = []`, crée une liste vide et affecte le nom `x` à cette liste vide. Le nom de variable `x` n'est plus associé à la liste `l`, argument de `f()`.

La fonction `g()`, elle, ne réaffecte pas le nom `x` de son argument à une autre liste, mais modifie directement la valeur de `x`, c'est à dire la valeur de l'argument c'est à dire la valeur de `l`.

Il est possible de définir des fonctions à nombre d'arguments variable. Il y a plusieurs façons de faire, qui peuvent être combinées.

Valeurs d'argument par défaut. La technique la plus utile consiste à spécifier une *valeur par défaut* pour un ou plusieurs arguments. Cela crée une fonction qui peut être appelée avec moins d'arguments qu'il n'en a été défini. Les arguments par défaut doivent être les derniers de la liste d'arguments. Par exemple, si l'on définit la fonction suivante :

```

1 def interrogatoire(nom, prenom='', presume='coupable') :
2     print("Bonjour, monsieur ", nom, prenom )
3     print("Vous êtes présumé ", presume)

```

Cette fonction peut être appelée soit en utilisant une ou des valeurs par défaut :

```

1 >>> interrogatoire('Puisseux')
2 Bonjour, monsieur Puisseux
3 Vous êtes présumé coupable

```

ou en précisant la valeur de certains arguments :

```

1 >>> interrogatoire('Puissant', '', 'innocent')
2 Bonjour, monsieur Puissant
3 Vous êtes présumé innocent

```


Si une valeur par défaut est de type modifiable, les effets de bord peuvent être difficile à déceler. Dans l'exemple suivant, la valeur par défaut est une liste (modifiable) :

```

1 >>> def f(l=[0,0]) :
2 ...     l[0] += 1
3 ...     return l
4 >>> f()
5 [1, 0]
6 >>> f()
7 [2, 0]
8 >>> f([0,0])
9 [1, 0]
```

L'argument par défaut `l=[0, 0]` est créé lorsque la fonction `f()` est compilée. La fonction n'est compilée qu'une fois, la liste n'est donc pas recréée entre les différents appels, et la liste reste en l'état entre les différents appels. Après le premier appel elle vaut donc `[1,0]`, état qu'elle conserve jusqu'au second appel lors duquel elle passe à l'état `[2, 0]`. Lorsque la fonction est appelée avec un paramètre, la valeur par défaut n'est pas modifiée, c'est le paramètre effectif qui est modifié.

Arguments à mot-clé (ou avec étiquettes). Les fonctions peuvent aussi être appelées en utilisant des arguments mots-clés de la forme `motcle=valeur`. Dans ce cas, l'ordre des arguments peut être modifié. Par exemple, la fonction précédente peut être appelée ainsi :

```

1 >>> interrogatoire(prenom='Pierre', presume='innocent', nom='
    Puiseux')
2 Bonjour, monsieur Puiseux Pierre
3 Vous êtes présumé innocent
```

Listes arbitraire d'arguments. Il est également possible de spécifier un nombre arbitraire d'arguments. Ces arguments seront récupérés dans un **tuple**. Avant le nombre variable d'arguments, des arguments normaux peuvent être présents.

Dans cet exemple, le **tuple** est `x`, et `*x` est le tuple *dépaqueté* ou *déballé*

```

1 >>> def f(*X) :
2 ...     for x in X : print(x, end=' ')
3 >>> f(1,2,3)
4 1 2 3
```

Arguments dans un dictionnaire. Les arguments de la fonction peuvent être quelconques. Ils doivent être nommés *au moment de l'appel* comme pour une liste d'arguments à mots-clé. Lors d'un appel, **Python** passe en argument à la fonction un dictionnaire dont les clés sont les noms des arguments nommés. La fonction est appelée avec une liste d'arguments comme une fonction ordinaire. Par exemple :

```

1 >>> def f(**D) :
2 ...     for key, val in D.items() :
3 ...         print(key, ' : ', type(val), val)
4 >>> f(a='Zorro', b=3.14, c=[1,2])
5 a : <class 'str'> Zorro
6 c : <class 'list'> [1, 2]
7 b : <class 'float'> 3.14
8 >>> f(L=[], F=0.1, I=1953)
9 I : <class 'int'> 1953
10 L : <class 'list'> []
11 F : <class 'float'> 0.1

```

Avant l'argument dictionnaire, d'autres arguments ou même une liste arbitraire d'arguments (*X) peuvent être présents. Dans l'exemple qui suit, dans la portée de la fonction `f()`, l'argument `L` est un tuple, et l'argument `D` est un dictionnaire :

```

1 >>> def f(p, *L, **D) :
2 ...     print(p, L, D, sep='\n')
3 >>> f(0, 1, 2, Paul=1980, Jean=1979)
4 0
5 (1, 2)
6 {'Paul' : 1980, 'Jean' : 1979}

```

5.1.3. Les formes lambda. Elles sont une manière d'écrire des fonctions à la volée, sans avoir besoin de passer par une instruction `def`

Pour définir une fonction sous *forme lambda*, on utilise la syntaxe :

```

1 lambda param1,param2, ... : une_instruction_unique

```

Par exemple, dans l'instruction suivante on définit une fonction qui fait la somme de deux variable `x` et `y` et on l'applique au couple `(1, 3)`.

```

1 >>> (lambda x, y : x+y)(1,3)
2 4

```

`fabrique_incrementeur` crée et retourne une nouvelle fonction qui incrémente son argument à partir de `n` :

```

1 >>> def metaMultiplication(n) :
2 ...     return lambda x, mult=n : x*mult
3 ...
4 >>> M3 = metaMultiplication(3) #c'est une fonction

```

```

5 >>> M3(5)
6 15

```

Attention à ne pas en abuser, car les formes lambda sont un excellent moyen d'écrire des programmes abstrus. On trouve dans la *faq* de la documentation [Python](#) cet exemple qui est supposé calculer un ensemble de Mandelbrot.

```

1 print(lambda Ru,Ro,Iu,Io,IM,Sx,Sy :reduce(lambda x,y :x+y,map(
   lambda y,
2 Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,
   i=IM,
3 Sx=Sx,Sy=Sy :reduce(lambda x,y :x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru
   ,Ro=Ro,
4 i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f : (k<=0)or (x
   *x+y*y
5 >=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f) :f(xc,yc,x,y,k,f)
   :chr(
6 64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))) :L(Iu+y*(Io-Iu)/Sy),
   range(Sy
7 ))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24)
8 # \_ _ _ / \_ _ _ / | | | _ lines on screen
9 #      V      V   | | _ _ _ columns on screen
10 #      |      |   | _ _ _ _ _ maximum of "iterations"
11 #      |      |   | _ _ _ _ _ range on y axis
12 #      | _ _ _ _ _ range on x axis

```

5.2. Règles de portée

5.2.1. Espace de noms. Un *espace de noms* (*name space*) est une relation entre des noms et des objets. Par exemple :

- L'espace de noms appelé `__builtin__` contient l'ensemble des noms intégrés (les fonctions telles que `abs()`, et les noms d'exception intégrés). Il est créé au lancement de l'interpréteur [Python](#), et n'est jamais effacé.
- L'espace de noms global pour un module contient les noms globaux définis par le module. Il est créé quand la définition du module est chargée. Les instructions exécutées à l'invocation de l'interpréteur font partie d'un module appelé `__main__`, elles ont donc leur propre espace de noms global.
- L'espace de noms local à une fonction, contient les noms locaux au cours d'un appel de fonction. Il est créé quand celle-ci est appelée et il est effacé quand la fonction se termine.

il n'y a absolument aucune relation entre les noms contenus dans les différents espaces de noms.

par exemple, deux modules différents (disons `truc` et `troc` peuvent définir tous les deux une fonction `maximise()` sans confusion possible les utilisateurs des modules doivent préfixer par le nom du module à l'utilisation : `truc.maximise()` et `troc.maximise()`

5.2.2. Portée, la règle LGI. Une *portée* (*scope*) est une région textuelle d'un programme *Python* dans laquelle un espace de noms est directement accessible. A n'importe quel moment de l'exécution, exactement trois portées imbriquées sont utilisées (exactement trois espaces de noms sont accessibles directement, règle *LGI*) :

- (1) la *portée locale*, qui est explorée en premier, contient les noms locaux,
- (2) la *portée globale*, explorée ensuite, contient les noms globaux du module courant, et
- (3) la *portée interne* (explorée en dernier) correspond à l'espace de noms contenant les noms intégrés.

Si un nom est déclaré *global* alors les références et affectations le concernant sont directement adressées à la portée qui contient les noms globaux du module.

Chaque fonction définit son propre *espace de noms* (*namespace*). Une variable déclarée dans une fonction est *inaccessible* en dehors de la fonction. Elle appartient à l'espace de noms de la fonction.

On ne peut pas affecter directement une valeur aux variables globales depuis l'intérieur d'une fonction (à moins de les déclarer avec une instruction *global*), bien qu'on puisse y faire référence.

Voici un exemple de variable `x` non globale :

```
1 >>> def f() :
2 ...     x=12
3 ...
4 x = 1
5 >>> f()
6 >>> x
7 1
```

et un exemple de variable `x` globale :

```
1 >>> def f() :
2 ...     global x
3 ...     x=12
4 ...
5 >>> x = 0
6 >>> f()
```

```
7 >>> x
8 12
```

Une fonction définie dans un module *Python*, est accessible partout ailleurs (dans une session *Python* ou dans un autre module) à condition de l'importer à l'aide de l'instruction *import*.

Par exemple, dans un module *bidon* (fichier *bidon.py*) vous définissez la fonction :

```
def f(x) : print('x vaut :',x)
```

En ligne de commande vous pouvez l'utiliser ainsi :

```
1 >>> import bidon,math
2 >>> bidon.f(math.pi)
```

5.3. Modules et paquetages

5.3.1. Modules.

les instructions import, from. Un module est associé à un fichier avec l'extension *.py*. Le module porte le nom du fichier et peut contenir des définitions de fonction, de variables ou de classe, aussi bien que des instructions exécutables. Ces instructions sont destinées à initialiser le module et ne sont exécutées qu'une seule fois à la première importation du module.

Chaque module possède sa propre table de symboles privée.

On peut accéder aux variables globales d'un module avec la même notation que celle employée pour se référer à ses fonctions : *nommodule.nomvariable*.

Les noms du module importé sont placés dans la table globale de symboles du module importateur.

Pour importer et utiliser un module :

```
1 >>> import random, math
2 >>> random.randint(0,10)
3 >>> math.pi
```

On peut aussi importer un (ou des) nom(s) particulier(s) d'un module. Dans ce cas, ces éléments sont directement référencés par leur nom : *randint*, *choice* sans qu'il soit nécessaire de les préfixer par le nom du module comme ci-dessous :

```
1 >>> from random import (randint,choice)
2 >>> randint(0,10)
3 >>> choice(range(100))
```

On peut donner un « alias » au nom du module :

```
1 >>> import random as rand
2 >>> rand.randint(1, 10)
```

Il y a une variante pour importer tous les noms d'un module, excepté ceux qui commencent par un tiret-bas (déconseillé) :

```
1 >>> from random import *
2 >>> choice(500)
```

La fonction reload(). Dans une session *Python*, la première fois qu'un module est importé par l'instruction *import* module ou *from* module *import* x,y, le module est compilé (à moins qu'un fichier bytecode module.pyc dont la date de dernière modification est plus récente que celle du fichier module.py ne soit présent dans le même répertoire). Si une autre instruction *import* module est rencontrée au cours de la même session, *le module n'est pas recompilé*. Bien souvent, en cours de développement, le module module.py est modifié. Dans la session *Python*, le re-importer pour prendre en compte ces modifications est sans effet puisqu'il n'est pas recompilé. Il faut alors utiliser la fonction *reload*(module) ce qui a pour effet de compiler et re-importer module.

Ce faisant, les attributs importés individuellement par *from* module *import* x,y ne seront pas mis à jour. Puisque cette instruction n'est qu'une affectation des objets x et y de module aux noms x et y, dans l'espace de nommage *__main__*. Ces deux objets n'ont pas été modifiés par le *reload()* et sont toujours référencés sous le nom *__main__.x* et *__main__.y*. Il faut donc les ré-importer par *from* module *import* x,y. Les noms *__main__.x* et *__main__.y* seront alors mis en correspondance avec les *nouveaux* objets module.x et module.y. Cet exemple illustre ces différents points :

```
1 >>> import module
2 >>> dir(module)
3 ['__builtins__', ..., 'parfait', 'sqrt']
4 >>> module.parfait
5 <function parfait at 0xa439924>
6 >>> from module import parfait
7 >>> parfait
8 <function parfait at 0xa439924>
9 >>> reload(module)
10 <module 'module' from './module.pyc'>
11 >>> module.parfait
12 <function parfait at 0xa439bc4>
13 >>> parfait
14 <function parfait at 0xa439924>
15 >>> from module import parfait
16 >>> parfait
```

```
17 <function parfait at 0xa439bc4>
```

- 1 à 3 :** importation de module, qui contient une fonction parfait ;
- 4, 5 :** la fonction parfait est à l'adresse `..924` ;
- 6 :** la fonction `module.parfait` est affectée au nom `__main__.parfait` ;
- 7, 8 :** son adresse est donc celle de `module.parfait` ;
- 9, 10 :** compilation et importation de `module.py` ;
- 11, 12 :** la fonction `module.parfait` est un nouvel objet, avec une nouvelle adresse `..bc4` ;
- 13, 14 :** la fonction pointée par le nom `__main__.parfait` est toujours à la même (ancienne) adresse `..924` ;
- 15 à 17 :** le nom `__main__.parfait` est réaffecté au (nouvel) objet `module.parfait` à la nouvelle adresse, l'ancien parfait est détruit

5.3.2. Paquetages. Un *paquetage Python* est une arborescence de paquetages et de modules. Un paquetage possède un répertoire racine dans lequel se trouve un fichier `__init__.py`. Usuellement un paquetage possède une unité et une cohérence interne, mais rien n'empêche un paquetage *Python* de contenir côte à côte une base de données de recette de cuisines et une bibliothèque de fonctions mathématiques.

Un paquetage peut se charger comme un module par l'instruction

```
1 >>> import paquetage
```

Dans ce cas on accède aux modules du paquetage par la notation usuelle `paquetage.module`.

Le fichier `__init__.py` du paquetage, qui se trouve à la racine du répertoire-paquetage, est un fichier script *Python* ordinaire exécuté au chargement du paquetage, dans lequel sont faites diverses initialisations concernant le paquetage. On peut y trouver en particulier une liste

```
1 __all__=[mod1,mod2,...,var1,var2,...]
```

qui sera utilisée lors de l'instruction

```
1 >>> from paquetage import *
```

`mod1, mod2, ...` sont des noms de modules, `var1, var2, ...` sont des noms de variables.

Rien n'empêche cependant d'importer un module existant qui n'est pas listé dans `__all__` par une instruction explicite

```
1 >>> from paquetage import module_non_liste
```

5.3.3. Le chemin de recherche des paquetages et modules. Quand un module nommé `spam` est importé, l'interpréteur recherche un fichier nommé `spam.py`

- dans le répertoire courant, et puis
- dans la liste de répertoires indiquée par la variable d'environnement `PYTHONPATH`.
- dans un chemin d'accès par défaut, dépendant de l'installation ; sur *Unix*, c'est habituellement `/usr/local/lib/python`.

En fait, les modules sont recherchés dans la liste de répertoires donnée par la variable `sys.path`.

Ainsi, si votre module `karakoram` se trouve dans un répertoire exotique, disons `/home/puiseux/vacances/pakistan`, il vous suffit d'ajouter le nom de ce répertoire à la liste `sys.path` :

```
1 >>> import sys
2 >>> sys.path.append('/home/puiseux/vacances/pakistan')
3 >>> import karakoram
```

5.3.4. Fichiers bytecode. Pour accélérer le temps de lancement, si un fichier appelé `spam.pyc` existe dans le répertoire où se trouve `spam.py`, il est supposé contenir une version du module `spam` déjà compilée en *bytecode*.

Au lancement de `spam.py`

- recherche de `spam.pyc` (fichier déjà compilé en *bytecode*) dans le même répertoire que `spam.py`.
- si `spam.pyc` existe, et si sa date de création est postérieure à celle de `spam.py`, alors il est lancé.
- sinon, `spam.py` est compilé et `spam.pyc` est exécuté

Toutes les fois que `spam.py` est compilé avec succès, une tentative est faite pour écrire la version compilée sur `spam.pyc`.

Le contenu du fichier `spam.pyc` est *indépendant* de la plate-forme, ainsi un répertoire de module de *Python*. Il peut être partagé par des machines d'architectures différentes.

5.3.5. Modules standard. *Python* est livré avec une bibliothèque de modules standard, décrite dans un document séparé, *Python Library Reference*. Cette bibliothèque sera examinée plus en détail au chapitre 7. Quelques modules standard de base :

- `math` : fonctions et constantes mathématiques de base (`sin`, `cos`, `exp`, `pi`...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard...
- `os` : dialogue avec le système d'exploitation (e.g. permet de sortir de *Python*, lancer une commande en shell, puis de revenir à *Python*).
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder à l'heure de l'ordinateur et aux fonctions gérant le temps.
- `calendar` : fonctions de calendrier.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (*profilage* ou *profiling* en anglais).

- `urllib2` : permet de récupérer des données sur internet depuis *Python* .
- `Tkinter` : interface python avec *Tk* (permet de créer des objets graphiques ; nécessite d'installer *Tk*).
- `re` : gestion des expressions régulières.
- `pickle` : écriture et lecture de structures *Python* (comme les dictionnaires par exemple).
- `doctest` : un module pour le *développement dirigé par la documentation*, c'est à dire permettant d'écrire en même temps les tests *et* la documentation voir [2.12 page 45](#).

5.4. Décorateurs

Un décorateur *Python* est une fonction qui prend en paramètre une autre fonction, pour la modifier, lui ajouter des fonctionnalités, la substituer ou simplement exécuter un travail avant ou après l'avoir appelé.

Un décorateur nommé `monDecorateur()` est appelé en plaçant l'instruction `@monDecorateur` au dessus de la fonction ciblée.

Un exemple de décorateur pour rendre la comparaison de deux chaînes de caractères insensible à la casse :

```
1 def caseInsensitive(func) :
2     def minuscule(x) :
3         return func(*[a.lower() for a in x])
4
5 @caseInsensitive
6 def isEqual(x, y) :
7     return (x == y)
```

Comparons les chaînes de caractères `ChomoLunga` et `chomolunga` :

```
1 >>> isEqual("ChomoLunga", "chomolunga")
2 True
```

Voici un deuxième exemple de décorateur, totalement inutile, mais illustrant bien le fonctionnement :

```
1 def monDecorateur(f) :
2     def _monDecorateur() :
3         print("decorator stuff")
4         f()
5         print("other stuff")
6         return
7     return _monDecorateur
8
9 @monDecorateur
```

```

10 def cible() :
11     print ("fonction stuff")

```

et lorsque l'on appelle la fonction `cible()`, son comportement a été modifié par le décorateur :

```

1 >>> cible()
2 decorator stuff
3 fonction stuff
4 other stuff

```

Comme on le remarque sur cet exemple, dès qu'on applique un décorateur sur une fonction celui-ci prend le contrôle, et peut même complètement ignorer la fonction `cible()`.

En termes mathématiques, la fonction f décorée par la fonction g est la fonction $g \circ f$

5.5. Fonctions intégrées, le module `__builtin__`

Contient les fonctions *intégrées* (*builtins*) et les exceptions. Les fonctions *builtins* sont « codées en dur » dans l'interpréteur *Python* et aucun `import` n'est nécessaire pour y accéder. Quelques unes ont été présentées dans les sections précédentes .

Une liste complète (environ 68 fonctions) et documentée des fonctions intégrées se trouve sur <http://docs.python.org/library/functions.html>.

On présente ici les plus courantes.

5.5.1. Manipulation d'attributs. Un *attribut* d'un objet est une valeur associée à cet objet, référencée par son nom, via une expression « pointée ». Par exemple `math.pi` est l'attribut `pi` du module `math`.

Tous les objets *Python* ont des attributs.

Les attributs de classe seront décrits au chapitre 6 page 153.

Les fonctions suivantes permettent de manipuler ces attributs.

- `hasattr(object, attr)` retourne `True` ou `False` selon que l'objet *Python* `object` possède ou non un attribut nommé `attr`.
- `getattr(object, name[, default])` retourne l'attribut `name` de l'objet `object`. Si l'attribut n'existe pas, une exception `AttributeException` est levée. L'expression `getattr(X, 'a')` équivaut à `X.a`. Le paramètre `name` est une chaîne de caractères.
- `delattr(object, name)` supprime l'attribut `name` de l'objet `object`. Si l'attribut n'existe pas, une exception `AttributeException` est levée. L'expression `delattr(X, 'a')` équivaut à `del X.a`
- `setattr(object, name, value)` ajoute un attribut `name` de valeur `value` à l'objet `object`. L'expression `setattr(X, 'a', v)` équivaut à `X.a = v`.

Voici un exemple utilisant ces fonctions :

```

1 >>> class A : pass
2 ...
3 >>> a = A()
4 >>> setattr(a, 'x', 1)
5 >>> hasattr(a, 'x')
6 True
7 >>> delattr(a, 'y')
8 Traceback (most recent call last) :
9   [...]
10 AttributeError : A instance has no attribute 'y'
11 >>> delattr(a, 'x')
12 >>> dir(a)
13 ['__doc__', '__module__']

```

Il est possible d'ajouter des attributs à la plupart des objets *Python*. Par exemple :

```

1 >>> import math
2 >>> math.toto = 7
3 >>> hasattr(math, 'toto')
4 True
5 >>> math.toto
6 7

```

5.5.2. Conversion de type, construction. De nombreuses fonctions intégrées permettent la conversion d'un type à un autre. On peut distinguer les *constructeurs*, qui sont des fonctions de même nom que le type instance, et les fonctions de conversion proprement dit. Les principaux constructeurs ont déjà été évoqués au chapitre 4 page 79.

Les autres fonctions intégrées de conversion sont :

- `hex(i)` convertit un nombre entier en chaîne hexadécimale.
- `oct(x)` convertit un nombre entier en chaîne octale.
- `chr(i)` retourne le caractère dont le code *Unicode* est l'entier *i*. Par exemple, `chr(97)` retourne `'a'`. L'argument *i* doit être compris dans l'intervalle `[0, 1114111]`.
- `ord(c)` est l'inverse de la fonction `chr()`. Retourne le code *Unicode* (*Unicode code point* dans la terminologie *Unicode*) de l'argument, qui doit être un caractère *Unicode*.

Exemple :

```

1 >>> for i in range(34, 50) :
2     ... print(chr(i), end=' ')
3 " # $ % & ' ( ) * + , - . / 0 1
4

```

```

5 >>> for c in string.ascii_lowercase :
6 ...   print ('ord({0})={1}'.format(c,ord(c)),end=' ; ')
7
8 ord(a)=97 ; ord(b)=98 ; ord(c)=99 ; [...] ; ord(z)=122 ;

```

5.5.3. Logique.

- `all(iterable)` retourne True si tous les éléments de l'itérable sont vrais et False sinon.
- `any(iterable)` retourne True s'il existe un item vrai dans iterable, False sinon.
- `callable(object)` (à partir de *Python* 3.2) retourne True si `object` est callable (essentiellement une fonction, méthode ou classe, à partir de *Python* 3.2).

```

1 >>> all(range(12))
2 False
3 >>> all(range(1,12))
4 True
5 >>> any(range(1))
6 False
7 >>> any(range(2))
8 True

```

5.5.4. Programmation.

- `compile(source, filename, mode)` retourne la chaîne source compilée en objet code qui peut-être exécuté par `eval()` ou `exec()`. D'autres paramètres sont disponibles, voir la documentation *Python* .
 - ▷ `filename` est le nom du fichier où source a été lu. Si source n'est pas à lire sur un fichier, mettre `filename='<string>'`.
 - ▷ `mode` prendra une des valeurs `'exec'` si source est une suite d'instructions, ou bien `'eval'` s'il s'agit d'une simple expression ou encore `'single'` s'il s'agit d'une instruction interactive comme l'affichage d'une valeur.
- `eval(expression, globals=None, locals=None)` évalue une *expression*. Les arguments sont une chaîne de caractère (*expression*), et deux dictionnaires (`globals` et `locals`). L'argument *expression* est évalué comme une expression *Python* en utilisant les dictionnaires `globals` et `locals` comme espaces de nommage global et local. La valeur de retour est le résultat de l'évaluation. Exemple :

```

1 >>> x = 1
2 >>> print (eval('x+1'))
3 2

```

Cette fonction peut également être utilisée pour exécuter des codes arbitraire comme ceux créés par `compile()`. Dans ce cas, *expression* doit-être un objet code.

- `exec(object, globals=None, locals=None)`¹ exécute une *instruction* ou une liste d'instructions. Le paramètre `object` doit être une chaîne de caractères représentant la suite d'instructions, ou bien un objet code (comme produit par `compile()`). Les paramètres `globals` et `locals` ont le même rôle que pour la fonction `eval()`. La fonction `exec()` n'a pas de valeur de retour. Exemple, comparaison `exec()` versus `eval()` :

```

1  >>> exec('x = 5')
2  >>> exec('x = 5\n print(x)')
3  5
4  >>> eval('x = 5')
5  Traceback (most recent call last) :
6    [...]
7      x = 5
8      ^
9  SyntaxError : invalid syntax
10 >>> eval('5+2')
11 7

```

- `locals()` met à jour et retourne un dictionnaire représentant le table des symboles locale.
- `globals()` retourne un dictionnaire représentant la table des symboles courante, c'est à dire le dictionnaire du module courant. À l'intérieur d'une fonction ou d'une méthode, il s'agit du module définissant la fonction, et non pas le module appelant.

