# Python Frequently Asked Questions

*Version 3.13.3*

**Guido van Rossum and the Python development team**

**avril 27, 2025**

**Python Software Foundation**
**Email : docs@python.org**

# Table des matières


**1 FAQ générale sur Python** **1**
- 1.1 Informations générales . . . 1
  - 1.1.1 Qu'est-ce que Python ? . . . 1
  - 1.1.2 Qu'est ce que la Python Software Foundation ? . . . 1
  - 1.1.3 Existe-t-il des restrictions liées à la propriété intellectuelle quant à l'utilisation de Python ? 1
  - 1.1.4 Pourquoi Python a été créé ? . . . 2
  - 1.1.5 Pour quoi Python est-il fait ? . . . 2
  - 1.1.6 Comment fonctionne le numérotage des versions de Python ? . . . 2
  - 1.1.7 Comment obtenir une copie du code source de Python ? . . . 3
  - 1.1.8 Comment obtenir la documentation de Python ? . . . 3
  - 1.1.9 Je n'ai jamais programmé avant. Existe-t-il un tutoriel Python ? . . . 3
  - 1.1.10 Y a-t-il un forum ou une liste de diffusion dédié à Python ? . . . 3
  - 1.1.11 Comment obtenir une version bêta test de Python ? . . . 3
  - 1.1.12 Comment soumettre un rapport de bogues ou un correctif pour Python ? . . . 4
  - 1.1.13 Existe-t-il des articles publiés au sujet de Python auxquels je peux me référer ? . . . 4
  - 1.1.14 Y a-t-il des livres au sujet de Python ? . . . 4
  - 1.1.15 Où www.python.org est-il localisé dans le monde ? . . . 4
  - 1.1.16 Pourquoi le nom Python ? . . . 4
  - 1.1.17 Dois-je aimer "Monty Python's Flying Circus" ? . . . 4
- 1.2 Python c'est le monde réel . . . 4
  - 1.2.1 Quel est le niveau de stabilité de Python ? . . . 4
  - 1.2.2 Combien de personnes utilisent Python ? . . . 4
  - 1.2.3 Y a-t-il un nombre de projets significatif réalisés en Python ? . . . 5
  - 1.2.4 Quelles sont les nouveautés en développement attendues pour Python ? . . . 5
  - 1.2.5 Est-il raisonnable de proposer des changements incompatibles dans Python ? . . . 5
  - 1.2.6 Existe-t-il un meilleur langage de programmation pour les programmeurs débutants ? . . . 5

**2 FAQ de programmation** **7**
- 2.1 Questions générales . . . 7
  - 2.1.1 Existe-t-il un débogueur de code source avec points d'arrêts, exécution pas-à-pas, etc. ? . . . 7
  - 2.1.2 Existe-t-il des outils pour aider à trouver des bogues ou faire de l'analyse statique de code ? 7
  - 2.1.3 Comment créer un binaire autonome à partir d'un script Python ? . . . 8
  - 2.1.4 Existe-t-il des normes de développement ou un guide de style pour écrire des programmes Python ? . . . 8
- 2.2 Fondamentaux . . . 8
  - 2.2.1 Pourquoi une UnboundLocalError est levée alors qu'une variable a une valeur ? . . . 8
  - 2.2.2 Quelles sont les règles pour les variables locales et globales en Python ? . . . 9
  - 2.2.3 Pourquoi des expressions lambda définies dans une boucle avec des valeurs différentes retournent-elles le même résultat ? . . . 9
  - 2.2.4 Comment partager des variables globales entre modules ? . . . 10
  - 2.2.5 Quelles sont les « bonnes pratiques » pour utiliser import dans un module ? . . . 11

2.2.6 Pourquoi les arguments par défaut sont-ils partagés entre les objets ? . . . 11
2.2.7 Comment passer des paramètres optionnels ou nommés d'une fonction à l'autre ? . . . 12
2.2.8 Quelle est la différence entre les arguments et les paramètres ? . . . 12
2.2.9 Pourquoi modifier la liste 'y' modifie aussi la liste 'x' ? . . . 13
2.2.10 Comment écrire une fonction qui modifie ses paramètres ? (passage par référence) . . . 14
2.2.11 Comment construire une fonction d'ordre supérieur en Python ? . . . 15
2.2.12 Comment copier un objet en Python ? . . . 16
2.2.13 Comment récupérer les méthodes ou les attributs d'un objet ? . . . 16
2.2.14 Comment un code peut-il obtenir le nom d'un objet ? . . . 16
2.2.15 Qu'en est-il de la précédence de l'opérateur virgule ? . . . 17
2.2.16 Existe-t-il un équivalent à l'opérateur ternaire « ?: » du C ? . . . 17
2.2.17 Est-il possible d'écrire des programmes obscurcis (*obfuscated*) d'une ligne en Python ? . . . 17
2.2.18 Que signifie la barre oblique (/) dans la liste des paramètres d'une fonction ? . . . 18
2.3 Nombres et chaînes de caractères . . . 18
2.3.1 Comment écrire des entiers hexadécimaux ou octaux ? . . . 18
2.3.2 Pourquoi `-22 // 10` donne-t-il `-3` ? . . . 19
2.3.3 Pourquoi ai-je une erreur de syntaxe en essayant de lire un attribut d'un entier littéral ? . . . 19
2.3.4 Comment convertir une chaîne de caractères en nombre ? . . . 19
2.3.5 Comment convertir un nombre en chaîne de caractères ? . . . 19
2.3.6 Comment modifier une chaîne de caractères « sur place » ? . . . 20
2.3.7 Comment utiliser des chaînes de caractères pour appeler des fonctions/méthodes ? . . . 20
2.3.8 Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings ? . . . 21
2.3.9 Is there a `scanf()` or `sscanf()` equivalent ? . . . 21
2.3.10 What does `UnicodeDecodeError` or `UnicodeEncodeError` error mean ? . . . 21
2.3.11 Can I end a raw string with an odd number of backslashes ? . . . 21
2.4 Performances . . . 22
2.4.1 Mon programme est trop lent. Comment l'accélérer ? . . . 22
2.4.2 Quelle est la manière la plus efficace de concaténer un grand nombre de chaînes de caractères ? . . . 23
2.5 Séquences (*n*-uplets / listes) . . . 23
2.5.1 Comment convertir les listes en *n*-uplets et inversement ? . . . 23
2.5.2 Qu'est-ce qu'un indice négatif ? . . . 23
2.5.3 Comment itérer à rebours sur une séquence ? . . . 23
2.5.4 Comment retirer les doublons d'une liste ? . . . 24
2.5.5 Comment retirer les doublons d'une liste . . . 24
2.5.6 Comment construire un tableau en Python ? . . . 24
2.5.7 Comment créer une liste à plusieurs dimensions ? . . . 25
2.5.8 Comment appliquer une méthode ou une fonction à une séquence d'objets ? . . . 25
2.5.9 Pourquoi `a_tuple[i] += ['item']` lève-t-il une exception alors que l'addition fonctionne ? . . . 25
2.5.10 Je souhaite faire un classement compliqué : peut-on faire une transformation de Schwartz en Python ? . . . 27
2.5.11 Comment ordonner une liste en fonction des valeurs d'une autre liste ? . . . 27
2.6 Objets . . . 27
2.6.1 Qu'est-ce qu'une classe ? . . . 27
2.6.2 Qu'est-ce qu'une méthode ? . . . 27
2.6.3 Qu'est-ce que self ? . . . 28
2.6.4 Comment vérifier si un objet est une instance d'une classe donnée ou d'une sous-classe de celle-ci ? . . . 28
2.6.5 Qu'est-ce que la délégation ? . . . 29
2.6.6 Comment appeler une méthode définie dans une classe mère depuis une classe dérivée qui la surcharge ? . . . 30
2.6.7 Comment organiser un code pour permettre de changer la classe mère plus facilement ? . . . 30
2.6.8 Comment créer des données statiques de classe et des méthodes statiques de classe ? . . . 30
2.6.9 Comment surcharger les constructeurs (ou méthodes) en Python ? . . . 31
2.6.10 J'essaie d'utiliser `__spam` et j'obtiens une erreur à propos de `_SomeClassName__spam`. . . . 31
2.6.11 Ma classe définit `__del__` mais elle n'est pas appelée lorsque je supprime l'objet. . . . 32
2.6.12 Comment obtenir toutes les instances d'une classe ? . . . 32
2.6.13 Pourquoi le résultat de `id()` peut-il être le même pour deux objets différents ? . . . 32

2.6.14 Quand puis-je raisonnablement utiliser le test d'identité *is* ? . . . 33
2.6.15 Comment définir dans une classe fille les attributs d'une instance immuable ? . . . 34
2.6.16 Comment mettre en cache le résultat d'une méthode ? . . . 35
2.7 Modules . . . 36
2.7.1 Comment créer des fichiers `.pyc` ? . . . 36
2.7.2 Comment obtenir le nom du module actuel ? . . . 37
2.7.3 Comment avoir des modules qui s'importent mutuellement ? . . . 37
2.7.4 `__import__('x.y.z')` renvoie `<module 'x'>` ; comment accéder à `z` ? . . . 38
2.7.5 Quand j'édite un module et que je le réimporte, je ne vois pas les changements. Pourquoi ? 38

**3 FAQ sur l'histoire et la conception** **39**
3.1 Pourquoi Python utilise-t-il l'indentation pour grouper les instructions ? . . . 39
3.2 Pourquoi ai-je d'étranges résultats suite à de simples opérations arithmétiques ? . . . 39
3.3 Pourquoi les calculs à virgules flottantes sont si imprécis ? . . . 40
3.4 Pourquoi les chaînes de caractères Python sont-elles immuables ? . . . 40
3.5 Pourquoi *self* doit-il être explicitement utilisé dans les définitions et les appels de méthodes ? . . . 40
3.6 Pourquoi ne puis-je pas utiliser d'assignation dans une expression ? . . . 41
3.7 Pourquoi Python utilise des méthodes pour certaines fonctionnalités (ex : `list.index()`) mais des fonctions pour d'autres (ex : `len(list)`) ? . . . 41
3.8 Pourquoi `join()` est une méthode de chaîne plutôt qu'une méthode de liste ou de *n*-uplet ? . . . 41
3.9 À quel point les exceptions sont-elles rapides ? . . . 42
3.10 Pourquoi n'y a-t-il pas une instruction *switch* ou une structure similaire à *switch / case* en Python ? . . . 42
3.11 Est-il possible d'émuler des fils d'exécution dans l'interpréteur plutôt que se baser sur les implémentations spécifiques aux systèmes d'exploitation ? . . . 43
3.12 Pourquoi les expressions lambda ne peuvent pas contenir d'instructions ? . . . 43
3.13 Python peut-il être compilé en code machine, en C ou dans un autre langage ? . . . 43
3.14 Comment Python gère la mémoire ? . . . 44
3.15 Pourquoi CPython n'utilise-t-il pas un modèle de ramasse-miettes plus traditionnel ? . . . 44
3.16 Pourquoi toute la mémoire n'est pas libérée lorsque *CPython* s'arrête ? . . . 44
3.17 Pourquoi les *n*-uplets et les *list* sont deux types de données séparés ? . . . 45
3.18 Comment les listes sont-elles implémentées dans CPython ? . . . 45
3.19 Comment les dictionnaires sont-ils implémentés dans CPython ? . . . 45
3.20 Pourquoi les clés du dictionnaire sont immuables ? . . . 45
3.21 Pourquoi `list.sort()` ne renvoie pas la liste triée ? . . . 47
3.22 Comment spécifier une interface et appliquer une spécification d'interface en Python ? . . . 47
3.23 Pourquoi n'y a-t-il pas de `goto` en Python ? . . . 47
3.24 Pourquoi les chaînes de caractères brutes (r-strings) ne peuvent-elles pas se terminer par un *backslash* ? 48
3.25 Pourquoi la déclaration `with` pour les assignations d'attributs n'existe pas en Python ? . . . 48
3.26 Pourquoi l'instruction `with` ne prend-elle pas en charge les générateurs ? . . . 49
3.27 Pourquoi les deux-points sont-ils nécessaires pour les déclarations `if/while/def/class` ? . . . 49
3.28 Pourquoi Python permet-il les virgules à la fin des listes et des *n*-uplets ? . . . 50

**4 FAQ sur la bibliothèque et les extensions** **51**
4.1 Questions générales sur la bibliothèque . . . 51
4.1.1 Comment trouver un module ou une application pour effectuer la tâche X ? . . . 51
4.1.2 Où sont stockés les fichiers sources *math.py*, *socket.py*, *regex.py*, etc ? . . . 51
4.1.3 Comment rendre un script Python exécutable sous Unix ? . . . 51
4.1.4 Existe-t-il un module *curses* ou *termcap* en Python ? . . . 52
4.1.5 Existe-t-il un équivalent à la fonction C `onexit()` en Python ? . . . 52
4.1.6 Pourquoi mes gestionnaires de signaux ne fonctionnent-ils pas ? . . . 52
4.2 Tâches fréquentes . . . 52
4.2.1 Comment tester un programme ou un composant Python ? . . . 52
4.2.2 Comment générer la documentation à partir des *docstrings* ? . . . 53
4.2.3 Comment détecter qu'une touche est pressée ? . . . 53
4.3 Fils d'exécution . . . 53
4.3.1 Comment programmer avec des fils d'exécution ? . . . 53
4.3.2 Aucun de mes fils ne semble s'exécuter : pourquoi ? . . . 53
4.3.3 Comment découper et répartir une tâche au sein d'un ensemble de fils d'exécutions ? . . . 54

