Python Tutorial

Version 3.11.3

Guido van Rossum and the Python development team

mai 08, 2023

Python Software Foundation
Email : docs@python.org

Table des matières

Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et permet une approche simple mais efficace de la programmation orientée objet. Parce que sa syntaxe est élégante, que son typage est dynamique et qu'il est interprété, Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur la plupart des plateformes.

L'interpréteur Python et sa vaste bibliothèque standard sont disponibles librement, sous forme de sources ou de binaires, pour toutes les plateformes majeures depuis le site Internet https://www.python.org/ et peuvent être librement redistribués. Ce même site distribue et pointe vers des modules, des programmes et des outils tiers. Enfin, il constitue une source de documentation.

L'interpréteur Python peut être facilement étendu par de nouvelles fonctions et types de données implémentés en C ou C++ (ou tout autre langage appelable depuis le C). Python est également adapté comme langage d'extension pour personnaliser des applications.

Dans ce tutoriel, nous introduisons, de façon informelle, les concepts de base ainsi que les fonctionnalités du langage Python et de son écosystème. Il est utile de disposer d'un interpréteur Python à portée de main pour mettre en pratique les notions abordées. Si ce n'est pas possible, pas de souci, les exemples sont inclus et le tutoriel est adapté à une lecture "hors ligne".

Pour une description des objets et modules de la bibliothèque standard, reportez-vous à library-index. reference-index présente le langage de manière plus formelle. Pour écrire des extensions en C ou en C++, lisez extending-index et c-api-index. Des livres sont également disponibles qui couvrent Python dans le détail.

L'ambition de ce tutoriel n'est pas d'être exhaustif et de couvrir chaque fonctionnalité, ni même toutes les fonctionnalités les plus utilisées. Il vise, en revanche, à introduire les fonctionnalités les plus notables et à vous donner une bonne idée de la saveur et du style du langage. Après l'avoir lu, vous serez capable de lire et d'écrire des modules et des programmes Python et vous serez prêt à en apprendre davantage sur les modules de la bibliothèque Python décrits dans library-index.

Pensez aussi à consulter le *Glossaire*.

CHAPITRE 1

Mise en bouche

Lorsqu'on travaille beaucoup sur ordinateur, on finit souvent par vouloir automatiser certaines tâches : par exemple, effectuer une recherche et un remplacement sur un grand nombre de fichiers de texte ; ou renommer et réorganiser des photos d'une manière un peu compliquée. Pour vous, ce peut être créer une petite base de données, une application graphique ou un simple jeu.

Quand on est un développeur professionnel, le besoin peut se faire sentir de travailler avec des bibliothèques C/C++/Java, mais on trouve que le cycle habituel écriture/compilation/test/compilation est trop lourd. Vous écrivez peut-être une suite de tests pour une telle bibliothèque et vous trouvez que l'écriture du code de test est pénible. Ou bien vous avez écrit un logiciel qui a besoin d'être extensible grâce à un langage de script, mais vous ne voulez pas concevoir ni implémenter un nouveau langage pour votre application.

Python est le langage parfait pour vous.

Vous pouvez écrire un script shell Unix ou des fichiers batch Windows pour certaines de ces tâches. Les scripts shell sont appropriés pour déplacer des fichiers et modifier des données textuelles, mais pas pour une application ayant une interface graphique ni pour des jeux. Vous pouvez écrire un programme en C/C++/Java, mais cela peut prendre beaucoup de temps, ne serait-ce que pour avoir une première maquette. Python est plus facile à utiliser et il vous aidera à terminer plus rapidement votre travail, que ce soit sous Windows, macOS ou Unix.

Python reste facile à utiliser, mais c'est un vrai langage de programmation : il offre une bien meilleure structure et prise en charge des grands programmes que les scripts shell ou les fichiers batch. Par ailleurs, Python permet de beaucoup mieux vérifier les erreurs que le langage C et, en tant que *langage de très haut niveau*, il possède nativement des types de données très évolués tels que les tableaux de taille variable ou les dictionnaires. Grâce à ses types de données plus universels, Python est utilisable pour des domaines beaucoup plus variés que ne peuvent l'être *Awk* ou même Perl. Pourtant, vous pouvez faire de nombreuses choses au moins aussi facilement en Python que dans ces langages.

Python vous permet de découper votre programme en modules qui peuvent être réutilisés dans d'autres programmes Python. Il est fourni avec une grande variété de modules standards que vous pouvez utiliser comme base de vos programmes, ou comme exemples pour apprendre à programmer. Certains de ces modules donnent accès aux entrées/sorties, aux appels système, aux connecteurs réseaux et même aux outils comme Tk pour créer des interfaces graphiques.

Python est un langage interprété, ce qui peut vous faire gagner un temps considérable pendant le développement de vos programmes car aucune compilation ni édition de liens n'est nécessaire. L'interpréteur peut être utilisé de manière interactive, pour vous permettre d'expérimenter les fonctionnalités du langage, d'écrire des programmes jetables ou de tester des fonctions lors d'un développement incrémental. C'est aussi une calculatrice de bureau bien pratique.

Python permet d'écrire des programmes compacts et lisibles. Les programmes écrits en Python sont généralement beaucoup plus courts que leurs équivalents en C, C++ ou Java. Et ceci pour plusieurs raisons :

— les types de données de haut niveau vous permettent d'exprimer des opérations complexes en une seule instruction ;
— les instructions sont regroupées entre elles grâce à l'indentation, plutôt que par l'utilisation d'accolades ;
— aucune déclaration de variable ou d'argument n'est nécessaire.