5.5.5. Itérables. Une des principales différence entre *Python* 2 et *Python* 3 tient au mode de gestion des itérables. Grosso-modo, lorsqu'en *Python* 2 une *séquence* (liste, tuple, ...) est retournée, en *Python* 3 c'est un *itérateur* qui est renvoyé. La différence en terme d'encombrement est évidente puisque dans le premier cas, une séquence entière est créée et stockée, tandis qu'en *Python* 3, les termes de l'itérable sont calculés et fournis au fur et à mesure de la demande, sans être stockés. Par exemple pour une liste et un itérateur de 10^7 entiers l'encombrement est donné par :

```

1  >>> import sys
2  >>> sys.getsizeof(range(10000000))
3  20
4  >>> sys.getsizeof(list(range(10000000)))
5  45000056

```

Par contre en terme de temps d'exécution, les deux approches (liste/itérateur) ne sont pas très différentes :

¹La fonction `exec()` remplace l'instruction `exec` et la fonction `execfile()` de *Python* 2. Cette dernière n'existe plus en *Python* 3

```

1 >>> import profile
2 >>> profile.run("for i in range(10000000) : i*i")
3         4 function calls in 2.636 CPU seconds
4         [...]
5 >>> profile.run("for i in list(range(10000000)) : i*i")
6         4 function calls in 2.648 CPU seconds
7         [...]

```

- `enumerate(iterable, start=0)` retourne un objet de type `enumerate`. L'argument `iterable` doit être une séquence, un itérateur, ou tout autre objet supportant l'itération. Cette fonction est utilisée essentiellement pour obtenir une série indexée : `(0, seq[0])`, `(1, seq[1])`, `(2, seq[2])`, etc.. Exemple :

```

1 >>> notes = ['mi', 'la', 'ré', 'sol', 'si', 'mi']
2 >>> for i, note in enumerate(notes) :
3 ...     print ('Corde numéro {} : {}'.format(i+1, note))
4 Corde numéro 1 : mi
5 Corde numéro 2 : la
6 Corde numéro 3 : ré
7 Corde numéro 4 : sol
8 Corde numéro 5 : si
9 Corde numéro 6 : mi

```

- `filter(function, iterable)` construit un *itérateur* sur les items de `iterable` pour lesquels la fonction `function` retourne `True`. L'argument `iterable` peut être une séquence, un conteneur supportant les itérations, ou un itérateur. Si `function` vaut `None`, elle est supposée égale à l'identité, tous les éléments faux de `iterable` sont supprimés.
- `map(function, iterable, ...)` retourne un *itérateur* qui applique `function` à tous les termes de `iterable`. Si plusieurs n itérables sont passés en arguments, la fonction doit prendre n arguments. L'itérateur stoppe lorsque l'itérable le plus court a été entièrement parcouru.²
- `max(iterable[, args...][, key])` avec le seul argument `iterable` (de type itérable), retourne le plus grand de ses items. Avec plusieurs arguments, retourne le plus grand des arguments. L'argument optionnel `key` est une fonction à un argument appliquée à chaque item avant le tri qui détermine le max. Autrement dit, `max()` calcule le maximum de la séquence `[key(i) for i in iterable]`, disons `key(it0)` et retourne `it0`.

²En *Python* 2, `map(function, iterable, ...)` applique `function` à tous les items de `iterable` et en retourne la liste. Si la liste d'arguments contient d'autres itérables, alors `function` les traite en parallèle. Si un itérable est plus court que les autres, il est complété par `None` autant de fois que nécessaire. Si `function` vaut `None`, elle est remplacée par l'identité. La fonction `map()` renvoie une liste de tuples, la longueur de chaque tuple est égale au nombre d'itérables passés en arguments.

- `min(iterable[, args...][, key])` avec le seul argument `iterable` (de type itérable), retourne le plus petit de ses items. Avec plusieurs arguments, retourne le plus petit des arguments. L'argument optionnel `key` fonctionne de manière analogue à l'argument `key` de la fonction `max()`
- `next(iterator[, default])` retourne l'item suivant de `iterator` par appel à sa méthode `next()`. L'argument `iterator` doit être de type itérateur. Si l'argument `default` est présent, il est retourné lorsque l'itérateur est épuisé. Sinon l'exception **StopIteration** est levée.
- `range(start=0, stop, step=1)` permet la création de progressions arithmétiques itérables. Voir 3.5 page 65
- `reversed(seq)` retourne un « itérateur inverse » (*reverse iterator*). L'argument `seq` doit être un objet possédant une méthode `__reversed__()` ou bien supportant le protocole des séquences (c'est à dire possédant une méthode `__len__()` et une méthode `__getitem__()` avec argument dans \mathbb{N}).
- `sorted(iterable[, key][, reverse])` retourne une nouvelle liste triée, à partir des items de l'itérable. Les deux arguments `key` et `reverse` sont optionnels. Si `key` est présent, ce doit être une fonction à un seul argument destinée à être appliquée à tous les éléments avant comparaison (typiquement, `key=str.lower` transformera toutes les chaînes en minuscule avant de les trier). Si `reverse` vaut `True`, le tri se fera en ordre inverse. La valeur par défaut de `reverse` est `False`.
- `zip(*iterables)` retourne un itérateur dont les éléments sont des tuples. Le *i*-ème tuple est constitué des *i*-èmes éléments de chacun des itérables passés en argument. Si les itérables n'ont pas la même longueur, l'itérateur retourné a la longueur du plus court des itérables.

5.5.6. Entrées-sorties.

- `format(value[, format_spec])`. Cette fonction a été décrite 4.5.6 page 96
- `input([prompt])` affiche `prompt` à l'écran et se met en attente d'une réponse de l'utilisateur. La réponse est retournée par la fonction sous forme de chaîne de caractères. La chaîne de caractères ne doit pas être mise entre guillemets. Pour obtenir autre chose qu'une chaîne de caractères, il faut le convertir explicitement.³

```

1  >>> nomfichier = input("nom de fichier ? ")
2  nom de fichier ? toto.txt
3  >>> print(nomfichier)
4  'toto.txt'
5  >>> age = int(input("Age du capitaine ? "))
6  Age du capitaine ? 40

```

³Python 2 propose deux fonctions de saisie :

- ▷ la fonction `raw_input([prompt])` qui se comporte comme la fonction `input()` de Python 3 en retournant une chaîne de caractère et
- ▷ la fonction `input()` est équivalente à `eval(raw_input())`. Elle attend une expression Python syntaxiquement correcte.

- `open(file, mode='r', encoding=None, buffering=-1)` Ouvre un fichier et retourne le flux correspondant. Si le fichier ne peut pas être ouvert, l'exception **`IOError`** est levée.
 - ▷ `file` est soit une chaîne de caractères donnant le chemin du fichier (chemin absolu ou relatif au répertoire courant), soit un descripteur de fichier⁴.
 - ▷ `mode` est une chaîne de caractères optionnelle spécifiant le mode d'ouverture du fichier, contenant un ou plusieurs caractères spécifiés dans le tableau suivant. Sa valeur par défaut est `'rt'`, le fichier est ouvert en lecture et en mode texte. En mode texte, si le paramètre `encoding` n'est pas spécifié, le mode d'encodage utilisé dépend de la plateforme.

Caractère	Signification
'r'	Lecture
'w'	Écriture
'a'	Ajout
'b'	Binaire
't'	Texte
'+'	Mise à jour (lecture et écriture)

Tab. 1. Modes d'ouverture d'un fichier

Un fichier ouvert en mode binaire (`mode='b'`) retourne son contenu sous forme d'octets, sans décodage. Un fichier ouvert en mode texte (`mode='t'`) retourne son contenu sous forme de chaîne de caractères (`str`) après un décodage utilisant le mode spécifié par l'argument `encoding`.

- ▷ `encoding` Les principaux encodage pour l'Europe de l'ouest sont `'ascii'`, `'cp1252'`, `'latin_1'`, `'utf_8'`, etc., avec des alias possible. Une liste complète des encodages disponible est donnée dans <http://docs.python.org/py3k/library/codecs.html#standard-encodings>.
- ▷ `buffering=-1` est un entier (optionnel), indiquant la stratégie de « buffering » (mise en mémoire tampon) à adopter. Utile pour les gros fichiers, la valeur 0 désactive le buffering, le fichier entier est lu en une seule passe (autorisé en mode binaire seulement). La valeur 1 demande un buffering (mode texte seulement) et un entier supérieur à 1 fixe la taille du buffer. Par défaut les fichiers binaires sont bufferisés, la taille du buffer dépend du système (typiquement 4096 ou 8192 octets). En mode texte, la taille du buffer est une ligne.

D'autres arguments sont disponibles :

- ▷ `errors=None` pour préciser la stratégie de gestion des erreurs,
 - ▷ `newline=None` pour la gestion des fins de lignes,
 - ▷ `closefd=True`
- consulter la documentation *Python* pour plus de précisions.

⁴Un descripteur de fichier est un entier. Si `f` est un fichier ouvert, la méthode `f.fileno()` retourne son descripteur.

5.5.7. Classes.

- `isinstance(object, classinfo)` retourne True si l'argument `object` est une instance de `classinfo` ou d'une classe fille. `classinfo` doit être une classe, un type ou un tuple de classes ou de types. Sinon, une exception `TypeError` est levée.
- `issubclass(class, classinfo)` retourne True si `class` est une classe héritant de `classinfo`. Une classe est considérée comme héritant d'elle-même. `classinfo` peut aussi être un tuple de classes. Sinon, une exception `TypeError` est levée.

5.5.8. Mathématiques.

- `divmod(a, b)` voir 2.6 page 35.
- `pow(x, y[, z])` retourne x^y , équivaut à `x**y`. Si `z` est présent, `x` et `y` doivent être entiers, et `y` doit être positif ou nul. Dans ce cas, retourne `x` à la puissance `y` modulo `z` calculé de manière plus efficace que `pow(x, y)%z`
- `round(x, n=0)` retourne `x` arrondi à `n` décimales. Return the floating point value `x` rounded to `n` digits after the decimal point.
- `abs(x)` valeur absolue ou module pour un complexe.

5.5.9. Divers.

- `id(object)` retourne « l'identité » d'un objet : un entier unique tout au long de la vie de l'objet.
- `hash(object)` retourne la valeur de hachage de l'objet (un entier) si elle existe.
- `dir(module=None)` renvoie une liste des noms définis par le module `module`.

```
1 >>> import fibo
2 >>> dir(fibo)
```

Sans arguments, énumère les noms définis par l'utilisateur :

```
1 >>> a = [1, 2, 3, 4, 5]
2 >>> import fibo, sys
3 >>> fib = fibo.fib
4 >>> dir()
```

Enumère tous les types de noms : les variables, les modules, les fonctions, etc... *sauf* les noms des fonctions et des variables intégrées.⁵

- `type(objet)` renvoie le type d'un objet
- `len(obj)` renvoie la taille d'un objet
- `quit()` pour quitter *Python*
- `assert(condition)` déclenche une *exception* si la condition est fausse.

⁵L'instruction `dir(__builtins__)` retourne une liste des noms intégrés (exceptions, types, fonctions, ...)

```
1 >>> import os
2 >>> filename='toto.txt'
3 >>> assert(os.path.isfile(filename))
```

donne le même résultat que

```
1 >>> if not os.path.isfile(filename) :
2 ...     raise AssertionError
```

Exercices

Exercice 46. Navigation dans les modules

- (1) Quel est le contenu du module `matplotlib` ?
- (2) Dans quel répertoire se trouve ce module ?
- (3) Trouvez un tutoriel `matplotlib` et testez les premières instructions.

Exercice 47. Fonctions

Que fait la séquence d'instructions suivante ?

```
1 >>> def echo(msg) :
2 ...     print(msg)
3 >>> x = echo
4 >>> x("ca marche")
```

Exercice 48. Arguments : modifier une liste

- (1) Ecrire une fonction `modif(L)` qui modifie la liste `L`, en changeant son premier élément en `'change'` et qui ne retourne rien.
- (2) Ecrire une fonction `fidom(L)` qui ne modifie pas la liste `L`, mais qui renvoie une copie de `L` dont le premier terme est modifié en `'change'`

Exercice 49. Suite de Fibonacci :

Écrire la fonction `fibonacci` dans un fichier `fibonacci.py` et testez les instructions

```
1 >>> from fibo import fib, fib2
2 >>> fib(500)
3 >>> import fibo
4 >>> fibo.fib(20)
```

Exercice 50. Richesse lexicale

Reprendre l'exercice 4.8 page 106, et créer les fonctions

- `wordList(text, n)` qui retourne une liste des mots du texte passé en argument, étant postulé qu'un mot comporte au minimum n lettres, $n = 3$ par défaut
- `frequence(words)` qui prend en argument une liste de mots, qui retourne un dictionnaire dont les clés sont les mots utilisés, et les valeurs sont les fréquences d'apparition des mots.
- `lexicalRichness(text)` qui calcule et retourne la richesse lexicale du texte passé en argument
- `wordCount(text, n)` qui compte le nombre de mots distincts du texte passé en argument, attendu qu'un mot doit comporter au moins n caractères, $n = 3$ par défaut.

Exercice 51. Passage d'arguments

Tester les instructions, interprétez les résultats :

```

1 >>> def modif(x) : x[0] += 10
2 ...
3 >>> a = [1,2,3]
4 >>> modif(a)
5 >>> print(a)
6 >>>
7 >>> def fidom(x) : x=0
8 ...
9 b=1
10 >>> fidom(b)
11 >>> print(b)

```

Exercice 52. Espace de nommage

Écrire un fichier `tests.py` contenant ces lignes :

```

1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 import os
4
5 print ("La variable __name__ vaut : ",__name__)
6
7 def inters(l1,l2) :
8     """Intersection de deux listes"""
9     return list(set(l1)&set(l2))
10
11 if __name__=="__main__" :
12     print (os.listdir('.'))
13     print (inters([1,2,3],[2,3,4]))

```

Dans une session **Python** , importez le module `tests`.

Dans un terminal, exécuter le script `tests.py`.

Interprétez le résultat.

Exercice 53. Règles de portée, comment planter **Python** .

- (1) Quelle est la nature de la variable `__builtins__` ?
- (2) Quelles sont les clés de `__builtins__` ?
- (3) Que contient `__builtins__['max']` ?
- (4) Quel est l'espace de noms de la fonction `max()` ?
- (5) Que se passe-t-il si l'on masque la fonction `max()`, par exemple avec l'instruction

```
>>> max=0.5 ?
```
- (6) Que se passe-t-il si l'on efface `__builtins__['max']` ?
- (7) Que se passe-t-il si l'on efface `__builtins__` ?
- (8) Que se passe-t-il si l'on masque le dictionnaire `__builtins__`, par exemple avec l'instruction

```
>>> __builtins__=12 ?
```
- (9) Qu'advient-il lorsque l'on écrit :

```
>>> def f(x) :global x ?
```

Exercice 54. Fonctions builtins

- (1) Utilisation élémentaire
 - Voici des notes d'étudiants `N=[5, 3, 2.5, 7, 0.5, 19]`. Calculez le *max*, le *min* et la moyenne de ces notes.
 - Depuis une session **Python** , lire le fichier de donnée `data.txt` et créer les quatres variables `a`, `b`, `c` et `d` en les initialisant avec les valeurs précisées dans ce fichier.
 - quelle est la différence entre les fonctions `exec()` et `eval()` ?
 - Créez une liste de `n` nombres entiers aléatoires (module `random`).
- (2)

Exercice 55. Récursivité : somme des chiffres d'un nombre entier.

Écrire une fonction `som(S)` qui a pour argument `S` une chaîne de caractères constituée de chiffres uniquement, et qui calcule récursivement la somme des tous ses chiffres jusqu'à obtenir un résultat à un seul chiffre, et le renvoie sous forme de caractère.

Exercice 56. Récursivité : les tours de Hanoï

- (1) Ecrire une fonction récursive `hanoi(n, depuis, vers, par)` qui implémente l'algorithme célèbre des tours de Hanoï. Voir http://fr.wikipedia.org/wiki/Tours_de_Hanoi
- (2) Ajouter un compteur `N` du nombre de déplacements, afficher le nombre de déplacements en fin d'exécution.

Exercice 57. `csv2vcf` : lire une feuille Excel au format csv

Un fichier de contacts, `contacts.csv`, a été exporté par un tableur ou un gestionnaire de contacts, au format csv sous la forme :

prénom, nom, email, tel, commentaire

- (1) dans un premier temps lire ce fichier en **Python** et en faire un dictionnaire dont les clés sont les couples (nom, prénom) et les valeurs les tuples (email, tel, commentaire). Attention aux virgules à l'intérieur de certains champs.
- (2) Sauvegarder les contacts au format vcard, dans un fichier `contacts.vcf`, une carte par contact au format suivant :

```
BEGIN :VCARD
VERSION :3.0
N :puiseux;pierre;;;
FN :pierre puiseux
EMAIL ;type=INTERNET ;type=HOME ;type=pref :pierre@puiseux.name
TEL ;TYPE=CELL :06-89-70-79-94
END :VCARD
```

Exercice 58. Fichier IGC : lecture d'une trace GPS.

Une trace GPS est une suite de points de $(t_n, P_n) \in \mathbb{R}^+ \times \mathbb{R}^3$ représentant la trajectoire effectuée par un parapente lors d'un vol. Le format IGC est assez simple, et décrit partiellement en (5.5.9).

Créer un nouveau dossier GPS à l'intérieur duquel, un module **Python** nommé `lecture` (c'est à dire un fichier `lecture.py`) proposera les fonctions suivantes :

- (1) une fonction `latitude(r)` aura comme argument un enregistrement et retournera la latitude en radians (pour faire ce calcul, on notera que 42 deg 58.954 mn font 42 + 58.954/60 degrés décimaux, que l'on transforme facilement en radians en multipliant par $\frac{\pi}{180}$),
- (2) une fonction `longitude(r)` aura un comportement analogue,
- (3) une fonction `altitude(r)` retournera l'altitude du barographe (en mètres),
- (4) une fonction `heure(r)` retournera l'heure sous la forme d'une liste d'entiers `[hh, mm, ss]`,
- (5) une fonction `lecture(nomfichier)` retournera un triplé (date, modèle, points) où date est l'enregistrement contenant la date, modèle est l'enregistrement contenant le modèle et points est la liste des enregistrements contenant des points GPS,
- (6) une fonction `distance(a, b, unit='km')` retournera la distance entre deux enregistrements-points. En m ou bien km, suivant la valeur du paramètre `unit`.
- (7) des instructions testant ces fonctionnalités.

Exercice 59. Fichier IGC (suite).

Dans un nouveau module **Python**, nommé `resume`, on programmera une fonction `resume()` fournissant un résumé du vol sous la forme :

- Modèle de la voile
- Date du vol
- Durée du vol
- Vitesse moyenne (réelle et projetée)
- Finesse moyenne⁶

⁶il s'agit du rapport distance horizontale/distance verticale

- Distance parcourue (distance réelle et projetée)
- Dénivelé

Pour pouvoir accéder aux fonctions du module `lecture`, il faudra bien sûr les importer dans le module `resume`.

Exercice 60. Fichier IGC, analyse détaillée (suite).

- (1) Une fonction `decompose(L)` prendra comme argument la liste `L` retournée par la fonction `lecture()` et retournera une liste d'enregistrements décomposé [heure, latitude, longitude, altitude]
- (2) Une fonction `vitesse(D)` prendra comme argument la liste `D` renvoyée par fonction `decompose(L)` et renverra une liste des vitesses.
- (3) Des fonctions `altitudes(D)`, `longitude(D)`, `latitudes(D)` renverront la liste des altitudes, longitudes et latitudes sur le même principe.
- (4) Une fonction `distances(D)` renverra la liste des distances parcourues depuis le départ.

Exercice 61. Décorateur pour vérification de type.

Prévoir le comportement des instructions suivantes :

```

1 def ArgEntier(f) :
2     def _ArgEntier(arg) :
3         if not isinstance(arg, int) :
4             raise TypeError("%s : %s n'est pas de type entier" \
5                             % (arg, type(arg)))
6         else : return f(arg)
7     return _ArgEntier
8
9 @Entier
10 def h(x) :
11     return x*x
12
13 print(h(1))
14 print(h(1.2))

```

Exercice 62. Décorateurs.

Soit la fonction

```

1 def f(x) : return x*x

```

- (1) Ajouter un décorateur à la fonction `f`, qui vérifie que l'argument `x` est de type `float` et lève une exception `TypeError` si ça n'est pas le cas.
- (2) Ajouter un décorateur à la fonction `f`, qui lève une exception adaptée si le nombre d'arguments est différent de 1

Exercice 63. Décorateur

Soit la fonction `def g(x) : print(x)`.

Ecrire un décorateur *Majuscule*, appliquer le à *g* de sorte que :

```
1 >>> g('hello')
```

produise le résultat

```
1 HELLO
```

Exercice 64. Profilage

Il s'agit d'évaluer les performances des list comprehension. Dans un script *Python* :

(1) créez une grande liste *L* d'entiers aléatoires (de taille $N = 1000$ par exemple) compris dans l'intervalle $[-100, 100]$. Créez également une fonction

```
1 def f(x) :
2     return sin(x/2.0)
```

- (2) Créez (et testez) une fonction `mapWithoutListComprehension()` qui répète 100 fois le calcul de la liste des $f(x)$ pour x dans L , en utilisant la fonction `map()`. Cette fonction n'effectuera aucune impression ni affectation.
- (3) Créez (et testez) une fonction `mapWithListComprehension()` qui effectue la même tâche mais avec la technique de list comprehension.
- (4) Utiliser la fonction `run()` du module `profile` pour comparer les temps d'exécution des deux fonctions.
- (5) Conclusion ?

Le format IGC (International Gliding Commission). Un fichier IGC se présente ainsi :

<http://web.univ-pau.fr/~puiseux/enseignement/python/2010-2011/Chap2-Prog/GPS/test.igc>

Dans ces enregistrements (1 ligne = 1 enregistrement), seule nous intéresse la série des enregistrements suivants :

- HFDTE260309 qui contient la date sous la forme jjmmaa
- HFGTYGLIDERTYPE :Kailash bivouac qui détermine le modèle d'aéronef
- B+38caractères contient des données enregistrées à intervalles réguliers, toutes les 1 ou 2 secondes. Par exemple, l'enregistrement B 084035 4258954N 00002136W A01231 01186 038 (auquel on a rajouté des espaces pour plus de lisibilité) se lit ainsi :
 - ▷ 084035 est l'heure, ici 08h 40mn 35sec
 - ▷ 4258954N est la latitude, ici 42 deg 58.954 mn, en degrés, minutes et fraction de minutes (mais pas en secondes), dans l'intervalle $[0, 90]$,

- ▷ 00002136W longitude (1 chiffre de plus que pour la latitude, car la longitude est dans l'intervalle $[0, 160]$),
- ▷ 01231 altitude barographe en mètres, elle est plus précise que l'altitude GPS,
- ▷ 01186 altitude GPS en mètres
- ▷ 038 vitesse calculée par la sonde si elle est présente

On considère deux points GPS de coordonnées (en radians) $M_1 = (x_1, y_1)$. La distance $d(M_1, M_2)$ est donnée, en mètres, par la formule

$$d(M_1, M_2) = r \times \arccos(\sin(x_1) * \sin(x_2) + \cos(x_1) * \cos(x_2) * \cos(y_2 - y_1))$$

où $r = 6378.7 \times 10^3$ (mètres) est la circonférence de la terre.

Solutions des exercices

Solution 46 Navigation dans les modules

(1) Contenu du module matplotlib

```
1 >>> import matplotlib
2 >>> dir(matplotlib)
```

(2) Dans quel répertoire se trouve ce module ?

```
1 >>> matplotlib.path
2 ['/usr/lib/pymodules/python2.6/matplotlib']
```

(3) http://matplotlib.sourceforge.net/users/pyplot_tutorial.html#pyplot-tutorial

Solution 5.5.9 Elle définit une fonction echo (en lignes 1 et 2) lui donne un alias, x en ligne 3 et l'exécute (ligne 4).