4.3.4 Quels types de mutations sur des variables globales sont compatibles avec les programmes à fils d'exécution multiples ? sécurisé ? . . . 55
4.3.5 Pourquoi ne pas se débarrasser du verrou global de l'interpréteur ? . . . 56
4.4 Les entrées/sorties . . . 57
4.4.1 Comment supprimer un fichier ? (et autres questions sur les fichiers…) . . . 57
4.4.2 Comment copier un fichier ? . . . 57
4.4.3 Comment lire (ou écrire) des données binaires ? . . . 57
4.4.4 Il me semble impossible d'utiliser `os.read()` sur un tube créé avec `os.popen()` ; pourquoi ? . . . 58
4.4.5 Comment accéder au port de transmission en série (RS-232) ? . . . 58
4.4.6 Pourquoi fermer *sys.stdout*, *sys.stdin*, *sys.stderr* ne les ferme pas réellement ? . . . 58
4.5 Programmation réseau et Internet . . . 58
4.5.1 Quels sont les outils Python dédiés à au web ? . . . 58
4.5.2 Quel module utiliser pour générer du HTML ? . . . 58
4.5.3 Comment envoyer un courriel avec un script Python ? . . . 58
4.5.4 Comment éviter de bloquer dans la méthode `connect()` d'un connecteur réseau ? . . . 59
4.6 Bases de données . . . 60
4.6.1 Existe-t-il des modules Python pour s'interfacer avec des bases de données ? . . . 60
4.6.2 Comment implémenter la persistance d'objets en Python ? . . . 60
4.7 Mathématiques et calcul numérique . . . 60
4.7.1 Comment générer des nombres aléatoires en Python ? . . . 60

**5 FAQ extension/intégration** **61**
5.1 Puis-je créer mes propres fonctions en C ? . . . 61
5.2 Puis-je créer mes propres fonctions en C++ ? . . . 61
5.3 Écrire directement en C est difficile ; existe-t-il des alternatives ? . . . 61
5.4 Comment puis-je exécuter des instructions quelconques Python à partir de C ? . . . 61
5.5 Comment puis-je évaluer une expression quelconque de Python à partir de C ? . . . 62
5.6 Comment puis-je extraire des donnés en C d'un objet Python ? . . . 62
5.7 Comment utiliser Py_BuildValue() pour créer un *n*-uplet de longueur définie ? . . . 62
5.8 Comment puis-je appeler la méthode d'un objet à partir de C ? . . . 62
5.9 Comment puis-je récupérer la sortie de `PyErr_Print()` (ou tout ce qui s'affiche sur *stdout*/*stderr*) ? 63
5.10 Comment accéder à un module écrit en Python à partir de C ? . . . 63
5.11 Comment s'interfacer avec les objets C++ depuis Python ? . . . 63
5.12 J'ai ajouté un module en utilisant le fichier *Setup* et la compilation échoue ; pourquoi ? . . . 64
5.13 Comment déboguer une extension ? . . . 64
5.14 Je veux compiler un module Python sur mon système Linux, mais il manque certains fichiers. Pourquoi ? . . . 64
5.15 Comment distinguer une « entrée incomplète » (*incomplete input*) d'une « entrée invalide » (*invalid input*) ? . . . 64
5.16 Comment puis-je trouver les symboles g++ indéfinis `__builtin_new` ou `__pure_virtual` ? . . . 64
5.17 Puis-je créer une classe d'objets avec certaines méthodes implémentées en C et d'autres en Python (p. ex. en utilisant l'héritage) ? . . . 65

**6 FAQ : Python et Windows** **67**
6.1 Comment exécuter un programme Python sous Windows ? . . . 67
6.2 Comment rendre des scripts Python exécutables ? . . . 68
6.3 Pourquoi Python met-il du temps à démarrer ? . . . 68
6.4 Comment construire un exécutable depuis un script Python ? . . . 68
6.5 Est-ce qu'un fichier `*.pyd` est la même chose qu'une DLL ? . . . 69
6.6 Comment puis-je intégrer Python dans une application Windows ? . . . 69
6.7 Comment empêcher mon éditeur d'utiliser des tabulations dans mes fichiers Python ? . . . 70
6.8 Comment puis-je vérifier de manière non bloquante qu'une touche a été pressée ? . . . 70
6.9 Comment résoudre l'erreur « `api-ms-win-crt-runtime-l1-1-0.dll` manquante » ? . . . 70

**7 FAQ interface graphique** **71**
7.1 Questions générales sur l'interface graphique . . . 71
7.2 Quelles boîtes à outils IUG existent pour Python ? . . . 71
7.3 Questions à propos de *Tkinter* . . . 71

7.3.1 Comment puis-je geler (*freezer*) les applications *Tkinter* ? . . . 71
7.3.2 Puis-je modifier des événements *Tk* pendant l'écoute des *E/S* ? . . . 71
7.3.3 Je n'arrive pas à faire fonctionner les raccourcis clavier dans *Tkinter* : pourquoi ? . . . 71

**8 FAQ "Pourquoi Python est installé sur mon ordinateur ?" 73**
8.1 Qu'est-ce que Python ? . . . 73
8.2 Pourquoi Python est installé sur ma machine ? . . . 73
8.3 Puis-je supprimer Python ? . . . 73

**A Glossaire 75**

**B About this documentation 93**
B.1 Contributors to the Python documentation . . . 93

**C Histoire et licence 95**
C.1 Histoire du logiciel . . . 95
C.2 Conditions générales pour accéder à, ou utiliser, Python . . . 96
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 96
C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0 . . . 97
C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1 . . . 97
C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2 . . . 99
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . . . 99
C.3 Licences et remerciements pour les logiciels tiers . . . 99
C.3.1 Mersenne twister . . . 99
C.3.2 Interfaces de connexion (*sockets*) . . . 100
C.3.3 Interfaces de connexion asynchrones . . . 101
C.3.4 Gestion de témoin (*cookie*) . . . 101
C.3.5 Traçage d'exécution . . . 102
C.3.6 Les fonctions UUencode et UUdecode . . . 102
C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*) . . . 103
C.3.8 test_epoll . . . 103
C.3.9 Select kqueue . . . 104
C.3.10 SipHash24 . . . 104
C.3.11 *strtod* et *dtoa* . . . 105
C.3.12 OpenSSL . . . 105
C.3.13 expat . . . 108
C.3.14 libffi . . . 109
C.3.15 zlib . . . 109
C.3.16 cfuhash . . . 110
C.3.17 libmpdec . . . 111
C.3.18 Ensemble de tests C14N du W3C . . . 111
C.3.19 mimalloc . . . 112
C.3.20 *asyncio* . . . 112
C.3.21 Global Unbounded Sequences (GUS) . . . 113

**D Copyright 115**

**Index 117**

CHAPITRE 1

# FAQ générale sur Python

## 1.1 Informations générales

### 1.1.1 Qu'est-ce que Python ?

Python est un langage de programmation interprété, interactif et orienté objet. Il intègre des modules, des exceptions, un typage dynamique, des types de données dynamiques de très haut niveau et des classes. Il gère de multiples paradigmes de programmation au-delà de la programmation orientée objet, tels que la programmation procédurale et fonctionnelle. Python combine une puissance remarquable avec une syntaxe très claire. Il possède des interfaces avec de nombreux appels système et bibliothèques, ainsi qu'avec divers systèmes d'interfaces graphiques, et il peut être étendu avec du C ou du C++. Il est également utilisable comme langage d'extension pour les applications qui nécessitent une interface de programmation. Enfin, Python est portable : il fonctionne sur de nombreuses variantes d'Unix, y compris Linux et macOS, et sur Windows.

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-t-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.

See the license page to find further explanations and the full text of the PSF License.

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.

Le langage vient avec une bibliothèque standard importante qui couvre des domaines tels que le traitement des chaînes de caractères (expressions régulières, Unicode, calcul de différences entre les fichiers), les protocoles internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, script CGI), ingénierie logicielle (tests unitaires, enregistrement, analyse de code Python), et interfaces pour systèmes d'exploitation (appels système, système de fichiers, connecteurs TCP/IP). Regardez la table des matières library-index pour avoir une idée de ce qui est disponible. Une grande variété de greffons tiers existent aussi. Consultez le sommaire des paquets Python pour trouver les paquets qui pourraient vous intéresser.

### 1.1.6 Comment fonctionne le numérotage des versions de Python ?

Python versions are numbered "A.B.C" or "A.B" :

- *A* is the major version number -- it is only incremented for really major changes in the language.
- *B* is the minor version number -- it is incremented for less earth-shattering changes.
- *C* is the micro version number -- it is incremented for each bugfix release.

Not all releases are bugfix releases. In the run-up to a new feature release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized ; it's not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix :

- The suffix for an alpha version is "aN" for some small number *N*.
- The suffix for a beta version is "bN" for some small number *N*.

— The suffix for a release candidate version is ”rcN” for some small number *N*.

In other words, all versions labeled *2.0aN* precede the versions labeled *2.0bN*, which precede versions labeled *2.0rcN*, and *those* precede 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*.

See the Developer's Guide for more information about the development cycle, and **PEP 387** to learn more about Python's backward compatibility policy. See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

### 1.1.7 Comment obtenir une copie du code source de Python ?

La dernière version du code source déployée est toujours disponible sur python.org, à https://www.python.org/downloads/. Le code source de la dernière version en développement peut être obtenue à 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.

Consultez la section Premiers pas du Guide des Développeurs Python pour plus d'informations sur comment obtenir le code source et le compiler.

### 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. 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.

Announcements of new software releases and events can be found in comp.lang.python.announce, a low-traffic moderated list that receives about five postings per day. It's available as the python-announce mailing list.

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.

Vous pouvez aussi accéder aux de Python en développement grâce à Git. Voir Le Guide du Développeur Python pour plus de détails.

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

Pour signaler un bogue ou soumettre un correctif, utilisez le système de suivi des problèmes à l'adresse https://github.com/python/cpython/issues.

Pour davantage d'informations sur comment Python est développé, consultez le Guide du Développeur Python.

### 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 sur Python a été écrit en 1991 et est aujourd'hui assez dépassé.

> Guido van Rossum et 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 ?

L'infrastructure du projet Python est située dans le monde entier et est gérée par l'équipe de l'infrastructure Python. Plus de détails ici.

### 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 ?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. As of version 3.9, Python will have a new feature release every 12 months (**PEP 602**).

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. Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore**.

### 1.2.2 Combien de personnes utilisent Python ?

Il y a probablement des millions d'utilisateurs, bien qu'il soit difficile d'en déterminer le 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-rendus des conférences Python précédentes il s'avère que les contributions proviennent de nombreux organismes et entreprises divers.

Les projets Python les plus connus 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 les propositions d'amélioration de Python (« *Python Enhancement Proposals* », ou *PEP*) sur https://peps.python.org/. Les PEP 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.

New development is discussed on the python-dev mailing list.

### 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__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

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

Avec l'interpréteur, la documentation n'est jamais loin des étudiants quand ils travaillent.

Il y a aussi de bons environnements de développement intégrés (EDIs) pour Python. IDLE est un EDI multiplate-formes pour Python qui est écrit en Python en utilisant Tkinter. Les utilisateurs d'Emacs 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.

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.

Plusieurs débogueurs sont décrits ci-dessous et la fonction native `breakpoint()` permet d'utiliser n'importe lequel d'entre eux.

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/idle3) 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 pywin32 et fait partie de la distribution ActivePython.

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

trepan3k est un débogueur semblable à GDB.

Visual Studio Code est un EDI qui contient des outils de débogage. Il sait interagir avec les outils de gestion de versions.

Il existe de nombreux EDI Python propriétaires qui embarquent un débogueur graphique, notamment :

- Wing IDE ;
- Komodo IDE ;
- PyCharm ;

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

Oui.

Pylint et Pyflakes font des vérifications de base et vous aideront à trouver des bogues plus tôt.

Static type checkers such as Mypy, Pyre, and Pytype can check type hints in Python source code.

### 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 ; avec un compilateur C, vous pouvez 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'importation (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.

Voici quelques paquets qui permettent de créer des exécutables en ligne de commande comme graphiques :

- Nuitka (multiplateformes) ;
- PyInstaller (Cross-platform)
- PyOxidizer (multi-plateforme) ;
- cx_Freeze (multi-plateforme) ;
- py2app (uniquement pour macOS) ;
- py2exe (uniquement pour Windows).

### 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**.

## 2.2 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
```

results in an `UnboundLocalError` :

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

Cela est dû au fait que, quand une variable est affectée dans un contexte, cette variable devient locale à ce contexte et remplace toute variable du même nom du contexte appelant. Vu que la dernière instruction dans *foo* affecte une nouvelle valeur à `x`, le compilateur la traite comme une nouvelle variable. Par conséquent, quand le `print(x)` essaye d'afficher la variable non initialisée, une erreur se produit.

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
```

Une alternative dans un contexte imbriqué consiste à utiliser le mot-clé `nonlocal` :

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
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 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`, c.-à-d. `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 — p. ex. `sys`, `os`, `getopt`, `re`
2. les modules externes (tout ce qui est installé dans le dossier *site-packages* de Python) — par exemple `dateutil`, `requests`, `PIL.Image`
3. les modules développés en local

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 importations ne devraient être déplacées dans un espace de nommage 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 importations 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 nouveaux 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 modifiables 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 modifiables 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 can only provide two parameters and optionally pass _cache by keyword
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)
```

### 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 d'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 *mutable*, ce qui signifie que leur contenu peut être modifié.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

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 mutable (`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 a 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 façons d'en émuler un.

1) En renvoyant un *n*-uplet de résultats :

```
>>> def func1(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
>>> func1(x, y)
('new-value', 100)
```

   C'est presque toujours la meilleure solution.

2) En utilisant des variables globales. Cette approche ne fonctionne pas dans des contextes à plusieurs fils d'exécution (elle n'est pas *thread-safe*), et n'est donc pas recommandée.

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

```
>>> def func2(a):
...     a[0] = 'new-value'      # 'a' references a mutable list
...     a[1] = a[1] + 1         # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['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)
>>> args
{'a': 'new-value', 'b': 100}
```

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

```
>>> class Namespace:
...     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 Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

Faire quelque chose d'aussi compliqué est rarement une bonne idée.

