Python Frequently Asked Questions

Version 2.7.18

Guido van Rossum
and the Python development team

mai 20, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

CHAPITRE 1

FAQ générale sur Python

1.1 Informations générales

1.1.1 Qu'est-ce que Python ?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable : it runs on many Unix variants, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

Pour en savoir plus, commencez par tutorial-index. Le « Guide des Débutants pour Python » renvoie vers d'autres tutoriels et ressources d'initiation pour apprendre Python.

1.1.2 Qu'est ce que la Python Software Foundation ?

La Python Software Foundation (PSF) est une organisation indépendante à but non lucratif qui détient les droits d'auteur sur les versions Python 2.1 et plus récentes. La mission de la PSF est de faire progresser les technologies ouvertes (*open source*) relatives au langage de programmation Python et de promouvoir son utilisation. Le page d'accueil de la PSF se trouve à l'adresse suivante : https://www.python.org/psf/.

Si vous utilisez Python et que vous le trouvez utile, merci de contribuer par le biais de la page de donation de la PSF.

1.1.3 Existe-il des restrictions liées à la propriété intellectuelle quant à l'utilisation de Python ?

Vous pouvez faire ce que vous souhaitez avec la source, tant que vous respecterez la licence d'utilisation et que vous l'afficherez dans toute documentation que vous produirez au sujet de Python. Si vous respectez ces règles, vous pouvez utiliser Python dans un cadre commercial, vendre le code source ou la forme binaire (modifiée ou non), ou vendre des produits qui incorporent Python sous une forme quelconque. Bien entendu, nous souhaiterions avoir connaissance de tous les projets commerciaux utilisant Python.

Voir la page de licence d'utilisation de la PSF pour trouver davantage d'informations et un lien vers la version intégrale de la licence d'utilisation.

Le logo de Python est une marque déposée, et dans certains cas une autorisation est nécessaire pour l'utiliser. Consultez la politique d'utilisation de la marque pour plus d'informations.

1.1.4 Pourquoi Python a été créé ?

Voici un *très* bref résumé de comment tout a commencé, écrit par Guido van Rossum (puis traduit en français) :

> J'avais une expérience complète avec la mise en œuvre du langage interprété ABC au sein du CWI, et en travaillant dans ce groupe j'ai appris beaucoup à propos de la conception de langage. C'est l'origine de nombreuses fonctionnalités de Python, notamment l'utilisation de l'indentation pour le groupement et l'inclusion de types de très haut niveau (bien que dans les détails ils soient tous différents dans Python).
>
> J'avais un certain nombre de différends avec le langage ABC, mais j'aimais aussi beaucoup de ses fonctionnalités. Il était impossible d'étendre le langage ABC (ou ses implémentations) pour remédier à mes réclamations – en vérité le manque d'extensibilité était l'un des plus gros problème. J'avais un peu d'expérience avec l'utilisation de Modula-2+ et j'en ai parlé avec les concepteurs de Modula-3 et j'ai lu le rapport sur Modula-3. Modula-3 est à l'origine de la syntaxe et de la sémantique utilisée pour les exceptions, et quelques autres fonctionnalités en Python.
>
> Je travaillais sur un groupe de systèmes d'exploitation distribués Amoeba au CWI. Nous avions besoin d'un meilleur moyen pour gérer l'administration système qu'écrire un programme en C ou en script Bourne shell, puisque l'Amoeba avait sa propre interface d'appels système qui n'était pas facilement accessible depuis les scripts Bourne shell. Mon expérience avec le traitement des erreurs dans l'Amoeba m'a vraiment fait prendre conscience de l'importance des exceptions en tant que fonctionnalité d'un langage de programmation.
>
> Il m'est venu à l'esprit qu'un langage de script avec une syntaxe comme ABC mais avec un accès aux appels systèmes d'Amoeba remplirait les besoins. J'ai réalisé que ce serait idiot d'écrire un langage spécifique à Amoeba, donc j'ai décidé que j'avais besoin d'un langage qui serait généralement extensible.
>
> Pendant les vacances de Noël 1989, j'avais beaucoup de temps à disposition, donc j'ai décidé de faire un essai. Durant l'année suivante, j'ai encore beaucoup travaillé dessus sur mon propre temps. Python a été utilisé dans le projet Amoeba avec un succès croissant, et les retours de mes collègues m'ont permis d'ajouter beaucoup des premières améliorations.
>
> En Février 1991, juste après un peu plus d'un an de développement, j'ai décidé de le poster sur USENET. Le reste se trouve dans le fichier « Misc/HISTORY ».

1.1.5 Pour quoi Python est-il fait ?

Python est un langage de programmation haut niveau généraliste qui peut être utilisé pour pallier à différents problèmes.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for library-index to get an idea of what's available. A wide variety of third-party extensions are also available. Consult the Python Package Index to find packages of interest to you.

1.1.6 Comment fonctionne le numérotage des versions de Python ?

Les versions de Python sont numérotées A.B.C ou A.B. A est une version majeure – elle est augmentée seulement lorsqu'il y a des changements conséquents dans le langage. B est une version mineure, elle est augmentée lors de changements de moindre importance. C est un micro-niveau – elle est augmentée à chaque sortie de correctifs de bogue.

Toutes les sorties ne concernent pas la correction de bogues. A l'approche de la sortie d'une nouvelle version majeure, une série de versions de développement sont créées, dénommées *alpha*, *beta*, *release candidate*. Les alphas sont des versions primaires dans lesquelles les interfaces ne sont pas encore finalisées ; ce n'est pas inattendu de voir des changements d'interface entre deux versions alpha. Les *betas* sont plus stables, préservent les interfaces existantes mais peuvent ajouter de nouveaux modules, les *release candidate* sont figées, elles ne font aucun changement à l'exception de ceux nécessaires pour corriger des bogues critiques.

Les versions *alpha*, *beta* et *release candidate* ont un suffixe supplémentaire. Le suffixe pour une version alpha est « aN » où N est un petit nombre, le suffixe pour une version *beta* est *bN* où N est un petit nombre, et le suffixe pour une *release candidate* est « cN » où N est un petit nombre. En d'autres mots, toutes les versions nommées *2.0.aN* précèdent les versions *2.0.bN*, qui elles-mêmes précèdent 2.0cN, et *celles-ci* précèdent la version 2.0.

Vous pouvez aussi trouver des versions avec un signe « + » en suffixe, par exemple « 2.2+». Ces versions sont non distribuées, construites directement depuis le dépôt de développement de CPython. En pratique, après la sortie finale d'une version mineure, la version est augmentée à la prochaine version mineure, qui devient la version *a0*, c'est-à-dire *2.4a0*.

Voir aussi la documentation pour for `sys.version`, `sys.hexversion`, et `sys.version_info`.

1.1.7 Comment obtenir une copie du code source de Python ?

The latest Python source distribution is always available from python.org, at https://www.python.org/downloads/. The latest development sources can be obtained at https://github.com/python/cpython/.

Le code source est dans une archive *gzippée* au format *tar*, elle contient le code source C complet, la documentation formatée avec Sphinx, les libraires Python, des exemples de programmes, et plusieurs morceaux de code utiles distribuables librement. Le code source sera compilé et prêt à fonctionner immédiatement sur la plupart des plateformes UNIX.

Consult the Getting Started section of the Python Developer's Guide for more information on getting the source code and compiling it.

1.1.8 Comment obtenir la documentation de Python ?

La documentation standard pour la version stable actuelle est disponible à https://docs.python.org/3/. Des versions aux formats PDF, texte et HTML sont aussi disponibles à https://docs.python.org/3/download.html.

La documentation est écrite au format *reStructuredText* et traitée par l'outil de documentation Sphinx <http://sphinx-doc.org/>'__. La source du *reStructuredText* pour la documentation constitue une partie des sources de Python.

1.1.9 Je n'ai jamais programmé avant. Existe t-il un tutoriel Python ?

Il y a de nombreux tutoriels et livres disponibles. La documentation standard inclut tutorial-index.

Consultez le Guide du Débutant afin de trouver des informations pour les développeurs Python débutants, incluant une liste de tutoriels.

1.1.10 Y a-t-il un forum ou une liste de diffusion dédié à Python ?

Il y a un forum, `comp.lang.python` et une liste de diffusion, python-list. Le forum et la liste de diffusion sont des passerelles l'un vers l'autre – si vous pouvez lire les *news* ce n'est pas inutile de souscrire à la liste de diffusion. `comp.lang.python` a beaucoup d'activité, il reçoit des centaines de messages chaque jour, et les lecteurs du réseau Usenet sont souvent plus capables de faire face à ce volume.

Les annonces pour les nouvelles versions et événements peuvent êtres trouvées dans *comp.lang.python.announce*, une liste diminuée peu active qui reçoit environ 5 messages par jour. C'est disponible à liste de diffusion des annonces Python.

Plus d'informations à propos des autres listes de diffusion et forums peuvent être trouvées à https://www.python.org/community/lists/.

1.1.11 Comment obtenir une version bêta test de Python ?

Les versions alpha et bêta sont disponibles depuis https://www.python.org/downloads/. Toutes les versions sont annoncées sur les *newsgroups comp.lang.python* et *comp.lang.python.announce* ainsi que sur la page d'accueil de Python à https://www.python.org/ ; un flux RSS d'actualités y est aussi disponible.

You can also access the development version of Python through Git. See The Python Developer's Guide for details.

1.1.12 Comment soumettre un rapport de bogues ou un correctif pour Python ?

Pour reporter un bogue ou soumettre un correctif, merci d'utiliser https://bugs.python.org/.

Vous devez avoir un compte Roundup pour reporter des bogues ; cela nous permet de vous contacter si nous avons des questions complémentaires. Cela permettra aussi le suivi de traitement de votre bogue. Si vous avez auparavant utilisé SourceForge pour reporter des bogues sur Python, vous pouvez obtenir un mot de passe Roundup grâce à la procédure de réinitialisation de mot de passe de Roundup.

For more information on how Python is developed, consult the Python Developer's Guide.

1.1.13 Existe-t-il des articles publiés au sujet de Python auxquels je peux me référer ?

C'est probablement mieux de vous référer à votre livre favori à propos de Python.

Le tout premier article à propos de Python a été écrit en 1991 et est maintenant obsolète.

> Guido van Rossum and Jelke de Boer, « Interactively Testing Remote Servers Using the Python Programming Language », CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283–303.

1.1.14 Y a-t-il des livres au sujet de Python ?

Oui, il y en a beaucoup, et d'autres sont en cours de publication. Voir le wiki python à https://wiki.python.org/moin/PythonBooks pour avoir une liste.

Vous pouvez aussi chercher chez les revendeurs de livres en ligne avec le terme « Python » et éliminer les références concernant les Monty Python, ou peut-être faire une recherche avec les termes « langage » et « Python ».

1.1.15 Où www.python.org est-il localisé dans le monde ?

The Python project's infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details here.

1.1.16 Pourquoi le nom Python ?

Quand il a commencé à implémenter Python, Guido van Rossum a aussi lu le script publié par « Monty Python's Flying Circus », une série comique des années 1970 diffusée par la BBC. Van Rossum a pensé qu'il avait besoin d'un nom court, unique, et un peu mystérieux, donc il a décidé de l'appeler le langage Python.

1.1.17 Dois-je aimer « Monty Python's Flying Circus » ?

Non, mais ça peut aider. :)

1.2 Python c'est le monde réel

1.2.1 Quel est le niveau de stabilité de Python ?

Très stable. Les versions stables sont sorties environ tous les 6 à 18 mois depuis 1991, et il semble probable que ça continue. Actuellement il y a habituellement environ 18 mois entre deux sorties de version majeure.

The developers issue « bugfix » releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability ; only fixes for known problems are included in a bugfix release, and it's guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the Python download page. There are two production-ready versions of Python : 2.x and 3.x. The recommended version is 3.x, which is supported by most widely used libraries. Although 2.x is still widely used, it will not be maintained after January 1, 2020.

1.2.2 Combien de personnes utilisent Python ?

Il y a probablement des dizaines de milliers d'utilisateurs, cependant c'est difficile d'obtenir un nombre exact.

Python est disponible en téléchargement gratuit, donc il n'y a pas de chiffres de ventes, il est disponible depuis de nombreux sites différents et il est inclus avec de beaucoup de distributions Linux, donc les statistiques de téléchargement ne donnent pas la totalité non plus.

Le forum *comp.lang.python* est très actif, mais tous les utilisateurs de Python ne laissent pas de messages dessus ou même ne le lisent pas.

1.2.3 Y a-t-il un nombre de projets significatif réalisés en Python ?

Voir https://www.python.org/about/success pour avoir une liste des projets qui utilisent Python. En consultant les comptes-rendu des conférences Python précédentes il s'avère que les contributions proviennent de nombreux organismes et entreprises divers.

Les projets Python à grande visibilité incluent Mailman mailing list manager et l'application serveur Zope. Plusieurs distributions Linux, notamment Red Hat, qui a écrit tout ou partie de son installateur et de son logiciel d'administration système en Python. Les entreprises qui utilisent Python en interne comprennent Google, Yahoo, et Lucasfilm Ltd.

1.2.4 Quelles sont les nouveautés en développement attendues pour Python ?

Regardez https://www.python.org/dev/peps/ pour Python Enhancement Proposals (PEPs). PEPs sont des documents techniques qui décrivent une nouvelle fonctionnalité qui a été suggérée pour Python, en fournissant une spécification technique concise et logique. Recherchez une PEP intitulée « Python X.Y Release Schedule », où X.Y est la version qui n'a pas encore été publiée.

Le nouveau développement est discuté sur la liste de diffusion python-dev.

1.2.5 Est-il raisonnable de proposer des changements incompatibles dans Python ?

En général, non. Il y a déjà des millions de lignes de code de Python tout autour du monde, donc n'importe quel changement dans le langage qui rend invalide ne serait-ce qu'une très petite fraction du code de programmes existants doit être désapprouvé. Même si vous pouvez fournir un programme de conversion, il y a toujours des problèmes de mise à jour dans toutes les documentations, beaucoup de livres ont été écrits au sujet de Python, et nous ne voulons pas les rendre invalides soudainement.

En fournissant un rythme de mise à jour progressif qui est obligatoire si une fonctionnalité doit être changée.

1.2.6 Existe-t-il un meilleur langage de programmation pour les programmeurs débutants ?

Oui.

Il reste commun pour les étudiants de commencer avec un langage procédural et à typage statique comme le Pascal, le C, ou un sous-ensemble du C++ ou Java. Les étudiants pourraient être mieux servis en apprenant Python comme premier langage. Python a une syntaxe très simple et cohérente ainsi qu'une vaste libraire standard, plus important encore, utiliser Python dans les cours d'initiation à la programmation permet aux étudiants de se concentrer sur les compétences de programmation les cruciales comme les problèmes de découpage et d'architecture. Avec Python, les étudiants peuvent rapidement aborder des concepts fondamentaux comme les boucles et les procédures. Ils peuvent même probablement travailler avec des objets définis dans leurs premiers cours.

Pour un étudiant qui n'a jamais programmé avant, utiliser un langage à typage statique peut sembler contre-nature. Cela représente une complexité additionnelle que l'étudiant doit maîtriser ce qui ralentit le cours. Les étudiants essaient d'apprendre à penser comme un ordinateur, décomposer les problèmes, établir une architecture propre, et résumer les données. Apprendre à utiliser un langage typé statiquement est important sur le long terme, ce n'est pas nécessairement la meilleure idée pour s'adresser aux étudiants durant leur tout premier cours.

De nombreux autres aspects de Python en font un bon premier langage. Comme Java, Python a une large bibliothèque standard donc les étudiants peuvent être assigner à la programmation de projets très tôt dans leur apprentissage qui *fait* quelque chose. Les missions ne sont pas restreintes aux quatre fonction standards. En utilisant la bibliothèque standard, les étudiants peuvent ressentir de la satisfaction en travaillant sur des applications réalistes alors qu'ils apprennent les fondamentaux de la programmation. Utiliser la bibliothèque standard apprend aussi aux étudiants la réutilisation de code. Les modules tiers tels que PyGame sont aussi très utiles pour étendre les compétences des étudiants.

L'interpréteur interactif de Python permet aux étudiants de tester les fonctionnalités du langage pendant qu'ils programment. Ils peuvent garder une fenêtre avec l'interpréteur en fonctionnement pendant qu'ils rentrent la source de leur programme dans une autre fenêtre. S'ils ne peuvent pas se souvenir des méthodes pour une liste, ils peuvent faire quelque chose comme ça :

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -- append object to end

>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as he's programming.

Il y a aussi de bons environnements de développement intégrés (EDIs) pour Python. IDLE est un EDI multiplateforme pour Python qui est écrit en Python en utilisant Tkinter. *PythonWin* est un IDE spécifique à Windows. Les utilisateurs d'Emcs seront heureux d'apprendre qu'il y a un très bon mode Python pour Emacs. Tous ces environnements de développement intégrés fournissent la coloration syntaxique, l'auto-indentation, et l'accès à l'interpréteur interactif durant le codage. Consultez le wiki Python pour une liste complète des environnements de développement intégrés.

Si vous voulez discuter de l'usage de Python dans l'éducation, vous devriez intéressé pour rejoindre la liste de diffusion pour l'enseignement.

1.3 Upgrading Python

1.3.1 What is this bsddb185 module my application keeps complaining about ?

Starting with Python2.3, the distribution includes the *PyBSDDB package <http ://pybsddb.sf.net/>* as a replacement for the old bsddb module. It includes functions which provide backward compatibility at the API level, but requires a newer version of the underlying Berkeley DB library. Files created with the older bsddb module can't be opened directly using the new module.

Using your old version of Python and a pair of scripts which are part of Python 2.3 (db2pickle.py and pickle2db.py, in the Tools/scripts directory) you can convert your old database files to the new format. Using your old Python version, run the db2pickle.py script to convert it to a pickle, e.g. :

```
python2.2 <pathto>/db2pickley.py database.db database.pck
```

Rename your database file :

```
mv database.db olddatabase.db
```

Now convert the pickle file to a new format database :

```
python <pathto>/pickle2db.py database.db database.pck
```

The precise commands you use will vary depending on the particulars of your installation. For full details about operation of these two scripts check the doc string at the start of each one.

CHAPITRE 2

FAQ de programmation

2.1 Questions générales

2.1.1 Existe-t'il un débogueur de code source avec points d'arrêts, exécution pas-à-pas, etc. ?

Oui.

Le module pdb est un débogueur console simple, mais parfaitement adapté à Python. Il fait partie de la bibliothèque standard de Python, sa documentation se trouve dans le `manuel de référence`. Vous pouvez vous inspirer du code de pdb pour écrire votre propre débogueur.

L'environnement de développement interactif IDLE, qui est fourni avec la distribution standard de Python (normalement disponible dans Tools/scripts/idle) contient un débogueur graphique.

PythonWin est un environnement de développement intégré (EDI) Python qui embarque un débogueur graphique basé sur pdb. Le débogueur PythonWin colore les points d'arrêts et possède quelques fonctionnalités sympathiques, comme la possibilité de déboguer des programmes développés sans PythonWin. PythonWin est disponible dans le projet Extensions Python pour Windows et fait partie de la distribution ActivePython (voir https://www.activestate.com/activepython).

Boa Constructor est un EDI et un constructeur d'interface homme-machine basé sur wxWidgets. Il propose la création et la manipulation de fenêtres, un inspecteur d'objets, de nombreuses façons de visualiser des sources comme un navigateur d'objets, les hiérarchies d'héritage, la documentation html générée par les docstrings, un débogueur avancé, une aide intégrée et la prise en charge de Zope.

Eric est un EDI basé sur PyQt et l'outil d'édition Scintilla.

Pydb est une version du débogueur standard Python pdb, modifié pour être utilisé avec DDD (Data Display Debugger), un célèbre débogueur graphique. Pydb est disponible sur http://bashdb.sourceforge.net/pydb/ et DDD est disponible sur https://www.gnu.org/software/ddd.

Il existe de nombreux EDI Python propriétaires qui embarquent un débogueur graphique. Notamment :

- Wing IDE (https://wingware.com/)
- Komodo IDE (https://komodoide.com/)
- PyCharm (https://www.jetbrains.com/pycharm/)

2.1.2 Existe-t'il des outils pour aider à trouver des bogues ou faire de l'analyse statique de code ?

Oui.

PyChecker est un outil d'analyse statique qui trouve les bogues dans le code source Python et émet des avertissements relatifs à la complexité et au style du code. PyChecker est disponible sur http://pychecker.sourceforge.net/.

Pylint <https://www.pylint.org/>‘_ est un autre outil qui vérifie si un module satisfait aux normes de développement, et qui permet en plus d'écrire des greffons pour ajouter des fonctionnalités personnalisées. En plus de la vérification des bogues effectuée par PyChecker, Pylint effectue quelques vérifications supplémentaires comme la longueur des lignes, les conventions de nommage des variables, que les interfaces déclarées sont implémentées en totalité, et plus encore. https://docs.pylint.org/ fournit la liste complète des fonctionnalités de Pylint.

2.1.3 Comment créer un binaire autonome à partir d'un script Python ?

Pour créer un programme autonome, c'est-à-dire un programme que n'importe qui peut télécharger et exécuter sans avoir à installer une distribution Python au préalable, il n'est pas nécessaire de compiler du code Python en code C. Il existe en effet plusieurs outils qui déterminent les modules requis par un programme et lient ces modules avec un binaire Python pour produire un seul exécutable.

Un de ces outils est freeze, qui se trouve dans `Tools/freeze` de l'arborescence des sources de Python. Il convertit le code intermédiaire (*bytecode*) Python en tableaux C ; un compilateur C permet d'intégrer tous vos modules dans un nouveau programme, qui est ensuite lié aux modules standards Python.

Il fonctionne en cherchant de manière récursive les instructions d'import (sous les deux formes) dans le code source et en recherchant ces modules dans le chemin Python standard ainsi que dans le répertoire source (pour les modules natifs). Il transforme ensuite le code intermédiaire des modules écrits en Python en code C (des tableaux pré-remplis qui peuvent être transformés en objets code à l'aide du module *marshal*) et crée un fichier de configuration personnalisé qui contient uniquement les modules natifs qui sont réellement utilisés dans le programme. Il compile ensuite le code C généré et le lie au reste de l'interpréteur Python pour former un binaire autonome qui fait exactement la même chose que le script.

Bien évidemment, freeze nécessite un compilateur C. Il existe d'autres outils qui peuvent s'en passer. Un de ceux-ci est py2exe de Thomas Heller (pour Windows uniquement) disponible sur

http://www.py2exe.org/

Another tool is Anthony Tuininga's cx_Freeze.

2.1.4 Existe-t'il des normes de développement ou un guide de style pour écrire des programmes Python ?

Oui. Le style de développement que les modules de la bibliothèque standard doivent obligatoirement respecter est documenté dans la **PEP 8**.

2.1.5 Mon programme est trop lent. Comment l'accélérer ?

That's a tough one, in general. There are many tricks to speed up Python code ; consider rewriting parts in C as a last resort.

In some cases it's possible to automatically translate Python to C or x86 assembly language, meaning that you don't have to modify your code to gain increased speed.

Pyrex can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms.

Psyco is a just-in-time compiler that translates Python code into x86 assembly language. If you can use it, Psyco can provide dramatic speedups for critical functions.