Solution 48 Modifier une liste

(1) Modifie la liste

```
1 >>> def modif(L) :
2 ...     L[0]='change'
```

(2) Ne modifie pas la liste

```
1 >>> def fidom(L) :
2 ...     return ['change'] + L[1 :]
```

(3) test des deux fonctions :


```
1 >>> l=[0,1]
2 >>> fidom(l)
3 ['change', 1]
4 >>> l
5 [0, 1]
6 >>> modif(l)
7 >>> l
8 ['change', 1]
```

Solution 49 Suite de Fibonacci**Solution 50** Richesse lexicale

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3
4 def wordCount(text, n=3) :
5     return len(set(wordList(text,n)))
6
7 def lexicalRichness(text, n=3) :
8     words = wordList(text, n)
9     lsw = float(len(set(words)))
10    lw = len(words)
11    return lsw/lw
12
13 def frequence(words) :
14     frequence={}
15     for word in words :
16         try : frequence[word]+=1
17         except KeyError : frequence[word]=1
18     return frequence
19
20 def wordList(text, n=3) :
21     text = unicode(text,"utf-8")
22     text = text.replace(u'\'',u' ')
23     words = [w.strip("'.,;!?()' :").lower() for w in text.split()
24               if len(w)>n]
25     return(words)
26
27 if __name__=="__main__" :
28     text=open('petittexte.txt').read()
29     print wordCount(text)
```

```

29     print wordCount(text, 4)
30     print lexicalRichness(text, 4)
31     mots = wordList(text, 4)
32     print sorted(mots)
33     print frequence(mots),

```

Solution 51 Passage d'arguments

`modif()` modifie son argument car c'est une liste (modifiable).

`fidom()` ne modifie pas son argument car c'est un entier (non modifiable).

Solution 52 Espace de nommage

Dans tout espace de nommage, une variable `__name__` est en permanence accessible, et contient le nom de l'espace de nommage.

Dans une session *Python*, l'espace de nommage est `__main__`.

Dans le module `tests` l'espace de nommage est `tests`.

Lors de l'importation du module `tests`, les instructions qui le composent sont exécutées. L'espace de nommage est alors celui du module : `__name__` vaut `tests`. L'expression booléenne `__name__=="__main__"` prend donc la valeur `False` les instructions du bloc ne sont pas exécutées.

Lors d'une exécution du script en ligne de commande Unix, l'espace de nommage est `__main__`. L'expression booléenne `__name__=="__main__"` : prend donc la valeur `True` et les instructions contenues dans le bloc sont exécutées.

Solution 53 Règles de portée

(1) `>>> type(__builtins__)` fournit la réponse : `__builtins__` est de type `dict` ;

(2) `>>> __builtins__.keys()` donne la réponse ;

(3) `__builtins__['max']` contient la fonction `max()` ;

(4) l'espace de noms de la fonction `max()` est fournit par l'instruction

```
>>> max.__module__;
```

(5) l'instruction

```
>>> max=0.5
```

masque la fonction `max()` qui n'est donc plus accessible ;

(6) si l'on efface `__builtins__['max']` par exemple avec

```
>>> __builtins__.pop('max')
```

... ça plante ;

(7) *Python* refuse d'effacer `__builtins__` sans donner aucun message ;

(8) si l'on masque `__builtins__`, avec l'instruction

```
>>> __builtins__=12
```

la fonction `max()` par exemple, n'est plus visible. Il suffit d'effacer la variable `builtins` qui masque la vraie `__builtins__` par l'instruction

```
>>> del __builtins__
```

(9) **SyntaxError** : name '**x**' **is** local **and global** (<input>, line 2)

Solution 54 Fonctions builtins

(1) Utilisation élémentaire

– Max, min et moyenne d'une liste N d'entiers

```
>>> max(N), min(N), sum(N)/len(N)
```

– Lecture de données

```
1 >>> for v in open('data.txt').readlines() :
2 ...     exec(v.strip('#\n ').replace(' ', '='))
```

(2) La fonction **eval**(expr) permet d'évaluer l'expression expr tandis que la fonction **exec**(inst) évalue l'instruction inst. Par exemple :

```
1 >>> eval('1+1')
2 2
3 >>> exec('a = 1 + 1')
4 >>> print(a)
5 2
```

Solution 55 Somme des chiffres d'un nombre entier

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 def som(N) :
4     """La somme des chiffres dans un nombre entier N, recursif"""
5     assert(type(N) is str)
6     if len(N)==1 :
7         return int(N)
8     else :
9         P=str(sum([int(i) for i in N]))
10        return som(P)
11
12 if __name__=="__main__" :
13     n='123'
14     print "som(%s)=%s"%(n, som(n))
```

Solution 56 Tours de Hanoï

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 """source : http://fr.wikipedia.org/wiki/Tours_de_Hanoi"""
4 N=0
```

```

5  def hanoi(n,de,vers,par) :
6      """
7      jeu de hanoi
8      il s'agit de deplacer des disques
9      de la tour A vers la tour B
10     """
11     global N
12
13     if n>0 :
14         N+=1
15         hanoi(n-1, de, par, vers)
16         print str(de),"-->",str(vers)
17         hanoi(n-1, par, vers, de)
18     return N
19
20 if __name__ == '__main__' :
21     print hanoi(5,'A','B','C'), ' déplacements'

```

Solution 57

Solution 58 Lecture d'une trace GPS sur un fichier IGC.

```

1  #!/usr/bin/python
2  #-*-coding : utf-8 -*-
3
4  import os, numpy, scipy
5  from math import pi, cos, sin, acos, asin, sqrt
6  from lecture import *
7  def duree(P, Q, unit='mn') :
8      """Duree entre deux enregistrements"""
9      h,m,s=(f-d for d,f in zip(heure(P),heure(Q)))
10     if unit.lower() in ('m','mn') :
11         return 60*h+m+s/60.0
12     elif unit.lower() in ('h') :
13         return h+m/60.0
14     elif unit.lower() in ('sec','s') :
15         return (60*h+m)*60+s
16     else :
17         return None
18
19 def dureeTotale(records,unit='mn') :
20     """Renvoit la duree totale d'un vol"""
21     return duree(records[0],records[-1],unit)

```

```

22
23 def vitesse(P,Q,unit='km/h') :
24     """Renvoit le couple (vitesse reelle, vitesse projetee) entre
        les deux enregistrements P et Q"""
25     if unit.lower() in ('km/h') :
26         d = duree(P,Q,'h')
27         dr,dp = distance(P,Q,'km')
28         return (dr/d,dp/d)
29     elif unit.lower() in 'm/s' :
30         d = duree(P,Q,'s')
31         dr,dp = distance(P,Q,'m')
32         return (dr/d,dp/d)
33     else :
34         raise NotImplementedError,'Unite de vitesse non prise en
            compte : %s'%unit
35
36 def vitesseMoyenne(records,unit='km/h') :
37     """Renvoit le couple (vitesse moyenne reelle, vitesse moyenne
        projetee)"""
38     return vitesse(records[0],records[-1],unit)
39
40 def finesse(P,Q) :
41     """Renvoit le la finesse entre deux enregistrements, c'est a
        dire le rapport distance horizontale/distance verticale"""
42     dp,dr=distance(P,Q,'m')
43     return dp/deniv(P,Q,'m')
44 def finesseMoyenne(records) :
45     return finesse(records[0],records[-1])
46
47 def deniv(P,Q,unit='m') :
48     """Retourne le denivele entre deux enregistrements. Unite=metre
        todo :proposer d'autre unites"""
49     return altitude(P)-altitude(Q)
50
51
52 def resume(trace) :
53     try : daterecord,modelerecord,pointsrecords = lecture(trace)
54     except IOError :
55         print 'fichier %s introuvable'%trace
56         exit()
57     print "Modele      :",modele(modelerecord)
58     print "Date        :",date(daterecord)
59     unite='h'

```

```

60     print 'Duree      : %.2f %s'%(dureeTotale(pointsrecords,unite),
        unite)
61     unite='km/h'
62     vp,vr=vitesseMoyenne(pointsrecords,unite)
63     print 'Vitesse reelle : %.2f %s, projetee : %.2f %s'%(vr,
        unite, vp,unite)
64     print 'Finesse      : %.1f'%(finesseMoyenne(pointsrecords))
65     dp,dr = distance(pointsrecords[0],pointsrecords[-1])
66     unit='km'
67     print 'Distance reelle : %.2f %s, projetee : %.2f %s'%(dr,
        unit,dp,unit)
68     unit='m'
69     print 'Denivelee    : %.2f %s'%(deniv(pointsrecords[0],
        pointsrecords[-1]),unit)
70
71 if __name__ == '__main__' :
72     resume('test.igc0')

```

On a rajouté la fonction `date(r)` pour extraction de la date, ainsi que la fonction `modele(r)` pour extraction du modèle d'aéronef

Solution 59 Résumé d'un fichier IGC (suite).

Voir (5.5.9)

Solution 60 Fichier IGC, analyse détaillée (suite).

voir (5.5.9)

Solution 61 Todo

Solution 62 Décorateurs

```

1  def unArgument(f) :
2      def _unArgument(*arg) :
3          #print '##', len(arg)
4          if len(arg) >1 :
5              print('Je ne prend que le premier argument : %s'%arg[0])
6              return f(arg[0])
7          return _unArgument
8
9  @unArgument
10 def g(x, y=0) : return x*x
11
12 if __name__ == "__main__" :
13     print(g(2))

```

```
14 print(g(1,3))
```

Solution 63 Décorateur

```
1 def Majuscule(f) :  
2     def _Majuscule(arg) :  
3         return f(arg.upper())  
4     return _Majuscule  
5  
6 @Majuscule  
7 def g(x) : print(x)
```

Solution 64 Profilage

```
1 #!/usr/bin/python  
2 # -*- coding : utf-8 -*-  
3 import random  
4 import profile as prof  
5 from math import sin  
6 N=1000  
7 biglist = [random.randint(-100,100) for i in range(N  
    )]
```

```
1  
2 def map_without_list_comprehension() :  
3     for i in range(100) :
```

```
1  
2 def map_with_list_comprehension() :  
3     for i in range(100) :
```

```
1  
2 print ("==== map_without_list_comprehension ====")  
3 prof.run('map_without_list_comprehension()')  
4 print ("==== map_with_list_comprehension ====")
```

On obtient les résultats suivants :

```

===== map_without_list_comprehension =====
200105 function calls in 1.040 CPU seconds
...
===== map_with_list_comprehension =====
200005 function calls in 1.176 CPU seconds
...

```

En effectuant plusieurs fois le test, on calcule la moyenne des temps d'exécution :

	<code>map</code>	<i>List comprehension</i>
	1.024	1.080
	1.208	1.116
	1.072	1.164
	1.040	1.176
	1.104	1.136
	1.124	1.212
<i>moyenne</i>	<i>1.095</i>	<i>1.147</i>

Tab. 2. Comparaison des temps d'exécution `map()`/*list comprehension*

L'utilisation des *list comprehension* produit un code environ 5% plus lent que la fonction `map()`.

CHAPITRE 6

Programmation orientée objet

Les principales caractéristiques des classes *Python* sont :

- l'héritage permet la multiplicité des classes de base ;
- une classe dérivée peut *surcharger* n'importe quelle méthode de sa ou ses classes de base ;
- une méthode peut appeler une méthode de sa classe de base avec le même nom ;
- les objets peuvent contenir un nombre arbitraire d'*attributs privés* ;
- tous les membres d'une classe (dont les données membres) sont *publics* ;
- toutes les méthodes sont *virtuelles*. Il n'y a pas de constructeurs ou de destructeurs particuliers ;
- toute méthode est déclarée avec un premier argument *explicite* (`self` le plus souvent) qui représente l'objet, qui est fourni implicitement à l'appel ;
- les classes sont elles-mêmes des *objets*, mais dans un sens plus large : en *Python*, tous les types de données sont des objets ;
- les *types intégrés* peuvent être utilisés comme classes de base pour des extensions par l'utilisateur ;
- la plupart des *opérateurs intégrés* qui ont une syntaxe particulière (opérateurs arithmétiques, indiciage, etc.) peuvent être *surchargés*.

6.1. Une première approche des classes

Si les types de données de base de *Python* (listes, entiers, réels, dictionnaires...) ne sont pas adaptés à vos besoins, créez une nouvelle classe.

Une classe créée par un programmeur *est* un nouveau type. Supposant que les complexes n'existent pas en *Python* natif, si un mathématicien a besoin du type complexe, il créera une classe `Complexe` dont le comportement sera à sa convenance.

À l'utilisation, il se pourrait que la déclaration `>>> z=Complexe(1, 2.3)` convienne à notre mathématicien, qui voudra peut-être également écrire `>>> w = Complexe(1.0, 3.1)` puis `Z = w * z`.

La création de la classe complexe devra répondre à ce *cahier des charges* (simplifié).

Comme toute entité *Python*, une classe est un *object* du langage *Python* que nous examinons un peu plus loin (voir paragraphe 6.1.3 page 155).

L'objet *class* `Complexe` (de type *class*) et l'objet `z` (de type `Complexe`) sont deux choses différentes. L'objet `z` est une *instance* de la classe `Complexe`.

6.1.1. Syntaxe. La forme la plus simple de définition de classe ressemble à ceci :

```
1 class NomClasse :
2     instructions
```

Les définitions de classe, comme les définitions de fonction doivent être exécutées pour entrer en effet.¹

Dans la pratique, les instructions seront souvent des définitions de *méthodes*, mais d'autres instructions sont acceptées.

A l'entrée d'une définition de classe, un nouvel espace de noms est créé et utilisé comme portée locale.

Lorsque la définition de la classe est achevée de façon normale, un objet de type **class** est créé.

6.1.2. Une classe Complexe. Voici le code d'une classe Complexe sommaire et incomplète :

```
1 class Complexe(object) :
2     """Une classe complexe sommaire"""
3     def __init__(self, re=0.0, im=0.0) :
4         self._x, self._y = re, im
5
6     def mod(self) :
7         return sqrt(self._x*self._x+self._y*self._y)
```

- **class** est un mot clé, Complexe le nom de la classe. Cette ligne alerte l'interpréteur : je veux créer un objet *Python* de type **class**, une nouvelle classe (et non pas un objet de type Complexe qui, lui, sera créé plus loin). La classe Complexe hérite de la classe **object**, c'est à dire que tous les attributs de la classe **object** peuvent être utilisés par notre nouvelle classe. (voir la section 6.2 page 157)
- il est recommandé d'écrire une *docstring* destinée expliquer le fonctionnement, avant toute autre instruction.
- la première méthode est un constructeur (méthode spéciale) : `__init__` ses trois arguments sont
 - (1) `self` : désigne l'objet non encore instancié, de type Complexe, qui appellera cette méthode. Autrement dit, `self` est un Complexe, qui n'existe pas encore.²
 - (2) `re` et `im` sont les parties réelle et imaginaire, (par défaut 0.0 et 0.0)
 Il y a création de deux attributs `_x` et `_y`, dans `self`. Ces attributs prendront corps en même temps que `self`, lorsque un Complexe sera instancié.

¹Il est possible de placer une définition de classe dans une branche d'une instruction **if**, ou à l'intérieur d'une fonction.

² De manière analogue, au moment de la définition d'une fonction (par exemple **def** `f(x) : return x*x`) `x` ne désigne « aucune » variable, mais prendra corps seulement lorsque la fonction sera appelée par exemple `a=3 ; f(a)`.

- `mod()` est une méthode qui calcule le module du complexe `self`. L'argument `self` prendra la valeur `z` lorsque sera exécutée l'instruction `z.mod()`.

Les classes introduisent trois nouveaux types d'objets : les *objets classe*, les *objets instances* et les *objets méthodes*, que nous examinons maintenant.

6.1.3. Objets Classes. En *Python*, une classe est de type *type*, tout comme les types de base *int*, *str*, etc..³

Les objets classe admettent deux sortes d'opérations :

- Les références aux attributs : utilisent la syntaxe standard utilisée pour toutes les références d'attribut en *Python* : `objet.nom`. Par exemple pour la classe `Complexe` définie plus loin, les références suivantes sont valides :
 - ▷ `Complexe.mod`, `Complexe.__init__` renvoient des objets de type *function*
 - ▷ `Complexe._x`, `Complexe._y` renvoient des objets de type *float*
 - ▷ `Complexe.__doc__` renvoie la docstring, de type *str*
 - ▷ Attention, il est possible d'affecter une valeur aux attributs de classe, par simple affectation. Par exemple l'instruction suivante est valide

```
1 >>> def f() : print 'hello'
2 >>> Complexe.mod = f
```

- L'instantiation de classe, consiste à créer un objet de la classe. Utilisez la notation d'appel de fonction. Faites comme si l'objet classe était une fonction qui renvoie une instance nouvelle de la classe. Par exemple l'instruction

```
1 >>> z = Complexe(1,2)
```

crée une nouvelle instance de la classe et affecte cet objet à la variable locale `z`. Cette instruction déclenche un appel à `Complexe.__init__(1,2)`. Les attributs `z._x`, `z._y` deviennent effectif, à l'intérieur de `z`. On dit que `z` est instancié.

6.1.4. Objets Instances.

Les seules opérations acceptées par des objets instance sont des références à leurs attributs. Il y a deux sortes de noms d'attributs valides.

- les données (*data attributes*) : `_x` et `_y` dans la classe `Complexe`
- les méthodes (*methods*) : une méthode est une fonction liée à une instance. Par exemple `z.__init__()`, `z.mod()` dans la classe `Complexe`

³En *Python* 2, les objets classes sont de type `classobj`, contrairement aux types de base qui sont de type *type*. En ce sens, une classe n'est pas un équivalent strict d'un type de base.

6.1.5. Objets Méthodes. *Python* fait une distinction entre la *méthode* `z.mod()` associée à une instance, et la *fonction* `Complexe.mod()` associée à la classe⁴. Ce ne sont pas les même objets :

```

1 >>> class A :
2 ...     def f(self) :pass
3 ...
4 >>> A.f
5 <function f at 0xb7107b2c>
6 >>> a=A()
7 >>> a.f
8 <bound method A.f of <__main__.A object at 0xb71141ac>>

```

On adoptera la même terminologie que *Python* en parlant de fonction (ou fonction-méthode) pour une classe et de méthode pour une instance.

Usuellement, une méthode est appelée de façon directe, par exemple : `>>> z.mod()`. Dans l'exemple de la classe `Complexe`, cette instruction renverrait le module du complexe `z`. Dans cette instruction, la méthode a été appelée sans argument, alors que sa définition en spécifiait un. *Python* aurait dû lever une exception ?

Non, et c'est la particularité des méthodes : l'objet est passé comme premier argument à la fonction.

<code>z.mod()</code> est strictement équivalent à <code>Complexe.mod(z)</code>
--

La différence entre une méthode et une fonction tient à ce que la méthode est appelée sans son premier argument, qui est implicitement l'instance de l'appelant.

Autrement dit : dans la méthode `Complexe.mod(self)`, l'argument `self` vaut `z`. Ainsi il est parfaitement légitime, bien que moins élégant, d'appeler `Complexe.mod(z)` au lieu de `z.mod()`.

6.1.6. Quelques remarques.

- Le premier argument d'une méthode est, par convention, appelé `self`, ça n'est qu'une convention qu'il est fortement conseillé de suivre, à défaut de quoi le programme peut devenir rapidement illisible.
- Contrairement au `C++` où le `this` est facultatif, en *Python*, il n'y a pas de raccourci pour faire référence aux attributs d'une classe à partir d'une méthode. Ainsi l'exemple suivant est valide :

⁴En *Python* 2, `Complexe.mod()` était une méthode, mais une *méthode non liée* (*unbound method*)

```

1 class Complexe :
2     def mod(self) :
3         return sqrt(self._x*self._x+self._y*self._y)

```

Tandis que l'exemple suivant lèvera l'exception **NameError** indiquant que `_x` n'est pas défini :

```

1 class Complexe :
2     def mod(self) :
3         return sqrt(_x*_x+_y*_y)

```

- La définition des méthodes n'est pas nécessairement comprise dans la définition de la classe. L'exemple suivant est valide :

```

1 def ext(self,x) : return 2*x
2 class C :
3     f=ext
4     #etc...

```

6.2. Héritage simple, multiple

L'instruction :

```

1 class Complexe(object) :

```

signifie que la classe `Complexe` hérite de la classe `object` (qui est une classe *Python* de base).

Donc `Complexe` est un `object` de *Python*.

Grâce à cet héritage, tous les attributs et les méthodes de la classe `object` peuvent être utilisés et/ou surchargés par les instances de la classe `Complexe`.

Une classe *Python* doit toujours hériter d'une autre classe. À minima, si l'on crée une classe ex-abrupto, on la fera hériter de la classe de base `object`. Même si on omet cet héritage, *Python* le fera de lui même. Par exemple :

```

1 >>> class A :pass
2 ...
3 >>> type(A)
4 <class 'type'>
5 >>> issubclass(A, object)
6 True

```

Une classe utilisateur peut hériter indifféremment d'une classe de base *Python* ou bien d'une autre classe utilisateur.

6.2.1. Héritage ou pas ? C'est une question d'importance fondamentale pour le développement de classes réutilisables. Une mauvaise réponse à cette question débouche inmanquablement sur des problèmes inextricables. Des programmeurs chevronnés se sont laissés prendre au piège.

On utilise un héritage lorsque l'héritier *est* un parent, comme dans une filiation de famille ordinaire. Le fils Dupond *est* un Dupond.

La seule bonne question à se poser pour décider si B doit hériter de A est :

B *is* A ou bien B *has* A ?

- Si la réponse est *B is A*, alors B doit *hériter* de A,
- Si la réponse est *B has A*, alors B doit avoir un *attribut* A.

Par exemple :

- un cercle n'est pas un Complexe, ni même un centre. Un cercle *possède* un centre (donc un Complexe), une classe Cercle n'héritera pas de Complexe, mais possèdera un attribut Complexe, qu'on appellera probablement `_centre`.
- Par contre un carré *est* un polygône. Un triangle également. Donc les classes Carre et Triangle hériteront de la classe Polygone.

Nous développons cet exemple dans les paragraphes suivants.

6.2.2. Exemple de la classe Polygone TODO : commenter. En supposant que l'on dispose d'une classe Polygone, comme celle-ci :

```

1  class Polygone(object) :
2      def __init__(self, liste_de_points) :
3          self.points = liste_de_points
4
5      def isClosed(self) :
6          return self.points[0] == self.points[-1]
7
8      def close(self) :
9          if not self.isClosed() :
10             self.points.append(self.points[0])
11
12     def longueur(self) :
13         return sum([abs(z1-z0) for (z0,z1) in
14                     zip(self.points[ :-1],self.points[1 :])])

```

on pourra définir une classe Triangle héritant de la classe Polygone comme ceci :

```

1  from polygone import Polygone
2
3  class Triangle(Polygone) :
4
5      def __init__(self, liste) :
```

et l'utiliser ainsi :

```

1      self.close()
2
3      def __str__(self) :
4          return 'Triangle : '+Polygone.__str__(self)
```

etc...

6.2.3. Surcharge. Dans la terminologie C++, toutes les méthodes en *Python* sont virtuelles, ce qui signifie d'une part que l'on peut les surcharger. D'autre part, lorsqu'une méthode est surchargée, l'interpréteur *Python* décide dynamiquement -à l'exécution donc- laquelle des deux méthodes il doit utiliser (la méthode surchargée ou bien la méthode parent), suivant le type de l'appelant :

```

1  >>> class A :
2  ...   def f(self) : return 'Ah, bonjour'
3  >>> class B(A) : pass
4  >>> class C(A) :
5  ...   def f(self) : return "C'est bien"
6  >>> a, b, c = A(), B(), C()
7  >>> a.f()
8  'Ah, bonjour'
9  >>> b.f()
10 'Ah, bonjour'
11 >>> c.f()
12 "C'est bien"
```

Dans cet exemple,

- l'invocation de la méthode `a.f()` déclenche un appel à la fonction `A.f()` de la classe `A`,
- l'instruction `b.f()` appelle également la fonction `A.f()` de la classe `A`. Puisque la classe `B` n'a pas de fonction `f()`, l'interpréteur parcourt les ancêtres de `B` pour trouver une fonction nommée `f()`, qu'il trouve dans la classe `A`.
- à l'invocation de `b.f()`, la fonction `B.f()` est appelée. Cette fonction *masque* la fonction `A.f()`. On dit que la fonction `B.f()` *surcharge* la fonction `A.f()`.

De même, dans l'exemple 6.2.2 page 158, la méthode spéciale `Triangle.__str__()` surcharge celle de `Polygone.__str__()`.

Une question récurrente survient lorsque la méthode de l'héritier doit seulement *compléter* la méthode du parent plutôt que la redéfinir en totalité. Dans ce cas, on voudrait appeler la méthode parent, puis effectuer le traitement spécifique à la méthode enfant. À la lumière de ce qu'on a vu au paragraphe 6.1.5 page 156, il suffit d'appeler la méthode du parent comme une fonction ordinaire. Voici un exemple :

```
1 >>> class A :
2 ...     def f(self) : return 'Hello'
3 >>> class B(A) :
4 ...     def f(self) :
5 ...         return A.f(self)+' , world !'
6 >>> b = B()
7 >>> b.f()
8 >>> 'Hello, world !'
```

La méthode `b.f()` appelle tout d'abord explicitement la fonction `A.f()`, puis effectue son propre traitement. L'instance `b` ne peut accéder à la `A.f()` qu'en la qualifiant complètement. Le masquage de la fonction `A.f()` par la méthode `b.f()` n'est pas total, elle reste accessible comme fonction de la classe `A`.

6.2.4. Héritage multiple. *Python* supporte aussi une forme limitée d'héritage multiple. Une définition de classe avec plusieurs classes de base s'écrit ainsi :

```
1 class NomClasseDerivee(Base1, Base2, Base3) :
2     instructions ...
```

La seule règle permettant d'expliquer la sémantique de l'héritage multiple est la règle de résolution utilisée pour les références aux attributs. *La résolution se fait en profondeur d'abord*, de gauche à droite. Donc, si un attribut n'est pas trouvé dans `NomClasseDerivee`, il est cherché dans `Base1`, puis (récursivement) dans les classes de base de `Base1`, et seulement s'il n'y est pas trouvé, il est recherché dans `Base2`, et ainsi de suite.

L'utilisation banalisée de l'héritage multiple est un cauchemar de maintenance.

6.3. La classe object

```
1 >>> A.mro()
2 [<class '__main__.A'>, <class 'object'>]
3 >>> class A(object) :pass
4 ...
5 >>> A.mro()
```