La meilleure option reste de renvoyer un *n*-uplet contenant les différents résultats.

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

Deux possibilités : on peut utiliser des portées imbriquées ou bien des objets appelables. Par exemple, supposons que l'on souhaite 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):
        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 leurs 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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()`. Tous les objets ne peuvent pas être copiés, mais la plupart le peuvent.

Certains objets peuvent être copiés plus facilement que d'autres. 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 récupérer les méthodes ou les attributs 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 un code peut-il obtenir le nom d'un objet ?

C'est impossible en général, parce qu'un objet n'a pas de nom à proprement parler. Schématiquement, l'affectation fait correspondre un nom à une valeur ; c'est vrai aussi pour les instructions `def` et `class`, à la différence près que, dans ce cas, la valeur est un appelable. Par exemple, dans le code suivant :

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

Affirmer que la classe a un nom est discutable. Bien qu'elle soit liée à deux noms, et qu'elle soit appelée via le nom `B`, l'instance créée 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 étant attachés à la même valeur.

De façon générale, une application ne devrait pas avoir besoin de « connaître le nom » d'une valeur particulière. À moins d'être délibérément en train d'écrire un programme introspectif, c'est souvent l'indication qu'un changement d'approche serait 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 peut 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 nommage) si c'est leur chat (objet)…
>
> …et ne soyez pas surpris si vous découvrez qu'il est connu sous plusieurs noms, ou s'il n'a 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 le code suivant :

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

Comme la virgule n'est pas un opérateur, mais un séparateur entre deux expressions, 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'affectation (=, += etc). Ce ne sont pas vraiment des opérateurs mais plutôt des délimiteurs syntaxiques dans les instructions d'affectation.

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

Oui. Sa syntaxe est la suivante :

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

x, y = 50, 25
small = x if x < y else y
```

Avant l'introduction de cette syntaxe dans Python 2.5, il était courant d'utiliser les opérateurs de logique :

```
[expression] and [on_true] or [on_false]
```

Cet idiome est dangereux, car il donne un résultat erroné quand *on_true* a la valeur booléenne fausse. Il faut donc toujours utiliser la forme `... if ... else ...`.

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

Oui. C'est souvent le cas en imbriquant des `lambda` dans des `lambda`. Par exemple les trois morceaux de code suivants, légèrement adaptés par Ulf Bartelt :

```
from functools import reduce

# Primes < 1000
print(list(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(list(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+'\n'+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))
```

(suite sur la page suivante)

(suite de la page précédente)

```
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |_________ maximum of "iterations"
#         |          |________________ range on y axis
#         |___________________________ range on x axis
```

Les enfants, ne faites pas ça chez vous !

### 2.2.18 Que signifie la barre oblique (/) dans la liste des paramètres d'une fonction ?

Une barre oblique dans la liste des arguments d'une fonction indique que les paramètres la précédant sont uniquement positionnels. Les paramètres uniquement positionnels ne peuvent pas être référencés par leur nom depuis l'extérieur. Lors de l'appel d'une fonction qui accepte des paramètres uniquement positionnels, les arguments sont affectés aux paramètres en fonction de leur position. Par exemple, la fonction `divmod()` n'accepte que des paramètres uniquement positionnels. Sa documentation est la suivante :

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

La barre oblique à la fin de la liste des paramètres signifie que les trois paramètres sont uniquement positionnels. Et donc, appeler `divmod()` avec des arguments nommés provoque une erreur :

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

## 2.3 Nombres et chaînes de caractères

### 2.3.1 Comment 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 pour affecter la valeur octale "10" (8 en décimal) à la variable "a", tapez :

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

L'hexadécimal est tout aussi simple, faites précéder le nombre hexadécimal par un zéro, puis un "x" majuscule ou minuscule. Les nombres hexadécimaux peuvent être écrits 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 dû à la volonté que `i % j` ait le même signe que j. Si vous voulez en plus que :

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

alors la division entière doit renvoyer l'entier inférieur. Le C impose également que cette égalité soit vérifiée, et donc les compilateurs 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 affiche 10 h maintenant, qu'affichait-elle il y a 200 heures ? `-190 % 12 == 2` est utile ; `-190 % 12 == -10` est un bogue en puissance.

### 2.3.3 Pourquoi ai-je une erreur de syntaxe en essayant de lire un attribut d'un entier littéral ?

Essayer d'utiliser l'opérateur d'accès à un attribut sur un entier littéral lève une `SyntaxError` car le point est compris comme un séparateur décimal en notation anglo-saxonne :

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

Il faut séparer l'entier du point, soit avec une espace, soit avec des parenthèses.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

### 2.3.4 Comment convertir une chaîne de caractères en nombre ?

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to a floating-point number, e.g. `float('144') == 144.0`.

Par défaut, ces fonctions interprètent les nombres comme des décimaux, de telle façon que `int('0144') == 144` et `int('0x144')` lève une `ValueError`. Le second argument (optionnel) de `int(string, base)` est la base dans laquelle convertir, donc `int('0x144', 16) == 324`. Si la base donnée est 0, le nombre est interprété selon les règles Python : un préfixe `0o` indique de l'octal et `0x` indique de l'hexadécimal.

N'utilisez pas la fonction native `eval()` pour convertir des chaînes de caractères en nombres. `eval()` est beaucoup plus lente et pose des problèmes de sécurité : quelqu'un pourrait vous envoyer une expression Python pouvant avoir des effets de bord indésirables. Par exemple, quelqu'un pourrait passer `__import__('os').system("rm -rf $HOME")` ce qui effacerait votre répertoire personnel.

`eval()` a aussi pour effet d'interpréter les nombres comme des expressions Python. Ainsi `eval('09')` produit une erreur de syntaxe, parce que Python ne permet pas les '0' en tête d'un nombre décimal (à l'exception du nombre '0').

### 2.3.5 Comment convertir un nombre en chaîne de caractères ?

Pour transformer, par exemple, le nombre `144` en la chaîne de caractères `'144'`, il faut utiliser la fonction native `str()`. Pour obtenir la représentation hexadécimale ou octale, il faut utiliser les fonctions natives `hex()` ou `oct()`. Pour des représentations non-conventionnelles, se référer aux sections f-strings et formatstrings, par exemple `"{:04d}".format(144)` produit `'0144'` et `"{:.3f}".format(1.0/3.0)` produit `'0.333'`.

### 2.3.6 Comment modifier une chaîne de caractères « sur place » ?

C'est impossible car les chaînes de caractères sont immuables. Dans la plupart des cas, il faut tout simplement construire une nouvelle chaîne à partir des morceaux de l'ancienne. Si toutefois vous avez besoin d'un objet capable de modifier de la donnée Unicode « sur place », essayez d'utiliser un objet `io.StringIO` ou le module `array` :

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

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

### 2.3.7 Comment utiliser des chaînes de caractères pour appeler des fonctions/méthodes ?

Il y a plusieurs façons de faire.

— La meilleure est d'utiliser un dictionnaire qui fait correspondre les chaînes de caractères à des fonctions. Le principal avantage de cette technique est que les chaînes n'ont pas besoin d'être égales aux noms de fonctions. C'est aussi la façon principale 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 inclut les classes, les instances de classes, les modules et ainsi de suite.

Ceci est utilisé à plusieurs reprises dans la bibliothèque standard, de cette façon :

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

(suite sur la page suivante)

(suite de la page précédente)

```
    def do_bar(self):
        ...

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

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

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

fname = "myFunc"

f = locals()[fname]
f()
```

### 2.3.8 Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings ?

Vous pouvez utiliser `S.rstrip("\r\n")` pour retirer toutes les occurrences de tout marqueur de fin de ligne à la fin d'une chaîne de caractère `S`, sans en enlever aucune espace. Si la chaîne `S` représente plus d'une ligne, avec plusieurs lignes vides, les marqueurs de fin de ligne de chaque ligne vide seront retirés :

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

Vu que cela ne sert presque qu'à lire un texte ligne à ligne, utiliser `S.rstrip()` de cette manière fonctionne correctement.

### 2.3.9 Is there a `scanf()` or `sscanf()` equivalent ?

Pas exactement.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

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

### 2.3.10 What does `UnicodeDecodeError` or `UnicodeEncodeError` error mean ?

Voir unicode-howto.

### 2.3.11 Can I end a raw string with an odd number of backslashes ?

A raw string ending with an odd number of backslashes will escape the string's quote :

```
>>> r'C:\this\will\not\work\'
  File "<stdin>", line 1
    r'C:\this\will\not\work\'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

There are several workarounds for this. One is to use regular strings and double the backslashes :

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Another is to concatenate a regular string containing an escaped backslash to the raw string :

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

It is also possible to use `os.path.join()` to append a backslash on Windows :

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Note that while a backslash will ”escape” a quote for the purposes of determining where the raw string ends, no escaping occurs when interpreting the value of the raw string. That is, the backslash remains present in the value of the raw string :

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Also see the specification in the language reference.

## 2.4 Performances

### 2.4.1 Mon programme est trop lent. Comment l'accélérer ?

Question difficile en général. Il faut garder en tête les points suivants avant d'aller plus loin :

- Les performances varient en fonction des implémentations de Python. Cette FAQ ne traite que de *CPython*.
- Les comportements peuvent différer d'un système d'exploitation à l'autre, tout particulièrement quand il s'agit d'entrée/sortie ou de fils d'exécution multiples.
- Il faut toujours essayer de trouver où sont les points de contention d'un programme *avant* d'essayer d'optimiser du code (voir le module `profile`).
- Écrire des scripts d'évaluation de performances permet de progresser rapidement dans la recherche d'améliorations (voir le module `timeit`).
- Il est très fortement recommandé d'avoir une bonne couverture de code (avec des tests unitaires ou autre) avant d'ajouter des erreurs dans des optimisations sophistiquées.

Ceci étant dit, il y a beaucoup d'astuces pour accélérer du code Python. Voici quelques principes généraux qui peuvent aider à atteindre des niveaux de performance satisfaisants :

- Améliorer les algorithmes (ou en changer pour des plus performants) peut produire de bien meilleurs résultats que d'optimiser çà et là de petites portions du code.
- Utiliser les structures de données adaptées. Se référer à la documentation des bltin-types et du module `collections`.
- Quand la bibliothèque standard fournit une implémentation pour quelque chose, il y a de fortes chances (même si ce n'est pas systématique) que cette implémentation soit plus rapide que la vôtre. C'est d'autant plus vrai pour les routines écrites en C, comme les routines natives et certaines extensions de types. Par exemple, il faut utiliser la méthode native `list.sort()` ou la fonction `sorted()` similaire pour classer (et se référer à la section sortinghowto pour des exemples d'utilisation courante).
- Les abstractions ont tendance à créer des indirections et obligent l'interpréteur à faire plus d'efforts. Si le niveau d'indirection dépasse la quantité de travail effectif, le programme sera ralenti. Il faut toujours éviter trop d'indirections, en particulier sous la forme de fonctions ou méthodes trop petites (qui nuisent aussi souvent à la clarté du code).

Si vous atteignez les limites de ce que du Python « pur » permet de faire, il y a des outils qui permettent d'aller plus loin. Par exemple, Cython peut compiler une version légèrement modifiée de code Python en une extension C et est disponible sur de nombreuses plate-formes. Cython peut bénéficier de la compilation (et de l'annotation, optionnelle, des types) pour rendre votre code beaucoup plus rapide que s'il était interprété. Si vous avez confiance en vos capacités de programmation en C, vous pouvez aussi écrire un module d'extension en C vous-même.

**Voir aussi**

La page wiki dédiée aux astuces de performance.

### 2.4.2 Quelle est la manière la plus efficace de concaténer un grand nombre de chaînes de caractères ?

Les objets `str` et `bytes` sont immuables, par conséquent concaténer un grand nombre de chaînes de caractères entre elles n'est pas très efficace car chaque concaténation crée un nouvel objet. Dans le cas général, la complexité est quadratique par rapport à la taille totale de la chaîne.

Pour mettre bout à bout un grand nombre d'objets `str`, la technique recommandée consiste à toutes les mettre dans une liste et appeler la méthode `str.join()` à la fin :

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(une autre technique relativement efficace consiste à utiliser `io.StringIO`)

Pour concaténer un grand nombre d'objets `bytes`, la technique recommandée consiste à étendre un objet `bytearray` en utilisant la concaténation en-place (l'opérateur +=) :

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

## 2.5 Séquences (*n*-uplets / listes)

### 2.5.1 Comment convertir les listes en *n*-uplets et inversement ?

Le constructeur de type `tuple(seq)` convertit toute séquence (plus précisément, tout itérable) en un *n*-uplet avec les mêmes éléments dans le même ordre.

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

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[:]`.

### 2.5.2 Qu'est-ce qu'un indice négatif ?

Les séquences Python sont indicées avec des nombres positifs aussi bien que négatifs. Pour les nombres positifs, 0 est le premier indice, 1 est le deuxième, et ainsi de suite. Pour les indices négatifs, `-1` est le dernier indice, `-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 indices négatifs peut être très pratique. Par exemple `S[:-1]` représente la chaîne tout entière à l'exception du dernier caractère, ce qui est pratique pour retirer un caractère de fin de ligne à la fin d'une chaîne.

### 2.5.3 Comment itérer à rebours sur une séquence ?

Utilisez la fonction native `reversed()` :

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

Cela ne modifie pas la séquence initiale, mais construit à la place une copie en ordre inverse pour itérer dessus.