The rest of this answer will discuss various tricks for squeezing a bit more speed out of Python code. *Never* apply any optimization tricks unless you know you need them, after profiling has indicated that a particular function is the heavily executed hot spot in the code. Optimizations almost always make the code less clear, and you shouldn't pay the costs of reduced clarity (increased development time, greater likelihood of bugs) unless the resulting performance benefit is worth it.

There is a page on the wiki devoted to performance tips.

Guido van Rossum has written up an anecdote related to optimization at https://www.python.org/doc/essays/list2str.

One thing to notice is that function and (especially) method calls are rather expensive ; if you have designed a purely OO interface with lots of tiny functions that don't do much more than get or set an instance variable or call another method, you might consider using a more direct way such as directly accessing instance variables. Also see the standard module `profile` which makes it possible to find out where your program is spending most of its time (if you have some patience – the profiling itself can slow your program down by an order of magnitude).

Remember that many standard optimization heuristics you may know from other programming experience may well apply to Python. For example it may be faster to send output to output devices using larger writes rather than smaller ones in order to reduce the overhead of kernel system calls. Thus CGI scripts that write all output in « one shot » may be faster than those that write lots of small pieces of output.

Also, be sure to use Python's core features where appropriate. For example, slicing allows programs to chop up lists and other sequence objects in a single tick of the interpreter's mainloop using highly optimized C implementations. Thus to get the same effect as :

```
L2 = []
for i in range(3):
    L2.append(L1[i])
```

it is much shorter and far faster to use

```
L2 = list(L1[:3])  # "list" is redundant if L1 is a list.
```

Note that the functionally-oriented built-in functions such as `map()`, `zip()`, and friends can be a convenient accelerator for loops that perform a single task. For example to pair the elements of two lists together :

```
>>> zip([1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

or to compute a number of sines :

```
>>> map(math.sin, (1, 2, 3, 4))
[0.841470984808, 0.909297426826, 0.14112000806, -0.756802495308]
```

The operation completes very quickly in such cases.

Other examples include the `join()` and `split()` methods of string objects. For example if s1..s7 are large (10K+) strings then `"".join([s1,s2,s3,s4,s5,s6,s7])` may be far faster than the more obvious `s1+s2+s3+s4+s5+s6+s7`, since the « summation » will compute many subexpressions, whereas `join()` does all the copying in one pass. For manipulating strings, use the `replace()` and the `format()` methods on string objects. Use regular expressions only when you're not dealing with constant string patterns. You may still use the old % operations `string % tuple` and `string % dictionary`.

Be sure to use the `list.sort()` built-in method to do sorting, and see the sorting mini-HOWTO for examples of moderately advanced usage. `list.sort()` beats other techniques for sorting in all but the most extreme circumstances.

Another common trick is to « push loops into functions or methods. » For example suppose you have a program that runs slowly and you use the profiler to determine that a Python function `ff()` is being called lots of times. If you notice that `ff()` :

```
def ff(x):
    ... # do something with x computing result...
    return result
```

tends to be called in loops like :

```
list = map(ff, oldlist)
```

ou :

```
for x in sequence:
    value = ff(x)
    ... # do something with value...
```

then you can often eliminate function call overhead by rewriting `ff()` to :

```
def ffseq(seq):
    resultseq = []
    for x in seq:
        ... # do something with x computing result...
        resultseq.append(result)
    return resultseq
```

and rewrite the two examples to `list = ffseq(oldlist)` and to :

```
for value in ffseq(sequence):
    ... # do something with value...
```

Single calls to `ff(x)` translate to `ffseq([x])[0]` with little penalty. Of course this technique is not always appropriate and there are other variants which you can figure out.

You can gain some performance by explicitly storing the results of a function or method lookup into a local variable. A loop like :

```
for key in token:
    dict[key] = dict.get(key, 0) + 1
```

resolves `dict.get` every iteration. If the method isn't going to change, a slightly faster implementation is :

```
dict_get = dict.get  # look up the method once
for key in token:
    dict[key] = dict_get(key, 0) + 1
```

Default arguments can be used to determine values once, at compile time instead of at run time. This can only be done for functions or objects which will not be changed during program execution, such as replacing

```
def degree_sin(deg):
    return math.sin(deg * math.pi / 180.0)
```

with

```
def degree_sin(deg, factor=math.pi/180.0, sin=math.sin):
    return sin(deg * factor)
```

Because this trick uses default arguments for terms which should not be changed, it should only be used when you are not concerned with presenting a possibly confusing API to your users.

2.2 Fondamentaux

2.2.1 Pourquoi une UnboundLocalError est levée alors qu'une variable a une valeur ?

Il est parfois surprenant d'obtenir une UnboundLocalError dans du code jusqu'à présent correct, quand celui-ci est modifié en ajoutant une instruction d'affectation quelque part dans le corps d'une fonction.

Le code suivant :

```
>>> x = 10
>>> def bar():
...     print x
>>> bar()
10
```

fonctionne, mais le suivant :

```
>>> x = 10
>>> def foo():
...     print x
...     x += 1
```

lève une UnboundLocalError :

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to `x`, the compiler recognizes it as a local variable. Consequently when the earlier `print x` attempts to print the uninitialized local variable and an error results.

Dans l'exemple ci-dessus, la variable du contexte appelant reste accessible en la déclarant globale :

```
>>> x = 10
>>> def foobar():
...     global x
...     print x
...     x += 1
>>> foobar()
10
```

Cette déclaration explicite est obligatoire pour se rappeler que (contrairement au cas à peu près similaire avec des variables de classe et d'instance), c'est la valeur de la variable du contexte appelant qui est modifiée :

```
>>> print x
11
```

2.2.2 Quelles sont les règles pour les variables locales et globales en Python ?

En Python, si une variable n'est pas modifiée dans une fonction mais seulement lue, elle est implicitement considérée comme globale. Si une valeur lui est affectée, elle est considérée locale (sauf si elle est explicitement déclarée globale).

Bien que surprenant au premier abord, ce choix s'explique facilement. D'une part, exiger `global` pour des variables affectées est une protection contre des effets de bord inattendus. D'autre part, si `global` était obligatoire pour toutes les références à des objets globaux, il faudrait mettre `global` partout, car il faudrait dans ce cas déclarer globale chaque référence à une fonction native ou à un composant d'un module importé. Le codé serait alors truffé de déclarations `global`, ce qui nuirait à leur raison d'être : identifier les effets de bords.

2.2.3 Pourquoi des expressions lambda définies dans une boucle avec des valeurs différentes retournent-elles le même résultat ?

Supposons que l'on utilise une boucle itérative pour définir des expressions lambda (voire même des fonctions) différentes, par exemple :

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Le code précédent crée une liste de 5 expressions lambda qui calculent chacune `x**2`. En les exécutant, on pourrait s'attendre à obtenir `0`, `1`, `4`, `9` et `16`. Elles renvoient en réalité toutes `16` :

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Ceci s'explique par le fait que `x` n'est pas une variable locale aux expressions, mais est définie dans le contexte appelant. Elle est lue à l'appel de l'expression lambda – et non au moment où cette expression est définie. À la fin de la boucle, `x` vaut `4`, donc toutes les fonctions renvoient `4*2`, i.e. `16`. Ceci se vérifie également en changeant la valeur de `x` et en constatant que les résultats sont modifiés :

```
>>> x = 8
>>> squares[2]()
64
```

Pour éviter ce phénomène, les valeurs doivent être stockées dans des variables locales aux expressions lambda pour que celles-ci ne se basent plus sur la variable globale `x` :

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Dans ce code, `n=x` crée une nouvelle variable `n`, locale à l'expression. Cette variable est évaluée quand l'expression est définie donc `n` a la même valeur que `x` à ce moment. La valeur de `n` est donc `0` dans la première lambda, `1` dans la deuxième, `2` dans la troisième et ainsi de suite. Chaque expression lambda renvoie donc le résultat correct :

```
>>> squares[2]()
4
>>> squares[4]()
16
```

Ce comportement n'est pas propre aux expressions lambda, mais s'applique aussi aux fonctions normales.

2.2.4 Comment partager des variables globales entre modules ?

La manière standard de partager des informations entre modules d'un même programme est de créer un module spécial (souvent appelé *config* ou *cfg*) et de l'importer dans tous les modules de l'application ; le module devient accessible depuis l'espace de nommage global. Vu qu'il n'y a qu'une instance de chaque module, tout changement dans l'instance est propagé partout. Par exemple :

config.py

```
x = 0   # Default value of the 'x' configuration setting
```

mod.py

```
import config
config.x = 1
```

main.py

```
import config
import mod
print config.x
```

Pour les mêmes raisons, l'utilisation d'un module est aussi à la base de l'implémentation du patron de conception singleton.

2.2.5 Quelles sont les « bonnes pratiques » pour utiliser import dans un module ?

De manière générale, il ne faut pas faire `from modulename import *`. Ceci encombre l'espace de nommage de l'importateur et rend la détection de noms non-définis beaucoup plus ardue pour les analyseurs de code.

Les modules doivent être importés au début d'un fichier. Ceci permet d'afficher clairement de quels modules le code à besoin et évite de se demander si le module est dans le contexte. Faire un seul *import* par ligne rend l'ajout et la suppression d'une importation de module plus aisé, mais importer plusieurs modules sur une même ligne prend moins d'espace.

Il est recommandé d'importer les modules dans l'ordre suivant :

1. les modules de la bibliothèque standard — e.g. `sys`, `os`, `getopt`, `re`
2. les modules externes (tout ce qui est installé dans le dossier *site-packages* de Python) — e.g. *mx.DateTime, ZODB, PIL.Image*, etc.
3. les modules développés en local

Only use explicit relative package imports. If you're writing code that's in the `package.sub.m1` module and want to import `package.sub.m2`, do not just write `import m2`, even though it's legal. Write `from package.sub import m2` or `from . import m2` instead.

Il est parfois nécessaire de déplacer des importations dans une fonction ou une classe pour éviter les problèmes d'importations circulaires. Comme le souligne Gordon McMillan :

> Il n'y a aucun souci à faire des importations circulaires tant que les deux modules utilisent la forme « import <module> » . Ça ne pose problème que si le second module cherche à récupérer un nom du premier module (*« from module import name »*) et que l'importation est dans l'espace de nommage du fichier. Les noms du premier module ne sont en effet pas encore disponibles car le premier module est occupé à importer le second.

Dans ce cas, si le second module n'est utilisé que dans une fonction, l'importation peut facilement être déplacée dans cette fonction. Au moment où l'importation sera appelée, le premier module aura fini de s'initialiser et le second pourra faire son importation.

Il peut parfois être nécessaire de déplacer des importations de modules hors de l'espace de plus haut niveau du code si certains de ces modules dépendent de la machine utilisée. Dans ce cas de figure, il est parfois impossible d'importer tous

les modules au début du fichier. Dans ce cas, il est recommandé d'importer les modules adéquats dans le code spécifique à la machine.

Les imports ne devraient être déplacés dans un espace local, comme dans la définition d'une fonction, que si cela est nécessaire pour résoudre un problème comme éviter des dépendances circulaires ou réduire le temps d'initialisation d'un module. Cette technique est particulièrement utile si la majorité des imports est superflue selon le flux d'exécution du programme. Il est également pertinent de déplacer des importations dans une fonction si le module n'est utilisé qu'au sein de cette fonction. Le premier chargement d'un module peut être coûteux à cause du coût fixe d'initialisation d'un module, mais charger un module plusieurs fois est quasiment gratuit, cela ne coûte que quelques indirections dans un dictionnaire. Même si le nom du module est sorti du contexte courant, le module est probablement disponible dans `sys.modules`.

2.2.6 Pourquoi les arguments par défaut sont-ils partagés entre les objets ?

C'est un problème que rencontrent souvent les programmeurs débutants. Examinons la fonction suivante

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

Au premier appel de cette fonction, `mydict` ne contient qu'un seul élément. Au second appel, `mydict` contient deux éléments car quand `foo()` commence son exécution, `mydict` contient déjà un élément.

On est souvent amené à croire qu'un appel de fonction créé des nouveau objets pour les valeurs par défaut. Ce n'est pas le cas. Les valeurs par défaut ne sont créées qu'une et une seule fois, au moment où la fonction est définie. Si l'objet est modifié, comme le dictionnaire dans cet exemple, les appels suivants à cette fonction font référence à l'objet ainsi modifié.

Par définition, les objets immuables comme les nombres, les chaînes de caractères, les n-uplets et `None` ne sont pas modifiés. Les changements sur des objets muables comme les dictionnaires, les listes et les instances de classe peuvent porter à confusion.

En raison de cette fonctionnalité, il vaut mieux ne pas utiliser d'objets muables comme valeurs par défaut. Il vaut mieux utiliser `None` comme valeur par défaut et, à l'intérieur de la fonction, vérifier si le paramètre est à `None` et créer une nouvelle liste, dictionnaire ou autre, le cas échéant. Par exemple, il ne faut pas écrire

```
def foo(mydict={}):
    ...
```

mais plutôt

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Cette fonctionnalité a une utilité. Il est courant de mettre en cache les paramètres et la valeur de retour de chacun des appels d'une fonction coûteuse à exécuter, et de renvoyer la valeur stockée en cache si le même appel est ré-effectué. C'est la technique dite de « mémoïsation », qui s'implémente de la manière suivante

```
# Callers will never provide a third parameter for this function.
def expensive(arg1, arg2, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

Il est possible d'utiliser une variable globale contenant un dictionnaire à la place de la valeur par défaut ; ce n'est qu'une question de goût.

2.2.7 Comment passer des paramètres optionnels ou nommés d'une fonction à l'autre ?

Il faut récupérer les arguments en utilisant les sélecteurs `*` et `**` dans la liste des paramètres de la fonction ; ceci donne les arguments positionnels sous la forme d'un n-uplet et les arguments nommés sous forme de dictionnaire. Ces arguments peuvent être passés à une autre fonction en utilisant `*` et `**`

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

In the unlikely case that you care about Python versions older than 2.0, use `apply()` :

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    apply(g, (x,)+args, kwargs)
```

2.2.8 Quelle est la différence entre les arguments et les paramètres ?

Les *paramètres* sont les noms qui apparaissent dans une définition de fonction, alors que les *arguments* sont les valeurs qui sont réellement passées à une fonction lors de l'appel de celle-ci. Les paramètres définissent les types des arguments qu'une fonction accepte. Ainsi, avec la définition de fonction suivante

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* et *kwargs* sont des paramètres de `func`. Mais à l'appel de `func` avec, par exemple

```
func(42, bar=314, extra=somevar)
```

les valeurs `42`, `314`, et `somevar` sont des arguments.

2.2.9 Pourquoi modifier la liste "y" modifie aussi la liste "x" ?

Si vous avez écrit du code comme :

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

vous vous demandez peut-être pourquoi l'ajout d'un élément à `y` a aussi changé `x`.

Il y a deux raisons qui conduisent à ce comportement :

1) Les variables ne sont que des noms qui font référence à des objets. La ligne `y = x` ne crée pas une copie de la liste — elle crée une nouvelle variable `y` qui pointe sur le même objet que `x`. Ceci signifie qu'il n'existe qu'un seul objet (la liste) auquel `x` et `y` font référence.
2) Les listes sont des *muable*, ce qui signifie que leur contenu peut être modifié.

Après l'appel de `append()`, le contenu de l'objet muable est passé de `[]` à `[10]`. Vu que les deux variables font référence au même objet, il est possible d'accéder à la valeur modifiée `[10]` avec chacun des noms.

Si au contraire, on affecte un objet immuable à `x`

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

on observe que `x` et `y` ne sont ici plus égales. Les entiers sont des immuables (*immutable*), et `x = x + 1` ne change pas l'entier `5` en incrémentant sa valeur. Au contraire, un nouvel objet est créé (l'entier `6`) et affecté à `x` (c'est à dire qu'on change l'objet auquel fait référence `x`). Après cette affectation on a deux objets (les entiers `6` et `5`) et deux variables qui font référence à ces deux objets (`x` fait désormais référence à `6` mais `y` fait toujours référence à `5`).

Certaines opérations (par exemple, `y.append(10)` et `y.sort()`) modifient l'objet, alors que des opérations identiques en apparence (par exemple `y = y + [10]` et `sorted(y)`) créent un nouvel objet. En général, en Python, une méthode qui modifie un objet renvoie `None` (c'est même systématique dans la bibliothèque standard) pour éviter la confusion entre les deux opérations. Donc écrire par erreur `y.sort()` en pensant obtenir une copie triée de `y` donne `None`, ce qui conduit très souvent le programme à générer une erreur facile à diagnostiquer.

Il existe cependant une classe d'opérations qui se comporte différemment selon le type : les opérateurs d'affectation incrémentaux. Par exemple, += modifie les listes mais pas les n-uplets ni les entiers (`a_list += [1, 2, 3]` équivaut à `a_list.extend([1, 2, 3])` et modifie `a_list`, alors que `some_tuple += (1, 2, 3)` et `some_int += 1` créent de nouveaux objets).

En d'autres termes :

— Il est possible d'appliquer des opérations qui modifient un objet muable (`list`, `dict`, `set`, etc.) et toutes les variables qui y font référence verront le changement.
— Toutes les variables qui font référence à un objet immuable (`str`, `int`, `tuple`, etc.) renvoient la même valeur, mais les opérations qui transforment cette valeur en une nouvelle valeur renvoient toujours un nouvel objet.

L'opérateur `is` ou la fonction native `id()` permettent de savoir si deux variables font référence au même objet.

2.2.10 Comment écrire une fonction qui modifie ses paramètres ? (passage par référence)

En Python, les arguments sont passés comme des affectations de variables. Vu qu'une affectation crée des références à des objets, il n'y pas de lien entre un argument dans l'appel de la fonction et sa définition, et donc pas de passage par référence en soi. Il y a cependant plusieurs façon d'en émuler un.

1) En renvoyant un n-uplet de résultats

```
def func2(a, b):
    a = 'new-value'        # a and b are local names
    b = b + 1              # assigned to new objects
    return a, b            # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print x, y                 # output: new-value 100
```

C'est presque toujours la meilleure solution.

2) En utilisant des variables globales. Ce qui n'est pas *thread-safe*, et n'est donc pas recommandé.

3) En passant un objet muable (modifiable sur place) :

```
def func1(a):
    a[0] = 'new-value'     # 'a' references a mutable list
    a[1] = a[1] + 1        # changes a shared object

args = ['old-value', 99]
func1(args)
print args[0], args[1]     # output: new-value 100
```

4) En passant un dictionnaire, qui sera modifié :

```
def func3(args):
    args['a'] = 'new-value'     # args is a mutable dictionary
    args['b'] = args['b'] + 1   # change it in-place

args = {'a': 'old-value', 'b': 99}
func3(args)
print args['a'], args['b']
```

5) Ou regrouper les valeurs dans une instance de classe :

```
class callByRef:
    def __init__(self, **args):
        for (key, value) in args.items():
            setattr(self, key, value)

def func4(args):
    args.a = 'new-value'        # args is a mutable callByRef
    args.b = args.b + 1         # change object in-place

args = callByRef(a='old-value', b=99)
func4(args)
print args.a, args.b
```

Il n'y a pratiquement jamais de bonne raison de faire quelque chose d'aussi compliqué.

Votre meilleure option est de renvoyer un *tuple* contenant les multiples résultats.

2.2.11 Comment construire une fonction d'ordre supérieur en Python ?

Vous avez deux choix : vous pouvez utiliser les portées imbriquées ou vous pouvez utiliser des objets appelables. Par exemple, supposons que vous vouliez définir `linear(a, b)` qui renvoie une fonction `f(x)` qui calcule la valeur `a*x+b`. En utilisant les portées imbriquées :

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Ou en utilisant un objet appelable :

```
class linear:

    def __init__(self, a, b):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

dans les deux cas,

```
taxes = linear(0.3, 2)
```

donne un objet appelable où `taxes(10e6) == 0.3 * 10e6 + 2`.

L'approche par objet appelable a le désavantage d'être légèrement plus lente et de produire un code légèrement plus long. Cependant, il faut noter qu'une collection d'objet appelables peuvent partager leur signatures par héritage :

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Les objets peuvent encapsuler un état pour plusieurs méthodes :

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Ici `inc()`, `dec()` et `reset()` agissent comme des fonctions partageant une même variable compteur.

2.2.12 Comment copier un objet en Python ?

En général, essayez `copy.copy()` ou `copy.deepcopy()` pour le cas général. Tout les objets ne peuvent pas être copiés, mais la plupart le peuvent.

Certains objets peuvent être copiés plus facilement. Les Dictionnaires ont une méthode `copy()` :

```
newdict = olddict.copy()
```

Les séquences peuvent être copiées via la syntaxe des tranches :

```
new_l = l[:]
```

2.2.13 Comment puis-je trouver les méthodes ou les attribues d'un objet ?

Pour une instance x d'une classe définie par un utilisateur, `dir(x)` renvoie une liste alphabétique des noms contenants les attributs de l'instance, et les attributs et méthodes définies par sa classe.