```
6 [<class '__main__.A'>, <class 'object'>]
```

Dans cet exemple, on utilise la méthode `mro()` (initiales de Méthode Resolution Order), héritée du type `object`. Cette méthode indique l'ordre de parcours des classe parentes lors de la recherche d'un attribut d'une classe donnée.

6.4. Méthodes spéciales

Les méthodes spéciales commencent et finissent par `__` (deux underscores '`_`'). Elles sont héritées de la classe `object`.

La liste de ces méthodes spéciales se trouve dans la documentation *Python* de la classe `object` :

<http://docs.python.org/reference/datamodel.html>

Par exemple, si l'on veut pouvoir écrire `z = z1+z2` pour `z1` et `z2` de type Complexe, il suffit de surcharger la méthode `Complexe.__add__(self, z)` dans la classe `Complexe` en sachant que

`z = z1 + z2`
est strictement équivalent à
`z = z1.__add__(z2)`

On écrira donc dans la classe `Complexe` :

```
1 def __add__(self, z) :  
2     self._x += z._x  
3     self._y += z._y
```

Le tableau suivant énumère quelques méthodes spéciales que l'on peut surcharger dans une classe `C` donnée (à condition que celle-ci hérite de la classe `object`). La liste complète se trouve sur

<http://docs.python.org/reference/datamodel.html#specialnames>.

`x` et `y` sont deux instances de classe `C`

Méthode à surcharger	Exemple d'utilisation	Équivalence
<code>C.__add__(self, y)</code>	<code>x + y</code>	<code>x.__add__(y)</code>
<code>C.__sub__(self, y)</code>	<code>x - y</code>	<code>x.__sub__(y)</code>
<code>C.__mul__(self, y)</code>	<code>x * y</code>	<code>x.__mul__(y)</code>
<code>C.__and__(self, y)</code>	<code>x and y</code>	<code>x.__and__(y)</code>
<code>C.__or__(self, C)</code>	<code>x or y</code>	<code>x.__or__(y)</code>
<code>C.__len__(self)</code>	<code>len(x)</code>	<code>x.__len__()</code>
<code>C.__getitem__(self, i)</code>	<code>x[i]</code>	<code>x.__getitem__(i)</code>
<code>C.__setitem__(self, i, val)</code>	<code>x[i]=3</code>	<code>x.__setitem__(i, 3)</code>
<code>C.__call__(self[, args...])</code>	<code>x(args)</code>	<code>x.__call__(args)</code>
<code>C.__str__(self)</code>	<code>str(z)</code>	<code>x.__str__()</code>
<code>C.__repr__(self)</code>	<code>repr(x)</code>	<code>x.__repr__()</code>

Quelques remarques :

- `__repr__()` devrait retourner une représentation de son argument qui soit lisible par l'interpréteur. Autrement dit, `eval(repr(z))` devrait retourner une copie de `z`.
- L'instruction `str(z)` est en réalité un appel à `z.__str__()`
- `__str__()` devrait retourner une représentation de son argument qui soit confortablement lisible pour l'œil humain.
- La commande `print` ne sait afficher que des chaînes de caractères. L'instruction `print z` convertit tout d'abord `z` en string par un appel à `z.__str__()`, puis affiche la chaîne de caractères.

6.5. Variables privées

Il y a un support limité pour des identificateurs privés dans une classe. Tout identificateur de la forme `__spam` (au moins deux tirets-bas au début, au plus un tiret-bas à la fin) est maintenant textuellement remplacé par `_nomclasse_spam`, où `nomclasse` est le nom de classe courant, duquel les tirets-bas de début ont été enlevés.

Pour limiter ou interdire l'accès à un attribut d'une classe, on utilisera avec profit la notion de *property*, présentée au paragraphe 6.6.

Les décorateurs (5.4 page 129) fournissent un moyen efficace.

6.6. Les *properties*

Une *property* est un attribut de classe, gérée par un *getter* et un *setter*⁵.

Une *property* est déclarée par la fonction `property()`

⁵Un *setter* permet d'affecter (*set*) une valeur à un attribut, un *getter* permet de récupérer (*get*) la valeur de cet attribut. La traduction française, « accesseur en lecture » et « accesseur en écriture », en étant peu commode nous adoptons les mots anglais *getter* et *setter*.

C'est un moyen élégant pour autoriser l'utilisateur d'une classe à accéder -en lecture comme en écriture- à un attribut de la classe, avec une syntaxe allégée, tout en conservant le contrôle sur cet accès.

Supposons que l'on ait besoin de garder le contrôle sur le type d'une affectation : on veut être assuré que l'attribut `altitude` de la classe `Montagne` soit de type `float`. La solution usuelle (en C++ par exemple) est d'écrire une méthode de type *setter*, nommé `Montagne.setAltitude()` qui s'acquittera de cette vérification et renvoie un message en cas d'erreur. Supposant que `colline` est une instance de `Montagne`, on peut alors écrire :

```
1 >>> colline.setAltitude("hyper haut")
```

La syntaxe est lourde. On aimerait pouvoir écrire de manière plus naturelle :

```
1 >>> colline.altitude = "hyper haut"
```

tout en étant certain que la réponse soit identique, par exemple

```
1 Attention, altitude doit être un nombre réel.
```

La fonction `property()` autorise ce comportement.

Cet exemple basique illustre le fonctionnement de la fonction `property()` :

```
1 class Montagne(object) :  
2     def _getAltitude(self) :  
3         return self._alt  
4     def _setAltitude(self, alt) :  
5         if type(alt) in (float,int) :  
6             self._alt = float(alt)  
7         else :  
8             raise TypeError('%s' doit etre numerique'%alt)  
9     altitude = property(_getAltitude, _setAltitude)
```

Une instance `>>> s=Montagne()` est créée, puis l'utilisateur précise l'altitude avec une syntaxe simplifiée :

```
1 >>> s.altitude = 4807 !
```

L'interpréteur *Python* identifie `altitude` comme une *property* qui doit prendre la valeur 4807. Il appelle alors le setter de `altitude`, c'est à dire `_setAltitude()`.

Pour l'utilisateur, la notion de *property* est transparente. Elle permet de définir, comme en *C++* des *getters* et des *setters*, avec une vérification de type si nécessaire, mais avec une syntaxe légère pour l'utilisateur.

Au niveau de la programmation, cela demande au programmeur un effort initial.

La signature de *property* est :

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- `fget` est la méthode *getter* de l'attribut
- `fset` le *setter*
- `fdel` est le *destructeur* de l'attribut
- `doc` est une chaîne de caractères contenant la *documentation* de l'attribut

Une déclaration plus complète de la *property* `Montagne.altitude` pourrait être

```
1 altitude = property(fget=_getAltitude, fset=_setAltitude, doc="l'
    altitude en mètres")
```

La documentation de `Montagne.altitude` est comme de coutume dans sa docstring `Montagne.altitude.__doc__`, accessible par

```
1 >>> help(Montagne.altitude)
```

6.7. Manipulation dynamique des attributs

Python est un langage *reflectif*, qui supporte la *métaprogrammation*. Il est possible d'écrire un programme *Python* qui génère puis exécute un programme *Python*. Une illustration de cette technique est donnée par le package *PyPy* qui Les fonctions builtins `setattr()`, `getattr()`, `hasattr()` sont conçues pour créer, acquérir et consulter des attributs dans une classe. Voici un exemple simple d'utilisation :

```
1 >>> class A(object) :
2 ...     pass
3 ...
4 >>> a = A()
5 >>> setattr(a, 'un_attribut', 3.14)
6 >>> hasattr(a, 'un_attribut')
7 True
8 >>> a.un_attribut
9 3.1400000000000001
10 >>> delattr(a, 'un_attribut')
```

```
11 >>> hasattr(a, 'un_attribut')
12 False
```

- `setattr(objet, nom, valeur)` permet d'ajouter l'attribut `nom` à `objet`, avec la valeur `valeur`
- `delattr(objet, nom)` pour supprimer l'attribut `nom` à `objet`
- `hasattr(objet, nom)` renvoie `True` ou `False` suivant que `objet` possède ou non un attribut `nom`.

6.8. Slots

`__slots__` par défaut, chaque instance d'une classe possède son propre dictionnaire pour y stocker ses attributs. Lorsque le nombre d'instances est important, cette particularité est très consommatrice en mémoire vive. Ce comportement des classes *Python* peut être modifié par la déclaration de `__slots__`. Dans ce cas, lorsqu'une séquence d'instances de la classe est créée, seul l'espace mémoire nécessaire aux attributs *déclarés* est utilisé, et le dictionnaire n'est pas créé.

`__slots__` est une variable de classe à laquelle est affectée une chaîne, un itérable ou une séquence de chaînes, contenant les noms des attributs déclarés.

Exemple :

```
1 >>> class A(object) :
2 ...     __slots__="x"
3 >>> a = A()
4 >>> class B(object) :
5 ...     pass
6 >>> b = B()
7 >>> import sys
8 >>> sys.getsizeof(a)
9 56
10 >>> sys.getsizeof(b)
11 64
```

Les objets de classe A sont effectivement plus petits que les objets de type B :

```
1 >>> import sys
2 >>> sys.getsizeof(a)
3 56
4 >>> sys.getsizeof(b)
5 64
```

Comme il n'y a pas de dictionnaire attaché à une instance de la classe, il est impossible de lui ajouter un attribut :

```

1 >>> a.__dict__
2 Traceback (most recent call last) :[...]
3 AttributeError : 'A' object has no attribute '__dict__'
4 >>> b.__dict__
5 {}
6 >>> a.y = 7 # ou setattr(a, 'y', 7)
7 Traceback (most recent call last) :[...]
8 AttributeError : 'A' object has no attribute 'y'
9 >>> b.y = 7
10 >>> import sys

```

L'attribut de classe `__slots__` doit déclarer *tous* les attributs de la classe.

```

1 >>> class C(object) :
2     __slots__ = "x"
3     def __init__(self) :
4         self.x = 0
5         self.y = 1
6 >>> c = C()
7 Traceback (most recent call last) :[...]
8     self.y = 1
9 AttributeError : 'C' object has no attribute 'y'

```

Exercices

Exercice 65. Richesse lexicale d'un texte

À partir des résultats de l'exercice 5.5.9 page 138, créer une classe *Texte* capable de lire un fichier ASCII, compter le nombre de mots qui le compose, créer un dictionnaire des mots, avec leur fréquence, calculer la richesse lexicale du texte.

Voici un cahier des charges :

```

1 >>> RL = Texte('petittexte.txt')
2 >>> RL.wordCount()
3 22
4 >>> RL.wordsDict() #dictionnaire fréquences
5 {u'etage' : 2, u'dessus' : 1, ... u'actuellement' : 1}
6 >>> RL.lexicalRichness()
7 0.759...

```

Exercice 66. Polygones

Créer une classe *Polygone* représentant les polygones du plan, utilisant le type **complex** de **Python**, et dont le comportement répondra au cahier des charges minimum suivant :

```

1 >>> p = Polygone([0,1, 1+1j, 1j])
2 >>> print (p.point(1),p.point(2))
3 (1, (1+1j))
4 >>> print (p.isClosed())
5 False
6 >>> print (p.longueur())
7 3.0
8 >>> p.close()
9 >>> print (p.isClosed())
10 True
11 >>> print (p.longueur())
12 4.0
13 >>> print (p)
14 <__main__.Polygone object at ...>

```

Exercice 67. Classe, réutilisabilité

- (1) Dans la classe *Complexe*, écrire une méthode *arg()* qui retourne l'argument du complexe. Testez.
- (2) Écrire une version fonction de la méthode précédente qui se comportera de manière naturelle
 `>>> print arg(z).`
- (3) Créer une méthode de *Complexe*. *dist(...)* qui utilisée ainsi `>>> z.dist(z1)`, renvoie la distance entre les Complexes *z* et *z1* considérés comme des points du plan.
- (4) Créer une version fonction de la méthode précédente qui s'utilise ainsi : `>>> dist(z0, z1)`
- (5) Dans la classe *Complexe*, écrire une méthode *__repr__()* qui renvoie le complexe sous forme d'une chaîne de caractères agréable à lire.
- (6) Testez cette méthode par l'instruction `>>> print(z.__repr__())`, comparer avec `>>> print(z).`

Exercice 68. Héritage Triangle

Écrire une classe *Triangle* qui hérite la classe *Polygone* définie à l'exercice 66 et tester ses différents composants. Écrire en particulier une méthode *aire()* qui calcule l'aire du triangle.

Exercice 69. Héritage d'un type de base

Revisiter la classe *Polygone* de l'exercice 66.

Faites la hériter du type **list**

Profitez de cet héritage pour simplifier l'implémentation de la classe *Polygone*.

Vérifiez que cette nouvelle classe répond au cahier des charges de l'exercice 66

Ajouter ensuite les fonctionnalités répondant au cahier des charges :

```

1 >>> p=Polygone([0,1,1+1j])
2 >>> p.append(1j)
3 >>> p.close()
4 >>> p
5 [0j, (1+0j), (1+1j), 1j, 0j]
6 >>> p.aire()
7 1.0

```

Soit un polygône (fermé), de sommets complexes $(z_j)_{0 \leq j \leq n}$ avec $z_0 = z_n$. Alors l'aire du polygône est donné par⁶

$$A = \frac{1}{2} \sum_{1 \leq j \leq n} \Im(\bar{z}_j \cdot z_{j+1})$$

où $\Im(z)$ et \bar{z} désignent la partie imaginaire et le conjugué du complexe z .

Écrire et tester une méthode `Polygone.aire()`

Exercice 70. Héritage.

Ecrire une classe `Tableau` qui hérite de la classe `list`,

- dont la fonction `__init__(self, l)` permet de vérifier que l'argument `l` est une liste dont tous les éléments sont d'un type numérique : `int`, `long`, `float` ou `complex`
- qui permet d'écrire les instructions :

```

1 >>> a=Tableau([1,2,3])
2 >>> a.norm()
3 >>> a.norm(1)
4 >>> len(a)
5 >>> b=Tableau([2,3,4])
6 >>> a+b
7 >>> a*b #produit scalaire
8 >>> a(3) = 12
9 >>> a[3] = 12

```

Exercice 71. Méthodes spéciales

Dans la classe `Complexe`,

- (1) implémenter une méthode `__add__()` et une méthode `__mul__()`. Testez ensuite l'instruction
`>>> print z0+z1, z1*z0;`
- (2) implémentez correctement la méthode `__repr__` dans la classe `Complexe`, puis testez les instructions

⁶voir http://fr.wikipedia.org/wiki/Aire_et_centre_de_masse_d'un_polygone


```

1 >>> z = Complexe(1,2)
2 >>> z == eval(repr(z))

```

Exercice 72. Méthodes spéciales

En ligne de commande **Python**, déclarez une classe A : `>>> class A : pass`. Puis créez une instance a de la classe A

Créer une fonction p qui se contente d'imprimer "hello" et utiliser la fonction `setattr()` pour attacher p comme méthode de la classe A sous le nom de `__str__()` (fonction).

Testez l'instruction `>>> print a` et interprétez le résultat

Exercice 73. Property

Dans la classe Complexe, créer une property x, pour accéder à la partie réelle (lecture et écriture) avec la syntaxe suivante :

```

1 >>> z = Complexe(1,2)
2 >>> z.x
3 2
4 >>> z.x = 3.14

```

Modifier le setter de la property x pour qu'il effectue une vérification sur le type et lève une exception lorsque l'on tente d'affecter à la partie réelle un type non attendu, comme ceci :

```

1 >>> z0 = Complexe(1,2)
2 >>> z0.x = "12.0"
3 Traceback (most recent call last) :
4   File "./prop.py", line 40, in <module>
5     z0.x = "12"
6   File "./prop.py", line 29, in _setx
7     raise TypeError, "Complexe.x : %s non supporte"%(type(value))
8 TypeError : Complexe.x : <type 'str'> non supporte

```

Exercice 74. slots

Soient A et B les classes définies ainsi :

```

1 class A(object) :
2     __slots__="x"
3     def __init__(self) :
4         x = 0
5
6 class B(object) :
7     def __init__(self) :

```

```
8 x = 0
```

Écrire un script comprenant deux courtes fonctions qui testent la création de N objets de classe A , puis de classe B , utiliser la fonction `cProfile.run()` pour tester les temps d'exécution.

Solutions des exercices

Solution 65 Richesse lexicale d'un texte

```
1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 class Texte(object) :
4
5     def __init__(self, nom_fichier, word_size=4) :
6         self.size = word_size
7         try :
8             self.text = open(nom_fichier).read()
9         except IOError :
10             print ("Impossible de lire le fichier %s"%nom_fichier)
11         self.setWordList()
12
13     def setWordList(self) :
14         #self.text = unicode(self.text, "utf-8")
15         self.text = self.text.replace('\n', ' ')
16         self.words = [w.strip("'.,;!?()'" :").lower()
17                       for w in self.text.split() if len(w)>self.size]
18
19     def wordCount(self) :
20         return len(set(self.words))
21
22     def lexicalRichness(self) :
23         return float(len(set(self.words)))/len(self.words)
24
25     def wordsDict(self) :
26         self.frequency={}
27         for word in self.words :
28             try : self.frequency[word]+=1
29             except KeyError : self.frequency[word]=1
30         return self.frequency
31
32 if __name__=="__main__" :
33     RL = Texte('petittexte.txt')
34     print(RL.wordCount())
```

```

35     print(RL.wordsDict())
36     print(RL.lexicalRichness())

```

Solution 66 Polygônes

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3
4  class Polygone(object) :
5      def __init__(self, liste_de_points) :
6          self.points = liste_de_points
7
8      def isClosed(self) :
9          return self.points[0] == self.points[-1]
10
11     def close(self) :
12         if not self.isClosed() :
13             self.points.append(self.points[0])
14
15     def longueur(self) :
16         return sum([abs(z1-z0) for (z0,z1) in
17                     zip(self.points[ :-1],self.points[1 :])])
18     def point(self,i) : return self.points[i]
19
20 if __name__ == "__main__" :
21     p = Polygone([0,1, 1+1j, 1j])
22     print (p.point(1),p.point(2))
23     print (p.isClosed())
24     print (p.longueur())
25     p.close()
26     print (p.isClosed())
27     print (p.longueur())
28     print (p)

```

Solution 67 Classe, réutilisabilité

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3
4  from math import sqrt, atan2
5
6  class Complexe() :
7      """Une classe complexe sommaire"""

```

```

8     def __init__(self, re=0.0, im=0.0) :
9         self._x,self._y = re,im
10
11     def abs(self) :
12         return sqrt(self._x*self._x+self._y*self._y)
13
14     def arg(self) :
15         return atan2(self._x,self._y)
16
17     def __repr__(self) :
18         return "%f+%fI"%(self._x,self._y)
19
20     def dist(self,z) :
21         return Complexe(self._x-z._x,self._y-z._y).abs()
22
23     def arg(z) :
24         return z.arg()
25
26     def dist(z1,z2) :
27         return z1.dist(z2)
28
29     if __name__ == "__main__" :
30         z0 = Complexe(1,2)
31         z1 = Complexe()
32         print (z0.arg(),"==", arg(z0))
33         print (dist(z0,z1),"==",z0.dist(z1))
34         print (z0,"==", z0.__repr__())

```

Solution 68 Héritage Triangle

TODO

Solution 69 Héritage d'un type de base

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3
4  class Polygone(list) :
5      def __init__(self, liste_de_points) :
6          list.__init__(self,[complex(z) for z in liste_de_points])
7
8      def isClosed(self) :
9          return self[0] == self[-1]
10

```

```

11     def close(self) :
12         if not self.isClosed() :
13             self.append(self[0])
14
15     def longueur(self) :
16         return sum([abs(z1-z0) for (z0,z1) in zip(self[:-1],self[1:])]
17
18     def aire(self) :
19         return 0.5*sum([(z0.conjugate()*z1).imag
20             for (z0,z1) in zip(self[:-1],self[1:])]
21
22 if __name__ == "__main__" :
23     p=Polygone([0,1,1+1j])
24     p.append(1j)
25     p.close()
26     print (p, p.aire())

```

Solution 70 Todo

Solution 71 Méthodes spéciales

Voici une classe Complexe un peu plus complète :

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3
4  from math import sqrt, atan2
5  class Complexe(object) :
6      """Une classe complexe sommaire"""
7      def _getx ( self ) : return self._x
8      def _setx(self, a) :
9          try : self._x = float(a)
10         except ValueError : print ("float(%s) : impossible"%a)
11     x = property(_getx, _setx,doc="partie réelle")
12     def _gety ( self ) : return self._y
13     def _sety(self, a) :
14         try : self._y = float(a)
15         except ValueError : print ("float(%s) : impossible"%a)
16     y = property(_gety, _sety,doc="partie imaginaire")
17
18     def __init__(self, re=0.0, im=0.0) :
19         self.x, self.y = re, im
20
21     def __abs__(self) :

```

```

22     return sqrt(self.x*self.x+self.y*self.y)
23
24     def arg(self) : return atan2(self.x,self.y)
25
26     def __str__(self) : return '(%s + %si)'%(self.x,self.y)
27
28     def __repr__(self) : return "Complexe(%f,%f)"%(self.x,self.y)
29
30     def __add__(toto,z) :#toto+z
31         return Complexe(toto.x+z.x,toto.y+z.y)
32
33     def __mul__(self, z) :
34         return Complexe (self.x*z.x-self.y*z.y,self.x*z.y+self.y*z.x
35                             )
36
37     def conj(self) : return Complexe(self.x, -self.y)
38
39     def inv(self) :
40         a = self.x*self.x+self.y*self.y
41         return Complexe(self.x/a, -self.y/a)
42
43     def __getitem__(self,i) :
44         if i == 1 : return self.x
45         if i == 2 : return self.y
46         raise IndexError( "Complexe[%d] inaccessible"%i)
47
48     def __setitem__(self,i,val) :
49         if i == 1 :
50             self.x = val
51             return
52         elif i == 2 :
53             self.y = val
54             return
55         raise IndexError( "Complexe[%d] inaccessible"%i)
56
57     def inv(z) : return z.inv()
58     def conj(z) : return z.conj()
59
60     if __name__ == "__main__" :
61         z0=Complexe(1,2)
62         #z1=Complexe(2,3)
63         print (z0.x,z0._y)
64         z0.x = "0.5"

```

```

63     print (z0.x)
64     z0.x = "aaa"
65     print (z0.x)
66     print ("z0._x est toujours accessible : ",z0._x)
67     print (z0)
68     #print (z0.__repr__())
69     #print (z0.arg())
70     print (abs(z0))
71     #print (z1+z0)
72     print (z0[1],z0[2])
73     # print (z0[3])
74     z0[1]=1953
75     print (z0)
76     print (inv(z0), z0.inv())
77     print (conj(z0), z0.conj())

```

Solution 72 Méthodes spéciales

```

1  >>> class A : pass
2  ...
3  >>> a=A()
4  >>> def p(x) : return "hello"
5  ...
6  >>> setattr(A, '__str__',p)
7  >>> print (a)
8  hello

```

L'instruction `>>> setattr(A, '__str__', p)` attache la méthode spéciale `__str__()` à la classe A. Or, lorsque l'instruction `>>> print (a)` est exécutée, c'est cette méthode spéciale (méthode de classe) qui est automatiquement appelée. TODO

Solution 73 Property

Todo

Solution 74

```

1  def tests(classe, n=10000000) :
2      for i in range(n) : classe()
3
4  import cProfile
5  print("##### SLOTS #####")
6  cProfile.run('tests(A)')
7  print("##### NO SLOTS #####")
8  cProfile.run('tests(B)')

```


CHAPITRE 7

La bibliothèque standard

La bibliothèque standard de **Python** est extrêmement fournie. Elle propose, répartis en une quarantaine de thèmes, environ 300 modules très fiables.

De par leur compacité, et leur appartenance à la bibliothèque standard, ces modules sont utilisés, testés et débogués par la communauté **Python** entière, ce qui garantit leur fiabilité, à la différence de certains gros packages spécialisés comme *Apache*, *Django*, *PySide*, etc. dont le débogage est plus ardu du fait de leur taille et de leur relative confidentialité.

Parmi les thèmes abordés dans la bibliothèque standard, citons :

- les fonctions, les types, les exceptions et les constantes builtins ;
- les types de données élaborés (calendriers, dates, tableaux, conteneurs etc.) ;
- les chaînes de caractère (unicode, expressions régulières, etc.) ;
- les modules mathématiques et numériques (fonctions mathématiques, fractions, nombres aléatoires, etc.) ;
- l'accès aux fichiers et répertoires (manipulation de chemins, parcours d'arborescence, fichiers temporaires, etc.) ;
- la compression de données (*tar*, *bzip2*, *zip*, etc.) ;
- la persistance de données (bases de données, sauvegarde, etc.) ;
- les *markup languages* (*XML*, *HTML*, etc.) ;
- les systèmes d'exploitation (accès et modification des données des OS, etc.) ;
- les systèmes d'exploitation spécifiques (*Unix*, *MacOS*, *MS Windows*, etc.) ;
- internet (email, mime, *http*, *ftp*, *telnet* etc.) ;
- le multimedia (*wave*, *oss*, *aif*, etc.) ;
- l'interface graphique *Tkinter*
- les outils pour le développement (langages, débogage, profilage, tests unitaires, etc.) ;
- d'autres encore ...

Avant de développer votre propre code pour traiter d'un problème, consultez la documentation, vous y trouverez probablement un ou plusieurs modules pour vous faciliter la vie.

On décrit ici les modules les plus utiles pour la programmation et le calcul scientifique.

7.1. Le module *os*

Contient de nombreuses facilités pour interagir avec les commandes du système d'exploitation.

N'utilisez pas `from os import *` mais préférez lui `import os`. Sinon la fonction `os.open()` masquera la fonction intégrée `open()`. Toutes deux ont un fonctionnement différent.

7.1.1. Le module `os.path`. Traite de tout ce qui concerne les chemins (*path*). Il est vivement conseillé d'utiliser ce module pour composer ou décomposer des chemins. L'utilisation du module `os.path` assure la portabilité de vos programmes *Python* d'un environnement à l'autre (*MacOs*, *Windows*, *Linux*, etc.).

À contrario, si vous assemblez un chemin comme `/home/puiseux/devel/main.cxx`, soyez assurés que le programme sera inexploitable sous *Windows*.

Vous devriez plutôt écrire `os.path.join('home', 'puiseux', 'devel', 'main.cxx')`.

Les principales fonctions et attributs du module `os.path` sont les suivantes :

- `curdir` est le nom du répertoire courant,
- `sep` est le caractère séparateur de chemin. En *Unix*, c'est le `'/'` (*slash*) et en *MS Windows* le `'\'` (*antislash*).
- `abspath(chemin)` retourne le *path* absolu du paramètre chemin :