### 2.5.4 Comment retirer les doublons d'une liste ?

Lisez le « livre de recettes » Python pour trouver une longue discussion sur les nombreuses approches possibles :

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

Si changer l'ordre de la liste ne vous dérange pas, commencez par ordonner 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 (c'est-à-dire, qu'elles sont toutes *hachables*) ceci est souvent plus rapide

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

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

### 2.5.5 Comment retirer les doublons d'une liste

Comme pour supprimer les doublons, il est possible d'itérer explicitement à l'envers avec une condition de suppression. Cependant, il est plus facile et plus rapide d'utiliser le remplacement des tranches par une itération avant, implicite ou explicite. Voici trois variantes :

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

La liste en compréhension est peut-être la plus rapide :

### 2.5.6 Comment construire un tableau en Python ?

Utilisez une liste :

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

Les listes ont un coût équivalent à celui des tableaux 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 NumPy (et d'autres) fournissent différentes structures de type tableaux, avec des caractéristiques différentes.

Pour obtenir des listes chaînées à la sauce Lisp, vous pouvez émuler les *cons cells* en utilisant des *n*-uplets :

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

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

### 2.5.7 Comment créer une liste à plusieurs dimensions ?

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

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

Elle semble correcte si on l'affiche :

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

Mais quand vous affectez une valeur, celle-ci apparaît à plusieurs endroits :

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

Dupliquer une liste en utilisant `*` ne crée en réalité pas de copie mais 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 très probablement pas ce que vous souhaitiez.

L'approche suggérée est d'abord de créer une liste de la longueur désirée, puis de remplir tous les éléments avec une nouvelle chaîne :

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

Cela génère une liste contenant elle-même trois listes distinctes, de longueur deux. Vous pouvez aussi utiliser la syntaxe des listes en compréhension :

```
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.

### 2.5.8 Comment appliquer une méthode ou une fonction à une séquence d'objets ?

Pour appeler une méthode ou une fonction et accumuler les valeurs renvoyées dans une liste, une *compréhension de liste* est une solution élégante :

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

result = [function(obj) for obj in mylist]
```

To just run the method or function without saving the return values, a plain `for` loop will suffice :

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

### 2.5.9 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 mutables et immuables en Python.

Cette discussion est valable, en général, quand des opérateurs d'affectation incrémentale sont appliqués aux éléments d'un *n*-uplet qui pointe sur des objets mutables, 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']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__()` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement ; and (b) for lists, `__iadd__()` is equivalent to calling `extend()` on the list and returning the list. That's why we say that for lists, `+=` is a "shorthand" for `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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 mutables.

### 2.5.10 Je souhaite faire un classement compliqué : peut-on faire une transformation de Schwartz en Python ?

Cette technique, attribuée à Randal Schwartz de la communauté Perl, ordonne les éléments d'une liste à l'aide une transformation qui fait correspondre chaque élément à sa "valeur de tri". En Python, ceci est géré par l'argument `key` de la méthode `list.sort()` :

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

### 2.5.11 Comment ordonner une liste en fonction des valeurs d'une autre liste ?

Fusionnez-les dans un itérateur de *n*-uplets, ordonnez la liste obtenue, puis choisissez l'élément que vous voulez

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

## 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) classe(s) de base. Elle hérite alors des attributs et des méthodes de ses classes mères. Cela permet à un modèle d'objet d'être successivement raffiné par héritage. Vous pourriez avoir une classe générique `Mailbox`, qui fournit des méthodes d'accès de base pour une boîte aux lettres, et des sous-classes telles que `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` qui gèrent les plusieurs formats spécifiques de boîtes aux lettres.

### 2.6.2 Qu'est-ce qu'une méthode ?

Une méthode est une fonction sur un objet `x` qu'on appelle de manière générale sous la forme `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 ?

Par convention, le premier argument d'une méthode est appelé self. Une méthode `meth(self, a, b, c)` doit être appelée sous la forme `x.meth(a, b, c)` où `x` est une instance de la classe dans laquelle cette méthode est définie ; tout se passe comme si la méthode était appelée comme `meth(x, a, b, c)`.

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

### 2.6.4 Comment vérifier si un objet est une instance d'une classe donnée ou d'une sous-classe de celle-ci ?

Utilisez la fonction native `isinstance(obj, cls)`. Vous pouvez vérifier qu'un objet est une instance de plusieurs classes à la fois en fournissant un *n*-uplet à la place d'une seule classe, par exemple, `isinstance(obj, (class1, class2, ...))`. Vous pouvez également vérifier qu'un objet est l'un des types natifs de Python, par exemple `isinstance(obj, str)` ou `isinstance(obj, (int, float, complex))`.

Notez que `isinstance()` prend aussi en compte l'héritage virtuel d'une *classe mère abstraite*, c'est à dire qu'elle renvoie `True` pour une classe A enregistrée auprès d'une classe mère abstraite B même si A n'est pas directement ou indirectement une classe fille de B. Pour vérifier l'héritage dans le sens plus restreint, parcourez l'*ordre de résolution des méthodes* de la classe :

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # direct
True
>>> isinstance(c, P)        # indirect
True
>>> isinstance(c, Mapping)  # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

Notez que la plupart des programmes n'utilisent que rarement `isInstance()` sur les classes définies par l'utilisateur. Si vous développez vous-même des classes, une approche plus orientée-objet consiste définir des méthodes sur les classes qui sont porteuses d'un comportement particulier, plutôt que de vérifier la classe de l'objet et de faire un traitement ad-hoc. 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()` dans toutes les classes et qu'il suffit d'appeler de la manière suivante :

```
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 appelée « patron de conception). Prenons un objet `x` dont on souhaite modifier le comportement d'une seule de ses méthodes. On peut créer une nouvelle classe qui fournit une nouvelle implémentation de la méthode qui nous intéresse dans l'évolution et qui délègue toute autre méthode à 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

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

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

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__()` method ; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following :

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

Many `__setattr__()` implementations call `object.__setattr__()` to set an attribute on self without causing infinite recursion :

```
class X:
    def __setattr__(self, name, value):
        # Custom logic here...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

### 2.6.6 Comment appeler une méthode définie dans une classe mère depuis une classe dérivée qui la surcharge ?

Utilisez la fonction native `super()` :

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

Dans cet exemple, `super()` détermine automatiquement l'instance dont une méthode l'appelle (l'objet `self`), observe l'*ordre de résolution des méthodes* obtenu avec `type(self).__mro__`, et renvoie la classe qui suit `Derived` dans cet ordre, donc `Base`.

### 2.6.7 Comment organiser un code pour permettre de changer la classe mère plus facilement ?

Vous pouvez définir un alias pour la classe mère et dériver depuis l'alias. Ensuite, tout ce que vous devez changer est la valeur attribuée à cet alias. Accessoirement, cette astuce est également utile pour déterminer dynamiquement (par exemple en fonction de la disponibilité de certaines ressources) la classe mère à utiliser. Exemple :

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

### 2.6.8 Comment créer des données statiques de classe et des méthodes statiques de classe ?

Les données statiques et les méthodes statiques (au sens C++ ou Java) sont prises 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` tel que `isInstance (c, C)` est vrai, sauf remplacement par `c` lui-même ou par une classe sur le chemin de recherche de classe mère de `c.__class__` jusqu'à `C`.

Attention : dans une méthode de C, une affectation comme `self.count = 42` crée une nouvelle instance sans rapport avec le nom `count` 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
```

Il est possible d'utiliser des méthodes statiques :

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        # 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 surcharger les constructeurs (ou méthodes) en Python ?

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

En C++, on écrirait

```
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 des 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 sans les éventuels tirets bas du début.

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used :

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()
```

(suite sur la page suivante)

(suite de la page précédente)

```
class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute ; many Python programmers never bother to use private variable names at all.

**Voir aussi**

The private name mangling specifications for details and special cases.

### 2.6.11 Ma classe définit `__del__` mais elle n'est pas appelée lorsque je supprime l'objet.

Il y a plusieurs explications possibles.

The `del` statement does not necessarily call `__del__()` -- it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

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 subobjects. 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 !).

Finally, if your `__del__()` method raises an exception, a warning message is printed to `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
```

### 2.6.14 Quand puis-je raisonnablement utiliser le test d'identité *is* ?

L'opérateur `is` détermine si deux objets sont identiques, c'est-à-dire le même objet. Le test `a is b` est équivalent à `id(a) == id(b)`.

La propriété la plus importante du test d'identité est qu'un objet est toujours identique à lui-même. Quelle que soit la valeur de *a*, `a is a` vaut toujours `True`. Un test d'identité est généralement plus rapide qu'un test d'égalité. De plus, contrairement à l'opérateur `==`, l'opérateur `is` renvoie toujours un booléen, `True` ou `False`.

Cependant, les tests d'identité ne peuvent remplacer les tests d'égalité que si l'identité est garantie. C'est le cas dans les trois situations suivantes :

1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.
2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.
3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

Dans la plupart des autres cas, un test d'identité n'est pas approprié par rapport à un test d'égalité. En particulier, il ne faut pas utiliser `is` pour comparer à des constantes comme les entiers (type `int`) ou des chaînes de caractères (type `str`) car ces valeurs ne sont pas nécessairement des singletons :

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

De même, deux instances fraîchement créées d'un type de conteneurs mutables ne sont jamais identiques :

```
>>> a = []
>>> b = []
>>> a is b
False
```

Voici quelques exemples de la bibliothèque standard qui illustrent comment utiliser correctement les tests d'identité dans certaines situations particulières :

1) As recommended by **PEP 8**, an identity test is the preferred way to check for `None`. This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.
2) Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like `dict.pop()` :

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3) Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

For example, here is the implementation of `collections.abc.Sequence.__contains__()` :

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

### 2.6.15 Comment définir dans une classe fille les attributs d'une instance immuable ?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

Toutes les classes d'objets immuables suivantes ont des signatures de constructeur différentes de leur classe mère :

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

Ces classes s'utilisent comme ceci :

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
```

(suite sur la page suivante)

(suite de la page précédente)

```
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

### 2.6.16 Comment mettre en cache le résultat d'une méthode ?

Il existe deux outils principaux pour mettre en cache la valeur de retour d'une méthode, à savoir `functools.cached_property()`, qui stocke les valeurs au niveau de l'instance, et `functools.lru_cache()`, qui le fait au niveau de la classe.

La fonction `cached_property` n'est applicable qu'aux méthodes sans argument. Elle n'induit aucune référence vers l'instance. Le cache est simplement conservé aussi longtemps que l'instance elle-même.

L'avantage est que le cache est supprimé aussitôt que l'instance est détruite. L'inconvénient est que les caches peuvent s'accumuler avec les instances, sans limite de nombre.

La fonction `lru_cache` s'applique quant à elle aux méthodes dont les arguments sont *hachables*. Elle crée une référence forte à l'instance, sauf à passer par des circonvolutions pour que la référence soit faible.

L'avantage de l'algorithme LRU est d'offrir une limitation sur la taille du cache, que l'on peut régler avec le paramètre *maxsize* (NdT : LRU signifie *Least Recently Used* en anglais, les entrées du cache les plus anciennes sont évincées pour conserver la taille). L'inconvénient est qu'une référence forte vers l'instance est conservée dans le cache, ce qui maintient l'instance hors de portée du ramasse-miettes jusqu'à ce que l'entrée soit effacée du cache ou que le cache soit remis à zéro.

Voici une démonstration des différentes techniques :

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

Cet exemple suppose que l'attribut *station_id* n'est jamais changé. Si les attributs utilisés par la méthode sont susceptibles d'être modifiés, `cached_property` ne peut pas fonctionner car elle ne peut pas détecter les mutations pour invalider le cache.

To make the *lru_cache* approach work when the *station_id* is mutable, the class needs to define the `__eq__()` and `__hash__()` methods so that the cache can detect relevant attribute updates :

```
class Weather:
    "Example with a mutable station identifier"

```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

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

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

## 2.7 Modules

### 2.7.1 Comment créer des fichiers `.pyc` ?

Quand un module est importé pour la première fois (ou si le fichier source a été modifié depuis la création du fichier compilé), un fichier `.pyc` contenant le code précompilé est créé dans un sous-dossier `__pycache__` du dossier contentant le fichier `.py`. Le nom du fichier `.pyc` est identique au fichier `.py` et se termine par `.pyc`, avec une partie centrale qui dépend du binaire `python` qui l'a créé (voir la **PEP 3147** pour de plus amples précisions).

Une des raisons pour lesquelles un fichier `.pyc` peut ne pas être créé est un problème de droits sur le dossier qui contient le fichier source, ce qui veut dire qu'il est impossible de créer le sous-dossier `__pycache__`. Ceci peut arriver, par exemple, si vous développez en tant qu'un certain utilisateur, mais que le code est exécuté en tant qu'un autre utilisateur, par exemple pour tester un serveur Web.

La création du fichier `.pyc` est automatique durant l'importation d'un module si Python est capable (en termes de droits, d'espace disque, etc) de créer un sous-dossier `__pycache__` et d'écrire le module ainsi compilé dans ce sous-répertoire, à moins que la variable d'environnement `PYTHONDONTWRITEBYTECODE` soit définie.

Exécuter du Python dans un script de plus haut niveau n'est pas considéré comme une importation et le fichier `.pyc` n'est pas créé. Par exemple, si un module de plus haut niveau `foo.py` importe un autre module `xyz.py`, alors à l'exécution de `foo` (en tapant `python foo.py` dans la console), un fichier `.pyc` est créé pour `xyz` mais pas pour `foo` car `foo.py` n'est pas importé.

Pour créer un fichier `.pyc` pour `foo` — c'est-à-dire créer un fichier `.pyc` pour un module qui n'est pas importé — il existe les modules `py_compile` et `compileall`.

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')
```

Ces lignes écrivent le `.pyc` dans un sous-dossier `__pycache__` à côté de `foo.py` (le paramètre optionnel `cfile` permet de changer ce comportement).