2.2.14 Comment mon code peut il découvrir le nom d'un objet ?

De façon générale, il ne peut pas, par ce que les objets n'ont pas réellement de noms. Essentiellement, l'assignation attache un nom à une valeur ; C'est vrai aussi pour les instructions `def` et `class`, à la différence que dans ce cas la valeur est appelable. Par exemple, dans le code suivant :

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print b
<__main__.A instance at 0x16D07CC>
>>> print a
<__main__.A instance at 0x16D07CC>
```

Le fait que la classe ait un nom est discutable, bien qu'elles soit liée à deux noms, et qu'elle soit appelée via le nom B, l'instance crée déclare tout de même être une instance de la classe A. De même Il est impossible de dire si le nom de l'instance est a ou b, les deux noms sont attachés à la même valeur.

De façon général, il ne devrait pas être nécessaire pour votre application de « connaître le nom » d'une valeur particulière. À moins que vous soyez délibérément en train d'écrire un programme introspectif, c'est souvent une indication qu'un changement d'approche pourrait être bénéfique.

Sur *comp.lang.python*, Fredrik Lundh a donné un jour une excellente analogie pour répondre à cette question :

> C'est pareil que trouver le nom du chat qui traîne devant votre porte : Le chat (objet) ne peux pas vous dire lui même son nom, et il s'en moque un peu – alors le meilleur moyen de savoir comment il s'appelle est de demander à tous vos voisins (espaces de noms) si c'est leur chat (objet)….
>
> …et ne soyez pas surpris si vous découvrez qu'il est connus sous plusieurs noms différents, ou pas de nom du tout !

2.2.15 Qu'en est-il de la précédence de l'opérateur virgule ?

La virgule n'est pas un opérateur en Python. Observez la session suivante :

```
>>> "a" in "b", "a"
(False, 'a')
```

Comme la virgule n'est pas un opérateur, mais un séparateur entre deux expression, l'expression ci dessus, est évaluée de la même façon que si vous aviez écrit :

```
("a" in "b"), "a"
```

et non :

```
"a" in ("b", "a")
```

Ceci est vrai pour tous les opérateurs d'assignations (=, += etc). Ce ne sont pas vraiment des opérateurs mais des délimiteurs syntaxiques dans les instructions d'assignation.

2.2.16 Existe-t-il un équivalent à l'opérateur ternaire « ?: » du C ?

Oui, cette fonctionnalité à été ajouté à partir de Python 2.5. La syntaxe est la suivante :

```
[on_true] if [expression] else [on_false]

x, y = 50, 25

small = x if x < y else y
```

Pour les versions précédentes de python la réponse serait « Non ».

2.2.17 Est-il possible d'écrire des programmes obscurcis (*obfuscated*) d'une ligne en Python ?

Oui. Cela est généralement réalisé en imbriquant les `lambda` dans des `lambda`. Observez les trois exemples suivants de Ulf Bartelt :

```
# Primes < 1000
print filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))

# First 10 Fibonacci numbers
print map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1: f(x,f),
range(10))

# Mandelbrot set
print (lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
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
>=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(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24)
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |__________ maximum of "iterations"
#         |          |_________________ range on y axis
#         |____________________________ range on x axis
```

Les enfants, ne faîtes pas ça chez vous !

2.3 Nombres et chaînes de caractères

2.3.1 Comment puis-je écrire des entiers hexadécimaux ou octaux ?

Pour écrire un entier octal, faites précéder la valeur octale par un zéro, puis un « o » majuscule ou minuscule. Par exemple assigner la valeur octale « 10 » (8 en décimal) à la variable « a », tapez :

```
>>> a = 0o10
>>> a
8
```

L'hexadécimal est tout aussi simple, faîtes précéder le nombre hexadécimal par un zéro, puis un « x » majuscule ou minuscule. Les nombres hexadécimaux peuvent être écrit en majuscules ou en minuscules. Par exemple, dans l'interpréteur Python :

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 Pourquoi `-22 // 10` donne-t-il `-3` ?

Cela est principalement due à la volonté que `i % j` ait le même signe que j. Si vous voulez cela, vous voulez aussi :

```
i == (i // j) * j + (i % j)
```

Alors la division entière doit renvoyer l'entier inférieur. Le C demande aussi à ce que cette égalité soit vérifiée, et donc les compilateur qui tronquent `i // j` ont besoin que `i % j` ait le même signe que `i`.

Il y a peu de cas d'utilisation réels pour `i%j` quand `j` est négatif. Quand `j` est positif, il y en a beaucoup, et dans pratiquement tous, il est plus utile que `i % j` soit >=0. Si l'horloge dit *10h* maintenant, que disait-elle il y a 200 heures ? `-190%12 == 2` est utile ; `-192 % 12 == -10` est un bug qui attends pour mordre.

Note : On Python 2, `a / b` returns the same as `a // b` if `__future__.division` is not in effect. This is also known as « classic » division.

2.3.3 Comment puis-je convertir une chaine de caractère en nombre ?

Pour les entiers, utilisez la fonction native `int()` de type constructeur, par exemple `int('144') == 144`. De façon similaire, `float()` convertit en valeur flottante, par exemple `float('144') == 144.0`.

Par défaut, ces fonctions interprètent les nombre en tant que décimaux, de telles façons que `int('0144')==144` et `int('0x144')` remontent `ValueError`. `int(string, base)` prends la base depuis laquelle il faut convertir dans le second argument, optionnel, donc `int('0x144', 16) == 324`. Si la base donnée est 0, le nombre est interprété selon les règles Python : un “0” en tête indique octal, et “0x” indique un hexadécimal.

N'utilisez pas la fonction native `eval()` si tout ce que vous avez besoin est de convertir des chaines en nombres. `eval()` sera significativement plus lent et implique des risque de sécurité : quelqu'un pourrait vous envoyez une expression Python pouvant avoir des effets de bord indésirables. Par exemple, quelqu'un pourrait passer `__import__('os').system("rm -rf $HOME")` ce qui aurait pour effet d'effacer votre répertoire personnel.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python regards numbers starting with “0” as octal (base 8).

2.3.4 Comment convertir un nombre en chaine de caractère ?

To convert, e.g., the number 144 to the string "144", use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the formatstrings section, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:.3f}".format(1.0/3.0)` yields `'0.333'`. In Python 2, the division (/) operator returns the floor of the mathematical result of division if the arguments are ints or longs, but it returns a reasonable approximation of the division result if the arguments are floats or complex :

```
>>> print('{:.3f}'.format(1/3))
0.000
>>> print('{:.3f}'.format(1.0/3))
0.333
```

In Python 3, the default behaviour of the division operator (see **PEP 238**) has been changed but you can have the same behaviour in Python 2 if you import `division` from `__future__` :

```
>>> from __future__ import division
>>> print('{:.3f}'.format(1/3))
0.333
```

You may also use the % operator on strings. See the library reference manual for details.

2.3.5 Comment modifier une chaine de caractère « en place » ?

Vous ne pouvez pas, par ce que les chaines de caractères sont immuables, Si vous avez besoin d'un objet ayant une telle capacité, essayez de convertir la chaine en liste, ou utilisez le module array :

```
>>> import io
>>> s = "Hello, world"
>>> a = list(s)
>>> print a
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd']
>>> a[7:] = list("there!")
>>> ''.join(a)
'Hello, there!'

>>> import array
>>> a = array.array('c', s)
>>> print a
array('c', 'Hello, world')
>>> a[0] = 'y'; print a
array('c', 'yello, world')
>>> a.tostring()
'yello, world'
```

2.3.6 Comment utiliser des chaines de caractères pour appeler des fonctions/méthodes ?

Il y a différentes techniques.

— La meilleure est d'utiliser un dictionnaire qui fait correspondre les chaines de caractères à des fonctions. Le principal avantage de cette technique est que les chaines n'ont pas besoin d'être égales aux noms de fonctions. C'est aussi la principale façon d'imiter la construction « case » :

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

— Utiliser la fonction `getattr()` :

```
import foo
getattr(foo, 'bar')()
```

Notez que `getattr()` marche sur n'importe quel objet, ceci inclue les classes, les instances de classes, les modules et ainsi de suite.
Ceci est utilisé dans plusieurs endroit de la bibliothèque standard, de cette façon :

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

— Utilisez `locals()` ou `eval()` pour résoudre le nom de fonction :

```
def myFunc():
    print "hello"

fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()
```

Note : En utilisant `eval()` est lent est dangereux. Si vous n'avez pas un contrôle absolu sur le contenu de la chaine de caractère, quelqu'un peut passer une chaine de caractère pouvant résulter en l'exécution de code arbitraire.

2.3.7 Existe-t-il un équivalent à la fonction chomp() de Perl, pour retirer les caractères de fin de ligne d'une chaine de caractère ?

Starting with Python 2.2, you can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed :

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Du fait que ce soit principalement utile en lisant un texte ligne à ligne, utiliser `S.rstrip()` devrait marcher correctement.

Pour les versions plus anciennes de python, il y a deux substituts partiels disponibles.

- Si vous voulez retirer tous les espaces de fin de ligne, utilisez la méthode `rstrip()` des chaines de caractères. Cela retire tous les espaces de fin de ligne, pas seulement le caractère de fin de ligne.
- Sinon, s'il y a seulement une ligne dans la chaine `S`, utilisez `S.splitlines()[0]`.

2.3.8 Existe-t-il un équivalent à scanf() ou sscanf() ?

Pas exactement.

Pour une simple analyse de chaine, l'approche la plus simple est généralement de découper la ligne en mots délimités par des espaces, en utilisant la méthode `split()` des objets chaine de caractères, et ensuite de convertir les chaines de décimales en valeurs numériques en utilisant la fonction `int()` ou `float()`, `split()` supporte un paramètre optionnel « sep » qui est utile si la ligne utilise autre chose que des espaces comme séparateur.

Pour les analyses plus compliquées, les expressions rationnelles sont plus puissantes que la fonction `sscanf()` de C et mieux adaptées à la tâche.

2.3.9 What does "UnicodeError : ASCII [decoding,encoding] error : ordinal not in range(128)" mean ?

This error indicates that your Python installation can handle only 7-bit ASCII strings. There are a couple ways to fix or work around the problem.

If your programs must handle data in arbitrary character set encodings, the environment the application runs in will generally identify the encoding of the data it is handing you. You need to convert the input to Unicode data using that encoding. For example, a program that handles email or web input will typically find character set encoding information in Content-Type headers. This can then be used to properly convert input data to Unicode. Assuming the string referred to by `value` is encoded as UTF-8 :

```
value = unicode(value, "utf-8")
```

will return a Unicode object. If the data is not correctly encoded as UTF-8, the above call will raise a `UnicodeError` exception.

If you only want strings converted to Unicode which have non-ASCII data, you can try converting them first assuming an ASCII encoding, and then generate Unicode objects if that fails :

```
try:
    x = unicode(value, "ascii")
except UnicodeError:
    value = unicode(value, "utf-8")
else:
    # value was valid ASCII data
    pass
```

It's possible to set a default encoding in a file called `sitecustomize.py` that's part of the Python library. However, this isn't recommended because changing the Python-wide default encoding may cause third-party extension modules to fail.

Note that on Windows, there is an encoding known as « mbcs », which uses an encoding specific to your current locale. In many cases, and particularly when working with COM, this may be an appropriate default encoding to use.

2.4 Sequences (Tuples/Lists)

2.4.1 Comment convertir les listes en tuples et inversement ?

Le constructeur de type `tuple(seq)` convertit toute séquence (en fait tout itérable) en un tuple avec les mêmes éléments dans le même ordre….

Par exemple `tuple([1, 2, 3])` renvoi `(1, 2, 3)` et `tuple('abc')` renvoi `('a', 'b', 'c')`. Si l'argument est un tuple, cela ne crée pas une copie, mais renvoi le même objet, ce qui fait de `tuple()` un fonction économique à appeler quand vous ne savez pas si votre objet est déjà un tuple.

Le constructeur de type `list(seq)` convertit toute séquence ou itérable en liste contenant les mêmes éléments dans le même ordre. Par exemple, `list((1,2,3))` renvoie `[1,2,3]` et `list('abc')` renvoie `['a','b','c']`. Si l'argument est une liste, il renvoie une copie, de la même façon que `seq[:]`.

2.4.2 Qu'est-ce qu'un indexe négatif ?

Les séquences Python sont indexées avec des nombres positifs aussi bien que négatifs. Pour les nombres positifs, 0 est le premier index, 1 est le second, et ainsi de suite. Pour les indexes négatifs, `-1` est le dernier index, `-2` est le pénultième (avant dernier), et ainsi de suite. On peut aussi dire que `seq[-n]` est équivalent à `seq[len(seq)-n]`.

Utiliser des indexes négatifs peut être très pratique. Par exemple `S[:-1]` indique la chaine entière a l'exception du dernier caractère, ce qui est pratique pour retirer un caractère de fin de ligne en fin d'une chaine.

2.4.3 Comment itérer à rebours sur une séquence ?

Utilisez la fonction embarquée `reversed()`, qui est apparue en Python 2.4 :

```
for x in reversed(sequence):
    ...  # do something with x ...
```

Cela ne modifiera pas votre séquence initiale, mais construira à la place une copie en ordre inverse pour itérer dessus.

Avec Python 2.3 vous pouvez utiliser la syntaxe étendue de tranches :

```
for x in sequence[::-1]:
    ...  # do something with x ...
```

2.4.4 Comment retirer les doublons d'une liste ?

Lisez le Python Cookbook pour trouver une longue discussion sur les nombreuses façons de faire cela :

https://code.activestate.com/recipes/52560/

Si changer l'ordre de la liste ne vous dérange pas, commencez par trier celle ci, puis parcourez la d'un bout à l'autre, en supprimant les doublons trouvés en chemin :

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

Si tous les éléments de la liste peuvent être utilisés comme des clés de dictionnaire (cad, elles sont toutes hashables) ceci est souvent plus rapide :

```
d = {}
for x in mylist:
    d[x] = 1
mylist = list(d.keys())
```

En Python 2.5 et suivant, la forme suivante est possible à la place :

```
mylist = list(set(mylist))
```

Ceci convertis la liste en un ensemble, ce qui supprime automatiquement les doublons, puis la transforme à nouveau en liste.

2.4.5 Comment construire un tableau en Python ?

Utilisez une liste :

```
["this", 1, "is", "an", "array"]
```

Les listes ont un cout équivalent à celui des tableau C ou Pascal ; la principale différence est qu'une liste Python peut contenir des objets de différents types.

Le module `array` fournit des méthodes pour créer des tableaux de types fixes dans une représentation compacte, mais ils sont plus lents à indexer que les listes. Notez aussi que l'extension `Numeric` et d'autres, fournissent différentes structures de types tableaux, avec des caractéristiques différentes.

Pour obtenir des listes chainées de type Lisp, vous pouvez émuler les *cons cells* en utilisant des tuples :

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

Si vous voulez pouvoir modifier les éléments, utilisez une liste plutôt qu'un tuple. Ici la version équivalente au *car* de Lisp est `lisp_list[0]` et l'équivalent à *cdr* est `list_lip[1]`. Ne faites ceci que si vous êtes réellement sûr d'en avoir besoin, cette méthode est en générale bien plus lente que les listes Python.

2.4.6 Comment puis-je créer une liste à plusieurs dimensions ?

Vous avez probablement essayé de créer une liste à plusieurs dimensions de cette façon :

```
>>> A = [[None] * 2] * 3
```

Cela semble correct quand vous essayer de l'afficher :

```
>>> A
[[None, None], [None, None], [None, None]]
```

Mais quand vous assignez une valeur, elle apparait en de multiples endroits :

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

La raison en est que dupliquer une liste en utilisant `*` ne crée pas de copies, cela crée seulement des références aux objets existants. Le `*3` crée une liste contenant trois références à la même liste de longueur deux. Un changement dans une colonne apparaîtra donc dans toutes les colonnes. Ce qui n'est de façon quasi certaine, pas ce que vous souhaitez.

L'approche suggérée est de créer une liste de la longueur désiré d'abords, puis de remplir tous les éléments avec une chaîne nouvellement créée :

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Cette liste générée contient trois listes différentes de longueur deux. Vous pouvez aussi utilisez la notation de compréhension de listes :

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Vous pouvez aussi utiliser une extension qui fournit un type matriciel natif ; NumPy est la plus répandue.

2.4.7 Comment appliquer une méthode à une séquence d'objets ?

Utilisez une compréhension de liste :

```
result = [obj.method() for obj in mylist]
```

More generically, you can try the following function :

```
def method_map(objects, method, arguments):
    """method_map([a,b], "meth", (1,2)) gives [a.meth(1,2), b.meth(1,2)]"""
    nobjects = len(objects)
    methods = map(getattr, objects, [method]*nobjects)
    return map(apply, methods, [arguments]*nobjects)
```

2.4.8 Pourquoi a_tuple[i] += ["item"] lève-t'il une exception alors que l'addition fonctionne ?

Ceci est dû à la combinaison de deux facteurs : le fait que les opérateurs d'affectation incrémentaux sont des opérateurs d'*affectation* et à la différence entre les objets muables et immuables en Python.

Cette discussion est valable, en général, quand des opérateurs d'affectation incrémentale sont appliqués aux élément d'un n-uplet qui pointe sur des objets muables, mais on prendra `list` et += comme exemple.

Si vous écrivez :

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
   ...
TypeError: 'tuple' object does not support item assignment
```

La cause de l'exception est claire : `1` est ajouté à l'objet `a_tuple[0]` qui pointe sur (`1`), ce qui produit l'objet résultant `2`, mais, lorsque l'on tente d'affecter le résultat du calcul, `2`, à l'élément `0` du n-uplet, on obtient une erreur car il est impossible de modifier la cible sur laquelle pointe un élément d'un n-uplet.

Sous le capot, une instruction d'affectation incrémentale fait à peu près ceci :

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

C'est la partie de l'affectation de l'opération qui génère l'erreur, vu qu'un n-uplet est immuable.

Quand vous écrivez un code du style :

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

L'exception est un peu plus surprenante et, chose encore plus étrange, malgré l'erreur, l'ajout a fonctionné

```
>>> a_tuple[0]
['foo', 'item']
```

Pour comprendre ce qui se passe, il faut savoir que, premièrement, si un objet implémente la méthode magique c, celle-ci est appelée quand l'affectation incrémentale += est exécutée et sa valeur de retour est utilisée dans l'instruction d'affectation ; et que, deuxièmement, pour les listes, `__iadd__` équivaut à appeler `extend` sur la liste et à renvoyer celle-ci. C'est pour cette raison que l'on dit que pour les listes, += est un « raccourci » pour `list.extend` :

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

C'est équivalent à :

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

L'objet sur lequel pointe `a_list` a été modifié et le pointeur vers l'objet modifié est réaffecté à `a_list`. *In fine*, l'affectation ne change rien, puisque c'est un pointeur vers le même objet que sur lequel pointait `a_list`, mais l'affectation a tout de même lieu.

Donc, dans notre exemple avec un n-uplet, il se passe quelque chose équivalent à

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

L'appel à `__iadd__` réussit et la liste est étendue, mais bien que `result` pointe sur le même objet que `a_tuple[0]`, l'affectation finale échoue car les n-uplets ne sont pas muables.

2.5 Dictionnaires

2.5.1 Comment puis-je faire afficher les éléments d'un dictionnaire dans un ordre consistant ?

Vous ne pouvez pas. Les dictionnaires enregistrent leurs clées dans un ordre non prévisible, l'ordre d'affichage des éléments d'un dictionnaire sera donc de la même façon imprévisible.

Cela peut être frustrant si vous voulez sauvegarder une version affichable dans un fichier, faire des changement puis comparer avec un autre dictionnaire affiché. Dans ce cas, utilisez le module `pprint` pour afficher joliement le dictionnaire ; les éléments seront présentés triés par clés.

Une solution plus compliquée est de sousclasser `dict` pour créer une classe"SorterDict" qui s'affiche de façon prévisible. Voici une implémentation simple d'une telle classe :

```
class SortedDict(dict):
    def __repr__(self):
        keys = sorted(self.keys())
        result = ("{!r}: {!r}".format(k, self[k]) for k in keys)
        return "{{{}}}".format(", ".join(result))

    __str__ = __repr__
```

Cela marchera dans la plupart des situations que vous pourriez rencontrer, même si c'est loin d'être une solution parfaite. Le plus gros problème avec cette solution est que si certaines valeurs dans le dictionnaire sont aussi des dictionnaire, alors elles ne seront pas présentées dans un ordre particulier.

2.5.2 Je souhaite faire un tri compliqué : peut on faire une transformation de Schwartz en Python ?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its « sort value ». In Python, use the `key` argument for the `sort()` function :

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.3 Comment puis-je trier une liste en fonction des valeurs d'une autre liste ?

Merge them into a single list of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs
[('what', 'something'), ("I'm", 'else'), ('sorting', 'to'), ('by', 'sort')]
>>> pairs.sort()
>>> result = [ x[1] for x in pairs ]
>>> result
['else', 'sort', 'to', 'something']
```

Une alternative pour la dernière étape est :

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

Si vous trouvez cela plus lisible, vous préférez peut-être utiliser ceci à la place de la compréhension de la liste finale. Toutefois, ceci est presque deux fois plus lent pour les longues listes. Pourquoi ? Tout d'abord, `append ()` doit réaffecter la mémoire, et si il utilise quelques astuces pour éviter de le faire à chaque fois, il doit encore le faire de temps en temps, ce qui coûte assez cher. Deuxièmement, l'expression `result.append` exige une recherche d'attribut supplémentaire, et enfin, tous ces appels de fonction impactent la vitesse d'exécution.

2.6 Objets

2.6.1 Qu'est-ce qu'une classe ?

Une classe est le type d'objet particulier créé par l'exécution d'une déclaration de classe. Les objets de classe sont utilisés comme modèles pour créer des objets, qui incarnent à la fois les données (attributs) et le code (méthodes) spécifiques à un type de données.

Une classe peut être fondée sur une ou plusieurs autres classes, appelée sa ou ses classes de base. Il hérite alors les attributs et les méthodes de ses classes de base. Cela permet à un modèle d'objet d'être successivement raffinés par héritage. Vous pourriez avoir une classe générique `Mailbox` qui fournit des méthodes d'accès de base pour une boîte aux lettres, et sous-classes telles que `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` qui gèrent les différents formats de boîtes aux lettres spécifiques.

2.6.2 Qu'est-ce qu'une méthode ?

Une méthode est une fonction sur un objet `x` appelez normalement comme `x.name(arguments...)`. Les méthodes sont définies comme des fonctions à l'intérieur de la définition de classe :

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 Qu'est-ce que self ?

Self est simplement un nom conventionnel pour le premier argument d'une méthode. Une méthode définie comme `meth(self, a, b, c)` doit être appelée en tant que `x.meth(a, b, c)`, pour une instance `x` de la classe dans laquelle elle est définie, la méthode appelée considérera qu'elle est appelée `meth(x, a, b, c)`.

Voir aussi *Pourquoi « self » doit-il être explicitement utilisé dans les définitions et les appels de méthode ?*.

2.6.4 Comment puis-je vérifier si un objet est une instance d'une classe donnée ou d'une sous-classe de celui-ci ?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, long, float, complex))`.

Notez que la plupart des programmes n'utilisent pas `isInstance()` sur les classes définies par l'utilisateur, très souvent. Si vous développez vous-même les classes, un style plus appropriée orientée objet est de définir des méthodes sur les classes qui encapsulent un comportement particulier, au lieu de vérifier la classe de l'objet et de faire quelque chose de différent en fonction de sa classe. Par exemple, si vous avez une fonction qui fait quelque chose :

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

Une meilleure approche est de définir une méthode `search()` sur toutes les classes et qu'il suffit d'appeler :

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox

class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

2.6.5 Qu'est-ce que la délégation ?

La délégation est une technique orientée objet (aussi appelé un modèle de conception). Disons que vous avez un objet `x` et que vous souhaitez modifier le comportement d'une seule de ses méthodes. Vous pouvez créer une nouvelle classe qui fournit une nouvelle implémentation de la méthode qui vous intéresse dans l'évolution et les délégués de toutes les autres méthodes la méthode correspondante de `x`.

Les programmeurs Python peuvent facilement mettre en œuvre la délégation. Par exemple, la classe suivante implémente une classe qui se comporte comme un fichier, mais convertit toutes les données écrites en majuscules :

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Ici, la classe `UpperOut` redéfinit la méthode `write()` pour convertir la chaîne d'argument en majuscules avant d'appeler la méthode sous-jacentes `self.__outfile.write()`. Toutes les autres méthodes sont déléguées à l'objet sous-jacent `self.__outfile`. La délégation se fait par la méthode `__getattr__`, consulter the language reference pour plus d'informations sur le contrôle d'accès d'attribut.

Notez que pour une utilisation plus générale de la délégation, les choses peuvent se compliquer. Lorsque les attributs doivent être définis aussi bien que récupérés, la classe doit définir une méthode `__setattr__()` aussi, et il doit le faire avec soin. La mise en œuvre basique de la méthode `__setattr__()` est à peu près équivalent à ce qui suit :

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

La plupart des implémentations de `__setattr__()` doivent modifier `self.__dict__` pour stocker l'état locale de self sans provoquer une récursion infinie.

2.6.6 Comment appeler une méthode définie dans une classe de base depuis une classe dérivée qui la surcharge ?

Si vous utilisez des *new-style classes*, Utilisez la fonction native `super()` :

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

If you're using classic classes : For a class definition such as `class Derived(Base): ...` you can call method `meth()` defined in `Base` (or one of `Base`'s base classes) as `Base.meth(self, arguments...)`. Here, `Base.meth` is an unbound method, so you need to provide the `self` argument.

2.6.7 Comment puis-je organiser mon code pour permettre de changer la classe de base plus facilement ?

Vous pouvez définir un alias pour la classe de base, lui attribuer la classe de base réelle avant la définition de classe, et utiliser l'alias au long de votre classe. Ensuite, tout ce que vous devez changer est la valeur attribuée à l'alias. Incidemment, cette astuce est également utile si vous voulez décider dynamiquement (par exemple en fonction de la disponibilité des ressources) la classe de base à utiliser. Exemple :