```
1 >>> abspath('toto.cxx')
2 /home/devel/toto.cxx'
```

- `basename(chemin)` retourne le nom de fichier du paramètre chemin :

```
1 >>> basename('/home/devel/toto.cxx')
2 'toto.cxx'
```

- `dirname(chemin)` retourne le nom du répertoire contenant chemin :

```
1 >>> dirname('/home/devel/toto')
2 '/home/devel'
3 >>> dirname('/home/devel/toto/')
4 '/home/devel/toto/'
```

- `expanduser(chemin)` expansion du `~`, le répertoire utilisateur (le contenu de la variable d'environnement `$HOME` en *Unix*),
- `getatime(fichier)` retourne la date de création d'un fichier,
- `getsize(fichier)` retourne la taille d'un fichier,
- `exists(chemin)` retourne `True` ou `False` suivant que chemin existe (répertoire, fichier ou lien),
- `isabs(chemin)` retourne `True` ou `False` suivant que chemin est ou n'est pas un chemin absolu,
- `isdir(chemin)` retourne `True` ou `False` suivant que chemin un répertoire,
- `isfile(chemin)` retourne `True` ou `False` suivant que chemin un fichier,
- `islink(chemin)` retourne `True` ou `False` suivant que chemin un lien,
- `ismount(chemin)` retourne `True` ou `False` suivant que chemin un point de montage,

- `join(liste)` pour reconstituer un chemin à partir de ses composants contenus dans `liste` :

```
1 >>> os.path.join('devel', 'toto.cxx')
2 'devel/toto.cxx'
```

- `splitext(chemin)` renvoie chemin séparé en (un tuple à) deux éléments, dont le second élément est l'extension, y compris le point :

```
1 >>> splitext('formations/python.pdf')
2 ('formations/python', '.pdf')
```

- `split(chemin)` renvoie un tuple « (head,tail) » dont le second élément « tail » est ce qui suit le dernier slash (ou *antislash* sur MS Windows) ; « tail » peut être vide :

```
1 >>> split('formations/python.pdf')
2 ('formations', 'python.pdf')
```

7.1.2. Autres services du module os. Le module `os` fournit de nombreuses autres fonctions. Les plus utiles sont listées ci-après :

- `system(cmd)` exécute de la commande shell précisée dans la chaîne de caractères `cmd` :

```
1 >>> os.system('mkdir toto')
```

- `walk()` permet de parcourir une arborescence :

```
1 >>> for root,dir,files in os.walk('.'):
2     ...     print ("Fichiers du répertoire %s => %s"%(root,files))
3     ...
```

- divers utilitaires :

```
1 >>> os.getcwd()
2 >>> os.chdir('~/.devel/python')
```

7.2. Le module sys

Contient des attributs caractéristiques de la plateforme, ainsi que quelques fonctions de bas niveau.

- `sys.path` est la liste des *path* dans lesquels seront recherchés les modules lors d'un `import` ;
- `sys.modules` est la liste des modules disponible sur votre configuration ;
- `sys.platform` est votre plateforme (linux, ...);
- `sys.argv` est la liste des arguments actuels de la ligne de commande ;

- `sys.float_info` est une liste contenant des informations sur la représentation des nombres réels ;
- `sys.stdin`, `sys.stdout` et `sys.stderr` sont des fichiers représentant les entrées et sorties standard et la sortie erreur :

```
1 >>> sys.stdout.write('Hello\n')
```

Ils peuvent être redirigés vers un fichier texte :

```
1 >>> out = sys.stdout #sauvegarde
2 >>> sys.stdout=open('testttt','a')
3 >>> print('hello')
4 >>> a=1
5 >>> a
6 >>> sys.stdout.close()
7 >>> a
8 Traceback (most recent call last) :
9 File "<input>", line 1, in <module>
10 ValueError : I/O operation on closed file
11 >>> sys.stdout=out
12 >>> a
13 1
14 >>>
```

La commande `print` autorise une syntaxe plus élégante pour rediriger la sortie standard sur un fichier texte :

```
1 >>> f = open('toto.txt','w')
2 >>> print >> f, 'hello'
```

- `sys.exit()` permet de terminer inconditionnellement un script *Python*.

7.3. Dates et heures TODO

Le module `datetime` fournit des classes pour manipuler les dates et les heures aussi bien de manière simple que de manière complexe. Bien que l'arithmétique des dates et des heures est supportée, le centre d'attention de l'implémentation a été porté sur l'efficacité de l'extraction des membres en vue de la mise en forme et du traitement de l'affichage. Ce module supporte aussi les objets liés aux fuseaux horaires.

```
1 >>> from datetime import date
2 >>> now = date.today()
3 >>> now
4 >>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B."
    )
```

```
5 >>> birthday = date(1964, 7, 31)
6 >>> age = now - birthday
7 >>> age.days
```

7.4. Fichiers

7.4.1. Le module `glob`. fournit des fonctions pour construire des listes de fichiers à partir de recherches avec jockers dans des répertoires :

- dans le répertoire courant avec `glob()`,
- dans un autre répertoire avec `glob1()` ;

```
1 >>> import glob
2 >>> glob.glob('*.py')
3 ['primes.py', 'random.py', 'quote.py']
4 >>> glob.glob1('.', '*.py')
5 ['primes.py', 'random.py', 'quote.py']
```

7.4.2. Le sous module `glob.fnmatch`. permet d'effectuer des tâches analogues :

- `glob.fnmatch.filter(names, pat)` pour filtrer dans la liste `names` les noms des fichiers correspondant à l'expression régulière `pat` :

```
1 >>> glob.fnmatch.filter(os.listdir('.'), '*.py')
```

- `glob.fnmatch.fnmatch(names, pat)` est analogue à `filter`, mais le *pattern* est en style *Unix*, la casse n'est pas prise en compte ;
- `glob.fnmatch.fnmatchcase` : idem, la casse est prise en compte ;
- `glob.fnmatch.translate(pat)` transforme un *pattern Unix* en expression régulière.

7.4.3. Le module `shutil`. fournit une interface de haut niveau pour les tâches courantes de gestion de fichiers et répertoires :

- `copy(src, dest)` copie les données et les modes (rwx) du fichier `src` vers le fichier `dest` ;
- `copyfile(src, dest)` copie les données seulement du fichier `src` vers le fichier `dest` ;
- `copytree(src, dest)` recopie récursive du répertoire `src` vers le répertoire `dest` ;
- `move(src, dest)` déplace `src` vers `dest`, comme la commande *Unix* `mv`
- `rmtree(dir)` suppression récursive de `dir`.

```
1 >>> import shutil
2 >>> shutil.copyfile('data.db', 'archive.db')
3 >>> shutil.move('/build/executables', 'installdir')
```

7.5. Compression de données **TODO**

Les formats communs pour archiver et compresser des données sont supportés par des modules, dont : `zlib`, `gzip`, `bz2`, `zipfile` et `tarfile`.

```
1 >>> import zlib
2 >>> s = u"Sans la liberte de blamer, il n'est point d'eloge
    flatteur"
3 >>> len(s)
4 >>> t = zlib.compress(s)
5 >>> len(t)
6 >>> zlib.decompress(t)
7 >>> zlib.crc32(s)
```

7.6. Arguments de la ligne de commande

Les scripts utilitaires requièrent souvent le traitement des arguments de la ligne de commande. Ces arguments sont stockés sous forme de liste dans l'attribut `sys.argv` :

```
1 >>> import sys
2 >>> print sys.argv
3 ['demo.py', 'one', 'two', 'three']
```

Le module `getopt` traite `sys.argv` en utilisant les mêmes conventions que la fonction *Unix* `getopt()`. En voici un exemple avec des options courtes :

```
1 >>> import getopt
2 >>> args = '-a -b -ctoto -d bar a1 a2'.split()
3 >>> args
4 ['-a', '-b', '-ctoto', '-d', 'bar', 'a1', 'a2']
5 >>> optlist, args = getopt.getopt(args, 'abc :d :')
6 >>> optlist
7 [('-a', ''), ('-b', ''), ('-c', 'toto'), ('-d', 'bar')]
8 >>> args
9 ['a1', 'a2']
```

et avec les options longues :

```
1 >>> s = '--condition=foo --test --output-file abc.def -x a1 a2'
2 >>> args = s.split()
3 >>> args
4 ['--condition=foo', '--testing', '--output-file', 'abc.def', '-x',
    'a1', 'a2']
```

```
5 >>> optlist, args = getopt.getopt(args, 'x', [  
6 ... 'condition=', 'output-file=', 'testing'])  
7 >>> optlist  
8 [('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc  
9 .def'), ('-x', '')]  
10 >>> args  
['a1', 'a2']
```

Un traitement de la ligne de commande plus puissant est fourni par le module `argparse`¹

Pour l'utiliser on commence par importer et instancier `argparse.ArgumentParser` :

```
1 from argparse import ArgumentParser  
2 parser = ArgumentParser(description='Ajouter des entiers.')
```

puis décrire les options attendues :

```
1 parser = argparse.ArgumentParser(description='Process some  
2 integers.')
```

```
2 parser.add_argument('entiers', metavar='N', type=int, nargs='+',  
3 help='opérer sur deux entiers')
```

7.7. Mesure des performances

7.7.1. Le module `time`. Permet de mesurer le temps écoulé et le temps *CPU* (le temps passé par un processus en mémoire centrale) :

- `time()` mesure le temps ordinaire, celui de l'horloge
- `clock()` donne le temps *CPU* consommé par le processus *Python* actuel depuis le début de son exécution.

```
1 >>> import time  
2 >>> e0 = time.time() # elapsed time since the epoch  
3 >>> c0 = time.clock() # total CPU time spent in the script  
4 >>> elapsed_time = time.time() - e0  
5 >>> cpu_time = time.clock() - c0
```

¹La documentation *Python* signale que le module `optparse` n'est plus maintenu et sera remplacé par `argparse`, décrit ici.

7.7.2. Le module `timeit`. Pour comparer certaines instructions, ou différentes options de programmation sur des portions de code de taille réduite, on peut utiliser le module `timeit`.

Ce module contient principalement la classe `Timer` dont le constructeur est

```
Timer(stmt='pass', setup='pass').
```

- L'argument `stmt` est la séquence d'instructions à chronométrer. Les instructions doivent être séparées par des `;`
- `setup` est une instruction d'initialisation, qui sera exécutée une seule fois avant `stmt`

Il suffit ensuite d'invoquer la méthode `timeit(number=1000000)` qui exécute et chronomètre `number` fois l'instruction `stmt` du constructeur, après exécution de l'initialisation `setup`.

L'exemple qui suit chronomètre la suite d'instructions `w = x ; x = y ; y = w` après initialisation `x = 1 ; x = 2` :

```
1 >>> from timeit import Timer
2 >>> t = Timer('w = x ; x = y ; y = w', 'x = 1 ; y = 2')
3 >>> t.timeit()
4 0.059217929840087891
```

Il est intéressant de comparer ce résultat avec celui de l'instruction suivante :

```
1 >>> t = Timer('x, y = y, x', 'x, y = 1, 2')
2 >>> t.timeit()
3 0.035509824752807617
```

Pour permettre au module `timeit` d'accéder à une fonction, il suffit de l'importer lors de l'initialisation :

```
1 def test() :
2     "Stupid test function"
3     L = []
4     for i in range(100) :
5         L.append(i)
6
7 if __name__=='__main__' :
8     from timeit import Timer
9     t = Timer("test()", "from __main__ import test")
10    print t.timeit()
```


7.7.3. Les modules `profile`, `cProfile` et `pstats`. Pour profiler des (portions de) codes plus importants, il est préférable d'utiliser les modules `profile` ou `cProfile` et `pstats`.

Les modules `profile` et `cProfile` proposent une interface utilisateur analogue. La doc de [Python](#) recommande d'utiliser plutôt `cProfile`, dont l'« overhead »² est moins important.

Pour profiler une portion de code, il suffit d'appeler la fonction

```
cProfile.run(command, filename=None, sort=-1)
```

en lui passant la commande à profiler, le nom du fichier résultat (par défaut sur la sortie standard) et l'ordre d'affichage des résultats (par défaut `sort='stdname'`, le tri se fait par nom de fichier et numéro de ligne). On peut également demander le tri des résultats par nombre d'appels (`sort='calls'`), par temps d'exécution (`sort='time'`) ou par temps d'exécution cumulé (`sort='cumulative'`).

Dans l'exemple qui suit, on applique une fonction donnée à tous les termes d'une (grande) liste numérique, on utilise la fonction `map()` ou bien une technique de liste en compréhension. Voici un moyen pour comparer les temps d'exécution de ces deux méthodes :

```
1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import random, cProfile
4  from math import sin
5  N=1000
6  biglist = [random.randint(-100,100) for i in range(N)]
7
8  def myFunction(x) :
9      return sin(x/2.0)
10 def map_without_list_comprehension() :
11     for i in range(100) :
12         list(map(myFunction, biglist))
13 def map_with_list_comprehension() :
14     for i in range(100) :
15         [myFunction(j) for j in biglist]
16
17 print ("==== map_without_list_comprehension =====")
18 cProfile.run('map_without_list_comprehension()')
19 print ("==== map_with_list_comprehension =====")
20 cProfile.run('map_with_list_comprehension()')
```

on obtient le résultat suivant³ :

²c'est à dire le temps d'exécution des fonctions de mesure du temps

³Le même test, avec le module `profile` au lieu de `cProfile`, affiche des temps d'exécution multipliés par 2.5 environ.

```

===== map_without_list_comprehension =====
200004 function calls in 0.528 CPU seconds
Ordered by : standard name
ncalls tottime percall cumtime percall filename :line-
no(function)
[...]
100000 0.259 0.000 0.385 0.000 listcomprehension-
test.py :8(myFunction)
100000 0.126 0.000 0.126 0.000 {built-in method sin}
[...]
===== map_with_list_comprehension =====
200104 function calls in 0.532 CPU seconds
Ordered by : standard name
ncalls tottime percall cumtime percall filename :line-
no(function)
[...]
100 0.145 0.001 0.531 0.005 listcomprehension-
test.py :16(<listcomp>)
100000 0.257 0.000 0.386 0.000 listcomprehension-
test.py :8(myFunction)
[...]
100000 0.129 0.000 0.129 0.000 {built-in method sin}
[...]

```

Si l'on sauvegarde le résultat dans un fichier, celui-ci peut ensuite être exploité et présenté, combiné avec d'autres ou seul, en utilisant la classe `Stats` du module `stats` dont les principales méthodes sont :

- `Stats(fichier)` est le constructeur avec le nom du fichier
- `add(*filenames)` ajoute des fichiers à présenter.
- `strip_dirs()` pour présenter les statistiques avec des noms de fichier courts (le nom de base du fichier seulement). Retourne l'instance `Stats`
- `sort_stats(*keys)` trie les statistiques suivant un ordre défini (voir le paramètre `sort` de `cProfile.run()`). Retourne l'instance de `Stats`
- `print_stats(*restrictions)` affiche les statistiques avec des restrictions sur le nombre de lignes ou de fonctions affichées.
- `dump_stats(filename)` sauvegarde les statistiques dans le fichier `filename`

Voici un exemple d'utilisation :

```

1 import pstats, cProfile
2 import polygone
3 cProfile.run('polygone.go0()', 'go0.prof')
4 cProfile.run('polygone.go1()', 'go1.prof')

```

```

5 s = pstats.Stats()
6 s.add('go0.prof', 'go1.prof')
7 s.strip_dirs().sort_stats("time").print_stats()

```

Contrôle de qualitéTODO

Nous avons déjà dit quelque mots, au paragraphe 2.12 page 45 à propos de la programmation dirigée par la documentation et le module doctest. Sur ce sujet, et sur le développement de projets informatiques, la lecture de l'excellent ouvrage [TZ] est fortement recommandée.

7.7.4. Le module unittest TODO. C'est un module plus contraignant que doctest, mais il permet de maintenir dans un fichier séparé des tests plus étendus.

```

1 import unittest
2 class TestStatisticalFunctions(unittest.TestCase) :
3     def test_average(self) :
4         self.assertEqual(average([20, 30, 70]), 40.0)
5         self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
6         self.assertRaises(ZeroDivisionError, average, [])
7         self.assertRaises(TypeError, average, 20, 30, 70)
8 unittest.main() # Calling from the command line invokes all tests

```

7.8. Calcul scientifique

Les packages *NumPy* et *SciPy*, de loin les plus populaires pour le calcul scientifique, ne sont pas actuellement disponibles en *Python 3*.

On examinera plus en détail les paquetages dédiés au calcul scientifique en *Python 2* au chapitre 8 page 197.

7.8.1. Le module math. Le module *math* donne accès aux fonctions mathématiques à virgule flottante de la bibliothèque C sous-jacente. La liste des fonctions et constantes disponibles est la suivante :

Constantes.

- *e* la constantes $e = \exp(1) \simeq 2.718281828459045$
- *pi* la constante $\pi \simeq 3.141592653589793$

Conversion.

- *degrees(x)* convertit *x* de radians en degré et
- *radians(x)* convertit *x* de degrés en radians.

Fonctions trigonométriques et inverses.

- `sin(x)`, `cos(x)`, `tan(x)` retournent $\sin(x)$, $\cos(x)$ et $\tan(x)$, l'argument est en radians.
- `hypot(x, y)` retourne $\sqrt{x^2 + y^2}$
- `asin(x)`, `acos(x)` sont les fonctions inverses des fonctions `sin` et `cos`. Lèvent l'exception **ValueError** si $|x| > 1$
- `atan(x)` la fonction inverse de `tan`.
- `atan2(x, y)` retourne $\arctan\left(\frac{y}{x}\right)$ si $x \neq 0$ et $\frac{\pi}{2}$ si $x = 0$.

Fonctions hyperboliques et inverses.

- `cosh(x)`, `sinh(x)` et `tanh(x)` retournent les cosinus hyperbolique, sinus hyperbolique, et tangente hyperbolique de x . L'exception **ValueError** est levée si x n'appartient pas aux domaines de définition.
- `acosh(x)`, `asinh(x)`, `atanh(x)` sont les fonctions hyperboliques inverses des précédentes.

Fonctions diverses.

- `ceil(x)` retourne $\min\{n \in \mathbb{Z}, n \geq x\}$;
- `floor(x)` retourne $\max\{n \in \mathbb{Z}, n \leq x\}$;
- `trunc(x)` retourne x tronqué au point décimal.
- `copysign(x, y)` retourne x avec le signe de y ;
- `fmod(x, y)` retourne le reste de la division de x par y , c'est à dire $x - ny$ où $n \in \mathbb{Z}$ est l'unique entier vérifiant $|r| < |y|$ et $\text{signe}(y) = \text{signe}(r)$. Utiliser cette fonction si x et y sont réels (plus précis); lui préférer `x%y` pour x et y entiers (plus rapide);
- `modf(x)` retourne la partie fractionnaire et entière de x . Par exemple

```
1 >>> modf(pi)
2 (0.14159265358979312, 3.0)
```

- `fabs(x)` retourne la valeur absolue de x ;
- `fsum(iterable)` retourne la somme des items de `iterable`, tout comme la fonction intégrée `sum(iterable)`, mais le calcul est plus précis;
- `isfinite(x)` retourne `False` si x est infini ou `NaN`, `True` sinon;
- `isnan(x)` retourne `True` si x est `NaN`, `False` sinon;
- `isinf(x)` retourne `True` si x est infini (positif ou négatif), `False` sinon;

Fonctions logarithme, exponentielle et fonctions spéciales.

- `frexp(x)` retourne la mantisse et l'exposant de x sous la forme d'un couple (m, e) où m est un réel et e un entier tels que $x = m \times 2^e$ et $\frac{1}{2} \leq |m| < 1$. Retourne $(0.0, 0)$ si x vaut 0;
- `factorial(n)` retourne $n!$ pour $n \in \mathbb{N}$. Lève l'exception **ValueError** si $n \notin \mathbb{N}$;
- `ldexp(x, n)` retourne $x \times 2^n$. C'est l'inverse de la fonction `frexp()`.
- `erf(x)` retourne la fonction d'erreur de x , c'est à dire $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$,
- `erfc(x)` retourne la fonction d'erreur complémentaire de x définie par $\text{erfc}(x) = 1 - \text{erf}(x)$
- `exp(x)` retourne e^x

- `expm1(x)` retourne $e^x - 1$ calculé avec une précision accrue lorsque x est proche de 0.
- `gamma(x)` retourne l'image par la fonction *gamma* du réel x , c'est à dire $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$ qui vérifie $\Gamma(n+1) = n!$ pour $n \in \mathbb{N}$
- `lgamma(x)` retourne $\ln(|\Gamma(x)|)$.
- `log(x, base=e)` retourne le logarithme de x de base $base$, c'est à dire $\frac{\ln(x)}{\ln(base)}$. Lève l'exception **ValueError** si $x \leq 0$
- `log10(x)` retourne $\log(x, 10)$
- `log1p(x)` retourne $\ln(1+x)$. Lorsque x est voisin de 0, le calcul est raffiné.
- `pow(x, y)` retourne x^y . `pow(1.0, y)` et `pox(x, 0.0)` retournent 1 dans tous les cas. L'exception **ValueError** est levée si $x < 0$ et $y \notin \mathbb{Z}$
- `sqr t(x)` retourne la racine carrée de x . L'exception **ValueError** est levée si $x < 0$.

Quelques exemples d'utilisation

```

1 >>> from math import *
2 >>> cos(pi / 4.0)
3 0.7071067811865476
4 >>> log(1024, 2)
5 10.0

```

7.8.2. Le module `cmath`. Contient des fonctions analogues à celles du module `math`, mais dédiées aux nombres complexes, ainsi que les fonctions suivantes :

- `phase(z)` retourne l'argument du complexe z ;
- `polar(z)` retourne le couple module, argument des coordonnées polaires du complexe z ;
- `rect(r, teta)` retourne le complexe z dont les coordonnées polaires sont r et $teta$.

La détermination des fonctions `log()` et trigonométriques inverses `asin()`, `acos()`, etc. correspondent au plan complexe muni de la coupure π , c'est à dire le demi-axe des x négatifs. Voir la documentation [Python](#).

7.8.3. Le module `random`. Le module `random` fournit des outils pour faire des sélections aléatoires :

- `random()` retourne un réel aléatoire dans l'intervalle $[0, 1]$;
- `randint(a, b)` retourne un entier aléatoire dans l'intervalle $[a, b]$;
- `randrange(a, b)` retourne un entier aléatoire dans l'intervalle $[a, b[$;
- `choice(seq)` choisi un élément aléatoirement dans la séquence `seq` ;
- `shuffle(liste)` mélange *in situ* les éléments de la liste ;
- `sample(population, k)` choisit un échantillon de k individus unique dans la séquence `population` ;

- `gauss(mu, sigma)` retourne un réel aléatoire suivant une distribution de Gauss de moyenne `mu` et d'écart type `sigma` ; d'autres distributions sont proposées avec les méthodes : `lognormvariate()`, `betavariate()`, `normalvariate()`, `expovariate()`, `gammavariate()`, `paretovariate()`, `vonmisesvariate()`, `triangular()`, `uniform()`, `weibullvariate()`.

Exemple d'utilisation :

```

1 >>> import random
2 >>> random.choice(['apple', 'pear', 'banana'])
3 'apple'
4 >>> random.sample(xrange(100), 10)
5 [55, 59, 52, 33, 66, 44, 78, 58, 62, 6]
6 >>> random.random()
7 0.27695553082643876
8 >>> random.randrange(6)
9 1
10 >>> l=range(-2,4)
11 >>> random.shuffle(l)
12 >>> l
13 [3, 1, -2, -1, 0, 2]
```

7.8.4.

7.9. Autres modules

- `urllib2` et `smtplib` sont deux modules des plus simples pour récupérer des données depuis des url et pour envoyer du courrier :

```

1 >>> import urllib2
2 >>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/
3   cgi-bin/timer.pl') :
4   ... if 'EST' in line : # look for Eastern Standard Time ...
5   print line <BR>Nov. 25, 09 :43 :32 PM EST
6 >>> import smtplib
7 >>> server = smtplib.SMTP('localhost')
8 >>> server.sendmail('soothsayer@example.org', 'jcaesar@example.
9   org',
10  """To : jcaesar@example.org
11  From : soothsayer@example.org
12  Beware the Ides of March.
13  """)
14 >>> server.quit()
```

- Le paquetage `email` pour gérer les messages électroniques, y compris les documents *MIME* et les autres documents basés sur la *RFC-2822*. Contrairement à `smtplib` et `poplib`, qui envoient et reçoivent effectivement des messages, le paquetage `email` a une boîte à outils complète pour construire ou décoder des messages à la structure complexe (incluant des pièces jointes) et pour le codage et les entêtes.
- Les paquetages `xml.dom` et `xml.sax` fournissent un support robuste pour analyser ces formats d'échange de données très répandus. De même, le module `csv` effectue des lectures et des écritures directement dans un format de base de données commun.
- L'internationalisation est supportée par un certain nombre de modules, incluant les paquetages `gettext`, `locale` et `codecs`.
- Le module `re` fournit des outils d'expressions régulières pour un traitement avancé des chaînes. Pour des appariements et des traitements complexes, les expressions régulières offrent des solutions succinctes et optimisées :