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*) ;
- La mécanique des importations tente de lire *foo_var* dans les variables globales de *foo* pour procéder à l'affectation `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 mères)
- 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 ?

Pour des raisons de performance et de cohérence, Python ne lit le fichier d'un module que la première fois où celui-ci est importé. Si ce n'était pas le cas, dans un programme composé d'un très grand nombre de modules qui importent tous le même module de base, ce module de base serait analysé et ré-analysé un très grand nombre de fois. Pour forcer la relecture d'un module, il faut faire :

```
import importlib
import modname
importlib.reload(modname)
```

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

```
from modname import some_objects
```

continuent de fonctionner avec l'ancienne version des objets importés. Si le module contient une définition de classe, les instances déjà existantes de celle-ci ne sont *pas* mises à jour avec la nouvelle définition de la classe. Ceci peut conduire au comportement paradoxal suivant :

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

La nature du problème apparaît clairement en affichant « l'identité » des objets de la classe :

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

CHAPITRE 3

# FAQ sur l'histoire et la conception

## 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 d'un programme Python ordinaire. La plupart des gens finissent par aimer cette particularité au bout d'un moment.

Comme il n'y a pas d'accolades de début et 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++` est exécutée si la condition est vraie, mais l'indentation conduit beaucoup de gens à penser le contraire. Mêmes des développeurs C expérimentés peuvent parfois rester pendant un moment à se demander pourquoi `y` est décrémenté même si `x > y`.

Comme il n'y a pas d'accolades de début et fin, Python est moins sujet aux conflits de style de code. En C, on peut placer les accolades de nombreuses façons. Après s'être habitué à lire et écrire selon un style particulier, il est normal de se sentir perturbé en lisant (ou en devant écrire) avec un autre style.

Nombre de styles de programmation utilisent des accolades de début et fin sur une ligne à part. Cela rend les programmes beaucoup plus longs et fait perdre une bonne partie de l'espace visible sur l'écran, empêchant un peu d'avoir une vue globale du programme. Idéalement, une fonction doit être visible sur un écran (environ 20 ou 30 lignes). 20 lignes de Python peuvent faire beaucoup plus que 20 lignes de C. Ce n'est pas seulement dû à l'absence d'accolades de début et fin — l'absence de déclarations et la présence de types de haut-niveau en sont également responsables — mais la syntaxe basée sur l'indentation aide sûrement.

## 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 n'en est pas un. Ceci n'a d'ailleurs que peu à voir avec Python, mais avec la manière dont la plateforme sous-jacente gère les nombres à virgule flottante.

La classe `float` dans CPython utilise le type double du langage C comme stockage. La valeur d'un objet `float` est stockée dans un format binaire à virgule flottante avec une précision fixe (généralement 53 bits). Python utilise des opérations qui proviennent du langage C qui à leur tour reposent sur l'implémentation au niveau du processeur afin d'effectuer des opérations en virgule flottante. Cela signifie que, dans le cadre des opérations sur les nombres à virgule flottante, Python se comporte comme beaucoup de langages populaires dont C et Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating point. For example, after :

```
>>> x = 1.2
```

la valeur stockée pour `x` est une (très bonne) approximation de la valeur décimale `1.2`, mais cette valeur n'est pas exacte. Sur une machine typique, la valeur stockée est en fait :

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

qui est, exactement :

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

La précision typique de 53 bits des *floats* Python permet une précision de 15–16 décimales.

For a fuller explanation, please see the floating-point arithmetic chapter in the Python tutorial.

## 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 *n*-uplets 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 ne changera la chaîne de caractères "huit" en autre chose.

## 3.5 Pourquoi *self* doit-il être explicitement utilisé dans les définitions et les appels de méthodes ?

L'idée a été empruntée à Modula-3. Cela s'avère 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 reconnaître 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 devriez 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.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator -- in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

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 nommage, et vous devez dire à Python quel espace de nommage utiliser.

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

Depuis Python 3.8, c'est possible !

Les expressions d'affectation qui utilisent l'opérateur morse `:=` affectent une variable dans une expression :

```
while chunk := fp.read(200):
   print(chunk)
```

Voir la **PEP 572** pour plus d'informations.

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

Comme l'a dit Guido :

> (a) Pour certaines opérations, la notation préfixe se lit mieux que celle suffixe — les opérations préfixe (et infixe !) sont une longue tradition en mathématique, où on apprécie les notations qui aident visuellement le mathématicien à réfléchir sur un problème. Comparez la facilité avec laquelle nous réécrivons une formule comme x*(a+b) en x*a + x*b à la lourdeur de faire la même chose avec une notation orientée objet brute.
>
> (b) Quand je lis du code qui dit `len(x)` *je sais* qu'il demande la longueur de quelque chose. Cela me dit deux choses : le résultat est un entier, et l'argument est une sorte de conteneur. Au contraire, quand je lis `x.len()`, je dois déjà savoir que x est une sorte de conteneur implémentant une interface ou héritant d'une classe qui a un `len()` standard. Voyez la confusion qui arrive parfois quand une classe qui n'implémente pas une interface de dictionnaire a une méthode `get()` ou `key()`, ou quand un objet qui n'est pas un fichier implémente une méthode `write()`.
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

## 3.8 Pourquoi `join()` est une méthode de chaîne plutôt qu'une méthode de liste ou de *n*-uplet ?

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 mal à l'aise 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 argument peut se résumer à : « c'est vraiment moche d'utiliser une méthode de chaîne littérale (c.-à-d. constante) ». La réponse est « certes, mais une chaîne littérale est une valeur comme une autre, juste fixe ». Si on utilise les méthodes sur des noms de variables, il n'y a pas de raison à les interdire sur des chaînes littérales.

Le second consiste à se demander : « suis-je réellement en train de dire à une séquence de joindre ses membres avec une constante de chaîne ? ». Malheureusement, ce n'est pas ça. Allez savoir, il semble être bien moins difficile de comprendre ce qui se passe avec `split()` en tant que méthode de chaîne, puisque dans ce cas il est facile de voir que

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

est une instruction à une chaîne littérale de renvoyer les sous-chaînes délimitées par le séparateur fourni (ou, par défaut, les espaces ou groupes d'espaces).

`join()` est une méthode de chaîne parce qu'en l'utilisant vous dites au séparateur de chaîne d'itérer une séquence de chaînes et de s'insérer entre les éléments adjacents. Cette méthode peut être utilisée avec n'importe quel argument qui obéit aux règles d'objets séquence, incluant n'importe quelles nouvelles classes que vous pourriez définir vous-même. Des méthodes similaires existent pour des objets `bytes` et `bytearray`.

## 3.9 À quel point les exceptions sont-elles rapides ?

A `try`/`except` block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom :

```
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)
```

Pour ce cas, vous pouvez également utiliser `value = dict.setdefault(key, getvalue(key))`, mais seulement si l'appel à `getvalue()` est suffisamment peu coûteux car il est évalué dans tous les cas.

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

In general, structured switch statements execute one block of code when an expression has a particular value or set of values. Since Python 3.10 one can easily match literal values, or constants within a namespace, with a `match ... case` statement. An older alternative is a sequence of `if... elif... elif... else`.

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 :

```
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 :

```
class MyVisitor:
    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.

Imitating switch with fallthrough, as with C's switch-case-default, is possible, much harder, and less needed.

## 3.11 Est-il possible d'émuler des fils d'exécution dans l'interpréteur plutôt que se baser sur les implémentations spécifiques aux systèmes d'exploitation ?

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 nécessiterait une gestion complète pour le C.

Réponse 2 : heureusement, il existe Stackless Python, qui a 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 que celui renvoyé par 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.

## 3.14 Comment Python gère la mémoire ?

Les détails de la gestion de la mémoire en Python dépendent de l'implémentation. En effet, l'implémentation standard de Python, *CPython*, utilise des compteurs de références afin de détecter des objets inaccessibles et un autre mécanisme pour collecter les références circulaires, exécutant périodiquement un algorithme de détection de cycles qui recherche les cycles inaccessibles et supprime les objets impliqués. Le module `gc` fournit des fonctions pour lancer le ramasse-miettes, d'obtenir des statistiques de débogage et ajuster ses paramètres.

Other implementations (such as Jython or PyPy), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Dans certaines implémentations de Python, le code suivant (qui fonctionne parfaitement avec *CPython*) aurait probablement manqué de descripteurs de fichiers :

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

Indeed, using CPython's reference counting and destructor scheme, each new assignment to `f` closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

Si vous souhaitez écrire du code qui fonctionne avec n'importe quelle implémentation de Python, vous devez explicitement fermer le fichier ou utiliser l'instruction `with` ; ceci fonctionne indépendamment du système de gestion de la mémoire :

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

## 3.15 Pourquoi CPython n'utilise-t-il pas un modèle de ramasse-miettes plus traditionnel ?

D'une part, ce n'est pas une caractéristique normalisée en C et, par conséquent, ce n'est pas portable. (Oui, nous connaissons la bibliothèque *GC Boehm*. Elle contient du code assembleur pour la plupart des plateformes classiques, mais pas toutes, et bien qu'elle soit le plus souvent transparente, elle ne l'est pas complètement ; des correctifs sont nécessaires afin que Python fonctionne correctement avec.)

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, CPython works with anything that implements `malloc()` and `free()` properly.

## 3.16 Pourquoi toute la mémoire n'est pas libérée lorsque *CPython* s'arrête ?

Les objets référencés depuis les espaces de nommage 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 allouées 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 `atexit` pour exécuter une fonction qui va forcer ces destructions.

## 3.17 Pourquoi les *n*-uplets et les *list* sont deux types de données séparés ?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal `records` or C `structs` ; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Les *n*-uplets sont immuables, ce qui signifie que lorsqu'un *n*-uplet a été créé, vous ne pouvez remplacer aucun de ses éléments par une nouvelle valeur. Les listes sont mutables, 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 *n*-uplets peuvent être utilisés comme clés.

## 3.18 Comment les listes sont-elles implémentées dans CPython ?

Les listes en CPython sont de vrais tableaux de longueur variable contrairement à des listes orientées *Lisp* (c.-à-d. des listes chaînées). L'implémentation utilise un tableau contigu de références à d'autres objets. Elle conserve également un pointeur vers ce tableau et la longueur du tableau dans une structure de tête de liste.

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.

## 3.19 Comment les dictionnaires sont-ils implémentés dans CPython ?

Les dictionnaires CPython sont implémentés sous forme de tables de hachage redimensionnables. Par rapport aux *B-trees*, cela donne de meilleures performances pour la recherche (l'opération la plus courante de loin) dans la plupart des circonstances, et leur implémentation est plus simple.

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 and a per-process seed ; for example, `'Python'` could hash to `-539294296` while `'python'`, a string that differs by a single bit, could hash 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 Big-O notation -- to retrieve a key.

## 3.20 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 mutable, 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 *n*-uplet ; la fonction `tuple(L)` crée un *n*-uplet avec les mêmes entrées que la liste `L`. Les *n*-uplets sont immuables et peuvent donc être utilisés comme clés du dictionnaire.

Certaines solutions insatisfaisantes ont été proposées :

— Hacher les listes 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]])
```

lèverait une exception `KeyError` car l'ID de `[1, 2]` utilisé dans la deuxième ligne diffère de celui 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 mutable, 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 *n*-uplet 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.

There is a trick to get around this if you need to, but use it at your own risk : You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other 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):
            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 soient impossibles 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.

In the case of `ListWrapper`, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

## 3.21 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 classer serait un gaspillage. Par conséquent, `list.sort()` classe la liste en place. Afin de vous le rappeler, elle ne renvoie pas la liste classée. De cette façon, vous ne serez pas dupés en écrasant accidentellement une liste lorsque vous avez besoin d'une copie triée tout en gardant sous la main la version non triée.

Si vous souhaitez qu'une nouvelle liste soit renvoyée, utilisez plutôt la fonction native `sorted()`. Cette fonction crée une nouvelle liste à partir d'un itérable fourni, la classe et la renvoie. Par exemple, voici comment itérer dans l'ordre sur les clés d'un dictionnaire :

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

## 3.22 Comment spécifier une interface et appliquer 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 facilite la construction de grands programmes.

Python 2.6 ajoute un module `abc` qui vous permet de définir des classes mères abstraites (ABC pour *abstract base classes* en anglais). Vous pouvez ensuite utiliser `isinstance()` et `issubclass()` pour vérifier si une instance ou une classe implémente une ABC particulière. Le module `collections.abc` définit un ensemble d'ABC utiles telles que `Iterable`, `Container` et `collections.abc.MutableMapping`.

Pour Python, la plupart des avantages des spécifications d'interface peuvent être obtenus par une discipline de test appropriée pour les composants.

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 cadriciels de test tiers peuvent être utilisés pour construire des suites de tests exhaustives qui éprouvent chaque ligne de code dans un module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `list.append()` method is expected to add new elements to the end of some internal list ; an interface specification cannot test that your `list.append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

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.

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

In the 1970s people realized that unrestricted goto could lead to messy "spaghetti" code that was hard to understand and revise. In a high-level language, it is also unneeded as long as there are ways to branch (in Python, with `if` statements and `or`, `and`, and `if`/`else` expressions) and loop (with `while` and `for` statements, possibly containing `continue` and `break`).

One can also use exceptions to provide a "structured goto" that works even across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the `go` or `goto` constructs of C, Fortran, and other languages. For example :

```
class label(Exception): pass  # declare a label

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

This doesn't allow you to jump into the middle of a loop, but that's usually considered an abuse of `goto` anyway. Use sparingly.