```
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
        ...
```

2.6.8 Comment puis-je créer des données statiques de classe et des méthodes statiques de classe ?

Tant les données statiques que les méthodes statiques (dans le sens de C + + ou Java) sont pris en charge en Python.

Pour les données statiques, il suffit de définir un attribut de classe. Pour attribuer une nouvelle valeur à l'attribut, vous devez explicitement utiliser le nom de classe dans l'affectation :

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

`c.count` se réfère également à `C.count` pour tout `c` telle que `isInstance (c, C)` est vrai, sauf remplacement par `c` lui-même ou par une classe sur le chemin de recherche de classe de base de `c.__class__` jusqu'à `C`.

Attention : dans une méthode de C, une affectation comme `self.count = 42` crée une nouvelle instance et sans rapport avec le nom `count` dans dans le dictionnaire de données de `self`. La redéfinition d'une donnée statique de classe doit toujours spécifier la classe que l'on soit à l'intérieur d'une méthode ou non :

```
C.count = 314
```

Les méthodes statiques sont possibles depuis Python 2.2 :

```
class C:
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
    static = staticmethod(static)
```

Avec les décorateurs de Python 2.4, cela peut aussi s'écrire :

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

Cependant, d'une manière beaucoup plus simple pour obtenir l'effet d'une méthode statique se fait par une simple fonction au niveau du module :

```
def getcount():
    return C.count
```

Si votre code est structuré de manière à définir une classe (ou bien la hiérarchie des classes connexes) par module, ceci fournira l'encapsulation souhaitée.

2.6.9 Comment puis-je surcharger les constructeurs (ou méthodes) en Python ?

Cette réponse s'applique en fait à toutes les méthodes, mais la question vient généralement en premier dans le contexte des constructeurs.

In C++ you'd write

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

En Python, vous devez écrire un constructeur unique qui considère tous les cas en utilisant des arguments par défaut. Par exemple :

```
class C:
    def __init__(self, i=None):
        if i is None:
            print "No arguments"
        else:
            print "Argument is", i
```

Ce n'est pas tout à fait équivalent, mais suffisamment proche dans la pratique.

Vous pouvez aussi utiliser une liste d'arguments de longueur variable, par exemple :

```
def __init__(self, *args):
    ...
```

La même approche fonctionne pour toutes les définitions de méthode.

2.6.10 J'essaie d'utiliser `__spam` et j'obtiens une erreur à propos de `_SomeClassName__spam`.

Les noms de variables commençant avec deux tirets bas sont «déformés», c'est un moyen simple mais efficace de définir variables privées à une classe. Tout identifiant de la forme `__spam` (commençant par au moins deux tirets bas et se terminant par au plus un tiret bas) est textuellement remplacé par `_classname__spam`, où `classname` est le nom de la classe en cours dont les éventuels tirets bas ont été retirés.

Cela ne garantit pas la privauté de l'accès : un utilisateur extérieur peut encore délibérément accéder à l'attribut `_classname__spam`, et les valeurs privées sont visibles dans l'objet `__dict__`. De nombreux programmeurs Python ne prennent jamais la peine d'utiliser des noms de variable privée.

2.6.11 Ma classe définit `__del__` mais il n'est pas appelé lorsque je supprime l'objet.

Il y a plusieurs raisons possibles pour cela.

La commande *del* n'appelle pas forcément `__del__()` — il décrémente simplement le compteur de références de l'objet, et si celui ci arrive à zéro `__del__()` est appelée.

Si la structure de données contient des références circulaires (e.g. un arbre dans lequel chaque fils référence son père, et chaque père garde une liste de ses fils), le compteur de références n'arrivera jamais à zéro. Python exécute périodiquement un algorithme pour détecter ce genre de cycles, mais il peut se passer un certain temps entre le moment où la structure est référencée pour la dernière fois et l'appel du ramasse-miettes, donc la méthode `__del__()` peut être appelée à un moment aléatoire et pas opportun. C'est gênant pour essayer reproduire un problème. Pire, l'ordre dans lequel les méthodes

`__del__()` des objets sont appelées est arbitraire. Il est possible de forcer l'appel du ramasse-miettes avec la fonction `gc.collect()`, mais il existe certains cas où les objets ne seront jamais nettoyés.

Despite the cycle collector, it's still a good idea to define an explicit `close()` method on objects to be called whenever you're done with them. The `close()` method can then remove attributes that refer to subobjecs. Don't call `__del__()` directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Une alternative pour éviter les références cycliques consiste à utiliser le module `weakref`, qui permet de faire référence à des objets sans incrémenter leur compteur de références. Par exemple, les structures d'arbres devraient utiliser des références faibles entre pères et fils (si nécessaire !).

If the object has ever been a local variable in a function that caught an expression in an except clause, chances are that a reference to the object still exists in that function's stack frame as contained in the stack trace. Normally, calling `sys.exc_clear()` will take care of this by clearing the last recorded exception.

Enfin, si la méthode `__del__()` lève une exception, un message d'avertissement s'affiche dans `sys.stderr`.

2.6.12 Comment obtenir toutes les instances d'une classe ?

Python ne tient pas de registre de toutes les instances d'une classe (ni de n'importe quel type natif). Il est cependant possible de programmer le constructeur de la classe de façon à tenir un tel registre, en maintenant une liste de références faibles vers chaque instance.

2.6.13 Pourquoi le résultat de `id()` peut-il être le même pour deux objets différents ?

La fonction native `id()` renvoie un entier dont l'unicité est garantie durant toute la vie de l'objet. Vu qu'en CPython cet entier est en réalité l'adresse mémoire de l'objet, il est fréquent qu'un nouvel objet soit alloué à une adresse mémoire identique à celle d'un objet venant d'être supprimé. Comme l'illustre le code suivant :

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

Les deux identifiants appartiennent à des objets entiers créés juste avant l'appel à `id()` et détruits immédiatement après. Pour s'assurer que les objets dont on veut examiner les identifiants sont toujours en vie, créons une nouvelle référence à l'objet :

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.7 Modules

2.7.1 Comment créer des fichiers .pyc ?

When a module is imported for the first time (or when the source is more recent than the current compiled file) a `.pyc` file containing the compiled code should be created in the same directory as the `.py` file.

One reason that a `.pyc` file may not be created is permissions problems with the directory. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server. Creation of a .pyc file is automatic if you're importing a module and Python has the ability (permissions, free space, etc…) to write the compiled module back to the directory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo`, `xyz.pyc` will be created since `xyz` is imported, but no `foo.pyc` file will be created since `foo.py` isn't being imported.

If you need to create `foo.pyc` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

Le module `py_compile` peut compiler n'importe quel module manuellement. Il est ainsi possible d'appeler la fonction `compile()` de manière interactive :

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

Tous les fichiers d'un ou plusieurs dossiers peuvent aussi être compilés avec le module `compileall`. C'est possible depuis l'invite de commande en exécutant `compileall.py` avec le chemin du dossier contenant les fichiers Python à compiler

```
python -m compileall .
```

2.7.2 Comment obtenir le nom du module actuel ?

Un module peut déterminer son propre nom en examinant la variable globale prédéfinie `__name__`. Si celle-ci vaut `'__main__'`, c'est que le programme est exécuté comme un script. Beaucoup de modules qui doivent normalement être importés pour pouvoir être utilisés fournissent aussi une interface en ligne de commande ou un test automatique. Ils n'exécutent cette portion du code qu'après avoir vérifié la valeur de `__name__` :

```
def main():
    print 'Running test...'
    ...

if __name__ == '__main__':
    main()
```

2.7.3 Comment avoir des modules qui s'importent mutuellement ?

Considérons les modules suivants :

foo.py

```
from bar import bar_var
foo_var = 1
```

bar.py

```
from foo import foo_var
bar_var = 2
```

Le problème réside dans les étapes que l'interpréteur va réaliser :

- *main* importe *foo*
- Les variables globales (vides) de *foo* sont créées
- *foo* est compilé et commence à s'exécuter
- *foo* importe *bar*
- Les variables globales (vides) de *bar* sont créées
- *bar* est compilé et commence à s'exécuter
- *bar* importe *foo* (en réalité, rien ne passe car il y a déjà un module appelé *foo*)
- bar.foo_var = foo.foo_var

La dernière étape échoue car Python n'a pas fini d'interpréter `foo` et le dictionnaire global des symboles de `foo` est encore vide.

Le même phénomène arrive quand on utilise `import foo`, et qu'on essaye ensuite d'accéder à `foo.foo_var` dans le code global.

Il y a (au moins) trois façons de contourner ce problème.

Guido van Rossum déconseille d'utiliser `from <module> import ...` et de mettre tout le code dans des fonctions. L'initialisation des variables globales et des variables de classe ne doit utiliser que des constantes ou des fonctions natives. Ceci implique que tout ce qui est fourni par un module soit référencé par `<module>.<nom>`.

Jim Roskind recommande d'effectuer les étapes suivantes dans cet ordre dans chaque module :

- les exportations (variables globales, fonctions et les classes qui ne nécessitent d'importer des classes de base)
- les instructions `import`
- le code (avec les variables globales qui sont initialisées à partir de valeurs importées).

van Rossum désapprouve cette approche car les importations se trouvent à un endroit bizarre, mais cela fonctionne.

Matthias Urlichs conseille de restructurer le code pour éviter les importations récursives.

Ces solutions peuvent être combinées.

2.7.4 __import__("x.y.z") renvoie <module "x">; comment accéder à z ?

Utilisez plutôt la fonction `import_module()` de `importlib`

```
z = importlib.import_module('x.y.z')
```

2.7.5 Quand j'édite un module et que je le réimporte, je ne vois pas les changements. Pourquoi ?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force rereading of a changed module, do this :

```
import modname
reload(modname)
```

Attention, cette technique ne marche pas systématiquement. En particulier, les modules qui contiennent des instructions comme

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour :

```
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> reload(cls)
<module 'cls' from 'cls.pyc'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

The nature of the problem is made clear if you print out the class objects :

```
>>> c.__class__
<class cls.C at 0x7352a0>
>>> cls.C
<class cls.C at 0x4198d0>
```

CHAPITRE 3

FAQ histoire et design

3.1 Pourquoi Python utilise-t-il l'indentation pour grouper les instructions ?

Guido van Rossum considère que l'usage de l'indentation pour regrouper les blocs d'instruction est élégant et contribue énormément à la clarté globale du programme Python. La plupart des gens finissent par aimer cette particularité au bout d'un moment.

Comme il n'y a pas d'accolades de début/fin, il ne peut y avoir de différence entre le bloc perçu par l'analyseur syntaxique et le lecteur humain. Parfois les programmeurs C pourront trouver un morceau de code comme celui-ci :

```
if (x <= y)
        x++;
        y--;
z++;
```

Seule l'instruction `x++` sera exécutée si la condition est vraie, mais l'indentation pourrait vous faire penser le contraire. Mêmes des développeurs C expérimentés resteront pendant un moment à se demander pourquoi `y` est décrémenté même si `x > y`.

Comme il n'y a pas d'accolades de début/fin, Python est moins sujet aux conflits de style de code. En C, on peut placer les accolades de nombreuses façons. Si vous êtes habitués à lire et écrire selon un style particulier, vous pourriez vous sentir perturbé en lisant (ou en devant écrire) avec un autre style.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

3.2 Pourquoi ai-je d'étranges résultats suite à de simples opérations arithmétiques ?

Voir la question suivante.

3.3 Pourquoi les calculs à virgules flottantes sont si imprécis ?

Les gens sont très souvent surpris par des résultats comme celui-ci :

```
>>> 1.2 - 1.0
0.19999999999999996
```

et pensent que c'est un bogue dans Python. Ça ne l'est pas. Ceci n'a rien à voir avec Python, mais avec la manière dont la plateforme C sous-jacente gère les nombres à virgule flottante et enfin, les imprécisions introduites lors de l'écriture des nombres en chaînes de caractères d'un nombre fixe de chiffres.

La représentation interne des nombres à virgule flottante utilise un nombre fixe de chiffres binaires pour représenter un nombre décimal. Certains nombres décimaux ne peuvent être représentés exactement en binaire, résultant ainsi à de petites erreurs d'arrondi.

En mathématiques, beaucoup de nombre ne peuvent être représentés par un nombre fixe de chiffres, par exemple 1/3 = 0.3333333333…….

En base 2, 1/2 = 0.1, 1/4 = 0.01, 1/8 = 0.001, etc. .2 est égale à 2/10 qui est égale à 1/5, ayant pour résultat le nombre fractionnel binaire 0.0011001100110011001…

Les nombres à virgule flottante ont une précision de seulement 32 ou 64 bits, donc les chiffres finissent par être tronqués, et le nombre résultant est 0.1999999999999999 en décimal, pas 0.2.

La fonction `repr()` d'un nombre décimal affiche autant de chiffres que nécessaire pour rendre l'expression `eval(repr(f)) == f` vraie pour tout nombre décimal f. La fonction `str()` affiche moins de chiffres et correspond généralement plus au nombre attendu

```
>>> 1.1 - 0.9
0.20000000000000007
>>> print 1.1 - 0.9
0.2
```

En conséquence comparer le résultat d'un calcul avec un nombre flottant avec l'opérateur "==" est propice à l'obtention d'erreurs. D'infimes imprécisions peuvent faire qu'un test d'égalité avec « == » échoue. Au lieu de cela, vous devez vérifier que la différence entre les deux chiffres est inférieure à un certain seuil

```
epsilon = 0.0000000000001  # Tiny allowed error
expected_result = 0.4

if expected_result-epsilon <= computation() <= expected_result+epsilon:
    ...
```

Veuillez vous référer au chapitre sur floating point arithmetic du tutoriel python pour de plus amples informations.

3.4 Pourquoi les chaînes de caractères Python sont-elles immuables ?

Il y a plusieurs avantages.

La première concerne la performance : savoir qu'une chaîne de caractères est immuable signifie que l'allocation mémoire allouée lors de la création de cette chaîne est fixe et figée. C'est aussi l'une des raisons pour lesquelles on fait la distinction entre les *tuples* et les listes.

Un autre avantage est que les chaînes en Python sont considérées aussi « élémentaires » que les nombres. Aucun processus ne changera la valeur du nombre 8 en autre chose, et en Python, aucun processus changera la chaîne de caractère « huit » en autre chose.

3.5 Pourquoi « self » doit-il être explicitement utilisé dans les définitions et les appels de méthode ?

L'idée a été empruntée à Modula-3. Il s'avère être très utile, pour diverses raisons.

Tout d'abord, il est plus évident d'utiliser une méthode ou un attribut d'instance par exemple au lieu d'une variable locale. Lire `self.x` ou `self.meth()` est sans ambiguïté sur le fait que c'est une variable d'instance ou une méthode qui est utilisée, même si vous ne connaissez pas la définition de classe par cœur. En C++, vous pouvez les reconnaitre par l'absence d'une déclaration de variable locale (en supposant que les variables globales sont rares ou facilement reconnaissables) – mais en Python, il n'y a pas de déclarations de variables locales, de sorte que vous devez chercher la définition de classe pour être sûr. Certaines normes de programmation C++ et Java préfixent les attributs d'instance par `m_`. Cette syntaxe explicite est ainsi utile également pour ces langages.

Ensuite, ça veut dire qu'aucune syntaxe spéciale n'est nécessaire si vous souhaitez explicitement référencer ou appeler la méthode depuis une classe en particulier. En C++, si vous utilisez la méthode d'une classe de base elle-même surchargée par une classe dérivée, vous devez utiliser l'opérateur `::` – en Python vous pouvez écrire `baseclass.methodname(self, <argument list>)`. C'est particulièrement utile pour les méthodes `__init__()`, et de manière générale dans les cas où une classe dérivée veut étendre la méthode du même nom de la classe de base, devant ainsi appeler la méthode de la classe de base d'une certaine manière.

Enfin, pour des variables d'instance, ça résout un problème syntactique pour l'assignation : puisque les variables locales en Python sont (par définition !) ces variables auxquelles les valeurs sont assignées dans le corps d'une fonction (et n'étant pas déclarées explicitement globales), il doit y avoir un moyen de dire à l'interpréteur qu'une assignation est censée assigner une variable d'instance plutôt qu'une variable locale, et doit de préférence être syntactique (pour des raisons d'efficacité). C++ fait ça au travers de déclarations, mais Python n'a pas de déclarations et ça serait dommage d'avoir à les introduire juste pour cette raison. Utiliser explicitement `self.var` résout ça avec élégance. Pareillement, pour utiliser des variables d'instance, avoir à écrire `self.var` signifie que les références vers des noms non-qualifiés au sein d'une méthode n'ont pas à être cherchés dans l'annuaire d'instances. En d'autres termes, les variables locales et les variables d'instance vivent dans deux différents espaces de noms, et vous devez dire à Python quel espace de noms utiliser.

3.6 Pourquoi ne puis-je pas utiliser d'assignation dans une expression ?

De nombreuses personnes habituées à C ou Perl se plaignent de vouloir utiliser cet idiome C :

```
while (line = readline(f)) {
    // do something with line
}
```

où en Python vous êtes forcé à écrire ceci :

```
while True:
    line = f.readline()
    if not line:
        break
    ...  # do something with line
```

La raison pour ne pas autoriser l'assignation dans les expressions en Python est un bug fréquent, et difficile à trouver dans ces autres langages, causé par cette construction :

```
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

Cette erreur est une simple coquille : `x = 0`, qui assigne 0 à la variable `x`, a été écrit alors que la comparaison `x == 0` est certainement ce qui était souhaité.

De nombreuses alternatives ont été proposées. La plupart économisaient quelques touches mais utilisaient des mots clefs ou des syntaxes arbitraires ou cryptiques, et manquaient à la règle que toute proposition de changement de langage devrait respecter : elle doit intuitivement suggérer la bonne signification au lecteur qui n'a pas encore été introduit à cette syntaxe.

Un phénomène intéressant est que la plupart des programmeurs Python expérimentés reconnaissent l'idiome `while True` et ne semblent pas manquer l'assignation dans la construction de l'expression ; seuls les nouveaux-venus expriment un fort désir d'ajouter ceci au langage.

Il y a une manière alternative de faire ça qui semble attrayante mais elle est généralement moins robuste que la solution `while True` :

```
line = f.readline()
while line:
    ...  # do something with line...
    line = f.readline()
```

Le problème avec ceci est que si vous changez d'avis sur la manière dont vous allez récupérer la prochaine ligne (ex : vous voulez changer en `sys.stdin.readline()`) vous devez vous souvenir de le changer à deux endroits dans votre programme – la deuxième occurrence est cachée en bas de la boucle.

La meilleur approche est d'utiliser les itérateurs, rendant possible de boucler au travers d'objets en utilisant la déclaration `for`. Par exemple, dans la version actuelle de Python, les fichiers objets supportent le protocole d'itérateur, vous pouvez alors simplement écrire :

```
for line in f:
    ...  # do something with line...
```

3.7 Pourquoi Python utilise des méthodes pour certaines fonctionnalités (ex : `list.index()`) mais des fonctions pour d'autres (ex : `len(list)`) ?

As Guido said :

> (a) For some operations, prefix notation just reads better than postfix – prefix (and infix !) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the easy with which we rewrite a formula like x*(a+b) into x*a + x*b to the clumsiness of doing the same thing using a raw OO notation.
>
> (b) When I read code that says len(x) I *know* that it is asking for the length of something. This tells me two things : the result is an integer, and the argument is some kind of container. To the contrary, when I read x.len(), I have to already know that x is some kind of container implementing an interface or inheriting from a class that has a standard len(). Witness the confusion we occasionally have when a class that is not implementing a mapping has a get() or keys() method, or something that isn't a file has a write() method.
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 Pourquoi `join()` est une méthode de chaîne plutôt qu'une de liste ou de tuple ?

Les chaînes sont devenues bien plus comme d'autres types standards à partir de Python 1.6, lorsque les méthodes ont été ajoutées fournissant ainsi les mêmes fonctionnalités que celles qui étaient déjà disponibles en utilisant les fonctions du module string. La plupart de ces nouvelles méthodes ont été largement acceptées, mais celle qui semble rendre certains programmeurs inconfortables est :

```
", ".join(['1', '2', '4', '8', '16'])
```

qui donne le résultat :

```
"1, 2, 4, 8, 16"
```

Il y a deux arguments fréquents contre cet usage.

Le premier se caractérise par les lignes suivantes : « C'est vraiment moche d'utiliser une méthode de chaîne littérale (chaîne constante) », à laquelle la réponse est qu'il se peut, mais une chaîne littérale est juste une valeur fixe. Si la méthode est autorisée sur des noms liés à des chaînes, il n'y a pas de raison logique à les rendre indisponibles sur des chaînes littérales.

La deuxième objection se réfère typiquement à : « Je suis réellement en train de dire à une séquence de joindre ses membres avec une constante de chaîne ». Malheureusement, vous ne l'êtes pas. Pour quelque raison, il semble être bien moins difficile d'avoir `split()` en tant que méthode de chaîne, puisque dans ce cas il est facile de voir que :

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space). In this case a Unicode string returns a list of Unicode strings, an ASCII string returns a list of ASCII strings, and everyone is happy.

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself.

Because this is a string method it can work for Unicode strings as well as plain ASCII strings. If `join()` were a method of the sequence types then the sequence types would have to decide which type of string to return depending on the type of the separator.

If none of these arguments persuade you, then for the moment you can continue to use the `join()` function from the string module, which allows you to write

```
string.join(['1', '2', '4', '8', '16'], ", ")
```

3.9 À quel point les exceptions sont-elles rapides ?

Un bloc `try` / `except` est extrêmement efficient tant qu'aucune exception ne sont levée. En effet, intercepter une exception s'avère coûteux. Dans les versions de précédant Python 2.0, il était courant d'utiliser cette pratique :

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

Cela n'a de sens que si vous vous attendez à ce que le dictionnaire ait la clé presque tout le temps. Si ce n'était pas le cas, vous l'auriez codé comme suit :

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

Note : In Python 2.0 and higher, you can code this as `value = mydict.setdefault(key, getvalue(key))`.

3.10 Pourquoi n'y a-t-il pas une instruction *switch* ou une structure similaire à *switch / case* en Python ?

Vous pouvez le faire assez facilement avec une séquence de `if... elif... elif... else`. Il y a eu quelques propositions pour la syntaxe de l'instruction `switch`, mais il n'y a pas (encore) de consensus sur le cas des intervalles. Voir la **PEP 275** pour tous les détails et l'état actuel.

Dans les cas où vous devez choisir parmi un très grand nombre de possibilités, vous pouvez créer un dictionnaire faisant correspondre des valeurs à des fonctions à appeler. Par exemple :

```
def function_1(...):
    ...

functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1, ...}

func = functions[value]
func()
```

Pour appeler les méthodes sur des objets, vous pouvez simplifier davantage en utilisant la fonction native `getattr()` pour récupérer les méthodes avec un nom donné :

```
def visit_a(self, ...):
    ...
...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

Il est suggéré que vous utilisiez un préfixe pour les noms de méthodes, telles que `visit_` dans cet exemple. Sans ce préfixe, si les valeurs proviennent d'une source non fiable, un attaquant serait en mesure d'appeler n'importe quelle méthode sur votre objet.