```

1 >>> import re
2 >>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
3 >>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')

```

Lorsque seules des opérations simples sont requises, les méthodes des chaînes sont préférables car elles sont plus simples à lire et à déboguer :

```

1 >>> 'tea for too'.replace('too', 'two')

```

Exercices TODO

Exercice 75. `os` et `os.path`

- (1) Retournez la liste des fichiers du répertoire courant (module `os`)
- (2) Lister récursivement l'adresse absolue de tous les fichiers `.py` et `.txt` qui s'y trouvent.

Exercice 76. *Parcours de répertoires.* (utiliser `os.walk()` et `os.path`) À l'occasion d'un important backup de sa photothèque, un photographe veut s'assurer que toutes ses images ont bien été sauvegardées. Il crée une liste des répertoires et des fichiers contenus avec la taille en octets, dans le dossier d'origine, puis dans le dossier de sauvegarde. Il compare ensuite les deux listes ainsi constituées et affiche les différences.

- (1) Écrire un script **Python** `walk.py`, qui prend comme argument un répertoire, en parcourt récursivement tous les sous répertoires, et liste à l'écran sous la forme (`nomfichier`, `taille`), tous les fichiers dont l'extension est dans la liste

```

1 IMAGES = ('.jpg', '.jpeg', '.tiff', '.tif', '.gif',
2           '.bmp', '.png', '.svg', '.xcf', '.mov',
3           '.mp3', '.mp4', '.mpeg', '.mpg', '.avi', '.cr2')

```

La liste sera triée avant affichage ;

- (2) exécuter ce script depuis le répertoire photothèque d'origine, et depuis le répertoire backup et produire deux fichiers `list.txt` et `listbackup.txt`. La commande Unix `$ walk.py > list.txt` permet de rediriger l'affichage dans le fichier `list.txt` ;
- (3) écrire une fonction qui prend en entrée les deux fichiers `list.txt` et `listbackup.txt` créés, parcourt `list.txt` ligne par ligne, vérifie que chaque fichier listé se trouve dans le fichier `listbackup.txt` avec la bonne taille. S'il ne s'y trouve pas, ou si sa taille n'est pas identique, afficher un message.

Exercice 77. Modules standards

- (1) Calculez le cosinus de $\pi/2$ (module `math`) ;
- (2) écrivez la racine carrée des nombres de 10 à 20 (module `math`) ;
- (3) générez une séquence aléatoire de 20 entiers tirés entre 1 et 4 (module `random`) ;
- (4) écrivez les nombres de 1 à 5 avec 3 secondes d'intervalle (module `time`) ;
- (5) faites afficher la date du jour (`time`) ;
- (6) déterminez votre jour (lundi, mardi...) de naissance (module `calendar`).
- (7) Trouver l'adresse absolue de votre répertoire utilisateur (module `os.path`).
- (8) Retournez la liste des fichiers du répertoire courant (fonction `os.system()`)

Exercice 78. Tri

- (1) Trouver la liste des codes ASCII des caractères imprimables (utiliser le module `string`)
- (2) Créer une liste aléatoire `L` de $N = 100$ caractères ASCII imprimables (utiliser le module `random`)
- (3) Transformer cette liste en chaîne de caractères `C`. (utiliser la méthode `str.join()`).
- (4) Est-il possible de trier cette chaîne in situ ? Pourquoi ?
- (5) Créer une nouvelle chaîne `C` contenant les caractères de `C` triés.
- (6) Créer une nouvelle chaîne `C` contenant les caractères de `C` triés indépendamment de la casse. (Utiliser l'argument `key` de `sorted()` et la méthode `str.lower()`)

Exercice 79. Mesure de performances.

- (1) Tester à l'aide du module `cProfile` la vitesse d'exécution des différentes méthodes de calcul du produit scalaire, proposées à l'exercice [20 page 70](#)
- (2) Idem pour le produit matriciel proposé à l'exercice [21 page 70](#)

Solutions des exercices

Solution 75 os et os.path

- (1) Adresse absolue d'un chemin `>>> os.path.abspath('.')`
- (2) contenu du répertoire courant : deux solutions
 - `>>> os.system('ls')`
 - `>>> os.listdir('.')`
- (3) Lister récursivement le contenu d'un répertoire


```

1 >>> racine = '~/devel/Version2/Concha/'
2 >>> for r,d,F in os.walk(racine) :
3 ...   for f in F :
4 ...       if os.path.splitext(f)[1] in ('.py','.txt') :
5 ...           print os.path.basename(f)

```

Solution 76 os.walk et os.path

- (1) La fonction `findFiles()` trouve toutes les images dans le répertoire et ses sous répertoires, et en retourne la liste. À noter que l'on supprime les points dans l'extension des fichiers par `strip('.')` et que l'on convertit l'extension en minuscule avec `lower()`, ce qui permet de ne pas oublier les fichiers avec extension Jpeg ou MPEG

```

1 #!/usr/bin/python
2 import sys, os
3 IMAGES = ('jpg','jpeg','tiff','tif','gif',
4           'bmp','png','svg','xcf','mov',
5           'mp3','mp4','mpeg','mpg','avi','cr2','pcd')
6
7 def findFiles(mainroot='', exts = IMAGES) :
8     liste = []
9     for root, dirs, files in os.walk(mainroot) :
10         for file in files :
11             if os.path.splitext(file)[1].strip('.').lower() in
12                 exts :
13                 af = os.path.join(root,file)
14                 size = os.path.getsize(af)/(1024.0**2)
15                 liste.append((af, size))
16
17     return liste
18
19 if __name__ == '__main__' :
20     try : root = sys.argv[1]
21     except IndexError : root = '.'
22     exts = IMAGES
23     liste = findFiles(root, exts)
24     for image,size in sorted(liste) :
25         print('{0} => {1 :.2f} Mo'.format(image,size))

```

- (2) La comparaison des listes peut se faire à l'aide de la fonction `compare()` que voici :

```

1 def compare (file1, file2) :
2     list2 = [line.strip().split('=>') for line in open(file2).
3               readlines()]

```

```

3     for line in open(file1) :
4         line = line.strip().split('=>')
5         if not line in list2 :
6             print('manque {}'.format(line))

```

Solution 77 Modules standards

(1) cosinus

```

1  >>> from math import cos, pi
2  >>> print(cos(pi/2))

```

;

(2) racines carrées

```

1  >>> from math import sqrt
2  >>> print([sqrt(n) for n in range(10,21)])
3  [3.1622776601683795, 3.3166247903553998, 3.4641016151377544,
4  3.6055512754639891, 3.7416573867739413, 3.872 983346207417,
5  4.0, 4.1231056256176606, 4.2426406871192848, 4.358898943540674,
6  4.4721359549995796]

```

;

(3) nombres aléatoires

```

1  >>> a=range(1,5)
2  >>> for i in range(20) :
3  ...     print(random.choice(a),)
4  2 2 4 1 3 1 3 4 3 3 1 4 4 1 2 2 3 3 4 1

```

;

(4) affichage à intervalle régulier

```

1  >>> for i in range(5) :
2  ...     i
3  ...     time.sleep(3)

```

;

(5) date du jour :

```

1  >>> import time
2  >>> time.localtime()

```

ou encore

```
1 >>> from datetime import date
2 >>> date.today()
```

;

(6) jour de naissance

```
1 >>> calendar.weekday(1953,12,02)
```

(7) Adresse absolue d'un chemin

```
1 >>> os.path.abspath('.')
```

(8) Liste des fichiers du répertoire courant

```
1 >>> os.listdir('.')
```

Solution 78 Tri

```
(1) >>> P = string.printable
(2) >>> L = [P[random.randint(0,len(P))]] for i in range(N)]
(3) >>> C = ''.join(L)
```

```
(4)
1 >>> C.sort()
2 Traceback (most recent call last) :
3 [...]
4 AttributeError : 'str' object has no attribute 'sort'
```

```
(5) >>> ''.join(sorted(C, key=str.lower))
```

Solution 79 (1) Produit scalaire :

```
1 # -*- coding : utf-8 -*-
2 import random
3 import cProfile as prof
4
5 N = 1000000
6 U = [random.random() for i in range(N)]
7 V = [random.random() for i in range(N)]
8
9 def dot0(X,Y) :
10     s = 0
11     for i in range(N) :
12         s += X[i]*Y[i]
13
14 def dot1(X,Y) :
```

```
15     sum([x*y for x,y in zip(X,Y)])
16
17 def dot2(X,Y) :
18     sum(x*y for x,y in zip(X,Y))
19
20 print ("==== dot à la C =====")
21 prof.run('dot0(U,V)')
22 print ("==== dot liste =====")
23 prof.run('dot1(U, V)')
24 print ("==== dot générateur =====")
25 prof.run('dot2(U,V)')
```

– En *Python* 2.6 on obtient le résultat suivant (extrait) :

```
==== dot à la C =====
 4 function calls in 0.245 CPU seconds
==== dot liste =====
 5 function calls in 1.291 CPU seconds
==== dot générateur =====
1000006 function calls in 3.498 CPU seconds
```

– En *Python* 3.1 on obtient une nette amélioration :

```
==== dot à la C =====
 4 function calls in 0.265 CPU seconds
==== dot liste =====
 6 function calls in 0.183 CPU seconds
==== dot générateur =====
1000006 function calls in 2.449 CPU seconds
```

CHAPITRE 8

Python pour le calcul scientifique

Suivant le lien http://www.scipy.org/Topical_Software, on trouvera une liste assez complète de packages scientifiques *Python* dans différents domaines d'application.

On relira avec profit la section 7.7 page 183

8.1. NumPy

Cette partie est largement inspirée du tutoriel http://www.scipy.org/Tentative_NumPy_Tutorial

NumPy est un package *Python*, disponible pour les versions 2.x seulement, dont le but principal est le traitement des tableaux *multidimensionnels*, *homogènes*, c'est à dire des tableaux *multi-indexés*, d'éléments de *même type*. Par exemple des vecteurs, matrices, images, ou feuilles de calcul.

Un tableau multidimensionnel possède plusieurs dimensions ou *axes*. Le terme dimension est ambigu, aussi nous parlerons plutôt de tableau multi-axes en cas d'ambiguïté. Dans un tableau multi-axes, les items sont repérés par des coordonnées entières, c'est pourquoi le terme multi-axes est plus adapté.

Par exemple, un vecteur n'a qu'un seul axe, les items y sont repérés par leur position c'est à dire leur unique coordonnée entière.

Une matrice possède deux axes, ses éléments sont repérés par deux coordonnées entières. Pour une matrice $A = (a_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$ l'élément $a_{i,j}$ a pour coordonnées entières i et j , la notation *NumPy* est $A[i, j]$. Les entiers n et m sont les *dimensions* de la matrice suivant les deux axes. En *NumPy*, on dira plutôt que la *forme (shape)* de la matrice est (m, n) .

8.1.1. Méthodes et fonctions de *NumPy*.

- Création de tableaux : `arange`, `array`, `copy`, `empty`, `eye`, `fromfile`, `fromfunction`, `identity`, `linspace`, `logspace`, `mgrid`, `ogrid`, `ones`, `r_`, `zeros`, `zeros_like` ;
- conversions : `astype`, `atleast`, `mat` ;
- manipulations : `split`, `column_stack`, `concatenate`, `diagonal`, `dsplit`, `dstack`, `hsplit`, `hstack`, `item`, `newaxis`, `ravel`, `repeat`, `reshape`, `resize`, `squeeze`, `swapaxes`, `take`, `transpose`, `vsplit`, `vstack` ;
- requêtes booléennes : `all`, `any`, `nonzero`, `where` ;
- ordre : `argmax`, `argmin`, `argsort`, `max`, `min`, `ptp`, `searchsorted`, `sort` ;

- opérations : choose, compress, cumprod, cumsum, inner, fill, imag, prod, put, putmask, real, `sum` ;
- statistique de base : cov, mean, std, var ;
- algèbre linéaire de base : cross, dot, outer, svd, vdot ;

8.1.2. Les bases de NumPy. Les tableaux multidimensionnels *NumPy* sont représentés par la classe `numpy.ndarray`. À ne pas confondre avec la classe `array` de *Python* qui ne représente que des tableaux uni-dimensionnels.

Les principaux attributs de la classe `ndarray` sont :

- `ndarray.ndim` est le nombre d'axes (on dit parfois aussi le rang)
- `ndarray.shape` est un tuple d'entiers donnant la taille du tableau suivant chaque axe. Le tuple `shape` est donc de longueur égale au nombre d'axes du tableau, `ndim`.
- `ndarray.size` est le nombre total d'éléments du tableau. C'est le produit des termes de `shape`. Par exemple, pour une matrice de $\mathbb{R}^{m,n}$, `size` vaut $m \times n$.
- `ndarray.dtype` est le type des éléments du tableau. Divers types standard *Python* peuvent être spécifiés. *NumPy* propose aussi les types `bool_`, `character`, `int_`, `int8`, `int16`, `int32`, `int64`, `float_`, `float8`, `float16`, `float32`, `float64`, `complex_`, `complex64`, `object_`.
- `ndarray.itemsize` est la taille en octet de chaque élément du tableau. Par exemple, pour un tableau de type `float64`, l'attribut `itemsize` vaut $64/8 = 8$.
- `ndarray.data` est le buffer contenant les éléments du tableau. On aura rarement besoin d'y accéder directement, mais plutôt au travers des opérateurs d'indexation et de slicing proposés par *NumPy*.
- `ndarray.base` est le tableau d'origine si le tableau est une vue ou un *slice* d'un autre tableau. S'il a été créé ex-nihilo (par exemple avec `ndarray.zeros()`), alors `base` vaut `None`.

Exemple :

```
1 >>> import numpy
2 >>> a = numpy.arange(10)
3 >>> a.shape = (2,5)
4 >>> a
5 array([[0, 1, 2, 3, 4],
6        [5, 6, 7, 8, 9]])
7 >>> a.shape
8 (2, 5)
9 >>> a.dtype.name
10 'int32'
```

On notera que le package s'appelle *NumPy*, mais que suivant les préconisations de Guido Van Rossum, le module (nom de fichier) s'importe par l'instruction `import numpy` tout en minuscule.

8.1.3. Création de tableaux. NumPy propose de nombreuses méthodes pour créer des tableaux :

à partir d'une liste *Python* , ou d'une liste de liste...

```
1 >>> import numpy as np
2 >>> a = np.array([1,2,3])
3 >>> type(a)
4 <type 'numpy.ndarray'>
5 >>> b = np.array([[1,2,3],[4,5,6]])
6 >>> c = np.array(b)
```

On peut préciser explicitement le type à la création :

```
1 >>> a = np.array([1,2,3])
2 >>> a.dtype
3 dtype('int32')
4 >>> b = np.array([1,2,3], dtype='complex')
5 >>> b.dtype
6 dtype('complex128')
7 >>> b
8 array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Les fonctions `zeros()`, `ones()`, `empty()`, `arange()`, `linspace()` permettent également la création de tableaux :

```
1 >>> a = np.zeros((2,3))
2 >>> b = np.ones(5,dtype='int')
3 >>> c = np.empty(3)
4 >>> c
5 array([ 5.96597778e-264,  0.00000000e+000,  0.00000000e+000])
6 >>> d = arange(start=5, stop=12, step=2.1) #ou arange(5, 12, 2.1)
7 >>> e = np.linspace(0,1,11)
```

`zeros(shape, dtype)`, `ones()` et `empty()` créent un tableau de la forme donnée, rempli de zéros, de uns et non initialisés.

`arange(start, stop, step, dtype)` renvoie un tableau de nombres également espacés, de start à stop, non compris, par pas de step, de type dtype.

`linspace(start, stop, num, endpoint=True)` renvoie un tableau de num nombres également espacés, de start à stop, dernier point compris.

8.1.4. Opérations sur les tableaux. Les opérations, les fonctions sont appliquées aux tableaux élément par élément avec d'éventuelles restrictions sur la compatibilité des dimensions.

Par exemple, pour appliquer la fonction *sinus* à tous les éléments d'un tableau :

```

1 >>> g = np.linspace(0,1,11)
2 >>> a = np.sin(g)
3 >>> a
4 array([ 0.        ,  0.09983342,  0.19866933,  0.29552021,  0.38941834,
5         0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691,
6         0.84147098])

```

Les opérations courantes (l'addition, le produit etc.) entre deux tableaux sont effectués élément par élément. La comparaison entre deux tableaux (de même taille) est possible et renvoie un tableau booléen de même taille.

Si l'on compare, additionne, multiplie, etc. un tableau par un nombre *a*, chaque élément du tableau est comparé, additionné, multiplié, etc. par *a*

```

1 >>> b = np.cos(g)
2 >>> a < b
3 array([ True,  True,  True,  True,  True,  True,  True,  True, False,
4        False, False], dtype=bool)
5 >>> a < 0.5
6 array([ True,  True,  True,  True,  True,  True, False, False, False,
7        False, False], dtype=bool)

```

Pour faire un produit matriciel, il faut utiliser la fonction `dot()` ou bien utiliser les classes de matrices (`matrix`, voir [8.1.10 page 208](#))

```

1 >>> A = array(arange(12)).reshape(3,4)
2 >>> B = array(arange(9)).reshape((3,3))
3 >>> B*A
4 Traceback (most recent call last) :
5   File "<ipython console>", line 1, in <module>
6   ValueError : shape mismatch : objects cannot be broadcast to a
   single shape
7 >>> dot(B,A)
8 array([[ 20,  23,  26,  29],
9        [ 56,  68,  80,  92],
10       [ 92, 113, 134, 155]])
11 >>> A = matrix(arange(12)).reshape(3,4)
12 >>> B = matrix(arange(9)).reshape((3,3))
13 >>> B*A
14 matrix([[ 20,  23,  26,  29],

```



```

15     [ 56, 68, 80, 92],
16     [ 92, 113, 134, 155]])

```

Lorsque l'on effectue une opération sur deux tableaux, de types distincts (mais compatibles pour cette opération), le résultat est du type le plus général (autrement dit le plus précis) :

```

1 >>> a = ones((2,3), dtype=int)
2 >>> b = np.random.random((2,3))
3 >>> c = a+b
4 >>> c.dtype
5 dtype('float64')

```

Attention toutefois, si l'on ajoute un (tableau de) réel(s) b à un tableau d'entiers a « in situ », a reste de type `int` :

```

1 >>> a += 1.5
2 >>> a
3 array([[2, 2, 2],
4        [2, 2, 2]])

```

On peut sommer les éléments d'un tableau, trouver le minimum, le maximum, sommer dans une direction d'axe :

```

1 >>> a = np.random.random((2,3))
2 >>> a
3 array([[ 9.59243730e-01,  1.57245491e-01,  7.87875050e-02],
4        [ 3.31722781e-04,  2.30972784e-01,  9.54687062e-01]])
5 >>> a.sum()
6 2.3812682944516514
7 >>> a.min()
8 0.00033172278084458995
9 >>> a.sum(1) #par ligne (axe 1)
10 array([ 1.19527673,  1.18599157])

```

8.1.5. Indicage, découpage, itérations sur les tableaux. Le *slicing* (découpage) est l'art de découper les tableaux. Les tableaux *unidimensionnels* peuvent être indexés, découpés, itérés comme les séquences *Python*.

Les tableaux multi-dimensionnels sont indexés par des tuples, avec un indice par axe. Les indices sur chaque axe suivent les mêmes règles que les indices de tableau unidimensionnels.

```
1 >>> def f(x,y) :  
2 ...     return 10*x+y  
3 ...  
4 >>> b = np.fromfunction(f,(3,4),dtype=int)  
5 >>> b  
6 array([[ 0,  1,  2,  3],  
7        [10, 11, 12, 13],  
8        [20, 21, 22, 23]])  
9 >>> b[2,3]  
10 23  
11 >>> b[:,1]  
12 array([ 1, 11, 21])  
13 >>> b[:,2:3]  
14 array([[ 2],  
15        [12],  
16        [22]])
```

Il est possible d'« oublier » certains indices en fin de tuple. Dans ce cas, *NumPy* complète par autant de `' , : '` que nécessaire.

Les points de suspension sont remplacés par autant de `' , : '` que nécessaire.

Exemple :

```
1 >>> a = arange(24).reshape(3,2,4)  
2 >>> a  
3 array([[[ 0,  1,  2,  3],  
4        [ 4,  5,  6,  7]],  
5  
6        [[ 8,  9, 10, 11],  
7        [12, 13, 14, 15]],  
8  
9        [[16, 17, 18, 19],  
10       [20, 21, 22, 23]])  
11  
12 >>> a[0] # idem a[0, :, :]  
13 array([[0, 1, 2, 3],  
14        [4, 5, 6, 7]])  
15 >>> a[:,2] # idem a[:, :, 2]  
16 array([[ 2,  6],  
17        [10, 14],  
18        [18, 22]])
```

Il est possible d'itérer sur les lignes (premier axe) d'un tableau :

```
1 >>> a = arange(6).reshape(3,2)
2 >>> for row in a :
3 ...     print(row)
4 [0 1]
5 [2 3]
6 [4 5]
```

Pour itérer sur tous les éléments d'un tableau, il faut le « mettre à plat » et utiliser l'attribut `flat` qui est un itérateur sur les items du tableau :

```
1 >>> a = arange(6).reshape(3,2)
2 >>> for x in a.flat :
3 ...     print x,
4 ...
5 0 1 2 3 4 5
```

Pour parcourir les indices et éléments d'un tableau, l'itérateur `ndenumerate()` fonctionne de manière analogue à `enumerate()` de *Python* en retournant les coordonnées entières et la valeur de chaque élément :

```
1 >>> a = arange(6).reshape(3,2)
2 >>> for ij,x in ndenumerate(a) : print ij,x
3 ...
4 ...
5 (0, 0) 0
6 (0, 1) 1
7 (1, 0) 2
8 (1, 1) 3
9 (2, 0) 4
10 (2, 1) 5
```

8.1.6. Remodelage des tableaux. Dans les tableaux *NumPy* les données sont alloués dans une zone de mémoire contigüe, la forme (nombre et taille des axes) n'affecte pas le tableau lui-même mais uniquement la manière d'accéder aux éléments. On peut modifier la forme (*shape*) de divers manières sans modifier les données elle-mêmes :

```
1 >>> a = floor(10*np.random.random((3,2)))
2 >>> a
3 array([[ 5.,  6.],
4        [ 1.,  1.],
5        [ 5.,  1.]])
```

```

6 >>> a.shape
7 (3, 2)
8 >>> a.ravel()
9 array([ 5., 6., 1., 1., 5., 1.])
10 >>> a.reshape(2,3)
11 array([[ 5., 6., 1.],
12        [ 1., 5., 1.]])
13 >>> a.shape
14 (3, 2) #a non modifié

```

Sauf demande expresse de l'utilisateur (paramètres `offset`, `stride` et `order` de `ndarray`), *NumPy* crée les tableaux « style C » ordonnés avec le dernier indice variant le plus vite (`A[0,0]` puis `A[0,1]`, etc.). Dans ce cas, les fonctions `ravel()`, `reshape()` retournent un nouveau tableau avec les *même* données, mais avec sa nouvelle forme (sans recopie). Il est cependant possible de demander aux fonctions `ravel()` et `reshape()` de retourner un tableau « style FORTRAN » c'est à dire avec le premier indice variant le plus vite. Dans ce cas, ces fonctions retournent une *recopie* du tableau. Dans tous les cas, le tableau lui-même n'est pas modifié.

Le méthode `resize()` modifie le tableau lui-même et ne retourne rien :

```

1 >>> a.resize(1,6)
2 >>> a.shape
3 (1,6)

```

Dans ces opérations de remodelage, le dernier indice peut prendre la valeur -1, auquel cas *NumPy* calcule lui-même la dernière dimension de sorte que la taille totale (nombre d'items) du tableau soit inchangée.

8.1.7. Empilage de tableaux. Les fonctions et méthodes `vstack`, `hstack`, `column_stack`, `concatenate`, `c_` et `r_` permettent d'empiler (*stack*) les tableaux *NumPy* suivant les différents axes. Les dimensions doivent être compatibles.

```

1 >>> a = np.arange(6).reshape(2,3)
2 >>> a
3 array([[0, 1, 2],
4        [3, 4, 5]])
5 >>> b = np.arange(9).reshape(3,3)
6 >>> b
7 array([[0, 1, 2],
8        [3, 4, 5],
9        [6, 7, 8]])
10 >>> np.vstack((a,b))
11 array([[0, 1, 2],
12        [3, 4, 5],

```

```
13     [0, 1, 2],
14     [3, 4, 5],
15     [6, 7, 8]])
```

La fonction `concatenate()` permet des actions analogues :

```
1 >>> b = np.arange(4).reshape(2,2)
2 >>> a = np.arange(4).reshape(2,2)
3 >>> np.concatenate((a,b))
4 array([[0, 1],
5        [2, 3],
6        [0, 1],
7        [2, 3]])
8 >>> np.concatenate((a,b),1)
9 array([[0, 1, 0, 1],
10        [2, 3, 2, 3]])
```

8.1.8. Copies et vues. Lorsque l'on manipule des tableaux *NumPy*, parfois les données sont dupliquées, parfois elles ne le sont pas. C'est une source de confusion fréquente.