## 3.24 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 la 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-ci

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

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

Python has a `with` statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some languages have a construct that looks like this :

```
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 Pascal, Delphi et 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 sera référencé 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)
```

The snippet assumes that `a` must have a member attribute called `x`. However, there is nothing in Python that tells the interpreter this. What should happen if `a` is, let us say, an integer ? If there is a global variable named `x`, will it be used inside the `with` block ? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of `with` and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of :

```
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.

Similar proposals that would introduce syntax to further reduce code volume, such as using a 'leading dot', have been rejected in favour of explicitness (see https://mail.python.org/pipermail/python-ideas/2016-May/040070.html).

## 3.26 Pourquoi l'instruction `with` ne prend-elle pas en charge les générateurs ?

For technical reasons, a generator used directly as a context manager would not work correctly. When, as is most common, a generator is used as an iterator run to completion, no closing is needed. When it is, wrap it as `contextlib.closing(generator)` in the `with` statement.

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

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

```
if a == b
    print(a)
```

et cela

```
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 français (et 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.

## 3.28 Pourquoi Python permet-il les virgules à la fin des listes et des *n*-uplets ?

Python vous permet d'ajouter une virgule à la fin des listes, des *n*-uplets 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 *n*-uplet 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 extensions

## 4.1 Questions générales sur la bibliothèque

### 4.1.1 Comment trouver un module ou une application pour effectuer la tâche X ?

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

Pour des paquets tiers, regardez dans l'index des paquets Python ou essayez Google ou un autre moteur de recherche (NDT : comme le moteur français Qwant). Rechercher « Python » accompagné d'un ou deux mots-clés se rapportant à ce qui vous intéresse donne souvent de bons résultats.

### 4.1.2 Où sont stockés les fichiers sources *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) les modules écrits en Python (*.py*) ;
2) les 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.

Il peut arriver que l'environnement d'un utilisateur soit si chargé que le programme **/usr/bin/env** échoue ; ou que le programme *env* n'existe pas du tout. Dans ce cas, vous pouvez utiliser l'astuce suivante, élaborée par 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 *curses* ou *termcap* en Python ?

Pour les dérivés d'Unix : la distribution standard de Python contient un module *curses* dans le sous-dossier Modules, bien qu'il ne soit pas compilé par défaut. Il n'est pas disponible en Windows — le module *curses* n'existant pas en 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.

### 4.1.5 Existe-t-il un équivalent à la fonction C `onexit()` en Python ?

The `atexit` module provides a register function that is similar to C's `onexit()`.

### 4.1.6 Pourquoi mes gestionnaires de signaux ne fonctionnent-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 ?

Pour les dérivés d'Unix, plusieurs solutions s'offrent à vous. C'est facile en utilisant le module *curses*, mais *curses* est un module assez conséquent à apprendre.

## 4.3 Fils d'exécution

### 4.3.1 Comment programmer avec des fils d'exécution ?

Veillez à bien utiliser le module `threading` et non le module `_thread`. Le module `threading` fournit une abstraction plus facile à manipuler que les primitives de bas-niveau du module `_thread`.

### 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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
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)
```

Au lieu d'essayer de trouver une bonne valeur d'attente pour la fonction `time.sleep()`, il vaut mieux utiliser un mécanisme basé sur les sémaphores. Une solution consiste à utiliser le module `queue` pour créer un objet file, faire en sorte que chaque fil ajoute un jeton à la file quand il se termine, et que le fil principal retire autant de jetons de la file qu'il y a de fils.

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

La manière la plus simple est d'utiliser le module `concurrent.futures`, en particulier la classe `ThreadPoolExecutor`.

Ou bien, si vous désirez contrôler plus finement l'algorithme de distribution, vous pouvez écrire votre propre logique « à la main ». Utilisez le module `queue` pour créer une file de tâches ; la classe `Queue` gère une liste d'objets et a une méthode `.put(objet)` pour ajouter un élément à la file, et une méthode `.get()` pour les récupérer. La classe s'occupe de gérer les verrous pour que chaque tâche soit exécutée une et une seule fois.

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)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
```

(suite sur la page suivante)

(suite de la page précédente)

```
            print('Worker', threading.current_thread(), end=' ')
            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 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consultez la documentation du module pour plus de détails ; la classe `Queue` fournit une interface pleine de fonctionnalités.

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

Le *verrou global de l'interpréteur* (GIL pour *global interpreter lock*) est utilisé en interne pour s'assurer que la machine virtuelle Python (MVP) n'exécute qu'un seul fil à la fois. De manière générale, Python ne change de fil qu'entre les instructions du code intermédiaire ; `sys.setswitchinterval()` permet de contrôler la fréquence de bascule entre les fils. Chaque instruction du code intermédiaire, et, par conséquent, tout le code C appelé par cette instruction est donc atomique du point de vue d'un programme Python.

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)
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a 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.

With the approval of **PEP 703** work is now underway to remove the GIL from the CPython implementation of Python. Initially it will be implemented as an optional compiler flag when building the interpreter, and so separate builds will be available with and without the GIL. Long-term, the hope is to settle on a single build, once the performance implications of removing the GIL are fully understood. Python 3.13 is likely to be the first release containing this work, although it may not be completely functional in this release.

The current work to remove the GIL is based on a fork of Python 3.9 with the GIL removed by Sam Gross. Prior to that, 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. Adam Olsen did a similar experiment in his python-safethread project. Unfortunately, both of these earlier experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL. The Python 3.9 fork is the first attempt at removing the GIL with an acceptable performance impact.

The presence of the GIL in current Python releases 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*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so ; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

Des extensions C appropriées peuvent aussi aider ; en utilisant une extension C pour effectuer une tâche longue, l'extension peut relâcher le GIL pendant que le fil est en train d'exécuter ce code et laisser les autres fils travailler. Des modules de la bibliothèque standard comme `zlib` ou `hashlib` utilisent cette technique.

An alternative approach to reducing the impact of the GIL is to make the GIL a per-interpreter-state lock rather than truly global. This was first implemented in Python 3.12 and is available in the C API. A Python interface to it is expected in Python 3.13. The main limitation to it at the moment is likely to be 3rd party extension modules, since these must be written with multiple interpreters in mind in order to be usable, so many older extension modules will not be usable.

## 4.4 Les entrées/sorties

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

Utilisez `os.remove(filename)` ou `os.unlink(filename)` ; pour la documentation, référez-vous au module `os`. Ces deux fonctions sont identiques, `unlink()` n'est tout simplement que le nom de l'appel système à cette fonction sous Unix.

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.

Pour supprimer le contenu d'un fichier, ouvrez celui-ci avec `f = open(nom_du_fichier, "rb+")`, puis exécutez `f.truncate(décalage)` où *décalage* est par défaut la position actuelle de la tête de lecture. Il existe aussi `os.ftruncate(df, décalage)` pour les fichiers ouverts avec `os.open()`, où *df* est le descripteur de fichier (un entier court).

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 ?

The `shutil` module contains a `copyfile()` function. Note that on Windows NTFS volumes, it does not copy alternate data streams nor resource forks on macOS HFS+ volumes, though both are now rarely used. It also doesn't copy file permissions and metadata, though using `shutil.copy2()` instead will preserve most (though not all) of it.

### 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

with open(filename, "rb") as f:
    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`.

**Note**

Pour lire et écrire de la donnée binaire, il est obligatoire d'ouvrir le fichier en mode binaire également (ici, en passant `"rb"` à `open()`). En utilisant `"r"` (valeur par défaut), le fichier est ouvert en mode textuel et `f.read()` renvoie des objets `str` au lieu d'objets `bytes`.

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

Pour Win32, OSX, Linux, BSD, Jython et IronPython :

pyserial

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

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

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

Les *objets fichiers* en Python sont des abstractions de haut niveau sur les descripteurs de fichier C de bas niveau.

Pour la plupart des objets fichiers créés en Python avec la fonction native `open()`, `f.close()` marque le fichier comme fermé du point de vue de Python et ferme aussi le descripteur C sous-jacent. Le même mécanisme est enclenché automatiquement dans le destructeur de `f`, lorsque `f` est recyclé.

Mais *stdin*, *stdout* et *stderr* ont droit à un traitement spécial en Python, car leur statut en C est lui aussi spécial. Exécuter `sys.stdout.close()` marque l'objet fichier comme fermé du point de vue de Python, mais le descripteur de fichier C associé n'est *pas* fermé.

Pour fermer le descripteur de fichier sous-jacent C de l'une de ces trois sorties, demandez-vous avant tout si vous êtes sûr de vous (cela peut, par exemple, perturber le bon fonctionnement de modules qui font des opérations d'entrée-sortie). Si c'est le cas, utilisez `os.close()` :

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

Il est aussi possible de fermer respectivement les constantes numériques 0, 1 ou 2.

## 4.5 Programmation réseau et Internet

### 4.5.1 Quels sont les outils Python dédiés à au web ?

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 web côté client comme côté serveur.

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

### 4.5.2 Quel module utiliser pour générer du HTML ?

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

### 4.5.3 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 = input("From: ")
toaddrs  = 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)
```

### 4.5.4 Comment éviter de bloquer dans la méthode `connect()` d'un connecteur réseau ?

Le module `select` est fréquemment utilisé pour effectuer des entrées-sorties asynchrones sur des connecteurs réseaux.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

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.select()` to check if it's writable.

**Note**

The `asyncio` module provides a general purpose single-threaded and concurrent asynchronous library, which can be used for writing non-blocking network code. The third-party Twisted library is a popular and feature-rich alternative.

## 4.6 Bases de données

### 4.6.1 Existe-t-il des modules Python pour s'interfacer avec des bases de données ?

Oui.

La distribution standard de Python fournit aussi des interfaces à des bases de données comme `DBM` ou `GDBM`. Il existe aussi le module `sqlite3` qui implémente une base de données relationnelle légère sur disque.

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 ?

Le module `pickle` répond à cela de manière large (bien qu'il ne soit pas possible de stocker des fichiers ouverts, des connecteurs ou des fenêtres par exemple), et le module `shelve` de la bibliothèque utilise *pickle* et *(g)dbm* pour créer des liens persistants vers des objets Python.

## 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()
```

This returns a random floating-point number in the range [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)` chooses a floating-point number in the range [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.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make 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 ?

La fonction de plus haut niveau pour ce faire est `PyRun_SimpleStringString()` qui prend une chaîne pour seul argument afin de l'exécuter dans le contexte du module `__main__` et renvoie `0` en cas de succès et `-1` quand une exception se produit (incluant `SyntaxError`). Pour une meilleure maîtrise, utilisez `PyRun_String()` ; voir le code source pour `PyRun_SimpleString()` dans `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 ?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

Pour tester le type d'un objet, assurez-vous d'abord qu'il ne soit pas `NULL`, puis utilisez `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

Il y a aussi une API de haut niveau pour les objets Python qui est fournie par l'interface dite « abstraite » — voir `Include/abstract.h` pour plus de détails. Elle permet l'interfaçage avec tout type de séquence Python en utilisant des appels tels que `PySequence_Length()`, `PySequence_GetItem()`, etc. ainsi que de nombreux autres protocoles utiles tels que les nombres (`PyNumber_Index()` et autres) et les correspondances dans les APIs PyMapping.

## 5.7 Comment utiliser Py_BuildValue() pour créer un *n*-uplet de longueur définie ?

Vous ne pouvez pas. Utilisez `PyTuple_Pack()` à la place.

## 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, const char *method_name,
                    const 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 *n*-uplet 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.

La façon la plus simple consiste à utiliser la classe `io.StringIO` :

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

Le code d'un objet à la fonctionnalité similaire ressemblerait à ceci :

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(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 ?

Most packaged versions of Python omit some files required for compiling Python extensions.

For Red Hat, install the python3-devel RPM to get the necessary files.

For Debian, run `apt-get install python3-dev`.

## 5.15 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.

## 5.16 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`).

## 5.17 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.

The Boost Python Library (BPL, https://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

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 pourriez avoir besoin d'être un peu guidé.

À moins que vous n'utilisiez une sorte d'environnement de développement, vous finirez par *taper* des commandes Windows dans ce qui est appelé une « invite de commande Windows ». En général vous pouvez ouvrir un telle fenêtre depuis votre barre de recherche en cherchant `cmd`. Vous devriez être capable de reconnaître quand vous avez lancé une telle fenêtre parce que vous verrez une « invite de commande » Windows, qui en en général ressemble à ça :

```
C:\>
```

La lettre peut être différente, et il peut y avoir d'autres choses à la suite, alors il se peut que ça ressemble également à ça :

```
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 ?

Tout d'abord, vous devez vous assurer que votre fenêtre d'invite de commande reconnaît le mot "python" comme une instruction pour démarrer l'interpréteur. Si vous avez ouvert une invite de commande, entrez la commande `py`, puis appuyez sur entrée :

```
C:\Users\YourName> py
```

Vous devez voir quelque chose comme ça :

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]␣
↪on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Vous avez lancé l'interpréteur dans son "mode interactif". Cela signifie que vous pouvez entrer des instructions ou des expressions Python de manière interactive pour qu'elles soient exécutées. Il s'agit là d'une des plus puissantes fonctionnalités de Python. Vous pouvez le vérifier en entrant quelques commandes de votre choix et en regardant le résultat :

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Beaucoup de gens utilisent le mode interactif comme une calculatrice pratique mais néanmoins hautement programmable. Lorsque vous souhaitez mettre fin à votre session Python interactive, appelez la fonction `exit()` ou maintenez la touche `Ctrl` enfoncée pendant que vous entrez un `Z`, puis appuyez sur la touche "`Enter`" pour revenir à votre invite de commande Windows.