3.11 Est-il possible d'émuler des fils d'exécution dans l'interpréteur plutôt que se baser sur les implémentations spécifique aux OS ?

Réponse 1 : Malheureusement, l'interpréteur pousse au moins un bloc de pile C (*stack frame*) pour chaque bloc de pile de Python. Aussi, les extensions peuvent rappeler dans Python à presque n'importe quel moment. Par conséquent, une implémentation complète des fils d'exécution nécessiterai un support complet en C.

Réponse 2 : Heureusement, il existe Stackless Python, qui à complètement ré-architecturé la boucle principale de l'interpréteur afin de ne pas utiliser la pile C.

3.12 Pourquoi les expressions lambda ne peuvent pas contenir d'instructions ?

Les expressions lambda de Python ne peuvent pas contenir d'instructions parce que le cadre syntaxique de Python ne peut pas gérer les instructions imbriquées à l'intérieur d'expressions. Cependant, en Python, ce n'est pas vraiment un problème. Contrairement aux formes lambda dans d'autres langages, où elles ajoutent des fonctionnalités, les expressions lambda de Python sont seulement une notation concise si vous êtes trop paresseux pour définir une fonction.

Les fonctions sont déjà des objets de première classe en Python et peuvent être déclarées dans une portée locale. L'unique avantage d'utiliser une fonction lambda au lieu d'une fonction définie localement est que vous n'avez nullement besoin d'un nom pour la fonction – Mais c'est juste une variable locale à laquelle est affecté l'objet fonction (qui est exactement le même type d'objet qui donne une expression lambda) !

3.13 Python peut-il être compilé en code machine, en C ou dans un autre langage ?

Cython compiles a modified version of Python with optional annotations into C extensions. Nuitka is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language. For compiling to Java you can consider VOC.

3.14 Comment Python gère la mémoire ?

The details of Python memory management depend on the implementation. The standard C implementation of Python uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Jython relies on the Java runtime so the JVM's garbage collector is used. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Sometimes objects get stuck in tracebacks temporarily and hence are not deallocated when you might expect. Clear the tracebacks with :

```
import sys
sys.exc_clear()
sys.exc_traceback = sys.last_traceback = None
```

Tracebacks are used for reporting errors, implementing debuggers and related things. They contain a portion of the program state extracted during the handling of an exception (usually the most recent exception).

In the absence of circularities and tracebacks, Python programs do not need to manage memory explicitly.

Why doesn't Python use a more traditional garbage collection scheme ? For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent ; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard malloc() and free() with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for malloc() and free(), and may not want Python's. Right now, Python works with anything that implements malloc() and free() properly.

In Jython, the following code (which is fine in CPython) will probably run out of file descriptors long before it runs out of memory :

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Using the current reference counting and destructor scheme, each new assignment to f closes the previous file. Using GC, this is not guaranteed. If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement ; this will work regardless of GC :

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 Why isn't all memory freed when Python exits ?

Les objets référencés depuis les espaces de noms globaux des modules Python ne sont pas toujours désalloués lorsque Python s'arrête. Cela peut se produire s'il y a des références circulaires. Il y a aussi certaines parties de mémoire qui sont alloués par la bibliothèque C qui sont impossibles à libérer (par exemple un outil comme *Purify* s'en plaindra). Python est, cependant, agressif sur le nettoyage de la mémoire en quittant et cherche à détruire chaque objet.

Si vous voulez forcer Python à désallouer certains objets en quittant, utilisez le module `texit` pour exécuter une fonction qui va forcer ces destructions.

3.16 Pourquoi les *tuples* et les *list* sont deux types de données séparés ?

Les listes et les *tuples*, bien que semblable à bien des égards, sont généralement utilisés de façons fondamentalement différentes. Les *tuples* peuvent être considérés comme étant similaires aux dossiers en Pascal ou aux structures en C ; Ce sont de petites collections de données associées qui peuvent être de différents types qui sont utilisées ensemble. Par exemple, un repère cartésien est correctement représenté comme un *tuple* de deux ou trois nombres.

Les listes, ressemblent davantage à des tableaux dans d'autres langues. Elles ont tendance à contenir un nombre variable d'objets de même type manipulés individuellement. Par exemple, `os.listdir('.')` renvoie une liste de chaînes représentant les fichiers dans le dossier courant. Les fonctions travaillant sur cette sortie accepteraient généralement sans aucun problème que vous ajoutiez un ou deux fichiers supplémentaire dans le dossier.

Les *tuples* sont immuables, ce qui signifie que lorsqu'un *tuple* a été créé, vous ne pouvez remplacer aucun de ses éléments par une nouvelle valeur. Les listes sont muables, ce qui signifie que vous pouvez toujours modifier les éléments d'une liste. Seuls des éléments immuables peuvent être utilisés comme clés de dictionnaires, et donc de `tuple` et `list` seul des *tuples* peuvent être utilisés comme clés.

3.17 How are lists implemented in CPython ?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

Cela rend l'indexation d'une liste `a[i]` une opération dont le coût est indépendant de la taille de la liste ou de la valeur de l'indice.

Lorsque des éléments sont ajoutés ou insérés, le tableau de références est redimensionné. Un savoir-faire ingénieux permet l'amélioration des performances lors de l'ajout fréquent d'éléments ; Lorsque le tableau doit être étendu, un certain espace supplémentaire est alloué de sorte que pour la prochaine fois, ceci ne nécessite plus un redimensionnement effectif.

3.18 How are dictionaries implemented in CPython ?

CPython's dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key ; for example, « Python » hashes to -539294296 while « python », a string that differs by a single bit, hashes to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time – O(1), in computer science notation – to retrieve a key. It also means that no sorted

order of the keys is maintained, and traversing the array as the `.keys()` and `.items()` do will output the dictionary's content in some arbitrary jumbled order.

3.19 Pourquoi les clés du dictionnaire sont immuables ?

L'implémentation de la table de hachage des dictionnaires utilise une valeur de hachage calculée à partir de la valeur de la clé pour trouver la clé elle-même. Si la clé était un objet muable, sa valeur peut changer, et donc son hachage pourrait également changer. Mais toute personne modifiant l'objet clé ne peut pas dire qu'elle a été utilisée comme une clé de dictionnaire. Il ne peut déplacer l'entrée dans le dictionnaire. Ainsi, lorsque vous essayez de rechercher le même objet dans le dictionnaire, il ne sera pas disponible parce que sa valeur de hachage est différente. Si vous essayez de chercher l'ancienne valeur, elle serait également introuvable car la valeur de l'objet trouvé dans cet emplacement de hachage serait différente.

Si vous voulez un dictionnaire indexé avec une liste, il faut simplement convertir la liste en un *tuple* ; la fonction `tuple(L)` crée un *tuple* avec les mêmes entrées que la liste `L`. Les *tuples* sont immuables et peuvent donc être utilisés comme clés du dictionnaire.

Certaines solutions insatisfaisantes qui ont été proposées :

— Les listes de hachage par leur adresse (*ID* de l'objet). Cela ne fonctionne pas parce que si vous créez une nouvelle liste avec la même valeur, elle ne sera pas retrouvée ; par exemple :

```
mydict = {[1, 2]: '12'}
print mydict[[1, 2]]
```

cela lèverait une exception de type *KeyError* car l'ID de `[1, 2]` utilisé dans la deuxième ligne diffère de celle de la première ligne. En d'autres termes, les clés de dictionnaire doivent être comparées à l'aide du comparateur == et non à l'aide du mot clé `is`.

— Faire une copie lors de l'utilisation d'une liste en tant que clé. Cela ne fonctionne pas puisque la liste, étant un objet muable, pourrait contenir une référence à elle-même ou avoir une boucle infinie au niveau du code copié.

— Autoriser les listes en tant que clés, mais indiquer à l'utilisateur de ne pas les modifier. Cela permettrait un ensemble de bogues difficiles à suivre dans les programmes lorsque vous avez oublié ou modifié une liste par accident. Cela casse également un impératif important des dictionnaires : chaque valeur de `d.keys()` est utilisable comme clé du dictionnaire.

— Marquer les listes comme étant en lecture seule une fois qu'elles sont utilisées comme clé de dictionnaire. Le problème est que ce n'est pas seulement l'objet de niveau supérieur qui pourrait changer sa valeur ; vous pourriez utiliser un tuple contenant une liste comme clé. Utiliser n'importe quoi comme une clé dans un dictionnaire nécessiterait de marquer tous les objets accessibles à partir de là comme en lecture seule – et encore une fois, les objets se faisant référence pourraient provoquer une boucle infinie.

Il y a un truc pour contourner ceci si vous en avez besoin, mais utilisez-le à vos risques et périls. Vous pouvez encapsuler une structure mutable à l'intérieur d'une instance de classe qui a à la fois une méthode `__eq__()` et `__hash__()`. Vous devez ensuite vous assurer que la valeur de hachage pour tous ces objets *wrapper* qui résident dans un dictionnaire (ou une autre structure basée sur le hachage), restent fixes pendant que l'objet est dans le dictionnaire (ou une autre structure). :

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
```

(suite sur la page suivante)

(suite de la page précédente)

```
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Notez que le calcul de hachage peut être compliqué car il est possible que certains membres de la liste peuvent être impossible à hacher et aussi par la possibilité de débordement arithmétique.

De plus, il faut toujours que, si `o1 == o2` (par exemple `o1.__eq__(o2) vaut True`) alors `hash(o1) == hash(o2)` (par exemple, `o1.__hash__() == o2.__hash__()`), que l'objet se trouve dans un dictionnaire ou pas. Si vous ne remplissez pas ces conditions, les dictionnaires et autres structures basées sur le hachage se comporteront mal.

Dans le cas de *ListWrapper*, chaque fois que l'objet *wrapper* est dans un dictionnaire, la liste encapsulée ne doit pas changer pour éviter les anomalies. Ne faites pas cela à moins que vous n'ayez pensé aux potentielles conséquences de ne pas satisfaire entièrement ces conditions. Vous avez été prévenus.

3.20 Pourquoi `list.sort()` ne renvoie pas la liste triée ?

Dans les situations où la performance est importante, faire une copie de la liste juste pour la trier serait un gaspillage. Par conséquent, `list.sort()` trie la liste en place. Afin de vous le rappeler, il ne retourne pas la liste triée. De cette façon, vous ne serez pas dupés en écrasant accidentellement une liste lorsque vous avez besoin d'une copie triée, mais vous devrez également garder sous la main la version non triée.

In Python 2.4 a new built-in function – `sorted()` – has been added. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order :

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

3.21 Comment spécifiez-vous et appliquez-vous une spécification d'interface en Python ?

Une spécification d'interface pour un module fourni par des langages tels que C++ et Java décrit les prototypes pour les méthodes et les fonctions du module. Beaucoup estiment que la vérification au moment de la compilation des spécifications d'interface aide à la construction de grands programmes.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

Pour Python, la plupart des avantages des spécifications d'interface peuvent être obtenus par une discipline de test appropriée pour les composants. Il existe aussi un outil, PyChecker, qui peut être utilisé pour trouver des problèmes d'héritage.

Une bonne suite de tests pour un module peut à la fois fournir un test de non régression et servir de spécification d'interface de module ainsi qu'un ensemble d'exemples. De nombreux modules Python peuvent être exécutés en tant que script pour fournir un simple « auto-test ». Même les modules qui utilisent des interfaces externes complexes peuvent souvent être testés isolément à l'aide d'émulations triviales embryonnaires de l'interface externe. Les modules `doctest` et `UnitTest` ou des frameworks de test tiers peuvent être utilisés pour construire des suites de tests exhaustives qui éprouvent chaque ligne de code dans un module.

Une discipline de test appropriée peut aider à construire des applications complexes de grande taille en Python aussi bien que le feraient des spécifications d'interface. En fait, c'est peut être même mieux parce qu'une spécification d'interface ne peut pas tester certaines propriétés d'un programme. Par exemple, la méthode `Append()` est censée ajouter de nouveaux éléments à la fin d'une liste « sur place » ; une spécification d'interface ne peut pas tester que votre implémentation de `append()` va réellement le faire correctement, mais il est trivial de vérifier cette propriété dans une suite de tests.

L'écriture des suites de tests est très utile, et vous voudrez peut-être concevoir votre code de manière à le rendre facilement testable. Une technique de plus en plus populaire, le développement dirigé par les tests, requiert d'écrire d'abord des éléments de la suite de tests, avant d'écrire le code réel. Bien sûr, Python vous permet d'être laxiste et de ne pas écrire de test du tout.

3.22 Pourquoi n'y a-t-il pas de `goto` en Python ?

Vous pouvez utiliser les exceptions afin de mettre en place un « `goto` structuré » qui fonctionne même avec les appels de fonctions. Beaucoup de personnes estiment que les exceptions peuvent émuler idéalement tout utilisation raisonnable des constructions `go` ou `goto` en C, en Fortran ou autres langages de programmation. Par exemple :

```
class label: pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

Cela ne vous permet pas de sauter au milieu d'une boucle. Néanmoins, dans tous les cas cela est généralement considéré comme un abus de `goto`. À Utiliser avec parcimonie.

3.23 Pourquoi les chaînes de caractères brutes (r-strings) ne peuvent-elles pas se terminer par un *backslash* ?

Plus précisément, elles ne peuvent pas se terminer par un nombre impair de *backslashes* : le *backslash* non appairé à la fin échappe le caractère de guillemet final, laissant une chaîne non terminée.

Les chaînes brutes ont été conçues pour faciliter la création de données pour les processeurs de texte (principalement les moteurs d'expressions régulières) qui veulent faire leur propre traitement d'échappement d'"*antislashes*. Ces processeurs considèrent un *antislash* de fin non-appairé comme une erreur, alors les chaînes brutes ne le permettent pas. En retour, elles vous permettent de transmettre le caractère de citation de la chaîne en l'échappant avec un *antislash*. Ces règles fonctionnent bien lorsque les chaînes brutes sont utilisées pour leur but premier.

Si vous essayez de construire des chemins d'accès Windows, notez que tous les appels système Windows acceptent également les *slashes* « classiques » :

```
f = open("/mydir/file.txt")  # works fine!
```

Si vous essayez de construire un chemin d'accès pour une commande DOS, essayez par exemple l'un de ceux-là :

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.24 Pourquoi la déclaration `with` pour les assignations d'attributs n'existe pas en Python ?

Python a une instruction `with` qui encapsule l'exécution d'un bloc, en appelant le code sur l'entrée et la sortie du bloc. Certains langages possèdent une construction qui ressemble à ceci :

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

En Python, une telle construction serait ambiguë.

Les autres langages, tels que le Pascal, le Delphi et le C++ utilisent des types statiques, il est donc possible de savoir d'une manière claire et directe ce à quoi est attribué un membre. C'est le point principal du typage statique –le compilateur connaît *toujours* la portée de toutes les variables au moment de la compilation.

Python utilise le typage dynamique. Il est impossible de savoir à l'avance quel attribut est utilisé comme référence lors de l'exécution. Les attributs membres peuvent être ajoutés ou retirés des objets à la volée. Il est donc impossible de savoir, d'une simple lecture, quel attribut est référencé : s'il est local, global ou un attribut membre ?

Prenons par exemple l'extrait incomplet suivant :

```
def foo(a):
    with a:
        print x
```

L'extrait suppose que « a » doit avoir un attribut membre appelé « x ». Néanmoins, il n'y a rien en Python qui en informe l'interpréteur. Que se passe-t-il si « a » est, disons, un entier ? Si une variable globale nommée « x » existe, sera-t-elle utilisée dans le bloc `with` ? Comme vous voyez, la nature dynamique du Python rend ces choix beaucoup plus difficiles.

L'avantage principal de `with` et des fonctionnalités de langage similaires (réduction du volume de code) peut, cependant, être facilement réalisé en Python par assignation. Au lieu de :

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

écrivez ceci :

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

Cela a également pour effet secondaire d'augmenter la vitesse d'exécution car les liaisons de noms sont résolues au moment de l'exécution en Python, et la deuxième version n'a besoin d'exécuter la résolution qu'une seule fois.

3.25 Pourquoi les deux-points sont-ils nécessaires pour les déclarations `if/while/def/class` ?

Le deux-points est principalement nécessaires pour améliorer la lisibilité (l'un des résultats du langage expérimental ABC). Considérez ceci :

```
if a == b
    print a
```

versus :

```
if a == b:
    print a
```

Remarquez comment le deuxième est un peu plus facile à lire. Remarquez aussi comment un deux-points introduit l'exemple dans cette réponse à la FAQ ; c'est un usage standard en anglais.

Une autre raison mineure est que les deux-points facilitent la tâche des éditeurs avec coloration syntaxique ; ils peuvent rechercher les deux-points pour décider quand l'indentation doit être augmentée au lieu d'avoir à faire une analyse plus élaborée du texte du programme.

3.26 Pourquoi Python permet-il les virgules à la fin des listes et des tuples ?

Python vous permet d'ajouter une virgule à la fin des listes, des tuples et des dictionnaires :

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

Il y a plusieurs raisons d'accepter cela.

Lorsque vous avez une valeur littérale pour une liste, un tuple ou un dictionnaire réparti sur plusieurs lignes, il est plus facile d'ajouter plus d'éléments parce que vous n'avez pas besoin de vous rappeler d'ajouter une virgule à la ligne précédente. Les lignes peuvent aussi être réorganisées sans créer une erreur de syntaxe.

L'omission accidentelle de la virgule peut entraîner des erreurs difficiles à diagnostiquer, par exemple :

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

Cette liste a l'air d'avoir quatre éléments, mais elle en contient en fait trois : « *fee* », « *fiefoo* » et « *fum* ». Toujours ajouter la virgule permet d'éviter cette source d'erreur.

Permettre la virgule de fin peut également faciliter la génération de code.

CHAPITRE 4

FAQ sur la bibliothèque et les extension

4.1 Questions générales sur la bibliothèque

4.1.1 Comment puis-je trouver un module ou une application pour exécuter la tâche X ?

Regardez si la bibliothèque standard contient un module approprié (avec l'expérience, vous connaitrez le contenu de la bibliothèque standard et pourrez sauter cette étape).

For third-party packages, search the Python Package Index or try Google or another Web search engine. Searching for « Python » plus a keyword or two for your topic of interest will usually find something helpful.

4.1.2 Où se situe le fichier source *math.py* (*socket.py*, *regex.py*, etc.) ?

Si vous ne parvenez pas à trouver le fichier source d'un module, c'est peut-être parce que celui-ci est un module natif ou bien un module implémenté en C, C++, ou autre langage compilé, qui est chargé dynamiquement. Dans ce cas, vous ne possédez peut-être pas le fichier source ou celui-ci est en réalité stocké quelque part dans un dossier de fichiers source C (qui ne sera pas dans le chemin Python), comme par exemple `mathmodule.c`.

Il y a (au moins) trois types de modules dans Python

1) modules écrits en Python (*.py*) ;
2) modules écrits en C et chargés dynamiquement (*.dll*, *.pyd*, *.so*, *.sl*, *.etc*) ;
3) les modules écrits en C et liés à l'interpréteur ; pour obtenir leur liste, entrez :

```
import sys
print sys.builtin_module_names
```

4.1.3 Comment rendre un script Python exécutable sous Unix ?

Deux conditions doivent être remplies : les droits d'accès au fichier doivent permettre son exécution et la première ligne du script doit commencer par `#!` suivi du chemin vers l'interpréteur Python.

La première condition est remplie en exécutant `chmod +x scriptfile` ou `chmod 755 scriptfile`.

Il y a plusieurs façons de remplir la seconde. La plus simple consiste à écrire au tout début du fichier

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

en utilisant le chemin de l'interpréteur Python sur votre machine.

Pour rendre ce script indépendant de la localisation de l'interpréteur Python, il faut utiliser le programme **`env`**. La ligne ci-dessous fonctionne sur la quasi-totalité des dérivés de Unix, à condition que l'interpréteur Python soit dans un dossier référencé dans la variable `PATH` de l'utilisateur :

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

Ne faites *pas* ceci pour des scripts CGI. La variable `PATH` des scripts CGI est souvent très succincte, il faut par conséquent préciser le chemin absolu réel de l'interpréteur.

Occasionally, a user's environment is so full that the **`/usr/bin/env`** program fails ; or there's no env program at all. In that case, you can try the following hack (due to Alex Rezinsky) :

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

Le léger inconvénient est que cela définit la variable *__doc__* du script. Cependant, il est possible de corriger cela en ajoutant

```
__doc__ = """...Whatever..."""
```

4.1.4 Existe-t'il un module *curse* ou *termcap* en Python ?

For Unix variants the standard Python source distribution comes with a curses module in the Modules subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution – there is no curses module for Windows.)

Le module `curses` comprend les fonctionnalités de base de *curses* et beaucoup de fonctionnalités supplémentaires provenant de *ncurses* et de *SYSV curses* comme la couleur, la gestion des ensembles de caractères alternatifs, la prise en charge du pavé tactile et de la souris. Cela implique que le module n'est pas compatible avec des systèmes d'exploitation qui n'ont que le *curses* de BSD mais, de nos jours, de tels systèmes d'exploitation ne semblent plus exister ou être maintenus.

Pour Windows : utilisez le module consolelib.

4.1.5 Existe-t'il un équivalent à la fonction C `onexit()` en Python ?

Le module `atexit` fournit une fonction d'enregistrement similaire à la fonction C `onexit()`.

4.1.6 Pourquoi mes gestionnaires de signaux ne fonctionnent-t'ils pas ?

Le problème le plus courant est d'appeler le gestionnaire de signaux avec les mauvais arguments. Un gestionnaire est appelé de la façon suivante

```
handler(signum, frame)
```

donc il doit être déclaré avec deux paramètres :

```
def handler(signum, frame):
    ...
```

4.2 Tâches fréquentes

4.2.1 Comment tester un programme ou un composant Python ?

Python fournit deux cadriciels de test. Le module `doctest` cherche des exemples dans les *docstrings* d'un module et les exécute. Il compare alors la sortie avec la sortie attendue, telle que définie dans la *docstring*.

Le module `unittest` est un cadriciel un peu plus élaboré basé sur les cadriciels de test de Java et de Smalltalk.

Pour rendre le test plus aisé, il est nécessaire de bien découper le code d'un programme. Votre programme doit avoir la quasi-totalité des fonctionnalités dans des fonctions ou des classes — et ceci a parfois l'avantage aussi plaisant qu'inattendu de rendre le programme plus rapide, les accès aux variables locales étant en effet plus rapides que les accès aux variables globales. De plus le programme doit éviter au maximum de manipuler des variables globales, car ceci rend le test beaucoup plus difficile.

La « logique générale » d'un programme devrait être aussi simple que

```
if __name__ == "__main__":
    main_logic()
```

à la fin du module principal du programme.

Une fois que la logique du programme est implémentée par un ensemble de fonctions et de comportements de classes, il faut écrire des fonctions de test qui vont éprouver cette logique. À chaque module, il est possible d'associer une suite de tests qui joue de manière automatique un ensemble de tests. Au premier abord, il semble qu'il faille fournir un effort conséquent, mais comme Python est un langage concis et flexible, c'est surprenamment aisé. Écrire simultanément le code « de production » et les fonctions de test associées rend le développement plus agréable et plus amusant, car ceci permet de trouver des bogues, voire des défauts de conception, plus facilement.

Les « modules auxiliaires » qui n'ont pas vocation à être le module principal du programme peuvent inclure un test pour se vérifier eux-mêmes.