Pas de recopie. Lors d'une simple affectation, il n'y a aucune recopie :

```
1 >>> import numpy as np
2 >>> a = np.arange(6)
3 >>> b = a
4 >>> b is a
5 True
```

Copie de surface. Des tableaux distincts peuvent partager les mêmes données, sous deux formes différentes ; ce sont deux vues distinctes des mêmes données. La méthode `ndarray.view()` retourne une nouvelle vue sur le tableau, qui partage donc les mêmes données.

```
1 >>> v = a.view()
2 >>> v is a
3 False
4 >>> v.base is a
5 True
6 >>> v.shape = 2,3
7 >>> v[0,0] = 314
8 >>> a[0,0]
9 314
```

Un *slice* de tableau retourne une vue (partielle) sur le tableau. Attention, ça n'est pas le fonctionnement ordinaire de *Python* pour qui un slice retourne une *copie* de la partie de tableau slicée.

```

1 >>> s = v[ :]
2 >>> s[1,1] = 44444
3 >>> v
4 array([[ 314,  1,    2],    [  3, 44444,  5]])
5 >>> a
6 array([ 314,  1,    2,    3, 44444,  5])
7 >>> s = v[ :1]
8 >>> s[0] = -10
9 >>> a
10 array([ -10, -10, -10,  3, 44444,  5])

```

Copie profonde. pour faire une recopie profonde du tableau, utiliser la méthode `ndarray.copy()` :

```

1 >>> a = np.arange(6)
2 >>> u = a.copy()
3 >>> u is a
4 False
5 >>> u.base is a
6 False
7 >>> u[0] = 3000
8 >>> a
9 array([0, 1, 2, 3, 4, 5])

```

8.1.9. Quelques fonctions plus évoluées. En plus des facilités d'indexation avec des entiers et des slices, *Numpy* propose l'indexation avec des tableaux d'indices ou de booléens.

Indexation par liste d'entiers. Pour faire référence aux éléments $a_1, a_1, a_1, a_3, a_{10}$ d'un tableau $A = \text{arange}(10)**2$, il suffit d'indexer A par la liste $L = [1, 1, 1, 6, 9]$ ou le tableau `array([1, 1, 1, 6, 9])` (voir l'exercice 8.5.5 page 219). On peut utiliser ce mode d'indexation en consultation mais aussi en affectation :

```

1 >>> L = [1, 1, 1, 6, 9]
2 >>> A[L]
3 np.array([ 1,  1,  1, 36, 81])
4 >>> A[L] = [2, 3, 4, -3, -10]
5 >>> A
6 np.array([ 0,  4,  4,  9, 16, 25, -3, 49, 64, -10])

```

Indexation par liste de booléens. Un tableau de booléen peut être passé pour indexer un tableau, dans ce cas, les éléments correspondant à la valeur True sont extraits du tableau, ceux indexés par False ne le sont pas. Le tableau extrait est mis à plat :

```

1 >>> a = np.arange(12)
2 >>> a.shape = (3,4)
3 >>> b = a > 4
4 >>> b
5 array([[False, False, False, False],
6        [False, True, True, True],
7        [True, True, True, True]], dtype=bool)
8 >>> a[b]
9 array([ 5, 6, 7, 8, 9, 10, 11])

```

Il est possible d'extraire des lignes ou des colonnes complètes, dans ce cas le tableau résultant n'est pas remis à plat :

```

1 >>> b1 = np.array([False,True,True])
2 >>> b2 = np.array([True,False,True,False])
3 >>> a[b1, :]
4 array([[ 4, 5, 6, 7],
5        [ 8, 9, 10, 11]])
6 >>> a[:,b2]
7 array([[ 0, 2],
8        [ 4, 6],
9        [ 8, 10]])

```

Ce type d'indexation est à manier avec précaution. Si l'on indexe une liste de booléens au lieu d'un array de booléens, le résultat est surprenant (bug ?) :

```

1 >>> b1 = [False,True,True]
2 >>> b2 = [True,False,True,False]
3 >>> a[b1, :]
4 >>> A[b1, :]
5 array([[0, 1, 2, 3],
6        [4, 5, 6, 7],
7        [4, 5, 6, 7]])
8 >>> a[:,b2]
9 array([[1, 0, 1, 0],
10       [5, 4, 5, 4],
11       [9, 8, 9, 8]])
12 array([[ 0, 2],
13       [ 4, 6],

```

```
14 [ 8, 10]])
```

8.1.10. Algèbre linéaire de base. *NumPy* propose quelques opérations d'algèbre linéaire élémentaires. Des fonctionnalités plus évoluées se trouvent dans le module `linalg` de *SciPy*, y compris la manipulation de systèmes linéaires creux (sparse). Voir [8.2 page suivante](#).

NumPy propose une classe `matrix` que l'on examine plus loin, ainsi que quelques opérations d'algèbre linéaire sur des tableaux :

```
1 >>> from numpy import *
2 >>> from numpy.linalg import *
3 >>> a = array([[1.0, 2.0], [4.0, 3.0]])
4 >>> a.transpose()
5 array([[ 1.,  3.],
6        [ 2.,  4.]])
7 >>> inv(a)
8 array([[ -0.6,  0.4],
9        [ 0.8, -0.2]])
10 >>> u = eye(2)
11 >>> j = array([[0.0, -1.0], [1.0, 0.0]])
12 >>> dot(j, j)
13 array([[ -1.,  0.],
14        [ 0., -1.]])
15 >>> trace(u)
16 2.0
17 >>> y = array([[5.], [7.]])
18 >>> solve(a, y)
19 array([[ -0.2],
20        [ 2.6]])
21 >>> eig(j)
22 (array([ 0.+1.j, 0.-1.j]),
23  array([[ 0.70710678+0.j , 0.70710678+0.j ],
24        [ 0.00000000-0.70710678j, 0.00000000+0.70710678j]]))
```

La classe `matrix`, une introduction sommaire

```
1 >>> A = matrix('1.0 2.0; 3.0 4.0')
2 >>> A
3 matrix([[ 1. 2.]
4         [ 3. 4.]])
5 >>> type(A)
6 <class 'numpy.matrixlib.defmatrix.matrix'>
7 >>> A.T
```



```
8 matrix([[ 1. 3.]
9         [ 2. 4.]])
10 >>> X = matrix('5.0 7.0')
11 >>> Y = X.T
12 >>> Y
13 matrix([[5.]
14         [7.]])
15 >>> A*Y
16 matrix([[19.]
17         [43.]])
18 >>> print A.I
19 [[-2. 1. ]
20  [ 1.5 -0.5]]
21 >>> solve(A, Y)
22 matrix([[-3.]
23         [ 4.]])
```

8.2. Scipy

Quelques tâches courantes en calcul scientifique :

- algèbre linéaire, `scipy.linalg`
- optimisation, `scipy.optimize`
- intégration et équations différentielles `scipy.integrate`
- fonctions spéciales, `scipy.special`
- transformation de Fourier, `scipy.fftpack`
- traitement du signal et imagerie, `scipy.signal`
- images multidimensionnelles `scipy.ndimage`
- statistiques `scipy.stats`
- entrées-sorties `scipy.io`
- inclure du code C dans du code *Python* avec `scipy.weave`

8.3. Calcul formel avec Sage

<http://www.sagemath.org/>

8.4. C/C++ en Python

voir <http://docs.scipy.org/doc/numpy/user/c-info.python-as-glue.html>

8.4.1. Écrire en *Python*, relire en C++ (binaire). Prenons un exemple concret. Un programme C++, une boîte noire, disons une simulation numérique, doit lire un maillage dans un format qui lui est propre. Mais le maillage est disponible dans un format autre. Un programme *Python* effectue la conversion de format du maillage, et doit produire un fichier binaire lisible par le programme C++.

Le format binaire est préféré au format texte pour des raisons de performance en lecture/écriture, le maillage étant très volumineux.

Le problème est donc de produire en *Python* un fichier binaire lisible en C++ .

Les méthodes et fonctions classiques de *Python* permettent d'écrire des fichiers texte, ou bien des fichiers binaires (module `pickle`) spécifiques à *Python*. Relire ces fichiers en C++ est très malcommode.

Le module array. Une solution basique consiste à utiliser le module `array` de *Python* pour écrire des fichiers binaires directement lisibles en C++ :

```
1 >>> import array
2 >>> A = array.array('d', range(10))
3 >>> A.tofile(open('myfile.array', 'wb'))
```

Ligne 2 : on précise que A est un tableau de 10 éléments de type 'd', c'est à dire réel double précision;

ligne 3 : le tableau A est écrit dans un fichier ouvert en binaire ('wb')

Le programme de lecture en C++ doit connaître exactement le type et le nombre d'éléments à lire.

Voici un petit programme permettant de vérifier que la lecture en C++ se passe bien :

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 int main(){
5     ifstream file("myfile.array", ios ::binary|ios ::in|ios ::ate);
6     int size = (int) file.tellg();
7     file.seekg (0, ios ::beg);
8     double a = 0.0;
9     n = size/sizeof(a);
10    for (int i = 0; i<n ; i++)
11    {
12        file.read((char*)&a, sizeof(a));
13        cout << a << " ";
14    }
15    cout << endl;
16    file.close();
```

```
17 }
```

Lignes 5-7 : ouverture du fichier en mode binaire (`ios :: binary`), positionnement en fin de fichier (`ios :: ate`), calcul de la taille du fichier en nombre d'octets (`file.tellg()`), repositionnement en début de fichier (`ios :: beg`) pour lecture.

Ligne 8-9 : les éléments sont des réels double précision, il y en a `n` à lire.

ligne 12 : le C++ est ainsi fait qu'il faut lire le nombre d'octets (`char`) requis (`sizeof(a)` octets) les affecter à la variable `a` sous forme brute. Si l'on passe d'un système d'exploitation à un autre ou bien d'un mode de représentation des double précision à un autre, le programme ne fonctionne plus.

Après compilation, et exécution, ce programme C++ affiche ce que l'on attendait :

```
1 0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 ;
```

Le module struct. Il permet de sauvegarder sous forme binaire des données hétérogènes, en précisant l'*endianness*¹. Dans l'exemple précédent supposons que l'on ait besoin de stocker plusieurs tableaux à la suite dans le même fichier binaire, en précisant leurs dimensions. L'instruction *Python* suivante retourne sous forme de chaîne de caractères (d'octets) la longueur du tableau `A` (`len(A)`), qui est un entier. (`'i'`)

```
1 >>> struct.pack('i', len(A))
```

Dans l'exemple ci-dessous, on crée en *Python* un tableau `A` (tableau d'entiers) et un tableau `B` (tableau de réels double précision).

```
1 >>> A = array.array('i', range(10))
2 >>> B = array.array('d', range(100, 200))
```

Pour créer un fichier binaire contenant la dimension de `A` suivie des valeurs de `A`, suivie de la dimension de `B` suivie des valeurs de `B`, il suffit donc d'écrire :²

```
1 >>> file = open('myfile.struct', 'wb')
2 >>> file.write(struct.pack('i', len(A)))
3 >>> A.tofile(file)
4 >>> file.write(struct.pack('i', len(B)))
5 >>> B.tofile(file)
6 >>> file.close()
```

¹L'*endianness* (ou le *boutisme*) est l'ordre dans lequel sont écrit les octets. La racine du mot fait référence à « *end* » et non pas à « *endian* ». Si les nombres sont écrits de gauche à droite, on parle de *big-endian*, (gros-boutisme) - on commence par l'octet de poids fort- sinon on parle de *little-endian*. Par extension, une date écrite sous forme AAAA/MM/JJ est *big-endian*. L'attribut `sys.byteorder` permet de connaître l'*endianness* de votre processeur.

²Le fichier `myfile.struct` généré par cette suite d'instructions *n'est pas portable*. Si l'on change de machine, de processeur, voire de compilateur, il ne sera pas possible de le relire en C++.

Les instructions de lecture en C++ peuvent prendre la forme : TODO commenter

```

1 ifstream file("myfile.struct", ios ::binary|ios ::in|ios ::ate);
2 int size = (int) file.tellg();
3 file.seekg (0, ios ::beg);
4 int m;
5 file.read((char*)&m, sizeof(m));
6 int *a = new int[m];
7 for (int i = 0; i<m; i++)
8     file.read((char*)&a[i], sizeof(int));
9 file.read((char*)&m, sizeof(m));
10 nbytes = sizeof(double) ;
11 double *b = new double[m];
12 for (int i = 0; i<m; i++)
13     file.read((char*)&b[i], nbytes);

```

8.4.2. pyrex, cython. TODO : prendre le problème à la base, voir sur <http://docs.cython.org/src/userguide/tutorial.html>

pyrex est un langage analogue à *Python*, utilisé pour créer des modules en C pour *Python*.

*cython*³ est un langage de programmation, surcouche de *Python*, proposant des extensions de syntaxe semblable à du C ou C++. Une de ces extensions permet la déclaration (optionnelle) de types statiques. Par exemple en *cython*, on pourra trouver des déclarations comme :

```

1 cdef inline invSquareInline(int i) :
2     return 1./i**2

```

Le mot clé *cdef* remplace dans ce cas l'instruction (*Python*) *def*. Pour les programmeurs C, l'instruction *inline* n'a pas de secret. Le paramètre *i* de la fonction est explicitement déclaré entier, comme en C/C++.

cython est basé sur *pyrex*, proposant (selon la documentation) quelques fonctionnalités et optimisations d'avant garde supplémentaires.

cython est actuellement à la version 0.12.1 et n'est pas exploitable de manière totalement opérationnelle. C'est cependant un outil prometteur, en conjonction avec *NumPy* et *SciPy*, pour accélérer radicalement l'exécution des programmes *Python*.

Dans un programme *cython*, le fichier source est traduit en code C/C++ optimisé, puis compilé comme une extension module *Python*.

³Le développement de *cython* est financé par *google* et *enthought*. Voir <http://cython.org/>.

Cette approche permet une exécution très rapide des programmes et une bonne intégration avec les bibliothèques externes **C**, tout en conservant la productivité du programmeur, typique de **Python**.

Ce qui suit, adapté du tutorial *cython*⁴, permet de tester une installation de *cython*, et d'en mesurer l'apport en terme de performance par rapport à un programme **Python** conventionnel. On considère le script **Python** suivant qui calcule 100 fois et retourne une valeur approchée de π basé sur la formule

$$\pi^2 = 6 \sum_{k \leq +\infty} \frac{1}{k^2}$$

Préliminaires : profilage d'un script **Python**. Le script **Python** est stocké dans le fichier `pi1.pyx`. Son nom (`.pyx`) indique qu'il s'agit d'un script *cython*, mais dans un premier temps il ne contient que des instructions en pur **Python**.

```

1 #fichier pi.pyx
2 @cython.profile(False)
3 def pynvSquare(i) :
4     return 1./i**2
5 def pypi(n=65535) :
6     for w in range(100) :
7         val = 0.0
8         for k in range(1,n+1) :
9             val += pynvSquare(k)
10    return (6 * val)**.5

```

Ligne 3 : en terme de performances, on pourrait bien évidemment se dispenser de la fonction `pynvSquare()` mais la performance du programme n'est pas ici l'objectif visé. La série est sommée jusqu'à $n = 65535$ seulement, car au delà, le compilateur **C** donne la valeur $n = 0$ ce qui produit une exception **ZeroDivisionError**.

Ligne 2 : un décorateur pour demander au *profiler cython* de ne pas détailler les appels à la fonction `pynvSquare()`, mais seulement ceux de `pypi()`. Le profilage en détail de la fonction `pynvSquare()` génère un surcoût important qui risque de biaiser le résultat.

Pour évaluer les performances de ce script `pi.pyx`, on l'appelle depuis un script de profiling contenant ces lignes :

```

1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 #fichier cython-profile.py
4 import pstats, cProfile
5 import pyximport
6 pyximport.install() #compilation auto de pi
7 import pi

```

⁴http://docs.cython.org/src/tutorial/profiling_tutorial.html

```

8  for p in ('pi.pypi()',) :
9      cProfile.runctx(p, globals(), locals(), "Profile.prof")
10     s = pstats.Stats("Profile.prof")
11     s.strip_dirs().sort_stats("time").print_stats()

```

Ligne 1 : indique que l'interpréteur est *Python*, c'est lui qui va lancer *cython*.

Lignes 5-6 : permet la compilation automatique du module `pi.pyx` par *cython*. Plus généralement, les scripts `.pyx` importés seront compilés par *cython*, les scripts pur python (`.py`) seront exécutés tel quels par *Python*.

Les lignes suivantes sont des instructions standard en *Python* destinées à « profiler » la fonction `pi.pypi()` du module `pi`.

Ligne 7 : importation du module à profiler.

Ligne 8 : pour le moment, seule la fonction `pi.pypi()` est à profiler. Par la suite, nous ajouterons diverses fonctions optimisées dans le but de comparer les temps d'exécution à celui de l'exemple de base présenté ici.

Ligne 9 : demande au profiler d'exécuter `pi.pypi()`, dans le contexte défini par `globals()` et `locals()`, et de stocker le résultat dans `Profile.prof`.

Ligne 10-11 : le module `pstats` est utilisé pour extraire du fichier `Profile.prof` les statistiques qui nous intéressent. La méthode `strip_dirs()` supprime les chemins absolus de la sortie, pour ne garder que les noms de fichier. Les méthodes `sort_stats()` et `print_stats()` ont des noms suffisamment évocateurs pour qu'il soit inutile de les commenter.

On lance le script de profilage :

```
$ ./cython-test0.py
```

et on obtient le résultat :

```

Mon Apr 18 06 :58 :01 2011 Profile.prof
4 function calls in 4.256 CPU seconds
Ordered by : internal time
ncalls  tottime  percall cumtime  percall filename :line-
no(function)
1 4.256 4.256 4.256 4.256 cypi0.pyx :9(pypi0)
1 0.000 0.000 4.256 4.256 <string> :1(<module>)
1 0.000 0.000 4.256 4.256 {cypi0.pypi0}
1 0.000 0.000 0.000 0.000 {method 'disable' of '_ls-
prof.Profiler' objects}

```

Nous nous attachons au seul résultat final, c'est à dire au temps d'exécution total, ici 4.256 secondes de *CPU*

cython au travail. Modifions maintenant le script `pi.pyx` en y introduisant une version supplémentaire du calcul de π en *cython* :

```

1  cimport cython
2  @cython.profile(False)
3  def invSquare(int i) :
4      return 1./i**2
5  def pi0(int n=65535) :
6      cdef double val = 0.
7      cdef int k
8      for w in range(100) :
9          val = 0.0
10         for k in range(1, n+1) :
11             val += invSquare(k)
12     return (6 * val)**.5

```

Ligne 1 : importation « à la cc ».

Ligne 3 et 5 : *cython* est averti que les paramètres `i` et `n` sont entiers, information qui sera prise en compte par *cython* pour produire un code C optimisé.

Lignes 6-7 : de même pour les variables `val` (réelle double précision) et `k` (entier).

Le reste est inchangé.

On demande ensuite au script de profiling d'exécuter les deux versions `pypi` et `pi0` de la fonction qui calcule π . Ce qui se fait en changeant la ligne 8 fichier `pi.pyx` en :

```

1  for p in ('pi.pypi()', 'pi.pi0()') :
2      etc...

```

L'exécution du `cython-profile.py` fournit le résultat :

```

4 function calls in 4.480 CPU seconds => pypi()
4 function calls in 0.786 CPU seconds => pi0()

```

qui montre qu'une utilisation élémentaire de *cython* divise déjà le temps d'exécution par 5,67

La dernière amélioration suggérée est de déclarer `inline` la fonction `invSquare(int i)`, en utilisant la définition :

```

1  @cython.profile(False)
2  cdef inline invSquareInline(int i) :
3      return 1./i**2
4  def pi1(int n=65535) :
5      cdef double val = 0.
6      cdef int k
7      for w in range(100) :

```

```

8     val = 0.0
9     for k in range(1,n+1) :
10         val += invSquareInline(k)
11     return (6 * val)**.5

```

Ligne 2 : la définition d'une fonction *inline* en *cython* se fait par `cdef`, au lieu de `def` en pur *Python*.

Cette fois-ci le gain en performance est réellement étonnant !

```

4 function calls in 4.480 CPU seconds => pypi()
4 function calls in 0.786 CPU seconds => pi0()
4 function calls in 0.100 CPU seconds => pi1()

```

Au total, sur cet exemple simple, *cython* à permis de diviser par 45 le temps d'exécution.

8.4.3. ctypes, swig.

8.4.4. boost.

8.4.5. shiboken TODO. *Shiboken* est un plugin de GeneratorRunner qui génère du code C++ pour des extensions CPython. La première génération de PySide était basée sur les templates de Boost.

Pour utiliser *Shiboken*, il faut avoir installé les paquetages *apiextractor*, *generatorrunner*, *pyside* et *shiboken* avec les versions de développement.

L'utilisation de *Shiboken* est plutôt complexe, le tutoriel http://developer.qt.nokia.com/wiki/PySide_Binding_Generation_Tutorial est un bon point d'entrée.

Pour résumer le contenu de ce tutoriel, les étapes nécessaires à la création et au test du module *foo*, à partir des codes sources fournis sont les suivantes :

(1) compiler la bibliothèque C++

```

1 $ cd libfoo
2 $ qmake
3 $ make

```

(2) générer, compiler et tester le binding : (cette étape est assez fastidieuse à décortiquer !)

```

1 $ cd foobinding-cmake
2 $ mkdir build
3 $ cd build
4 $ cmake ..
5 $ make
6 $ make test

```


8.4.6. weave.

8.5. Visualisation de données

8.5.1. matplotlib, pylab. http://matplotlib.sourceforge.net/users/pyplot_tutorial.html

matplotlib et pylab sont deux module très semblables pour les représentations graphiques (2d).

matplotlib est orienté objet, tandis que pylab est procédural. Les deux proposent une interface à la *Matlab*.

La galerie sur le site <http://matplotlib.sourceforge.net> est une fonctionnalité très simple pour aborder *matplotlib*. Il suffit de cliquer sur l'une des multiples figures proposées pour obtenir le code correspondant. Extrêmement efficace.

Il peut arriver que *matplotlib* se plaigne de ne pas posséder le « backend » requis et refuse d'afficher le graphique. Un backend est un processus qui gère les opérations graphiques en arrière plan.

Pour connaître la liste des backend reconnus (la documentation *matplotlib* ne semble pas à jour), on peut exécuter le code suivant dans une session *Python* :

```
1 >>> import matplotlib
2 >>> matplotlib.use('toto')
3 [...]
4 ValueError : Unrecognized backend string "toto" : valid strings
    are ['ps', 'Qt4Agg', 'GTK', 'GTKAgg', 'svg', 'agg', 'cairo', '
    MacOSX', 'GTKCairo', 'WXAgg', 'TkAgg', 'QtAgg', 'FltkAgg', 'pdf
    ', 'CocoaAgg', 'emf', 'gdk', 'template', 'WX']
```

Le plus sûr est d'utiliser le backend *tkAgg* ou *agg*.

8.5.2. SciTools, EasyViz. <http://code.google.com/p/scitools/SciTools> https://scitools.googlecode.com/hg/doc/easyviz/easyviz_sphinx_html/index.html

il s'agit d'un package proposant une interface *Python* unifiée pour divers packages de visualisation comme *Matplotlib*, *Gnuplot*, etc..

Les exemples du tutorial ne fonctionnent pas très bien.

8.5.3. Vtk. <http://www.vtk.org/Wiki/VTK/Examples/Python/Widgets/ContourWidget>

8.5.4. Mayavi. Attention, démarrer avec

```
$ ipython -wtrhead
```

<http://code.enthought.com/projects/mayavi/docs/development/html/mayavi/mlab.html>

8.5.5. Paraview. http://www.itk.org/Wiki/ParaView/Python_Scripting**Exercices****Exercice 80.** *Exercice***Exercice 81.** *Optimisation. Soit*

$$F(x, y, z) = \begin{pmatrix} x^2 - \frac{13}{4}x - \frac{y}{4} - \frac{z}{4} + \frac{3}{2} \\ y^2 - \frac{x}{2} + y - \frac{z}{2} - \frac{1}{2} \\ z^3 + 6z^2 + \frac{51}{4}z - \frac{x}{4} + \frac{y}{4} + 10 \end{pmatrix}$$

- (1) Trouver une solution $X^* = (x^*, y^*, z^*)$ de l'équation $F(x, y, z) = 0$ en partant de la valeur initiale $(x_0, y_0, z_0) = (0, 0, 0)$ ⁵. On peut utiliser le fait que, $\|\cdot\|_2$ désignant la norme euclidienne,

$$\begin{aligned} F(x^*, y^*, z^*) = (0, 0, 0) &\iff \|F(x^*, y^*, z^*)\|_2^2 = 0 \\ &\iff (x^*, y^*, z^*) \text{ est un minimum de } \|F(x, y, z)\|_2^2 \end{aligned}$$

- (2) Tracer la fonction $(x, y) \mapsto \Phi(x, y) = \|F(x, y, z^*)\|_2^2$ au voisinage de (x^*, y^*) .

Exercice 82. *Résolution d'une équation différentielle*

Résoudre sur $[0, 1]$ l'équation différentielle $y'' - 2(3 + 2t^3)y = 0$ avec les conditions initiales $y(0) = 0$ et $y'(0) = 1$

Mettre tout d'abord l'équation sous la forme $Y' = F(Y, t)$ avec des conditions initiales $Y(0) = Y_0$

Utiliser la fonction `scipy.integrate.odeint()`

Tracer la solution, comparer graphiquement avec la solution exacte $y(t) = te^{t^2}$

Calculer le gradient en Y de la fonction F et l'utiliser dans la résolution. Comparer les temps d'exécution pour obtenir une précision donnée

⁵Il n'y a pas unicité de la solution. Une solution possible est $(\frac{3}{2}, \frac{1}{2}, -2)$

Exercice 83. Résoudre l'équation différentielle :

$$y'' - ty = 0, t \in [0, 1]$$

avec les conditions initiales

$$y(0) = \frac{1}{3^{\frac{2}{3}} \Gamma(\frac{2}{3})}$$

$$y'(0) = -\frac{1}{3^{\frac{1}{3}} \Gamma(\frac{1}{3})}$$

Tracer la fonction $t \mapsto y(t)$

Exercice 84. Indexation multiple et performances.

Comparer en temps d'exécution les deux modes d'indexation multiple par tableau ou par liste :