Peut-être avez-vous remarqué une nouvelle entrée dans votre menu Démarrer telle que *Démarrer ‣ Programmes ‣ Python 3.x ‣ Python (ligne de commande)* qui a pour résultat que vous voyez l'invite `>>>` dans une nouvelle fenêtre. Si oui, la fenêtre va disparaître quand vous appellerez la fonction `exit()` ou entrez le caractère `Ctrl-Z`; Windows exécute une commande "python" dans la fenêtre et ferme celle-ci lorsque vous fermez l'interpréteur.

Maintenant que nous savons que la commande `py` est reconnue, vous pouvez lui donner votre script Python. Vous devez donner le chemin absolu ou relatif du script Python. Disons que votre script Python est situé sur votre bureau et est nommé `hello.py`, et votre invite de commande est bien ouvert dans votre répertoire d'accueil, alors vous voyez quelque chose comme :

```
C:\Users\YourName>
```

Alors maintenant, vous demanderez à la commande `py` de donner votre script à Python en tapant `py` suivi de votre chemin de script :

```
C:\Users\YourName> py Desktop\hello.py
hello
```

## 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 ?

Voir *Comment créer un binaire autonome à partir d'un script Python ?* pour une liste d'outils qui créent des exécutables.

## 6.5 Est-ce qu'un fichier `*.pyd` est la même chose qu'une DLL ?

Oui, les fichiers *.pyd* sont des fichiers *dll*, mais il y a quelques différences. Si vous avez une *DLL* `foo.pyd`, celle-ci doit posséder une fonction `PyInit_foo()`. Vous pouvez alors écrire en Python « *import foo* » et Python recherchera le fichier *foo.pyd* (ainsi que *foo.py* et *foo.pyc*) ; s'il le trouve, il tentera d'appeler `PyInit_foo()` pour l'initialiser. Ne liez pas votre *.exe* avec *foo.lib* car dans ce cas Windows aura besoin de la DLL.

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. Ne compilez **pas** Python directement dans votre fichier *.exe*. Sous Windows, Python doit être une DLL pour pouvoir importer des modules qui sont eux-mêmes des DLL (ceci constitue une information de première importance non documentée). Au lieu de cela faites un lien vers `python`*NN*`.dll` qui est généralement placé dans `C:\Windows\System`. *NN* étant la version Python, par exemple « 33 » pour Python 3.3.

   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.
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 `struct 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*.

Python va lever `IndentationError` ou `TabError` si un mélange de tabulation et d'indentation pose problème en début de ligne. Vous pouvez aussi utiliser le module `tabnanny` pour détecter ces erreurs.

## 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 Comment résoudre l'erreur « `api-ms-win-crt-runtime-l1-1-0.dll` manquante » ?

Ce problème peut se produire avec Python 3.5 et les versions ultérieures lorsque vous utilisez Windows 8.1 ou une version antérieure sans que toutes les mises à jour aient été installées. Assurez-vous d'abord que votre système d'exploitation est pris en charge et qu'il est à jour, et si cela ne résout pas le problème, visitez la page d'assistance de Microsoft pour obtenir des conseils sur l'installation manuelle de la mise à jour du *C Runtime*.

CHAPITRE 7

# FAQ interface graphique

## 7.1 Questions générales sur l'interface graphique

## 7.2 Quelles boîtes à outils IUG existent pour Python ?

Les versions standards de Python incluent une interface orientée objet pour le jeu d'objets graphiques *Tcl/Tk*, appelée tkinter. C'est probablement la plus facile à installer (puisqu'elle est incluse avec la plupart des distributions binaires de Python) et à utiliser. Pour plus d'informations sur *Tk*, y compris les liens vers les sources, voir la page d'accueil Tcl/Tk. *Tcl/Tk* est entièrement portable sur les plates-formes macOS, Windows et Unix.

D'autres outils existent. Le choix doit dépendre notamment de la ou des plateformes que vous visez. Sur le Wiki Python se trouvent des listes de bibliothèques graphiques multiplateformes et pour une seule plate-forme.

## 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.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

Various third-party freeze libraries such as py2exe and cx_Freeze have handling for Tkinter applications built-in.

### 7.3.2 Puis-je modifier des événements *Tk* pendant l'écoute des *E/S* ?

On platforms other than Windows, yes, and you don't even need threads ! But you'll have to restructure your I/O code a bit. Tk has the equivalent of Xt's `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See tkinter-file-handlers.

### 7.3.3 Je n'arrive pas à faire fonctionner les raccourcis clavier dans *Tkinter* : pourquoi ?

An often-heard complaint is that event handlers bound to events with the `bind()` method don't get handled even when the appropriate key is pressed.

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

# FAQ ”Pourquoi Python est installé sur mon ordinateur ?”

## 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.
— Une application tierce installée sur votre machine écrite en Python installera Python. Il existe de nombreuses applications de ce type, allant de programme avec interface graphique, jusqu’aux scripts d’administration, en passant par les serveurs.
— 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.
— Python est installé par défaut sur de nombreux systèmes compatibles Unix, comme macOS et certaines distributions Linux.

## 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

**>>>**
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

**...**
: Peut faire référence à :
— The default Python prompt of the *interactive* shell when entering the 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.
— La constante `Ellipsis`.

**classe mère abstraite**
: Les classes mères 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 subtilement fausses (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.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

**annotation**
: Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Les annotations de variables locales ne sont pas accessibles au moment de l'exécution, mais les annotations de variables globales, d'attributs de classe et de fonctions sont stockées dans l'attribut spécial `__annotations__` des modules, classes et fonctions, respectivement.

Voir *annotation de variable*, *annotation de fonction*, les **PEP 484** et **PEP 526**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

**argument**
: 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 *paramètre* dans le glossaire, la question *Différence entre argument et paramètre* de la FAQ et la **PEP 362**.

**gestionnaire de contexte asynchrone**
(*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `async with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été Introduit par la **PEP 492**.

**générateur asynchrone**
Fonction qui renvoie un *itérateur de générateur asynchrone*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi uns série de valeurs utilisables dans une boucle `async for`.

Générateur asynchrone fait généralement référence à une fonction, mais peut faire référence à un *itérateur de générateur asynchrone* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser l'ensemble des termes lève l'ambiguïté.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

**itérateur de générateur asynchrone**
Objet créé par un *générateur asynchrone*.

C'est un *asynchronous iterator* qui, lorsqu'il est appelé via la méthode `__anext__()` renvoie un objet *awaitable* qui exécute le corps de la fonction du générateur asynchrone jusqu'au prochain `yield`.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

**itérable asynchrone**
Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la **PEP 492**.

**itérateur asynchrone**
Objet qui implémente les méthodes `__aiter__()` et `__anext__()`. `__anext__()` doit renvoyer un objet *awaitable*. Tant que la méthode `__anext__()` produit des objets *awaitable*, le `async for` appelant les consomme. L'itérateur asynchrone lève une exception `StopAsyncIteration` pour signifier la fin de l'itération. A été introduit par la **PEP 492**.

**attribut**
Valeur associée à un objet et habituellement désignée par son nom *via* une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, cet attribut est référencé par *o.a*.

Il est possible de donner à un objet un attribut dont le nom n'est pas un identifiant tel que défini pour les identifiers, par exemple en utilisant `setattr()`, si l'objet le permet. Un tel attribut ne sera pas accessible à l'aide d'une expression pointée et on devra y accéder avec `getattr()`.

**attendable (*awaitable*)**
Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la **PEP 492**.

**BDFL**
Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

**fichier binaire**
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and

`gzip.GzipFile`.

Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

**référence empruntée**
: In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Il est recommandé d'appeler `Py_INCREF()` sur la *référence empruntée*, ce qui la transforme *in situ* en une *référence forte*. Vous pouvez faire une exception si vous êtes certain que l'objet ne peut pas être supprimé avant la dernière utilisation de la référence empruntée. Voir aussi la fonction `Py_NewRef()`, qui crée une nouvelle *référence forte*.

**objet octet-compatible**
: Un objet gérant le protocole tampon et pouvant exporter un tampon (*buffer* en anglais) C-*contigu*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets octets-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, `bytearray` ou une `memoryview` d'un `bytearray` en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (« *objets octets-compatibles en lecture seule* »), par exemple des `bytes` ou des `memoryview` d'un objet `bytes`.

**code intermédiaire (*bytecode*)**
: Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

La documentation du module dis fournit une liste des instructions du code intermédiaire.

**appelable (*callable*)**
: Un appelable est un objet qui peut être appelé, éventuellement avec un ensemble d'arguments (voir *argument*), avec la syntaxe suivante :

```
callable(argument1, argument2, argumentN)
```

Une *fonction*, et par extension une *méthode*, est un appelable. Une instance d'une classe qui implémente la méthode `__call__()` est également un appelable.

**fonction de rappel (*callback*)**
: Une fonction (classique, par opposition à une coroutine) passée en argument pour être exécutée plus tard.

**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.

**variable de classe**
: Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

**closure variable**
: A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable* :

```
def outer():
    x = 0
    def inner():
```

(suite sur la page suivante)

(suite de la page précédente)

```
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

**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.

**context**
: This term has different meanings depending on where and how it is used. Some common meanings :
— The temporary state or environment established by a *context manager* via a `with` statement.
— The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
— A `contextvars.Context` object. Also see *current context*.

**context management protocol**
: The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

**gestionnaire de contexte**
: An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

**variable de contexte**
: A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

**contigu**
: Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

**coroutine**
: Les coroutines sont une forme généralisée des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles peuvent être implémentées en utilisant l'instruction `async def`. Voir aussi la **PEP 492**.

**fonction coroutine**
: Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction `async def` et peut contenir les mots clés `await`, `async for` ainsi que `async with`. A été introduit par la **PEP 492**.

**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*.

**current context**
: The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

**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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

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 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 ou le guide pour l'utilisation des descripteurs.

**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.

**dictionnaire en compréhension (ou dictionnaire en intension)**
: Écriture concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats. `results = {n: n ** 2 for n in range(10)}` génère un dictionnaire contenant des clés `n` liées à leurs valeurs `n ** 2`. Voir compréhensions.

**vue de dictionnaire**
: Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir dict-views.

**chaîne de documentation (*docstring*)**
: A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

**typage canard (*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 mère 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**
: Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En

d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `while`. Les affectations sont également des instructions et non des 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.

**f-string**
Chaîne littérale préfixée de `'f'` ou `'F'`. Les "f-strings" sont un raccourci pour formatted string literals. Voir la **PEP 498**.

**objet fichier**
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

**objet fichier-compatible**
Synonyme de *objet fichier*.

**encodage du système de fichiers et gestionnaire d'erreurs associé**
Encodage et gestionnaire d'erreurs utilisés par Python pour décoder les octets fournis par le système d'exploitation et encoder les chaînes de caractères Unicode afin de les passer au système.

L'encodage du système de fichiers doit impérativement pouvoir décoder tous les octets jusqu'à 128. Si ce n'est pas le cas, certaines fonctions de l'API lèvent `UnicodeError`.

Cet encodage et son gestionnaire d'erreur peuvent être obtenus à l'aide des fonctions `sys.getfilesystemencoding()` et `sys.getfilesystemencodeerrors()`.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction `PyConfig_Read()` : regardez `filesystem_encoding` et `filesystem_errors` dans les membres de `PyConfig`.

Voir aussi *encodage régional*.

**chercheur**
Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

There are two types of finder : *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

**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**.

**free threading**
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

**free variable**
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

**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.

**annotation de fonction**
*annotation* d'un paramètre de fonction ou valeur de retour.

Les annotations de fonctions sont généralement utilisées pour des *indications de types* : par exemple, cette fonction devrait prendre deux arguments `int` et devrait également avoir une valeur de retour de type `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section function.

Voir *annotation de variable* et la **PEP 484**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

**__future__**
Une importation depuis le futur s'écrit `from __future__ import <fonctionnalité>`. Lorsqu'une importation du futur est active dans un module, Python compile ce module avec une certaine modification de la syntaxe ou du comportement qui est vouée à devenir standard dans une version ultérieure. Le module `__future__` documente les possibilités pour *fonctionnalité*. L'importation a aussi l'effet normal d'importer une variable du module. Cette variable contient des informations utiles sur la fonctionnalité en question, notamment la version de Python dans laquelle elle a été ajoutée, et celle dans laquelle elle deviendra standard :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**ramasse-miettes**
(*garbage collection* en anglais) 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. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

**générateur**
Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle *for* ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction génératrice mais peut faire référence à un *itérateur de générateur* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

**itérateur de générateur**
Objet créé par une fonction *générateur*.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**expression génératrice**
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function :

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

**fonction générique**
Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools.singledispatch()` et la **PEP 443**.

**type générique**
Un *type* qui peut être paramétré ; généralement un conteneur comme `list` ou `dict`. Utilisé pour les *indications de type* et les *annotations*.

Pour plus de détails, voir types alias génériques et le module `typing`. On trouvera l'historique de cette fonctionnalité dans les **PEP 483**, **PEP 484** et **PEP 585**.

**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.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

***pyc* utilisant le hachage**
Un fichier de cache de code intermédiaire (*bytecode* en anglais) qui utilise le hachage plutôt que l'heure de dernière modification du fichier source correspondant pour déterminer sa validité. Voir pyc-invalidation.

**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__()` 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*.

La plupart des types immuables natifs de Python sont hachables, mais les conteneurs mutables (comme les listes ou les dictionnaires) ne le sont pas ; les conteneurs immuables (comme les n-uplets ou les ensembles figés) ne sont hachables que si leurs éléments sont hachables. Les instances de classes définies par les utilisateurs sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

**IDLE**
Environnement d'apprentissage et de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

**immortal**
*Immortal objects* are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

**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.

**chemin des importations**
Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path` ; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.

**importation**
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 has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

**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*.

**arrêt de l'interpréteur**
Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

**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`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

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 (or passing it to the built-in function `next()`) 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.

**Particularité de l'implémentation CPython :** CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

**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 clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Par ailleurs `attrgetter()`, `itemgetter()` et `methodcaller()` permettent de créer des fonctions clés. Voir le guide pour le tri pour des exemples de création et d'utilisation de fonctions clefs.

**argument nommé**
Voir *argument*.

**lambda**
Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

**LBYL**
: Regarde avant de sauter, (*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 "sauter". 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.

**lexical analyzer**
: Formal name for the *tokenizer*; see *token*.

**liste**
: 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)**
: Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (`0x…`). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

**chargeur**
: An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also :
  — finders-and-loaders
  — `importlib.abc.Loader`
  — **PEP 302**

**encodage régional**
: Sous Unix, il est défini par la variable régionale LC_CTYPE. Il peut être modifié par `locale.setlocale(locale.LC_CTYPE, new_locale)`.

  Sous Windows, c'est un encodage ANSI (par ex. : `"cp1252"`).

  Sous Android et VxWorks, Python utilise `"utf-8"` comme encodage régional.

  `locale.getencoding()` can be used to get the locale encoding.

  Voir aussi l'*encodage du systèmes de fichiers et gestionnaire d'erreurs associé*.

**méthode magique**
: Un synonyme informel de *special method*.

**tableau de correspondances (*mapping* en anglais)**
: Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les classes mères abstraites des `tableaux de correspondances` (immuables) ou `tableaux de correspondances mutables` (voir les classes mères abstraites). Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

**chercheur dans les méta-chemins**
: Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

  Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

**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'auront 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 sûrs 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**
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

**module**
Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

**spécificateur de module**
Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

See also module-specs.

**MRO**
Voir *ordre de résolution des méthodes*.

**mutable**
Un objet mutable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

***n*-uplet nommé**
Le terme "n-uplet nommé" s'applique à tous les types ou classes qui héritent de la classe `tuple` et dont les éléments indexables sont aussi accessibles en utilisant des attributs nommés. Les types et classes peuvent avoir aussi d'autres caractéristiques.

Plusieurs types natifs sont appelés n-uplets, y compris les valeurs retournées par `time.localtime()` et `os.stat()`. Un autre exemple est `sys.float_info` :

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

**espace de nommage**
L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

**paquet-espace de nommage**
A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

Voir aussi *module*.

**portée imbriquée**
Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonc-

tion définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef `nonlocal` permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

**nouvelle classe**
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

**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*.

**optimized scope**
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note : most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

**paquet**
*module* Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

Voir aussi *paquet classique* et *namespace package*.

**paramètre**
Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

— *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* : définit un argument qui ne peut être fourni que par position. Les paramètres *positional-only* peuvent être définis en insérant un caractère "/" dans la liste de paramètres de la définition de fonction après eux. Par exemple : *posonly1* et *posonly2* dans le code suivant :

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

— *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (`*`) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

— *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.

Voir aussi *argument* dans le glossaire, la question sur *la différence entre les arguments et les paramètres* dans la FAQ, la classe `inspect.Parameter`, la section function et la **PEP 362**.

**entrée de chemin**
Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

**chercheur de chemins**
: *chercheur* renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

  Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

**point d'entrée pour la recherche dans *path***
: A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

**chercheur basé sur les chemins**
: L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

**objet simili-chemin**
: Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la **PEP 519**.

**PEP**
: *Python Enhancement Proposal* (Proposition d'amélioration de Python). Une PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

  Les PEP sont censées être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

  Voir la **PEP 1**.

**portion**
: Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la **PEP 420**.

**argument positionnel**
: Voir *argument*.

**API provisoire**
: Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

  Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

  Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la **PEP 411** pour plus de détails.

**paquet provisoire**
: Voir *provisional API*.

**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)
```