```
if __name__ == "__main__":
    self_test()
```

Les programmes qui interagissent avec des interfaces externes complexes peuvent être testés même quand ces interfaces ne sont pas disponibles, en utilisant des interfaces « simulacres » implémentées en Python.

4.2.2 Comment générer la documentation à partir des *docstrings* ?

Le module `pydoc` peut générer du HTML à partir des *docstrings* du code source Python. Il est aussi possible de documenter une API uniquement à partir des *docstrings* à l'aide de epydoc. Sphinx peut également inclure du contenu provenant de *docstrings*.

4.2.3 Comment détecter qu'une touche est pressée ?

For Unix variants there are several solutions. It's straightforward to do this using curses, but curses is a fairly large module to learn. Here's a solution without curses :

```
import termios, fcntl, sys, os
fd = sys.stdin.fileno()

oldterm = termios.tcgetattr(fd)
newattr = termios.tcgetattr(fd)
newattr[3] = newattr[3] & ~termios.ICANON & ~termios.ECHO
termios.tcsetattr(fd, termios.TCSANOW, newattr)

oldflags = fcntl.fcntl(fd, fcntl.F_GETFL)
fcntl.fcntl(fd, fcntl.F_SETFL, oldflags | os.O_NONBLOCK)

try:
    while 1:
        try:
            c = sys.stdin.read(1)
            print "Got character", repr(c)
        except IOError: pass
finally:
    termios.tcsetattr(fd, termios.TCSAFLUSH, oldterm)
    fcntl.fcntl(fd, fcntl.F_SETFL, oldflags)
```

You need the `termios` and the `fcntl` module for any of this to work, and I've only tried it on Linux, though it should work elsewhere. In this code, characters are read and printed one at a time.

`termios.tcsetattr()` turns off stdin's echoing and disables canonical mode. `fcntl.fnctl()` is used to obtain stdin's file descriptor flags and modify them for non-blocking mode. Since reading stdin when it is empty results in an `IOError`, this error is caught and ignored.

4.3 Fils d'exécution

4.3.1 Comment programmer avec des fils d'exécution ?

Be sure to use the `threading` module and not the `thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `thread` module.

Un ensemble de diapositives issues du didacticiel de Aahz sur les fils d'exécution est disponible à http://www.pythoncraft.com/OSCON2001/.

4.3.2 Aucun de mes fils ne semble s'exécuter : pourquoi ?

Dès que le fil d'exécution principal se termine, tous les fils sont tués. Le fil principal s'exécute trop rapidement, sans laisser le temps aux autres fils de faire quoi que ce soit.

Une correction simple consiste à ajouter un temps d'attente suffisamment long à la fin du programme pour que tous les fils puissent se terminer :

```
import threading, time

def thread_task(name, n):
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <----------------------------!
```

Mais à présent, sur beaucoup de plates-formes, les fils ne s'exécutent pas en parallèle, mais semblent s'exécuter de manière séquentielle, l'un après l'autre ! En réalité, l'ordonnanceur de fils du système d'exploitation ne démarre pas de nouveau fil avant que le précédent ne soit bloqué.

Une correction simple consiste à ajouter un petit temps d'attente au début de la fonction :

```
def thread_task(name, n):
    time.sleep(0.001) # <-----------------------!
    for i in range(n): print name, i

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `Queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

4.3.3 Comment découper et répartir une tâche au sein d'un ensemble de fils d'exécutions ?

Use the `Queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Voici un exemple trivial :

```
import threading, Queue, time

# The worker thread gets jobs off the queue.  When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print 'Running worker'
    time.sleep(0.1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    while True:
        try:
            arg = q.get(block=False)
        except Queue.Empty:
            print 'Worker', threading.currentThread(),
            print 'queue empty'
            break
        else:
            print 'Worker', threading.currentThread(),
            print 'running with argument', arg
            time.sleep(0.5)

# Create queue
q = Queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print 'Main thread sleeping'
time.sleep(5)
```

Quand celui-ci est exécuté, il produit la sortie suivante :

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started)> running with argument 0
Worker <Thread(worker 2, started)> running with argument 1
Worker <Thread(worker 3, started)> running with argument 2
Worker <Thread(worker 4, started)> running with argument 3
Worker <Thread(worker 5, started)> running with argument 4
Worker <Thread(worker 1, started)> running with argument 5
...
```

Consult the module's documentation for more details ; the `Queue` class provides a featureful interface.

4.3.4 Quels types de mutations sur des variables globales sont compatibles avec les programmes à fils d'exécution multiples ? sécurisé ?

A *global interpreter lock* (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions ; how frequently it switches can be set via `sys.setcheckinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

En théorie, cela veut dire qu'un décompte exact nécessite une connaissance parfaite de l'implémentation de la MVP. En pratique, cela veut dire que les opérations sur des variables partagées de type natif (les entier, les listes, les dictionnaires etc.) qui « semblent atomiques » le sont réellement.

Par exemple, les opérations suivantes sont toutes atomiques (*L*, *L1* et *L2* sont des listes, *D*, *D1* et *D2* sont des dictionnaires, *x* et *y* sont des objets, *i* et *j* des entiers) :

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

Les suivantes ne le sont pas :

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Les opérations qui remplacent d'autres objets peuvent invoquer la méthode `__del__()` de ces objets quand leur compteur de référence passe à zéro, et cela peut avoir de l'impact. C'est tout particulièrement vrai pour les des changements massifs sur des dictionnaires ou des listes. En cas de doute, il vaut mieux utiliser un mutex.

4.3.5 Pourquoi ne pas se débarrasser du verrou global de l'interpréteur ?

Le *verrou global de l'interpréteur* (GIL) est souvent vu comme un obstacle au déploiement de code Python sur des serveurs puissants avec de nombreux processeurs, car un programme Python à fils d'exécutions multiples n'utilise en réalité qu'un seul processeur. Presque tout le code Python ne peut en effet être exécuté qu'avec le GIL acquis.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the « free threading » patches) that removed the GIL and replaced it with fine-grained locking. Unfortunately, even on Windows (where locks are very efficient) this ran ordinary Python code about twice as slow as the interpreter using the GIL. On Linux the performance loss was even worse because pthread locks aren't as efficient.

Since then, the idea of getting rid of the GIL has occasionally come up but nobody has found a way to deal with the expected slowdown, and users who don't use threads would not be happy if their code ran at half the speed. Greg's free threading patch set has not been kept up-to-date for later Python versions.

This doesn't mean that you can't make good use of Python on multi-CPU machines ! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. Judicious use of C extensions will also help ; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done.

On a déjà proposé de restreindre le GIL par interpréteur, et non plus d'être complétement global ; les interpréteurs ne seraient plus en mesure de partager des objets. Malheureusement, cela n'a pas beaucoup de chance non plus d'arriver. Cela nécessiterait un travail considérable, car la façon dont beaucoup d'objets sont implémentés rend leur état global. Par exemple, les entiers et les chaînes de caractères courts sont mis en cache ; ces caches devraient être déplacés au niveau de l'interpréteur. D'autres objets ont leur propre liste de suppression, ces listes devraient être déplacées au niveau de l'interpréteur et ainsi de suite.

C'est une tâche sans fin, car les extensions tierces ont le même problème, et il est probable que les extensions tierces soient développées plus vite qu'il ne soit possible de les corriger pour les faire stocker leur état au niveau de l'interpréteur et non plus au niveau global.

Et enfin, quel intérêt y-a t'il à avoir plusieurs interpréteurs qui ne partagent pas d'état, par rapport à faire tourner chaque interpréteur dans un processus différent ?

4.4 Les entrées/sorties

4.4.1 Comment supprimer un fichier ? (et autres questions sur les fichiers...)

Use `os.remove(filename)` or `os.unlink(filename)` ; for documentation, see the `os` module. The two functions are identical ; `unlink()` is simply the name of the Unix system call for this function.

Utilisez `os.rmdir()` pour supprimer un dossier et `os.mkdir()` pour en créer un nouveau. `os.makedirs(chemin)` crée les dossiers intermédiaires de `chemin` qui n'existent pas et `os.removedirs(chemin)` supprime les dossiers intermédiaires si ceux-ci sont vides. Pour supprimer une arborescence et tout son contenu, utilisez `shutil.rmtree()`.

`os.rename(ancien_chemin, nouveau_chemin)` permet de renommer un fichier.

To truncate a file, open it using `f = open(filename, "r+")`, and use `f.truncate(offset)` ; offset defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where *fd* is the file descriptor (a small integer).

Le module `shutil` propose aussi un grand nombre de fonctions pour effectuer des opérations sur des fichiers comme `copyfile()`, `copytree()` et `rmtree()`.

4.4.2 Comment copier un fichier ?

Le module `shutil` fournit la fonction `copyfile()`. Sous MacOS 9, celle-ci ne copie pas le clonage de ressources ni les informations du chercheur.

4.4.3 Comment lire (ou écrire) des données binaires ?

Pour lire ou écrire des formats de données complexes en binaire, il est recommandé d'utiliser le module `struct`. Celui-ci permet de convertir une chaîne de caractères qui contient des données binaires, souvent des nombres, en objets Python, et vice-versa.

Par exemple, le code suivant lit, depuis un fichier, deux entiers codés sur 2 octets et un entier codé sur 4 octets, en format gros-boutiste :

```
import struct

f = open(filename, "rb")  # Open in binary mode for portability
s = f.read(8)
x, y, z = struct.unpack(">hhl", s)
```

« > » dans la chaîne de formatage indique que la donnée doit être lue en mode gros-boutiste, la lettre « h » indique un entier court (2 octets) et la lettre « l » indique un entier long (4 octets).

Pour une donnée plus régulière (p. ex. une liste homogène d'entiers ou de nombres à virgule flottante), il est possible d'utiliser le module `array`.

4.4.4 Il me semble impossible d'utiliser `os.read()` sur un tube créé avec `os.popen()` ; pourquoi ?

`os.read()` est une fonction de bas niveau qui prend en paramètre un descripteur de fichier — un entier court qui représente le fichier ouvert. `os.popen()` crée un objet fichier de haut niveau, du même type que celui renvoyé par la fonction native `open()`. Par conséquent, pour lire *n* octets d'un tube *p* créé avec `os.popen()`, il faut utiliser `p.read(n)`.

4.4.5 How do I run a subprocess with pipes connected to both input and output ?

Use the `popen2` module. For example :

```
import popen2
fromchild, tochild = popen2.popen2("command")
tochild.write("input\n")
tochild.flush()
output = fromchild.readline()
```

Warning : in general it is unwise to do this because you can easily cause a deadlock where your process is blocked waiting for output from the child while the child is blocked waiting for input from you. This can be caused by the parent expecting the child to output more text than it does or by data being stuck in stdio buffers due to lack of flushing. The Python parent can of course explicitly flush the data it sends to the child before it reads any output, but if the child is a naive C program it may have been written to never explicitly flush its output, even if it is interactive, since flushing is normally automatic.

Note that a deadlock is also possible if you use `popen3()` to read stdout and stderr. If one of the two is too large for the internal buffer (increasing the buffer size does not help) and you `read()` the other one first, there is a deadlock, too.

Note on a bug in popen2 : unless your program calls `wait()` or `waitpid()`, finished child processes are never removed, and eventually calls to popen2 will fail because of a limit on the number of child processes. Calling `os.waitpid()` with the `os.WNOHANG` option can prevent this ; a good place to insert such a call would be before calling `popen2` again.

In many cases, all you really need is to run some data through a command and get the result back. Unless the amount of data is very large, the easiest way to do this is to write it to a temporary file and run the command with that temporary file as input. The standard module `tempfile` exports a `mktemp()` function to generate unique temporary file names.

```
import tempfile
import os

class Popen3:
    """
    This is a deadlock-safe version of popen that returns
    an object with errorlevel, out (a string) and err (a string).
    (capturestderr may not work under windows.)
    Example: print Popen3('grep spam','\n\nhere spam\n\n').out
    """
    def __init__(self,command,input=None,capturestderr=None):
        outfile=tempfile.mktemp()
        command="( %s ) > %s" % (command,outfile)
        if input:
```

(suite sur la page suivante)

(suite de la page précédente)

```
            infile=tempfile.mktemp()
            open(infile,"w").write(input)
            command=command+" <"+infile
        if capturestderr:
            errfile=tempfile.mktemp()
            command=command+" 2>"+errfile
        self.errorlevel=os.system(command) >> 8
        self.out=open(outfile,"r").read()
        os.remove(outfile)
        if input:
            os.remove(infile)
        if capturestderr:
            self.err=open(errfile,"r").read()
            os.remove(errfile)
```

Note that many interactive programs (e.g. vi) don't work well with pipes substituted for standard input and output. You will have to use pseudo ttys (« ptys ») instead of pipes. Or you can use a Python interface to Don Libes" « expect » library. A Python extension that interfaces to expect is called « expy » and available from http://expectpy.sourceforge.net. A pure Python solution that works like expect is pexpect.

4.4.6 Comment accéder au port de transmission en série (RS-232) ?

Pour Win32, POSIX (Linux, BSD, etc.) et Jython :

http://pyserial.sourceforge.net

Pour Unix, référez-vous à une publication sur Usenet de Mitch Chapman :

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

4.4.7 Pourquoi fermer *sys.stdout*, *sys.stdin*, *sys.stderr* ne les ferme pas réellement ?

Python file objects are a high-level layer of abstraction on top of C streams, which in turn are a medium-level layer of abstraction on top of (among other things) low-level C file descriptors.

For most file objects you create in Python via the built-in `file` constructor, `f.close()` marks the Python file object as being closed from Python's point of view, and also arranges to close the underlying C stream. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C stream.

To close the underlying C stream for one of these three, you should first be sure that's what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use os.close :

```
os.close(0)   # close C's stdin stream
os.close(1)   # close C's stdout stream
os.close(2)   # close C's stderr stream
```

4.5 Programmation réseau et Internet

4.5.1 Quels sont les outils Python dédiés à la Toile ?

Référez-vous aux chapitres intitulés internet et netdata dans le manuel de référence de la bibliothèque. Python a de nombreux modules pour construire des applications de Toile côté client comme côté serveur.

Un résumé des cadriciels disponibles est maintenu par Paul Boddie à l'adresse https://wiki.python.org/moin/WebProgramming.

Cameron Laird maintient un ensemble intéressant d'articles sur les technologies Python dédiées à la Toile à l'adresse http://phaseit.net/claird/comp.lang.python/web_python.

4.5.2 Comment reproduire un envoi de formulaire CGI (METHOD=POST) ?

J'aimerais récupérer la page de retour d'un envoi de formulaire sur la Toile. Existe-t'il déjà du code qui pourrait m'aider à le faire facilement ?

Yes. Here's a simple example that uses httplib :

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

import httplib, sys, time

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
httpobj = httplib.HTTP('www.some-server.out-there', 80)
httpobj.putrequest('POST', '/cgi-bin/some-cgi-script')
# now generate the rest of the HTTP headers...
httpobj.putheader('Accept', '*/*')
httpobj.putheader('Connection', 'Keep-Alive')
httpobj.putheader('Content-type', 'application/x-www-form-urlencoded')
httpobj.putheader('Content-length', '%d' % len(qs))
httpobj.endheaders()
httpobj.send(qs)
# find out what the server said in response...
reply, msg, hdrs = httpobj.getreply()
if reply != 200:
    sys.stdout.write(httpobj.getfile().read())
```

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.urlencode()`. For example, to send `name=Guy Steele, Jr.` :

```
>>> import urllib
>>> urllib.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

4.5.3 Quel module utiliser pour générer du HTML ?

La page wiki de la programmation Toile (en anglais) répertorie un ensemble de liens pertinents.

4.5.4 Comment envoyer un courriel avec un script Python ?

Utilisez le module `smtplib` de la bibliothèque standard.

Voici un exemple très simple d'un envoyeur de courriel qui l'utilise. Cette méthode fonctionne sur tous les serveurs qui implémentent SMTP.

```
import sys, smtplib

fromaddr = raw_input("From: ")
toaddrs  = raw_input("To: ").split(',')
print "Enter message, end with ^D:"
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Sous Unix, il est possible d'utiliser *sendmail*. La localisation de l'exécutable *sendmail* dépend du système ; cela peut-être `/usr/lib/sendmail` ou `/usr/sbin/sendmail`, la page de manuel de *sendmail* peut vous aider. Par exemple :

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print "Sendmail exit status", sts
```

4.5.5 Comment éviter de bloquer dans la méthode `connect()` d'un connecteur réseau ?

The select module is commonly used to help with asynchronous I/O on sockets.

Pour empêcher une connexion TCP de se bloquer, il est possible de mettre le connecteur en mode lecture seule. Avec cela, au moment du `connect()`, la connexion pourra être immédiate (peu probable) ou bien vous obtiendrez une exception qui contient le numéro d'erreur dans `.errno`. `errno.EINPROGRESS` indique que la connexion est en cours, mais qu'elle n'a pas encore aboutie. La valeur dépend du système d'exploitation, donc renseignez-vous sur la valeur utilisée par votre système.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the errno value. To poll, you can call `connect_ex()` again later – 0 or `errno.EISCONN` indicate that you're connected – or you can pass this socket to select to check if it's writable.

4.6 Bases de données

4.6.1 Existe-t'il des modules Python pour s'interfacer avec des bases de données ?

Oui.

Python 2.3 includes the `bsddb` package which provides an interface to the BerkeleyDB library. Interfaces to disk-based hashes such as `DBM` and `GDBM` are also included with standard Python.

La gestion de la plupart des bases de données relationnelles est assurée. Voir la page wiki DatabaseProgramming pour plus de détails.

4.6.2 Comment implémenter la persistance d'objets en Python ?

The `pickle` library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the `shelve` library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects. For better performance, you can use the `cPickle` module.

A more awkward way of doing things is to use pickle's little sister, marshal. The `marshal` module provides very fast ways to store noncircular basic Python types to files and strings, and back again. Although marshal does not do fancy things like store instances or handle shared references properly, it does run extremely fast. For example, loading a half megabyte of data may take less than a third of a second. This often beats doing something more complex and general such as using gdbm with pickle/shelve.

4.6.3 Why is cPickle so slow ?

By default `pickle` uses a relatively old and slow format for backward compatibility. You can however specify other protocol versions that are faster :

```
largeString = 'z' * (100 * 1024)
myPickle = cPickle.dumps(largeString, protocol=1)
```

4.6.4 If my program crashes with a bsddb (or anydbm) database open, it gets corrupted. How come ?

Databases opened for write access with the bsddb module (and often by the anydbm module, since it will preferentially use bsddb) must explicitly be closed using the `.close()` method of the database. The underlying library caches database contents which need to be converted to on-disk form and written.

If you have initialized a new bsddb database but not written anything to it before the program crashes, you will often wind up with a zero-length file and encounter an exception the next time the file is opened.

4.6.5 I tried to open Berkeley DB file, but bsddb produces bsddb.error : (22, "Invalid argument"). Help ! How can I restore my data ?

Don't panic ! Your data is probably intact. The most frequent cause for the error is that you tried to open an earlier Berkeley DB file with a later version of the Berkeley DB library.

Many Linux systems now have all three versions of Berkeley DB available. If you are migrating from version 1 to a newer version use db_dump185 to dump a plain text version of the database. If you are migrating from version 2 to version 3 use db2_dump to create a plain text version of the database. In either case, use db_load to create a new native database for the latest version installed on your computer. If you have version 3 of Berkeley DB installed, you should be able to use db2_load to create a native version 2 database.

You should move away from Berkeley DB version 1 files because the hash file code contains known bugs that can corrupt your data.

4.7 Mathématiques et calcul numérique

4.7.1 Comment générer des nombres aléatoires en Python ?

Le module `random` de la bibliothèque standard comprend un générateur de nombres aléatoires. Son utilisation est simple :

```
import random
random.random()
```

Le code précédent renvoie un nombre à virgule flottante aléatoire dans l'intervalle [0, 1[.

Ce module fournit beaucoup d'autres générateurs spécialisés comme :

- `randrange(a, b)` génère un entier dans l'intervalle [a, b[.
- `uniform(a, b)` génère un nombre à virgule flottante aléatoire dans l'intervalle [a, b[.
- `normalvariate(mean, sdev)` simule la loi normale (Gaussienne).

Des fonctions de haut niveau opèrent directement sur des séquences comme :

- `choice(S)` sélectionne au hasard un élément d'une séquence donnée
- `shuffle(L)` mélange une liste en-place, c-à-d lui applique une permutation aléatoire

Il existe aussi une classe `Random` qu'il est possible d'instancier pour créer des générateurs aléatoires indépendants.

CHAPITRE 5

FAQ extension/intégration

5.1 Puis-je créer mes propres fonctions en C ?

Oui, vous pouvez créer des modules intégrés contenant des fonctions, des variables, des exceptions et même de nouveaux types en C. Ceci est expliqué dans le document extending-index.

La plupart des livres Python intermédiaires ou avancés couvrent également ce sujet.

5.2 Puis-je créer mes propres fonctions en C++ ?

Oui, en utilisant les fonctionnalités de compatibilité C existantes en C++. Placez `extern "C" { ... }` autour des fichiers Python inclus et mettez `extern "C"` avant chaque fonction qui va être appelée par l'interpréteur Python. Les objets C++ globaux ou statiques avec les constructeurs ne sont probablement pas une bonne idée.

5.3 Écrire directement en C est difficile ; existe-t-il des alternatives ?

Il y a un certain nombre de solutions existantes qui vous permettent d'écrire vos propres extensions C, selon ce que vous essayez de faire.

If you need more speed, Psyco generates x86 assembly code from Python bytecode. You can use Psyco to compile the most time-critical functions in your code, and gain a significant improvement with very little effort, as long as you're running on a machine with an x86-compatible processor.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Pyrex makes it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.

5.4 Comment puis-je exécuter des instructions quelconques Python à partir de C ?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns 0 for success and -1 when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()` ; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

5.5 Comment puis-je évaluer une expression quelconque de Python à partir de C ?

Appelez la fonction `PyRun_String()` de la question précédente avec le symbole de départ `Py_eval_input` ; il analyse une expression, l'évalue et renvoie sa valeur.

5.6 Comment puis-je extraire des donnés en C d'un objet Python ?

Cela dépend du type d'objet. Si c'est un tuple, `PyTuple_Size()` renvoie sa longueur et `PyTuple_GetItem()` renvoie l'élément à l'index spécifié. Les listes ont des fonctions similaires, `PyListSize()` et `PyList_GetItem()`.

For strings, `PyString_Size()` returns its length and `PyString_AsString()` a pointer to its value. Note that Python strings may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't *NULL*, and then use `PyString_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called "abstract" interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc.) as well as many other useful protocols.

5.7 Comment utiliser Py_BuildValue() pour créer un tuple de longueur définie ?

You can't. Use `t = PyTuple_New(n)` instead, and fill it with objects using `PyTuple_SetItem(t, i, o)` – note that this « eats » a reference count of `o`, so you have to `Py_INCREF()` it. Lists have similar functions `PyList_New(n)` and `PyList_SetItem(l, i, o)`. Note that you *must* set all the tuple items to some value before you pass the tuple to Python code – `PyTuple_New(n)` initializes them to NULL, which isn't a valid Python value.

5.8 Comment puis-je appeler la méthode d'un objet à partir de C ?

La fonction `PyObject_CallMethod()` peut être utilisée pour appeler la méthode d'un objet. Les paramètres sont l'objet, le nom de la méthode à appeler, une chaîne de caractères comme celle utilisée pour `Py_BuildValue()` et les valeurs des arguments :