```
1 A = arange(10)
2 il = [1, 2, 3]
3 ia = array([1, 2, 3])
4 A[il]
5 A[ia]
```

Exercice 85. cython

Considérons le code **Python** (fichier `integral.pyx`) :

```
1
2 def f(x) :
3     return x**2-x
4
5 def integrate_f(a, b, N) :
6     s = 0
7     dx = (b-a)/N
8     for i in range(N) :
9         s += f(a+i*dx)
10    return s * dx
```

- (1) Mettre en place un script **Python**, `intprof.py` pour profiler le code. (Voir 8.4.2 page 213)
- (2) Dupliquer le code pour définir `f1()` et `integrate_f1()`. Utiliser cython pour leur ajouter des déclarations de type de variables et paramètres, et profiler le code obtenu.
- (3) Ajouter une troisième version de `f` et `integrate_f`, dans laquelle la fonction `f` est typée par la déclaration `cdef double f2(double x)`. Comparer.

Exercice 86. *Intégration, vectorisation.*

Programmer la formule d'intégration approchée (trapèzes composée)

$$\int_a^b f(x) dx \simeq h \left(\frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{1 \leq k < n} f(a + ih) \right)$$

avec $h = \frac{b-a}{n}$

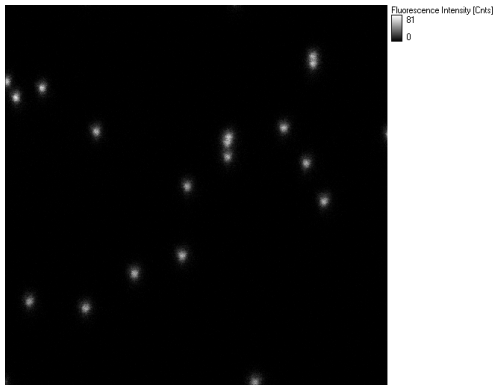
- (1) avec une boucle classique ;
- (2) en utilisant une vectorisation de la fonction f ;
- (3) Tracer à l'aide de Matplotlib, les deux courbes des temps d'exécution en fonction de n .

on utilisera les fonctions $f(x) = 1 + x$ et $g(x) = e^{-x^2} \ln(x + x \sin x)$

**Exercice 87.** *Ecrire une fonction $axpy(a, X, Y)$ calcule $aX + Y$ pour $X, Y \in \mathbb{R}^n$ et $a \in \mathbb{R}$ à l'aide d'une boucle*

Écrire une fonction analogue $vaxpy(a, X, Y)$ en utilisant la vectorisation dans NumPy.

Comparer les temps d'exécution.

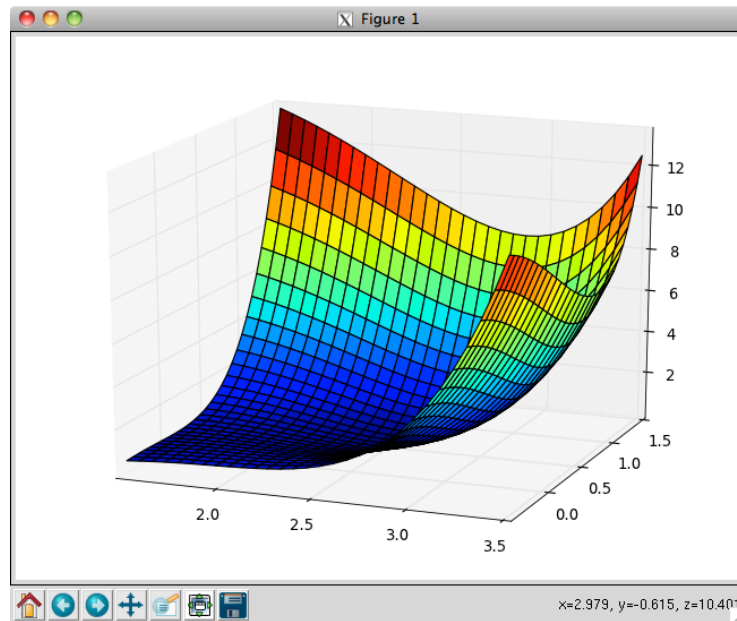
Exercice 88. *Traitement d'images : ajustement d'une gaussienne. TODO***Solutions des exercices****Solution 80** Solution**Solution 81** (1) Le script **Python** :

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  from scipy import array, zeros, meshgrid, optimize, r_
4  import matplotlib
5  matplotlib.use('TkAgg') #pour MacOSX
6  from pylab import figure, show
7  from mpl_toolkits.mplot3d import Axes3D
8  from matplotlib import cm
9
10 def F(xx) :
11     x, y, z = xx[0],xx[1],xx[2]
12     X = x*x -13*x/4.-y/4.-z/4.+3/2.
13     Y = y*y-x/2.+y-z/2.-1/2.
14     Z = z*z*z+6.*z*z+51*z/4.-x/4.+y/4.+10.
15     return X*X+Y*Y+Z*Z
16
17 def plot(F, x0, dx, r, z0) :
18     x0,y0 = x0[0], x0[1]
19     dx,dy = dx[0], dx[1]
20     x = r_[x0-dx :x0+dx :r*1j]
21     y = r_[y0-dy :y0+dy :r*1j]
22     x, y = meshgrid(x, y)
23     # ou bien en une ligne :
24     # x, y = mgrid[x0-dx :x0+dx :r*1j, y0-dy :y0+dy :r*1j]
25     z = F([x, y, z0])
26     fig = figure()
27     ax = Axes3D(fig)
28     ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.jet)
29     show()
30 if __name__=="__main__" :
31     A = optimize.fmin_cg(F, zeros(3))
32     print A
33     X0, z0 = (A[0], A[1]), A[2]
34     plot(F, X0, (1.,1.), 30, z0)

```

(2) Le tracé de la fonction $\phi(x, y) = \|F(x, y, -2)\|_2^2$ au voisinage de $(x^*, y^*) = (\frac{3}{2}, \frac{1}{2})$



Solution 82 Résolution d'une équation différentielle

```

1  #!/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import scipy as sp
4  from scipy.integrate import odeint
5  from scipy import array, r_
6
7  def F(Y,t) :
8      return array([2*(3+2*t*t*t)*Y[1],Y[0]])
9  def trace(N,T, s=None) :
10     import matplotlib
11     matplotlib.use('tkAgg')#pas obligatoire
12     from matplotlib import pyplot
13     fig = pyplot.figure()
14     pyplot.plot(N, T, '+r',N , s, '*b' )
15     pyplot.grid(True)
16     pyplot.show()
17
18  def solex(t) :
19     return t*sp.exp(t*t)
20
21  if __name__=="__main__" :
22     Y0 = array([1,0])
23     T = r_[0 :1 :100j]
24     Y = odeint(F,Y0,T)

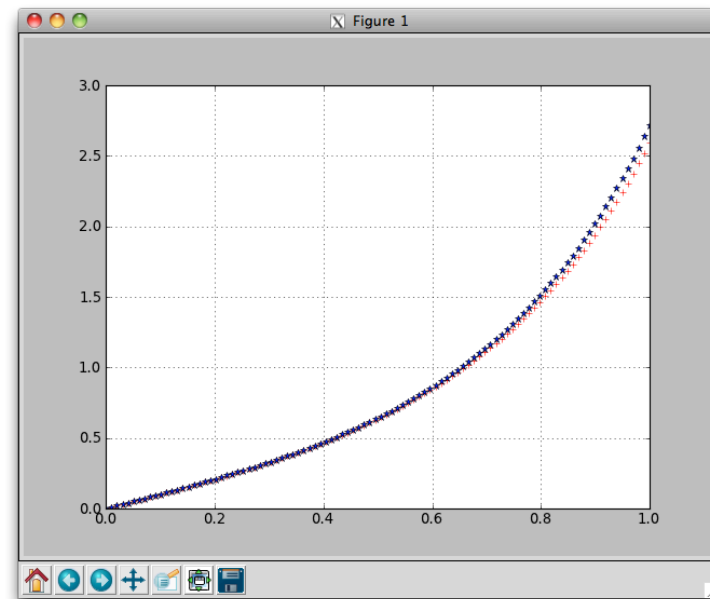
```

```

25     y = Y[ :,1]
26     s = solex(T)
27     trace(T,y,s)

```

Le tracé de la solution exacte et de la solution numérique :



Solution 83 Équation différentielle

Solution 84

```

1  #/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import cProfile
4  from numpy import arange
5  def tests(A,i,n=10000000) :
6      for k in range(n) : A[i]
7  A = arange(10000)
8  il = [1, 2, 3]
9  ia = array([1, 2, 3])
10 cProfile.run('tests(A,il)')
11 cProfile.run('tests(A,ia)')

```

et le résultat résumé :

```

4 function calls in 94.579 CPU seconds
4 function calls in 12.602 CPU seconds

```

L'indexation par array est 7,5 fois plus rapide que l'indexation par *liste*.

Solution 85 (1) Le profiler :

```

1  #/usr/bin/python
2  # -*- coding : utf-8 -*-
3  import pstats, cProfile
4  import pyximport
5  pyximport.install() #compilation auto de integral
6  import integral
7  pfile = 'integral.prof'
8  cProfile.run('integral.integrate_f0(0.0, 1.0, 1000000)', pfile)
9  s = pstats.Stats(pfile)
10 s.strip_dirs().sort_stats("time").print_stats()

```

(2) Le typage des variables et paramètres :

```

1  # -*- coding : utf-8 -*-
2  # cython : profile=True
3  # <ici le code Python pur>
4  #Typage variables et paramètres
5  def f1(double x) :
6      return x**2-x
7  def integrate_f1(double a, double b, int N) :
8      cdef int i
9      cdef double s, dx
10     s = 0
11     dx = (b-a)/N
12     for i in range(N) :
13         s += f1(a+i*dx)
14     return s * dx

```

(3) Le typage de la fonction

```

1  cdef double f2(double x) :
2      return x**2-x

```

Solution 86 Les fonctions test à intégrer

```

1  def f(x) : return 1+x
2  def g(x) : return exp(-x*x)*log(x+sin(x))

```

Intégration (méthode des Trapèzes composée), boucle simple :


```

1 def integ(f, a, b, n=100) :
2     h = float(b-a)/n
3     I = (f(a)+f(b))*0.5
4     for i in range(1,n) :
5         I += f(a+i*h)
6     return I*h

```

Intégration, fonction f vectorisée :

```

1 def v_integ(f, a, b, n=100) :
2     f = sp.vectorize(f)
3     h = float(b-a)/n
4     x = sp.linspace(a, b, n+1)
5     I = h*(sp.sum(f(x))-0.5*(f(a)+f(b)))
6     return I

```

Le calcul des temps d'exécution en fonction du nombre de points :

```

1 def temps(f, a=0, b=1, N=(100,10001,100)) :
2     import time
3     first, last, step = N
4     nbint = sp.arange(first, last, step)
5     times = sp.empty(len(nbint))
6     for k, n in enumerate(nbint) :
7         t = time.time()
8         integ(f,a,b,n)
9         times[k] = (time.time()-t)
10    return nbint, times

```

Le tracé des résultats :

```

1 def trace(N,T) :
2     """Tracés"""
3     import matplotlib
4     matplotlib.use('tkAgg')
5     from matplotlib import pyplot
6     fig = pyplot.figure()
7     pyplot.plot(N, T, 'r')
8     pyplot.grid(True)
9     pyplot.show()

```

Solution 87

```

1 def axpy(a,X,Y) :
2     Z = zeros(len(X))
3     for i in range(len(X)) :
4         Z[i] = a*X[i]+Y[i]
5     return Z
6
7 def vaxpy(a,X,Y) :
8     return a*X+Y
9
10 if __name__=="__main__" :
11     import cProfile
12     from scipy import random
13     X = random.random(100000)
14     Y = random.random(100000)
15     a = 2.0
16     cProfile.run ("axpy(a,X,Y)")
17     cProfile.run ("vaxpy(a,X,Y)")

```

À l'exécution on obtient :

```

7 function calls in 0.198 CPU seconds
3 function calls in 0.001 CPU seconds

```

La version vectorisée est 200 fois plus rapide.

Solution 88 Traitement d'images : ajustement d'une gaussienne

```

1 #!/usr/bin/python
2 # -*- coding : utf-8 -*-
3 import sys
4 import scipy as sp
5 from scipy import array, r_, where, exp, optimize, meshgrid
6 import matplotlib
7 matplotlib.use('TkAgg') #pour MacOSX
8 from pylab import figure, show
9 from mpl_toolkits.mplot3d import Axes3D
10 from matplotlib import cm
11
12 def readImage(filename) :
13     try : file = open(filename)
14     except IOError :
15         print "Impossible d'ouvrir le fichier %s"%filename

```

```

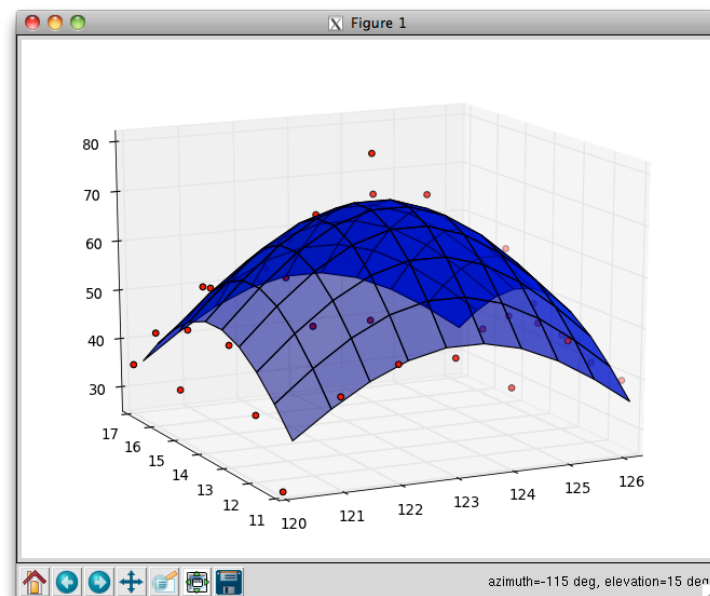
16         exit()
17     P = sp.loadtxt(filename, skiprows = 8)
18     return P
19
20 def support(pixels, i0, j0, n) :
21     return pixels[i0-n :i0+n+1,j0-n :j0+n+1]
22
23 def adjustGauss(pixels, i0, j0, n) :
24     supp = support(pixels, i0, j0, n)
25     sig0 = (pixels[i0,j0], n, n, 3.0, 0.0)
26     sig = optimize.fmin(phi, sig0, args=(supp,), full_output=0)
27     return sig
28
29 def gaussienne(X, sig) :
30     x,y = X[0],X[1]
31     g0, x0, y0, w, b = sig
32     return b + g0*exp(-(x-x0)**2/(2*w**2))*exp(-(y-y0)**2/(2*w**2))
33
34 def phi(sig, ph) :
35     g0, x0, y0, w, b = sig
36     s = 0
37     m,n = ph.shape
38     for i in range(m) :
39         for j in range(n) :
40             s += (ph[i,j]-gaussienne([i,j], sig))**2
41     return s
42
43 def plot(pixels, sig, i0, j0, n, r) :
44     g0, x0, y0, w, b = sig
45     fig = figure()
46     ax = Axes3D(fig)
47     x = r_[x0-n :x0+n :r*1j]
48     y = r_[y0-n :y0+n :r*1j]
49     x, y = meshgrid(x, y)
50     z = gaussienne([x,y], sig)
51     ax.plot_surface(i0-n+x, j0-n+y, z, rstride=1, cstride=1)#, cmap
        =cm.jet)
52
53     supp = support(pixels, i0, j0, n)
54     xdata = r_[i0-n :i0+n :(2*n+1)*1j]
55     ydata = r_[j0-n :j0+n :(2*n+1)*1j]
56

```

```

57 xdata,ydata = meshgrid(xdata, ydata)
58 # ax.plot_surface(xdata, ydata, supp, rstride=1, cstride=1, cmap=
    cm.jet)
59 # ax.plot_wireframe(xdata, ydata, supp)
60 for i in range(len(xdata)) :
61     ax.scatter(xdata[i, :], ydata[i, :], supp[i, :], c='r')
62 show()
63
64 if __name__=="__main__" :
65     try : filename = sys.argv[1]
66     except IndexError :
67         filename = 'srv_9.dat'
68     pixels = readImage(filename)
69
70     i0,j0 = sp.where(pixels == pixels.max())
71     n = 3
72
73     sig = adjustGauss(pixels, i0, j0, n)
74     r = 10
75     plot(pixels, sig, i0, j0, n, r)

```



Index

A

affectation multiple, 33, 59
alias, 87, 125
and, 62
argument, 119
ASCII, 91
attribut, 130, 154

B

bibliothèque standard, 128, 177
break, 59
buffering, 136
builtins, 130
bytecode, 128

C

cahier des charges, 153
camel case, 16
caractère, 37
chaîne de caractère, 36, 92
chaîne de documentation, 118
chemin, 178
complex, 81
concaténation, 37
constructeur, 154
continue, 59
convention de nommage, 15
conventions de programmation, 16
coupure du plan complexe, 189
court-circuit, 62
csv, 191
cython, 212

D

déballage, 86
 d'arguments, 121
 de tuple, 121
découpage, 87
data attribute, 155

descripteur de format, 99

dict

items(), 66
 keys(), 66
 values(), 66

dictionnaire, 41

dir(), 30

division entière, 35

divmod(), 35

docstring, 36, 118

doctest, 45

donnée, 155

dynamique, 13

dynamiquement typé, 13

E

effet de bord, 119

elif, 60

else, 59, 60

else (try-), 101

emballage, 86

empilage de tableaux, 204

ensembles, 99

Entrée-sortie, 42

enumerate(), 66

espace de noms, 123

except, 101

exception, 101

expression génératrice, 85

expression régulière, 191

F

False, 62

fichier module, 16

FIFO, 89

files d'attente, 89

filter(), 68

finally, 101

- float*, 81
- fonction-méthode, 156
- fonctions intégrées, 130
 - abs()*, 137
 - all()*, 132
 - any()*, 132
 - assert()*, 137
 - callable()*, 132
 - chr()*, 131
 - compile()*, 132
 - delattr()*, 130
 - dir()*, 30, 137
 - divmod()*, 35, 137
 - enumerate()*, 66, 134
 - eval()*, 132
 - exec()*, 133
 - filter()*, 68, 134
 - getattr()*, 130
 - globals()*, 133
 - hasattr()*, 130
 - hash()*, 137
 - help()*, 30
 - hex()*, 131
 - id()*, 79, 137
 - input()*, 50, 135
 - isinstance()*, 137
 - issubclass()*, 137
 - len()*, 137
 - map()*, 67, 134
 - max()*, 134
 - min()*, 135
 - next()*, 135
 - oct()*, 131
 - open()*, 42, 135
 - ord()*, 131
 - pow()*, 137
 - quit()*, 137
 - range()*, 50, 65, 135
 - raw_input()*, 135
 - reduce()*, 71
 - reload()*, 126
 - repr()*, 98
 - reversed()*, 67, 135
 - round()*, 137
 - setattr()*, 130
 - sorted()*, 67, 135
 - str()*, 98
 - type()*, 32, 137
 - zip()*, 66, 135
- for*, 61
- from*, 125
- G**
 - générateur, 82, 84
 - garbage collector, 13
 - getter, 162
- H**
 - héritage, 157
 - help()*, 30
 - historique, 12
- I**
 - if*, 60
 - immuable, 79
 - immutable*, 79
 - import*, 125
 - in*, 62
 - indenté, 59
 - indentation, 59
 - index, 100
 - input()*, 50
 - instance, 153, 155
 - instantiation de classe, 155
 - instructions
 - affectation, 80
 - break*, 59
 - continue*, 59
 - def*, 44
 - del*, 41, 90
 - else*, 59
 - else(try-)*, 101
 - except*, 101
 - finally*, 101
 - for*, 61
 - from*, 125
 - if*, 60
 - import*, 125
 - is*, 40
 - pass*, 59
 - return*, 44
 - try*, 101
 - while*, 59
 - int*, 81
 - inter-opérable, 13
 - introspectif, 13
 - is*, 40, 62
 - is not*, 62
 - ISO 8859, 91
 - itérable, 61, 82

itérateur, 61, 82, 83
inverse, 135

J

jockey, 181
Jython, 14

L

lambda, 122
LGI, 124
LIFO, 89
ligne de commande, 16
List comprehension, 68
liste, 88
Listes en comprehension, 68
locals(), 133
logiciel libre, 13

M

mémoire tampon, 136
méthode, 155
spéciale, 154
virtuelle, 13, 159
map(), 67
method, 155
MIME, 191
modifiables, 79
module, 125
fichier, 16
standard, 128
modules
cProfile, 185
doctest, 45
math, 33
os, 177
pickle, 42, 43, 51
profile, 185
pstats, 185
string, 95
timeit, 184
unittest, 187
mots réservés, 79
multi-thread, 13
mutable, 79

N

name space, 123
nombre parfait, 50
nombres de Fibonacci, 118
nombres parfaits, 50
non modifiables, 79

Numpy, 197

O

objet, 31
classe, 155
instances, 155
méthodes, 155
opérateur crochet, 37
opérations, 35
or, 62
orthogonal, 13

P

packing, 86
paquetages, 127
pass, 59
passage d'arguments
par affectation, 119
par référence, 119
par valeur, 119
path, 178
persistance, 44
pickle, 43, 51
pile, 89
plan Unicode, 90
point de code, 90
pointeurs, 13
portée, 124
globale, 124
interne, 124
locale, 124
profilage, 128, 143
profiling, 128
programmation dirigée par la documentation, 46
Programmation Orientée Objets, 45
prompt, 16
properties, 162
Python Software Foundation, 13

R

références, 155
réflectif, 13
réutilisabilité, 45
range(), 50, 65
règle *LGI*, 124
reduce(), 71
regular expression, 191
reverse iterator, 135
reversed(), 67
richesse lexicale, 106

S

séquence, 85
scope, 124
scripts *Python* , 16
setter, 162
slices, 38
slicing
 séquences, 87
 tableaux, 201
sorted(), 67
str, 36
surcharge, 13

T

table de symboles, 118
tabulations, 16
timeit, 184
tranches, 38
True, 62
try, 101
tuple, 85
 unpacking, 121
type(), 32
types
 complex, 81
 dict, 41, 100
 float, 81
 frozenset, 99
 int, 81
 list, 39, 85, 88
 set, 99
 str, 36
 str, bytes, bytearray, 90
 tuple, 85

U

Unicode, 90
unittest, 187
unpacking, 86
utf-8, 91

V

valeur d'indentation, 16
variable, 79
variables, 32
variables privées, 13
vue, 101

W

while, 59

X

xml.dom, 191
xml.sax, 191

Z

zip(), 66

Bibliographie

- [LA] Mark Lutz, David Ascher, Introduction à *Python*, ed O'Reilly, Janvier 2000, ISBN 2-84177-089-3
- [ML] Mark Lutz, Programming *Python*, Troisième Edition, ed O'Reilly & Associates, Août 2006, ISBN 0596009259 10
- [HPL] Hans Petter Langtangen, *Python* Scripting for Computational Science. Simula Research Laboratory and Department of Informatics University of Oslo. /home/puiseux/enseignement/CoursOutilsInformatiques/Python_scripting_for_computational_science.04.pdf
- [MP] Mark Pilgrim, dive into *Python* (Plongez au coeur de *Python*) <http://diveintopython.org/>, traduction française : Xavier Defrang, Jean-Pierre Gay, Alexandre Drahon. </home/puiseux/doc/python/frdiveintopython.pdf>
- [VR] Guido van Rossum, Tutoriel *Python*, Release 2.0.1. </home/puiseux/doc/python/tut-fr.pdf>
- [GS-1] Gérard Swinnen, Apprendre à programmer avec *Python*, http://inforef.be/swi/download/apprendre_python.pdf
- [GS-2] Gérard Swinnen, Apprendre à programmer avec *Python* 3, Éd. Eyrolles, 2010, ISBN 978-2-212-12708-9.
- [EDM] Jeffrey Elkner, Allen B. Downey and Chris Meyers, <http://openbookproject.net//thinkCSpy>
- [WK] [http://fr.wikipedia.org/wiki/Python_\(langage\)](http://fr.wikipedia.org/wiki/Python_(langage))
- [PLR] *Python* Library Reference. <http://docs.python.org/library/index.html>
- [TZ] Tarek Ziadé, *Python* Petit guide à l'usage du développeur agile, Éd. Dunos, collection études et développements, 2007, ISBN 978-2-10-050883-9. 187
- [MB] Matthieu Brucher, *Python* Les fondamentaux du langage, La programmation pour les scientifiques, Éd. eni, Collection Ressources Informatique, Janvier 2008, ISBN 978-2-7460-4088-5.