**nom qualifié**
: Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la **PEP 3155**. Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name* - *FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

**nombre de références**
: The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

**paquet classique**
: *paquet* traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

  Voir aussi *paquet-espace de nommage*.

**REPL**
: An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

**__slots__**
: Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

**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 `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

  The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and

`__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

**ensemble en compréhension (ou ensemble en intension)**
Une façon compacte de traiter tout ou partie des éléments d'un itérable et de renvoyer un *set* avec les résultats. `results = {c for c in 'abracadabra' if c not in 'abc'}` génère l'ensemble contenant les lettres « r » et « d » `{'r', 'd'}`. Voir comprehensions.

**distribution simple**
Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

**tranche**
(*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets `slice` en interne.

**soft deprecated**
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387 : Soft Deprecation.

**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`.

**static type checker**
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

**référence forte**
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

Une référence forte est créée à l'aide de la fonction `Py_NewRef()`. Il faut normalement appeler `Py_DECREF()` dessus avant de sortir de sa portée lexicale, sans quoi il y a une fuite de référence.

Voir aussi *référence empruntée*.

**encodages de texte**
Une chaîne de caractères en Python est une suite de points de code Unicode (dans l'intervalle `U+0000`--`U+10FFFF`). Pour stocker ou transmettre une chaîne, il est nécessaire de la sérialiser en suite d'octets.

Sérialiser une chaîne de caractères en une suite d'octets s'appelle « encoder » et recréer la chaîne à partir de la suite d'octets s'appelle « décoder ».

Il existe de multiples codecs pour la sérialisation de texte, que l'on regroupe sous l'expression « encodages de texte ».

**fichier texte**
*Objet fichier* capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*encodage de texte* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte (`'r'` ou `'w'`), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`.

Voir aussi *fichier binaire* pour un objet fichier capable de lire et d'écrire des *objets octets-compatibles*.

**token**
A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python's lexical analyzer. The `token` module contains information on the various types of tokens.

**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**
The type of a Python object determines what kind of object it is ; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

**alias de type**
Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci :

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

**indication de type**
L'*annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Les indications de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultées en utilisant `typing.get_type_hints()`.

Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

**retours à la ligne universels**
Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

**annotation de variable**
*annotation* d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

La syntaxe d'annotation de variable est expliquée dans la section annassign.

Reportez-vous à *annotation de fonction*, à la **PEP 484** et à la **PEP 526** qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

**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.

Voir aussi `venv`.

**machine virtuelle**
: Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *code intermédiaire* 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

# About this documentation

Python's documentation is generated from reStructuredText sources using Sphinx, a documentation generator originally created for Python and now maintained as an independent project.

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., the creator of the original Python documentation toolset and author of much of the content ;
— 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 Contributors to the Python documentation

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 was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

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, which became Zope Corporation. 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 was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible ; the table below summarizes the various releases.

| Version | Dérivé de | Année | Propriétaire | GPL-compatible ? (1) |
|---|---|---|---|---|
| 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 | yes (2) |
| 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 ultérieure | 2.1.1 | 2001-maintenant | PSF | oui |

**Note**

(1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL ; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

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

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

Certains logiciels faisant partie de Python sont soumis à d'autres licences. Ces licences sont incluses avec le code lié à celles-ci. Voir *Licences et remerciements pour les logiciels tiers* pour une liste non exhaustive de ces licences.

### C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

4. PSF is making Python 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 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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, 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
   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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   ("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
   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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 tiers

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

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code :

```
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).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(suite sur la page suivante)

(suite de la page précédente)

```
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*)

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://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.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Interfaces de connexion asynchrones

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice :

```
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.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

### C.3.5 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.3.6 Les fonctions UUencode et UUdecode

The `uu` codec contains the following notice :

```
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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

### C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
    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.3.8 test_epoll

The `test.test_epoll` module contains the following notice :

```
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.3.9 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.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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.
</MIT License>

Original location:
   https://github.com/majek/csiphash/
```

(suite sur la page suivante)

(suite de la page précédente)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

### C.3.11 *strtod* et *dtoa*

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice :

```
/****************************************************************
 *
 * 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
 * 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.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies :

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(suite sur la page suivante)

(suite de la page précédente)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

### C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--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.3.15 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) 1995-2011 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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
```

### C.3.16 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * 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.

  * Neither the name of the author nor the names of its
    contributors may 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.
```

### C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec` :

```
Copyright (c) 2008-2020 Stefan Krah. 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.3.18 Ensemble de tests C14N du W3C

Les tests de C14N version 2.0 du module `test` (`Lib/test/xmltestdata/c14n-20/`) proviennent du site du W3C à l'adresse https://www.w3.org/TR/xml-c14n2-testcases/ et sont distribués sous licence BSD modifiée :

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work 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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.3.19 mimalloc

MIT License :

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 *asyncio*

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license :

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License :

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 "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 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.
```

# ANNEXE D

## Copyright

Python et cette documentation sont :

Copyright © 2001-2024 Python Software Foundation. Tous droits réservés.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

# Index

## Non alphabétique


... , 75
>>>, 75
__future__, 81
__slots__, 88


## A


alias de type, 90
annotation, 75
annotation de fonction, 80
annotation de variable, 90
API provisoire, 87
appelable (*callable*), 77
argument, 75
    difference from parameter, 12
argument nommé, 83
argument positionnel, 87
arrêt de l'interpréteur, 83
attendable (*awaitable*), 76
attribut, 76


## B


BDFL, 76


## C


C-contiguous, 78
chaîne de documentation (*docstring*), 79
chaîne entre triple guillemets, 90
chargeur, 84
chemin des importations, 82
chercheur, 80
chercheur basé sur les chemins, 87
chercheur dans les méta-chemins, 84
chercheur de chemins, 87
classe, 77
classe mère abstraite, 75
closure variable, 77
code intermédiaire (*bytecode*), 77
context, 78
context management protocol, 78
contigu, 78
coroutine, 78
CPython, 78
current context, 78


## D


décorateur, 78
descripteur, 79
dictionnaire, 79
dictionnaire en compréhension (*ou dictionnaire en intension*), 79
distribution simple, 89
division entière, 80


## E


EAFP, 79
encodage du système de fichiers et gestionnaire d'erreurs associé, 80
encodage régional, 84
encodages de texte, 89
ensemble en compréhension (*ou ensemble en intension*), 89
entrée de chemin, 86
environnement virtuel, 90
espace de nommage, 85
expression, 79
expression génératrice, 81


## F


f-string, 80
fichier binaire, 76
fichier texte, 89
fonction, 80
fonction clé, 83
fonction coroutine, 78
fonction de rappel (*callback*), 77
fonction générique, 81
Fortran contiguous, 78
free threading, 80
free variable, 80


## G


générateur, 81
générateur asynchrone, 76
gestionnaire de contexte, 78
gestionnaire de contexte asynchrone, 76

GIL, **81**

## H

hachable, **82**

## I

IDLE, **82**
immortal, **82**
immuable, **82**
importateur, **82**
importation, **82**
indication de type, **90**
instruction, **89**
interactif, **82**
interprété, **82**
itérable, **83**
itérable asynchrone, **76**
itérateur, **83**
itérateur asynchrone, **76**
itérateur de générateur, **81**
itérateur de générateur asynchrone, **76**

## L

lambda, **83**
LBYL, **84**
Le zen de Python, **91**
lexical analyzer, **84**
liste, **84**
liste en compréhension (*ou liste en intension*), **84**

## M

machine virtuelle, **91**
magic
    méthode, 84
métaclasse, **84**
méthode, **85**
    magic, 84
    special, 89
méthode magique, **84**
méthode spéciale, **89**
module, **85**
module d'extension, **80**
MRO, **85**
mutable, **85**

## N

n-uplet nommé, **85**
nom qualifié, **88**
nombre complexe, **78**
nombre de références, **88**
nouvelle classe, **86**

## O

objet, **86**
objet fichier, **80**
objet fichier-compatible, **80**
objet octet-compatible, **77**
objet simili-chemin, **87**
optimized scope, **86**
ordre de résolution des méthodes, **85**

## P

paquet, **86**
paquet classique, **88**
paquet provisoire, **87**
paquet-espace de nommage, **85**
parameter
    difference from argument, 12
paramètre, **86**
PATH, 52
PEP, **87**
point d'entrée pour la recherche dans path, **87**
portée imbriquée, **85**
portion, **87**
pyc utilisant le hachage, **82**
Python 3000, **87**
Python Enhancement Proposals
    PEP 1, 87
    PEP 8, 8, 33, 70
    PEP 278, 90
    PEP 302, 84
    PEP 328, 80
    PEP 343, 78
    PEP 362, 76, 86
    PEP 373, 4
    PEP 387, 3
    PEP 411, 87
    PEP 420, 85, 87
    PEP 443, 81
    PEP 483, 81
    PEP 484, 75, 81, 90
    PEP 492, 76, 78
    PEP 498, 80
    PEP 519, 87
    PEP 525, 76
    PEP 526, 75, 90
    PEP 572, 41
    PEP 585, 81
    PEP 602, 4
    PEP 683, 82
    PEP 703, 56, 80, 82
    PEP 3116, 90
    PEP 3147, 36
    PEP 3155, 88
PYTHON_GIL, 82
PYTHONDONTWRITEBYTECODE, 36
Pythonique, **87**

## R

ramasse-miettes, **81**
référence empruntée, **77**
référence forte, **89**
REPL, **88**
retours à la ligne universels, **90**

## S

séquence, **88**
soft deprecated, **89**
special
    méthode, 89
spécificateur de module, **85**
static type checker, **89**

## T

tableau de correspondances (*mapping en anglais*), **84**
token, **89**
tranche, **89**
typage canard (*duck-typing*), **79**
type, **90**
type générique, **81**

## V

variable de classe, **77**
variable de contexte, **78**
variable d'environnement
    PATH, 52
    PYTHON_GIL, 82
    PYTHONDONTWRITEBYTECODE, 36
verrou global de l'interpréteur, **82**
vue de dictionnaire, **79**