```
PyObject *
PyObject_CallMethod(PyObject *object, char *method_name,
                    char *arg_format, ...);
```

Cela fonctionne pour tous les objets qui ont des méthodes — qu'elles soient intégrées ou définies par l'utilisateur. Vous êtes responsable de « `Py_DECREF()` *er* » la valeur de retour à la fin.

Pour appeler, p. ex., la méthode *seek* d'un objet *file* avec les arguments 10, 0 (en supposant que le pointeur de l'objet fichier est *f*) :

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Notez que `PyObject_CallObject()` veut *toujours* un tuple comme liste d'arguments. Aussi, pour appeler une fonction sans arguments, utilisez « () » pour être conforme au type et, pour appeler une fonction avec un paramètre, entourez-le de parenthèses, p. ex. « (i) ».

5.9 Comment puis-je récupérer la sortie de `PyErr_Print()` (ou tout ce qui s'affiche sur *stdout*/*stderr*) ?

Dans le code Python, définissez un objet qui possède la méthode `write()`. Affectez cet objet à `sys.stdout` et `sys.stderr`. Appelez *print_error* ou faites simplement en sorte que le mécanisme standard de remontée des erreurs fonctionne. Ensuite, la sortie sera dirigée vers l'endroit où votre méthode `write()` écrit.

The easiest way to do this is to use the StringIO class in the standard library.

Sample code and use for catching stdout :

```
>>> class StdoutCatcher:
...     def __init__(self):
...         self.data = ''
...     def write(self, stuff):
...         self.data = self.data + stuff
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print 'foo'
>>> print 'hello world!'
>>> sys.stderr.write(sys.stdout.data)
foo
hello world!
```

5.10 Comment accéder à un module écrit en Python à partir de C ?

Vous pouvez obtenir un pointeur sur l'objet module comme suit :

```
module = PyImport_ImportModule("<modulename>");
```

Si le module n'a pas encore été importé (c.-à-d. qu'il n'est pas encore présent dans `sys.modules`), cela initialise le module ; sinon il renvoie simplement la valeur de `sys.modules["<modulename>"]`. Notez qu'il n'inscrit le module dans aucun espace de nommage — il s'assure seulement qu'il a été initialisé et qu'il est stocké dans `sys.modules`.

Vous pouvez alors accéder aux attributs du module (c.-à-d. à tout nom défini dans le module) comme suit :

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Appeler `PyObject_SetAttrString()` pour assigner des valeurs aux variables du module fonctionne également.

5.11 Comment s'interfacer avec les objets C++ depuis Python ?

Selon vos besoins, de nombreuses approches sont possibles. Pour le faire manuellement, commencez par lire le document « Extension et intégration ». Sachez que pour le système d'exécution Python, il n'y a pas beaucoup de différence entre C et C++ — donc la méthode pour construire un nouveau type Python à partir d'une structure C (pointeur) fonctionne également avec des objets en C++.

Pour les bibliothèques C++, voir *Écrire directement en C est difficile ; existe-t-il des alternatives ?*.

5.12 J'ai ajouté un module en utilisant le fichier *Setup* et la compilation échoue ; pourquoi ?

Le fichier *Setup* doit se terminer par une ligne vide, s'il n'y a pas de ligne vide, le processus de compilation échoue (ce problème peut se régler en bidouillant un script shell, et ce bogue est si mineur qu'il ne mérite pas qu'on s'y attarde).

5.13 Comment déboguer une extension ?

Lorsque vous utilisez GDB avec des extensions chargées dynamiquement, vous ne pouvez pas placer de point d'arrêt dans votre extension tant que celle-ci n'est pas chargée.

Dans votre fichier `.gdbinit` (ou manuellement), ajoutez la commande :

```
br _PyImport_LoadDynamicModule
```

Ensuite, lorsque vous exécutez GDB :

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 Je veux compiler un module Python sur mon système Linux, mais il manque certains fichiers. Pourquoi ?

La plupart des versions pré-compilées de Python n'incluent pas le répertoire `/usr/lib/python2.x/config/`, qui contient les différents fichiers nécessaires à la compilation des extensions Python.

Pour Red Hat, installez le RPM *python-devel* pour obtenir les fichiers nécessaires.

Pour Debian, exécutez `apt-get install python-dev`.

5.15 What does « SystemError : _PyImport_FixupExtension : module yourmodule not loaded » mean ?

This means that you have created an extension module named « yourmodule », but your module init function does not initialize with that name.

Every module init function will have a line similar to :

```
module = Py_InitModule("yourmodule", yourmodule_functions);
```

If the string passed to this function is not the same name as your extension module, the `SystemError` exception will be raised.

5.16 Comment distinguer une « entrée incomplète » (*incomplete input*) d'une « entrée invalide » (*invalid input*) ?

Parfois vous souhaitez émuler le comportement de l'interpréteur interactif Python, quand il vous donne une invite de continuation lorsque l'entrée est incomplète (par exemple, vous avez tapé le début d'une instruction « if » ou vous n'avez pas fermé vos parenthèses ou triple guillemets) mais il vous renvoie immédiatement une erreur syntaxique quand la saisie est incorrecte.

En Python, vous pouvez utiliser le module `codeop`, qui se rapproche assez du comportement de l'analyseur. Par exemple, IDLE l'utilise.

La façon la plus simple de le faire en C est d'appeler `PyRun_InteractiveLoop()` (peut-être dans un autre fil d'exécution) et laisser l'interpréteur Python gérer l'entrée pour vous. Vous pouvez également définir `PyOS_ReadlineFunctionPointer()` pour pointer vers votre fonction d'entrée personnalisée. Voir `Modules/readline.c` et `Parser/myreadline.c` pour plus de conseils.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can't allow the `PyRun_InteractiveLoop()` to stop while waiting for user input. The one solution then is to call `PyParser_ParseString()` and test for `e.error` equal to `E_EOF`, which means the input is incomplete. Here's a sample code fragment, untested, inspired by code from Alex Farber :

```
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>
```

(suite sur la page suivante)

(suite de la page précédente)

```
int testcomplete(char *code)
  /* code should end in \n */
  /* return -1 for error, 0 for incomplete, 1 for complete */
{
  node *n;
  perrdetail e;

  n = PyParser_ParseString(code, &_PyParser_Grammar,
                           Py_file_input, &e);
  if (n == NULL) {
    if (e.error == E_EOF)
      return 0;
    return -1;
  }

  PyNode_Free(n);
  return 1;
}
```

Une autre solution est d'essayer de compiler la chaîne reçue avec `Py_CompileString()`. Si cela se compile sans erreur, essayez d'exécuter l'objet code renvoyé en appelant `PyEval_EvalCode()`. Sinon, enregistrez l'entrée pour plus tard. Si la compilation échoue, vérifiez s'il s'agit d'une erreur ou s'il faut juste plus de données — en extrayant la chaîne de message du tuple d'exception et en la comparant à la chaîne *« unexpected EOF while parsing »*. Voici un exemple complet d'utilisation de la bibliothèque *readline* de GNU (il vous est possible d'ignorer **SIGINT** lors de l'appel à `readline()`):

```
#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
  int i, j, done = 0;                           /* lengths of line, code */
  char ps1[] = ">>> ";
  char ps2[] = "... ";
  char *prompt = ps1;
  char *msg, *line, *code = NULL;
  PyObject *src, *glb, *loc;
  PyObject *exc, *val, *trb, *obj, *dum;

  Py_Initialize ();
  loc = PyDict_New ();
  glb = PyDict_New ();
  PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

  while (!done)
  {
    line = readline (prompt);

    if (NULL == line)                           /* Ctrl-D pressed */
    {
      done = 1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
    else
    {
      i = strlen (line);

      if (i > 0)
        add_history (line);                     /* save non-empty lines */

      if (NULL == code)                         /* nothing in code yet */
        j = 0;
      else
        j = strlen (code);

      code = realloc (code, i + j + 2);
      if (NULL == code)                         /* out of memory */
        exit (1);

      if (0 == j)                               /* code was empty, so */
        code[0] = '\0';                         /* keep strncat happy */

      strncat (code, line, i);                  /* append line to code */
      code[i + j] = '\n';                       /* append '\n' to code */
      code[i + j + 1] = '\0';

      src = Py_CompileString (code, "<stdin>", Py_single_input);

      if (NULL != src)                          /* compiled just fine - */
      {
        if (ps1  == prompt ||                   /* ">>> " or */
            '\n' == code[i + j - 1])            /* "... " and double '\n' */
        {                                                   /* so execute it */
          dum = PyEval_EvalCode ((PyCodeObject *)src, glb, loc);
          Py_XDECREF (dum);
          Py_XDECREF (src);
          free (code);
          code = NULL;
          if (PyErr_Occurred ())
            PyErr_Print ();
          prompt = ps1;
        }
      }                                         /* syntax error or E_EOF? */
      else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
      {
        PyErr_Fetch (&exc, &val, &trb);         /* clears exception! */

        if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
        {
          Py_XDECREF (exc);
          Py_XDECREF (val);
          Py_XDECREF (trb);
          prompt = ps2;
        }
        else                                    /* some other syntax error */
        {
          PyErr_Restore (exc, val, trb);
          PyErr_Print ();
```

(suite sur la page suivante)

(suite de la page précédente)

```
          free (code);
          code = NULL;
          prompt = ps1;
        }
      }
      else                                    /* some non-syntax error */
      {
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
      }

      free (line);
    }
  }

  Py_XDECREF(glb);
  Py_XDECREF(loc);
  Py_Finalize();
  exit(0);
}
```

5.17 Comment puis-je trouver les symboles g++ indéfinis `__builtin_new` ou `__pure_virtual` ?

Pour charger dynamiquement les modules d'extension g++, vous devez recompiler Python, effectuer l'édition de liens en utilisant g++ (modifiez *LINKCC* dans le *Python Modules Makefile*), et effectuer l'édition de liens de votre module d'extension avec g++ (par exemple, `g++ -shared -o mymodule.so mymodule.o`).

5.18 Puis-je créer une classe d'objets avec certaines méthodes implémentées en C et d'autres en Python (p. ex. en utilisant l'héritage) ?

Oui, vous pouvez hériter de classes intégrées telles que `int`, `list`, `dict`, etc.

La bibliothèque *Boost Python Library* (BPL, http://www.boost.org/libs/python/doc/index.html) fournit un moyen de le faire depuis C++ (c.-à-d. que vous pouvez hériter d'une classe d'extension écrite en C++ en utilisant BPL).

5.19 When importing module X, why do I get « undefined symbol : PyUnicodeUCS2* » ?

You are using a version of Python that uses a 4-byte representation for Unicode characters, but some C extension module you are importing was compiled using a Python that uses a 2-byte representation for Unicode characters (the default).

If instead the name of the undefined symbol starts with `PyUnicodeUCS4`, the problem is the reverse : Python was built using 2-byte Unicode characters, and the extension module was compiled using a Python with 4-byte Unicode characters.

This can easily occur when using pre-built extension packages. RedHat Linux 7.x, in particular, provided a « python2 » binary that is compiled with 4-byte Unicode. This only causes the link failure if the extension uses any of the `PyUnicode_*()` functions. It is also a problem if an extension uses any of the Unicode-related format specifiers for `Py_BuildValue()` (or similar) or parameter specifications for `PyArg_ParseTuple()`.

You can check the size of the Unicode character a Python interpreter is using by checking the value of sys.maxunicode :

```
>>> import sys
>>> if sys.maxunicode > 65535:
...     print 'UCS4 build'
... else:
...     print 'UCS2 build'
```

The only way to solve this problem is to use extension modules compiled with a Python binary built using the same size for Unicode characters.

CHAPITRE 6

FAQ : Python et Windows

6.1 Comment exécuter un programme Python sous Windows ?

Ce n'est pas forcément une question simple. Si vous êtes déjà familier avec le lancement de programmes depuis la ligne de commande de Windows alors tout semblera évident ; Sinon, vous auriez besoin d'être un peu guidé.

Unless you use some sort of integrated development environment, you will end up *typing* Windows commands into what is variously referred to as a « DOS window » or « Command prompt window ». Usually you can create such a window from your Start menu ; under Windows 7 the menu selection is *Start ▸ Programs ▸ Accessories ▸ Command Prompt*. You should be able to recognize when you have started such a window because you will see a Windows « command prompt », which usually looks like this :

```
C:\>
```

la lettre peut être différente, et il peut y avoir d'autres choses à la suite, alors il se peut aussi bien que vous voyez quelque chose tel que :

```
D:\YourName\Projects\Python>
```

selon la configuration de votre ordinateur et ce que vous avez récemment fait avec. Une fois que vous avez ouvert cette fenêtre, vous êtes bien partis pour pouvoir lancer des programmes Python.

Retenez que vos scripts Python doivent être traités par un autre programme appelé « l'interpréteur » Python. L'interpréteur lit votre script, le compile en *bytecode*, et exécute le *bytecode* pour faire tourner votre programme. Alors, comment faire pour donner votre code Python à l'interpréteur ?

First, you need to make sure that your command window recognises the word « python » as an instruction to start the interpreter. If you have opened a command window, you should try entering the command `python` and hitting return. :

```
C:\Users\YourName> python
```

You should then see something like :

```
Python 2.7.3 (default, Apr 10 2012, 22.71:26) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You have started the interpreter in « interactive mode ». That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python's strongest features. Check it by entering a few expressions of your choice and seeing the results :

```
>>> print "Hello"
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Beaucoup de personnes utilisent le mode interactif comme une calculatrice programmable. Lorsque vous souhaitez sortir du mode interactif de Python, maintenez la touche `Ctrl` appuyée, puis appuyez sur `Z`. Validez avec « `Entrée` » pour retrouver votre ligne de commande Windows.

You may also find that you have a Start-menu entry such as *Start ▸ Programs ▸ Python 2.7 ▸ Python (command line)* that results in you seeing the >>> prompt in a new window. If so, the window will disappear after you enter the `Ctrl-Z` character ; Windows is running a single « python » command in the window, and closes it when you terminate the interpreter.

Si, au lieu de vous afficher >>> la commande `python` vous affiche un message semblable à celui-ci :

```
'python' is not recognized as an internal or external command, operable program or␣
↪batch file.
```

ou :

```
Bad command or filename
```

alors, vous devez vous assurer que votre ordinateur sait où trouver l'interpréteur Python. Pour cela, vous devez modifier un paramètre, appelé « PATH », qui est une liste des répertoires dans lesquels Windows cherche les programmes.

You should arrange for Python's installation directory to be added to the PATH of every command window as it starts. If you installed Python fairly recently then the command

```
dir C:\py*
```

will probably tell you where it is installed ; the usual location is something like `C:\Python27`. Otherwise you will be reduced to a search of your whole disk … use *Tools ▸ Find* or hit the *Search* button and look for « python.exe ». Supposing you discover that Python is installed in the `C:\Python27` directory (the default at the time of writing), you should make sure that entering the command

```
c:\Python27\python
```

starts up the interpreter as above (and don't forget you'll need a « `Ctrl-Z` » and an « `Enter` » to get out of it). Once you have verified the directory, you can add it to the system path to make it easier to start Python by just running the `python` command. This is currently an option in the installer as of CPython 2.7.

More information about environment variables can be found on the Using Python on Windows page.

6.2 Comment rendre des scripts Python exécutables ?

Sous Windows, l'installateur Python associe l'extension *.py* avec un type de fichier (Python.File) et une commande qui lance l'interpréteur (`D:\Program Files\Python\python.exe "%1" %*`). Cela suffit pour pouvoir exécuter les scripts Python depuis la ligne de commande en saisissant *foo.py*. Si vous souhaitez pouvoir exécuter les scripts en saisissant simplement *foo* sans l'extension, vous devez ajouter *.py* au paramètre d'environnement PATHEXT.

6.3 Pourquoi Python met-il du temps à démarrer ?

Normalement, sous Windows, Python se lance très rapidement, mais parfois des rapports d'erreurs indiquent que Python commence soudain à prendre beaucoup de temps pour démarrer. C'est d'autant plus intrigant que Python fonctionne correctement avec d'autres Windows configurés de façon similaire.

Le problème peut venir d'un antivirus mal configuré. Certains antivirus sont connus pour doubler le temps de démarrage lorsqu'ils sont configurés pour surveiller toutes les lectures du système de fichiers. Essayez de regarder si les antivirus des deux machines sont bien paramétrés à l'identique. *McAfee* est particulièrement problématique lorsqu'il est paramétré pour surveiller toutes les lectures du système de fichiers.

6.4 Comment construire un exécutable depuis un script Python ?

See http://www.py2exe.org/ for a distutils extension that allows you to create console and GUI executables from Python code.

6.5 Est-ce qu'un fichier `*.pyd` est la même chose qu'une DLL ?

Yes, .pyd files are dll's, but there are a few differences. If you have a DLL named `foo.pyd`, then it must have a function `initfoo()`. You can then write Python « import foo », and Python will search for foo.pyd (as well as foo.py, foo.pyc) and if it finds it, will attempt to call `initfoo()` to initialize it. You do not link your .exe with foo.lib, as that would cause Windows to require the DLL to be present.

Notez que le chemin de recherche pour *foo.pyd* est *PYTHONPATH*, il est différent de celui qu'utilise Windows pour rechercher *foo.dll*. De plus, *foo.pyd* n'a pas besoin d'être présent pour que votre programme s'exécute alors que si vous avez lié votre programme avec une *dll* celle-ci est requise. Bien sûr *foo.pyd* est nécessaire si vous écrivez `import foo`. Dans une *DLL* le lien est déclaré dans le code source avec `__declspec(dllexport)`. Dans un *.pyd* la liaison est définie dans une liste de fonctions disponibles.

6.6 Comment puis-je intégrer Python dans une application Windows ?

L'intégration de l'interpréteur Python dans une application Windows peut se résumer comme suit :

1. Do _not_ build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to `python`*NN*`.dll`; it is typically installed in `C:\Windows\System`. *NN* is the Python version, a number such as « 27 » for Python 2.7.

Vous pouvez créer un lien vers Python de deux manières différentes. Un lien au moment du chargement signifie pointer vers `python`*NN*`.lib`, tandis qu'un lien au moment de l'exécution signifie pointer vers `python`*NN*`.dll`. (Note générale : `python`*NN*`.lib` est le soi-disant « *import lib* » correspondant à `python`*NN*`.dll`. Il définit simplement des liens symboliques pour l'éditeur de liens.)

La liaison en temps réel simplifie grandement les options de liaison ; tout se passe au moment de l'exécution. Votre code doit charger `python`*NN*`.dll` en utilisant la routine Windows `LoadLibraryEx()`. Le code doit aussi utiliser des routines d'accès et des données dans `python`*NN*`.dll` (c'est-à-dire les API C de Python) en utilisant des pointeurs obtenus par la routine Windows `GetProcAddress()`. Les macros peuvent rendre l'utilisation de ces pointeurs transparente à tout code C qui appelle des routines dans l'API C de Python.

Note Borland : convertir `python`*NN*`.lib` au format OMF en utilisant *Coff2Omf.exe* en premier.

2. Si vous utilisez SWIG, il est facile de créer un « module d'extension » Python qui rendra les données et les méthodes de l'application disponibles pour Python. SWIG s'occupera de tous les détails ennuyeux pour vous. Le résultat est du code C que vous liez *dans* votre *fichier.exe* (!) Vous n'avez _pas_ besoin de créer un fichier DLL, et cela simplifie également la liaison.

3. SWIG va créer une fonction d'initialisation (fonction en C) dont le nom dépend du nom du module d'extension. Par exemple, si le nom du module est *leo*, la fonction *init* sera appelée *initleo()*. Si vous utilisez des classes *shadow* SWIG, comme vous le devriez, la fonction *init* sera appelée *initleoc()*. Ceci initialise une classe auxiliaire invisible utilisée par la classe *shadow*.

   La raison pour laquelle vous pouvez lier le code C à l'étape 2 dans votre *fichier.exe* est que l'appel de la fonction d'initialisation équivaut à importer le module dans Python ! (C'est le deuxième fait clé non documenté.)

4. En bref, vous pouvez utiliser le code suivant pour initialiser l'interpréteur Python avec votre module d'extension.

```
#include "python.h"
...
Py_Initialize();  // Initialize Python.
initmyAppc();  // Initialize (import) the helper class.
PyRun_SimpleString("import myApp");  // Import the shadow class.
```

5. Il y a deux problèmes avec l'API C de Python qui apparaîtront si vous utilisez un compilateur autre que MSVC, le compilateur utilisé pour construire *pythonNN.dll*.

   Problème 1 : Les fonctions dites de « Très Haut Niveau » qui prennent les arguments FILE * ne fonctionneront pas dans un environnement multi-compilateur car chaque compilateur aura une notion différente de la structure de FILE. Du point de vue de l'implémentation, il s'agit de fonctions de très bas niveau.

   Problème 2 : SWIG génère le code suivant lors de la génération *d'encapsuleurs* pour annuler les fonctions :

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

   Hélas, *Py_None* est une macro qui se développe en référence à une structure de données complexe appelée *_Py_NoneStruct* dans *pythonNN.dll*. Encore une fois, ce code échouera dans un environnement multi-compilateur. Remplacez ce code par :

```
return Py_BuildValue("");
```

   Il est possible d'utiliser la commande `%typemap` de SWIG pour effectuer le changement automatiquement, bien que je n'ai pas réussi à le faire fonctionner (je suis un débutant complet avec SWIG).

6. Utiliser un script shell Python pour créer une fenêtre d'interpréteur Python depuis votre application Windows n'est pas une bonne idée ; la fenêtre résultante sera indépendante du système de fenêtrage de votre application. Vous (ou la classe *wxPythonWindow*) devriez plutôt créer une fenêtre d'interpréteur « native ». Il est facile de connecter cette fenêtre à l'interpréteur Python. Vous pouvez rediriger l'entrée/sortie de Python vers *n'importe quel* objet qui supporte la lecture et l'écriture, donc tout ce dont vous avez besoin est un objet Python (défini dans votre module d'extension) qui contient les méthodes *read()* et *write()*.

6.7 Comment empêcher mon éditeur d'utiliser des tabulations dans mes fichiers Python ?

La FAQ ne recommande pas l'utilisation des indentations et le guide stylistique de Python, la **PEP 8**, recommande l'utilisation de 4 espaces dans les codes Python. C'est aussi le comportement par défaut d'Emacs avec Python.

Quel que soit votre éditeur, mélanger des tabulations et des espaces est une mauvaise idée. *Visual C++*, par exemple, peut être facilement configuré pour utiliser des espaces : allez dans *Tools* ▸ *Options* ▸ *Tabs* et pour le type de fichier par défaut, vous devez mettre *Tab size* et *Indent size* à 4, puis sélectionner *Insert spaces*.

If you suspect mixed tabs and spaces are causing problems in leading whitespace, run Python with the `-t` switch or run the `tabnanny` module to check a directory tree in batch mode.

6.8 Comment puis-je vérifier de manière non bloquante qu'une touche a été pressée ?

Utilisez le module `msvcrt`. C'est une extension standard spécifique à Windows, qui définit une fonction `kbhit()` qui vérifie si une pression de touche s'est produite, et `getch()` qui récupère le caractère sans l'afficher.

6.9 How do I emulate os.kill() in Windows ?

Prior to Python 2.7 and 3.2, to terminate a process, you can use `ctypes` :

```
import ctypes

def kill(pid):
    """kill function for Win32"""
    kernel32 = ctypes.windll.kernel32
    handle = kernel32.OpenProcess(1, 0, pid)
    return (0 != kernel32.TerminateProcess(handle, 0))
```

In 2.7 and 3.2, `os.kill()` is implemented similar to the above function, with the additional feature of being able to send `Ctrl+C` and `Ctrl+Break` to console subprocesses which are designed to handle those signals. See `os.kill()` for further details.

6.10 Comment décompresser la documentation téléchargée sous Windows ?

Quelquefois, lorsque vous téléchargez de la documentation avec Windows en utilisant un navigateur internet, l'extension du fichier est .EXE. Il s'agit d'une erreur ; l'extension devrait être .TGZ.

Renommez simplement le fichier téléchargé pour lui donner l'extension .TGZ, puis utilisez WinZip pour le décompresser. Si WinZip ne peut pas décompresser le fichier, téléchargez une version plus à jour (https://www.winzip.com).

CHAPITRE 7

FAQ interface graphique

7.1 Quelles boites à outils multi-plateforme existe-t-il sur Python ?

Depending on what platform(s) you are aiming at, there are several.

7.1.1 Tkinter

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called Tkinter. This is probably the easiest to install and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page at https://www.tcl.tk. Tcl/Tk is fully portable to the Mac OS X, Windows, and Unix platforms.

7.1.2 *wxWidgets*

wxWidgets (https://www.wxwidgets.org) est une librairie de classe IUG portable et gratuite écrite en C++ qui fournit une apparence native sur un certain nombre de plates-formes, elle est notamment en version stable pour Windows, Mac OS X, GTK et X11. Des clients sont disponibles pour un certain nombre de langages, y compris Python, Perl, Ruby, etc.

wxPython (http://www.wxpython.org) est le portage Python de wxWidgets. Bien qu'il soit légèrement en retard sur les versions officielles de wxWidgets, il offre également des fonctionnalités propres à Python qui ne sont pas disponibles pour les autres langages. WxPython dispose de plus, d'une communauté d'utilisateurs et de développeurs active.

wxWidgets et *wxPython* sont tous deux des logiciels libres, open source, avec des licences permissives qui permettent leur utilisation dans des produits commerciaux ainsi que dans des logiciels gratuits ou contributifs (*shareware*).

7.1.3 *Qt*

Il existe des liens disponibles pour la boîte à outils *Qt* (en utilisant soit PyQt ou PySide) et pour *KDE* (PyKDE4). *PyQt* est actuellement plus mûre que *PySide*, mais*PyQt* nécessite d'acheter une licence de Riverbank Computing si vous voulez écrire des applications propriétaires. *PySide* est gratuit pour toutes les applications.

Qt >= *4.5* est sous licence LGPL ; de plus, des licences commerciales sont disponibles auprès de The Qt Company.

7.1.4 *Gtk+*

PyGtk bindings for the Gtk+ toolkit have been implemented by James Henstridge ; see <http://www.pygtk.org>.

7.1.5 *FLTK*

Les liaisons Python pour the FLTK toolkit, un système de fenêtrage multi-plateformes simple mais puissant et mûr, sont disponibles auprès de the PyFLTK project.

7.1.6 *OpenGL*

Pour les clients OpenGL, voir PyOpenGL.

7.2 Quelles boîtes à outils IUG spécifiques à la plate-forme existent pour Python ?

By installing the PyObjc Objective-C bridge, Python programs can use Mac OS X's Cocoa libraries.

Pythonwin de Mark Hammond inclut une interface vers les classes *Microsoft Foundation Classes* et un environnement de programmation Python qui est écrit principalement en Python utilisant les classes *MFC*.

7.3 Questions à propos de *Tkinter*

7.3.1 Comment puis-je geler (*freezer*) les applications *Tkinter* ?

Freeze est un outil pour créer des applications autonomes. Lors du *freezage* des applications Tkinter, les applications ne seront pas vraiment autonomes, car l'application aura toujours besoin des bibliothèques Tcl et Tk.

Une solution consiste à empaqueter les bibliothèques *Tcl* et *Tk* dans l'application et de les retrouver à l'exécution en utilisant les variables d'environnement `TCL_LIBRARY` et `TK_LIBRARY`.

Pour obtenir des applications vraiment autonomes, les scripts *Tcl* qui forment la bibliothèque doivent également être intégrés dans l'application. Un outil supportant cela est *SAM* (modules autonomes), qui fait partie de la distribution *Tix* (http://tix.sourceforge.net/).

Compilez Tix avec SAM activé, exécutez l'appel approprié à `Tclsam_init()`, etc. dans le fichier `Modules/tkappinit.c` de Python, et liez avec *libtclsam* et *libtksam* (il est également possible d'inclure les bibliothèques *Tix*).

7.3.2 Puis-je modifier des événements *Tk* pendant l'écoute des *E/S* ?

Sur d'autres plates-formes que Windows, oui, et vous n'avez même pas besoin de fils d'exécution multiples ! Mais vous devrez restructurer un peu votre code *E/S*. *Tk* possède l'équivalent de l'appel `XtAddInput()` de *Xt*, qui vous permet d'enregistrer une fonction de *callback* qui sera appelée par la boucle principale *Tk* lorsque des *E/S* sont disponibles sur un descripteur de fichier. Voir tkinter-file-handlers.

7.3.3 Je n'arrive pas à faire fonctionner les raccourcis clavier dans *Tkinter* : pourquoi ?

Une raison récurrente est que les gestionnaires d'évènements liés à des évènements avec la méthode `bind()` ne sont pas pris en charge même lorsque la touche appropriée est activée.

La cause la plus fréquente est que l'objet graphique auquel s'applique la liaison n'a pas de « focus clavier ». Consultez la documentation *Tk* pour la commande *focus*. Habituellement, un objet graphique reçoit le focus du clavier en cliquant dessus (mais pas pour les étiquettes ; voir l'option *takefocus*).

CHAPITRE 8

« Pourquoi Python est installé sur mon ordinateur ? » FAQ

8.1 Qu'est-ce que Python ?

Python est un langage de programmation. Il est utilisé dans de nombreuses applications. Souvent utilisé dans les lycées et universités comme langage d'introduction à la programmation pour sa simplicité d'apprentissage, il est aussi utilisé par des développeurs professionnels appartenant à des grands groupes comme Google, la NASA, Lucasfilm etc.

Si vous voulez en apprendre plus sur Python, vous pouvez commencer par le Guide des Débutants pour Python.

8.2 Pourquoi Python est installé sur ma machine ?

Si Python est installé sur votre système mais que vous ne vous rappelez pas l'avoir installé, il y a plusieurs raisons possibles à sa présence.

— Peut être qu'un autre utilisateur de l'ordinateur voulait apprendre la programmation et l'a installé, dans ce cas vous allez devoir trouver qui a utilisé votre machine et aurait pu l'installer.

— A third-party application installed on the machine might have been written in Python and included a Python installation. For a home computer, the most common such application is PySol, a solitaire game that includes over 1000 different games and variations.

— Certaines machines fonctionnant avec le système d'exploitation Windows possèdent une installation de Python. À ce jour, les ordinateurs des marques Hewlett-Packard et Compaq incluent nativement Python. Certains outils d'administration de ces marques sont écrits en Python.

— All Apple computers running Mac OS X have Python installed ; it's included in the base installation.

8.3 Puis-je supprimer Python ?

Cela dépend de l'origine de Python .

Si Python a été installé délibérément par une personne tierce ou vous même, vous pouvez le supprimer sans causer de dommage. Sous Windows, vous pouvez simplement utiliser l'icône d'Ajout / Suppression de programmes du Panneau de configuration.

Si Python a été installé par une application tierce, Python peut être désinstallé, l'application l'ayant installé cessera alors de fonctionner. Dans ce cas, désinstallez l'application en utilisant son outil de désinstallation est plus indiqué que supprimer Python.

Si Python a été installé avec votre système d'exploitation, le supprimer n'est pas recommandé. Si vous choisissez tout de même de le supprimer, tous les outils écrits en python cesseront alors de fonctionner, certains outils pouvant être importants. La réinstallation intégrale du système pourrait être nécessaire pour résoudre les problèmes suite à la désinstallation de Python.

ANNEXE A

Glossaire

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

2to3 est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. 2to3-reference.

classe de base abstraite Les classes de base abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes, ou subitement fausse (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles, qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (Voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections`), les nombres (dans le module `numbers`), les flux (dans le module `io`). Vous pouvez créer vos propres ABC avec le module `abc`.

argument Une valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :

— *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, `3` et `5` sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

— *argument positionnel* : Un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par `*`. Par exemple, `3` et `5` sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section calls à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi *parameter* dans le glossaire, et la question dans la FAQ à propos de *la différence entre argument et paramètre*.

attribut Valeur associée à un objet et désignée par son nom via une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, il sera référencé par *o.a*.

BDFL Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

Objet bytes-compatible Un objet gérant le bufferobjects, comme les classes `str`, `bytearray`, ou `memoryview`. Les objets bytes-compatibles peuvent manipuler des données binaires et ainsi servir à leur compression, sauvegarde, ou envoi sur une socket. Certaines actions nécessitent que la donnée binaire soit modifiable, ce qui n'est pas possible avec tous les objets byte-compatibles.

code intermédiaire (*bytecode*) Le code source, en Python, est compilé en un bytecode, la représentation interne à CPython d'un programme Python. Le bytecode est stocké dans un fichier nommé `.pyc` ou `.pyo`. Ces caches permettent de charger les fichiers plus rapidement lors de la deuxième exécution (en évitant ainsi de recommencer la compilation en bytecode). On dit que ce *langage intermédiaire* est exécuté sur une *machine virtuelle* qui exécute des instructions machine pour chaque instruction du bytecode. Notez que le bytecode n'a pas vocation à fonctionner entre différentes machines virtuelle Python, encore moins entre différentes version de Python.

La documentation du module dis fournit une liste des instructions du code intermédiaire.

classe Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

coercition The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

nombre complexe Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de `-1`, souvent écrite `i` en mathématiques ou `j` par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe `j`, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

gestionnaire de contexte Objet contrôlant l'environnement à l'intérieur d'un bloc `with` en définissant les méthodes `__enter__()` et `__exit__()`. Consultez la **PEP 343**.

CPython L'implémentation canonique du langage de programmation Python, tel que distribué sur python.org. Le terme « CPython » est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

décorateur Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

descripteur Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Pour plus d'informations sur les méthodes des descripteurs, consultez descriptors.

dictionnaire An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

vue de dictionnaire The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elles est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

duck-typing Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancane, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

EAFP Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LBYL* utilisé couramment dans les langages tels que C.

expression A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

module d'extension Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

objet fichier Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, une socket réseau, …). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

There are actually three categories of file objects : raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

objet fichier-compatible Synonyme de *objet fichier*.

chercheur An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See **PEP 302** for details.

division entière Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut `2`, contrairement à `11 / 4` qui vaut `2.75`. Notez que `(-11) // 4` vaut `-3` car l'arrondi se fait à l'entier inférieur. Voir la **PEP 328**.

fonction Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et function.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing :

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

ramasse-miettes (*garbage collection*) Le mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence, et un ramasse-miettes cyclique capable de détecter et casser les références circulaires.

générateur A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

expression génératrice Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une expression `for` définissant une variable de boucle, un intervalle et une expression `if` optionnelle. Toute cette expression génère des valeurs pour la fonction qui l'entoure :

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL Voir *global interpreter lock*.

verrou global de l'interpréteur (*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de CPython en rendant le modèle objet (incluant des parties critiques comme la classe native `dict`) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées / sorties.

Les tentatives précédentes d'implémenter un interpréteur Python avec une granularité de verrouillage plus fine ont toutes échouées, à cause de leurs mauvaises performances dans le cas d'un processeur unique. Il est admis que corriger ce problème de performance induit mènerait à une implémentation beaucoup plus compliquée et donc plus coûteuse à maintenir.

hachable An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

Tous les types immuables fournis par Python sont hachables, et aucun type mutable (comme les listes ou les dictionnaires) ne l'est. Toutes les instances de classes définies par les utilisateurs sont hachables par défaut, elles

sont toutes différentes selon `__eq__`, sauf comparées à elles mêmes, et leur empreinte (*hash*) est calculée à partir de leur `id()`.

IDLE Environnement de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

immuable Objet dont la valeur ne change pas. Les nombres, les chaînes et les n-uplets sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

importer Processus rendant le code Python d'un module disponible dans un autre.

importateur Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

interactif Python a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

interprété Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

itérable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

itérateur An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Vous trouverez davantage d'informations dans typeiter.

fonction clé Une fonction clé est un objet appelable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction `locale.strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clef pour maîtriser comment les éléments dont triés ou groupés. Typiquement les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, et `itertools.groupby()`.

La méthode `str.lower()` peut servir en fonction clef pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clef au besoin avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Finalement le module `operator` fournit des constructeurs de fonctions clef : `attrgetter()`, `itemgetter()`, et `methodcaller()`. Voir Comment Trier pour avoir des exemple de création et d'utilisation de fonctions clés.

argument nommé Voir *argument*.

lambda An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL Regarde devant avant de tomber, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquencement critique (*race condition* en anglais) entre le « regarde » et le « tomber ». Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé *key* du *mapping* après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche EAFP.

list A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

liste en compréhension (ou liste en intension) A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

chargeur An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

magic method An informal synonym for *special method*.

Tableau de correspondances Un conteneur permettant d'accéder à des éléments par clef et implémente les méthodes spécifiées dans `Mapping` ou `~collections.MutableMapping :ref:`classes de base abstraites`. Les classes suivantes sont des exemples de mapping : `dict`, `collections.defaultdict`, `collections.OrderedDict`, et `collections.Counter`.

métaclasse Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclasse a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'aura jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûr les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : metaclasses.

méthode Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

ordre de résolution des méthodes L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir The Python 2.3 Method Resolution Order pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

module Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de noms et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

MRO Voir *ordre de résolution des méthodes*.

muable Un objet muable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

n-uplet nommé (*named-tuple* en anglais) Classe qui, comme un *n-uplet* (*tuple* en anglais), a ses éléments accessibles par leur indice. Et en plus, les éléments sont accessibles par leur nom. Par exemple, `time.localtime()` donne un objet ressemblant à un *n-uplet*, dont *year* est accessible par son indice : `t[0]` ou par son nom : `t.tm_year`).

Un *n-uplet nommé* peut être un type natif tel que `time.struct_time` ou il peut être construit comme une simple classe. Un *n-uplet nommé* complet peut aussi être créé via la fonction `collections.namedtuple()`. Cette dernière approche fournit automatiquement des fonctionnalités supplémentaires, tel qu'une représentation lisible comme `Employee(name='jones', title='programmer')`.

espace de noms The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

portée imbriquée The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

nouvelle classe Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

objet N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

paquet *module* Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

paramètre A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters :

— *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, *foo* et *bar* dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

— *positional-only* : l'argument ne peut être donné que par sa position. Python n'a pas de syntaxe pour déclarer de tels paramètres, cependant des fonctions natives, comme `abs()`, en utilisent.

— *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une `*`. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

— *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par `**`. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

See also the *argument* glossary entry, the FAQ question on *the difference between arguments and parameters*, and the function section.

PEP Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

argument positionnel Voir *argument*.

Python 3000 Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

Pythonique Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)):
    print food[i]
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print piece
```

nombre de références Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module `sys` définit une fonction `getrefcount()` que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

`__slots__` A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

séquence An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

tranche An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

méthode spéciale (*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans specialnames.

instruction Une instruction (*statement* en anglais) est un composant d'un « bloc » de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

chaîne entre triple guillemets Chaîne qui est délimitée par trois guillemets simples (`'`) ou trois guillemets doubles (`"`). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

retours à la ligne universels A manner of interpreting text streams in which all of the following are recognized as ending a line : the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `str.splitlines()` for an additional use.

environnement virtuel Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

machine virtuelle Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *bytecode* produit par le compilateur de *bytecode*.

Le zen de Python Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant « `import this` » dans une invite Python interactive.

ANNEXE B

À propos de ces documents

Ces documents sont générés à partir de sources en reStructuredText par Sphinx, un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page reporting-bugs qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

- Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
- le projet Docutils pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
- Fredrik Lundh pour son projet Alternative Python Reference, dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez Misc/ACKS dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse – Merci !

ANNEXE C

Histoire et licence

C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir https://www.cwi.nl/) au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir https://www.cnri.reston.va.us/) de Reston, en Viriginie, d'où il diffusa plusieurs versions du logiciel.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation ; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

Toutes les versions de Python sont Open Source (voir https://www.opensource.org/ pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et supérieur	2.1.1	2001-maintenant	PSF	oui

Note : Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non compatible GPL » ne le peuvent pas.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   2.7.18 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 2.7.18 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 2.7.18 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 2.7.18 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   2.7.18.

4. PSF is making Python 2.7.18 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, OR ANY␣
↪DERIVATIVE
```

```
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
```

(suite sur la page suivante)

(suite de la page précédente)

```
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
```

(suite sur la page suivante)

(suite de la page précédente)

```
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 Licences et Remerciements pour les logiciels inclus

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

C.3.1 Mersenne twister

Le module `_random` inclut du code construit à partir d'un téléchargement depuis http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Voici mot pour mot les commentaires du code original :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

(suite sur la page suivante)

(suite de la page précédente)

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Interfaces de connexion (*sockets*)

Le module `socket` utilise les fonctions `getaddrinfo()` et `getnameinfo()` codées dans des fichiers source séparés et provenant du projet WIDE : http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.
```

(suite sur la page suivante)

(suite de la page précédente)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Virgule flottante et contrôle d'exception

Le code source pour le module `fpectl` inclut la note suivante :

```
  ----------------------------------------------------------------------
 /                      Copyright (c) 1996.                           \
|          The Regents of the University of California.                |
|                        All rights reserved.                          |
|                                                                      |
|   Permission to use, copy, modify, and distribute this software for  |
|   any purpose without fee is hereby granted, provided that this en-  |
|   tire notice is included in all copies of any software which is or  |
|   includes  a  copy  or  modification  of  this software and in all  |
|   copies of the supporting documentation for such software.          |
|                                                                      |
|   This  work was produced at the University of California, Lawrence  |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48  |
|   between  the  U.S.  Department  of  Energy and The Regents of the  |
|   University of California for the operation of UC LLNL.             |
|                                                                      |
|                              DISCLAIMER                              |
|                                                                      |
|   This  software was prepared as an account of work sponsored by an  |
|   agency of the United States Government. Neither the United States  |
|   Government  nor the University of California nor any of their em-  |
|   ployees, makes any warranty, express or implied, or  assumes  any  |
|   liability  or  responsibility  for the accuracy, completeness, or  |
|   usefulness of any information,  apparatus,  product,  or  process  |
|   disclosed,  or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-  |
|   cial  products,  process,  or  service  by trade name, trademark,  |
|   manufacturer, or otherwise, does not  necessarily  constitute  or  |
|   imply  its endorsement, recommendation, or favoring by the United  |
|   States Government or the University of California. The views  and  |
|   opinions  of authors expressed herein do not necessarily state or  |
|   reflect those of the United States Government or  the  University  |
|   of  California,  and shall not be used for advertising or product  |
 \  endorsement purposes.                                             /
  ----------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice :

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Gestion de témoin (*cookie*)

The `Cookie` module contains the following notice :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.8 Les fonctions UUencode et UUdecode

Le module `uu` contient la note suivante :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(suite sur la page suivante)

(suite de la page précédente)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

The `xmlrpclib` module contains the following notice :

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.10 test_epoll

Le module `test_epoll` contient la note suivante :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(suite sur la page suivante)

(suite de la page précédente)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Select kqueue

Le module `select` contient la note suivante pour l'interface kqueue :

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.12 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversions de *doubles* C vers et depuis les chaînes, et tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur http://www.netlib.org/fp/. Le fichier original, tel que récupéré le 16 mars 2009, contiens la licence suivante :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(suite sur la page suivante)

(suite de la page précédente)

```
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ************************************************************/
```

C.3.13 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
```

(suite sur la page suivante)

(suite de la page précédente)

```
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
   * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
   * OF THE POSSIBILITY OF SUCH DAMAGE.
   * ====================================================================
   *
   * This product includes cryptographic software written by Eric Young
   * (eay@cryptsoft.com).  This product includes software written by Tim
   * Hudson (tjh@cryptsoft.com).
   *
   */

Original SSLeay License
-----------------------

  /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
   * All rights reserved.
   *
   * This package is an SSL implementation written
   * by Eric Young (eay@cryptsoft.com).
   * The implementation was written so as to conform with Netscapes SSL.
   *
   * This library is free for commercial and non-commercial use as long as
   * the following conditions are aheared to.  The following conditions
   * apply to all code found in this distribution, be it the RC4, RSA,
   * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
   * included with this distribution is covered by the same copyright terms
   * except that the holder is Tim Hudson (tjh@cryptsoft.com).
   *
   * Copyright remains Eric Young's, and as such any Copyright notices in
   * the code are not to be removed.
   * If this package is used in a product, Eric Young should be given attribution
   * as the author of the parts of the library used.
   * This can be in the form of a textual message at program startup or
   * in documentation (online or textual) provided with the package.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   * 1. Redistributions of source code must retain the copyright
   *    notice, this list of conditions and the following disclaimer.
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in the
   *    documentation and/or other materials provided with the distribution.
   * 3. All advertising materials mentioning features or use of this software
   *    must display the following acknowledgement:
```

(suite sur la page suivante)

(suite de la page précédente)

```
   *    "This product includes cryptographic software written by
   *     Eric Young (eay@cryptsoft.com)"
   *    The word 'cryptographic' can be left out if the rouines from the library
   *    being used are not cryptographic related :-).
   * 4. If you include any Windows specific code (or a derivative thereof) from
   *    the apps directory (application code) you must include an acknowledgement:
   *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
   *
   * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
   * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
   * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
   * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
   * SUCH DAMAGE.
   *
   * The licence and distribution terms for any publically available version or
   * derivative of this code cannot be changed.  i.e. this code cannot simply be
   * copied and put under another distribution licence
   * [including the GNU Public Licence.]
   */
```

C.3.14 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

Le module `zlib` est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

ANNEXE D

Copyright

Python et cette documentation sont :

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Index

Non alphabétique

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V