Python est *extensible* : si vous savez écrire un programme en C, une nouvelle fonction ou module peut être facilement ajouté à l'interpréteur afin de l'étendre, que ce soit pour effectuer des opérations critiques à vitesse maximale ou pour lier des programmes en Python à des bibliothèques disponibles uniquement sous forme binaire (par exemple des bibliothèques graphiques dédiées à un matériel). Une fois que vous êtes à l'aise avec ces principes, vous pouvez relier l'interpréteur Python à une application écrite en C et l'utiliser comme un langage d'extensions ou de commandes pour cette application.

Au fait, le nom du langage provient de l'émission de la BBC « Monty Python's Flying Circus » et n'a rien à voir avec les reptiles. Faire référence aux sketchs des Monty Python dans la documentation n'est pas seulement permis, c'est encouragé !

Maintenant que vos papilles ont été chatouillées, nous allons pouvoir rentrer dans le vif du sujet Python. Et comme la meilleure façon d'apprendre un langage est de l'utiliser, ce tutoriel vous invite à jouer avec l'interpréteur au fur et à mesure de votre lecture.

Dans le prochain chapitre, nous expliquons l'utilisation de l'interpréteur. Ce n'est pas la section la plus passionnante, mais c'est un passage obligé pour que vous puissiez mettre en pratique les exemples donnés plus loin.

Le reste du tutoriel présente diverses fonctionnalités du langage et du système Python au travers d'exemples, en commençant par les expressions simples, les instructions et les types de données, jusqu'à aborder des concepts avancés comme les exceptions et les classes, en passant par les fonctions et modules.

CHAPITRE 2

Mode d'emploi de l'interpréteur Python

2.1 Lancement de l'interpréteur

En général, vous trouvez l'interpréteur Python sous `/usr/local/bin/python3.11` sur les machines où il est disponible ; ajoutez `/usr/local/bin` au chemin de recherche de votre shell Unix afin de pouvoir le lancer en tapant la commande :

```
python3.11
```

dans le shell. [1] Le choix du répertoire où se trouve l'interpréteur étant une option d'installation, d'autres chemins sont possibles ; voyez avec votre gourou Python local ou votre administrateur système (par exemple, `/usr/local/python` est un endroit courant).

Sur les machines Windows sur lesquelles vous avez installé Python à partir du Microsoft Store, la commande `python3.11` est disponible. Si le lanceur py.exe est installé, vous pouvez utiliser la commande `py`. Voir setting-envvars pour d'autres façons de lancer Python.

Tapez un caractère de fin de fichier (`Ctrl-D` sous Unix, `Ctrl-Z` sous Windows) dans une invite de commande primaire provoque la fermeture de l'interpréteur avec un code de sortie nul. Si cela ne fonctionne pas, vous pouvez fermer l'interpréteur en tapant la commande `quit()`.

Les fonctionnalités d'édition de l'interpréteur comprennent l'édition interactive, la substitution depuis l'historique et la complétion sur les systèmes qui gèrent la bibliothèque GNU Readline. Un moyen rapide de tester comment est gérée l'édition de la ligne de commande, c'est de taper `Control-P` à la première invite de commande que vous rencontrez. Si cela bipe, vous disposez de l'édition de la ligne de commande ; lisez l'appendice *Édition interactive des entrées et substitution d'historique* pour une introduction aux touches. Si rien ne semble se produire ou si `^P` s'affiche, l'édition de la ligne de commande n'est pas disponible ; vous serez seulement en mesure d'utiliser la touche retour arrière pour supprimer des caractères de la ligne courante.

L'interpréteur fonctionne de façon similaire au shell Unix : lorsqu'il est appelé avec l'entrée standard connectée à un périphérique tty, il lit et exécute les commandes de façon interactive ; lorsqu'il est appelé avec un nom de fichier en argument ou avec un fichier comme entrée standard, il lit et exécute un *script* depuis ce fichier.

1. Sous Unix, par défaut, l'interpréteur Python 3.x n'est pas installé sous le nom de `python` afin de ne pas entrer en conflit avec une éventuelle installation de Python 2.x.

Une autre façon de lancer l'interpréteur est `python -c commande [arg] ...`. Cela exécute les instructions de *commande* de façon analogue à l'option `-c` du shell. Parce que les instructions Python contiennent souvent des espaces et d'autres caractères spéciaux pour le shell, il est généralement conseillé de mettre *commande* entre guillemets simples.

Certains modules Python sont aussi utiles en tant que scripts. Ils peuvent être appelés avec `python -m module [arg] ...` qui exécute le fichier source de *module* comme si vous aviez tapé son nom complet dans la ligne de commande.

Quand un fichier de script est utilisé, il est parfois utile de pouvoir lancer le script puis d'entrer dans le mode interactif après coup. Cela est possible en passant `-i` avant le script.

Tous les paramètres utilisables en ligne de commande sont documentés dans using-on-general.

2.1.1 Passage d'arguments

Lorsqu'ils sont connus de l'interpréteur, le nom du script et les arguments additionnels sont représentés sous forme d'une liste assignée à la variable `argv` du module `sys`. Vous pouvez y accéder en exécutant `import sys`. La liste contient au minimum un élément ; quand aucun script ni aucun argument n'est donné, `sys.argv[0]` est une chaine vide. Quand `'-'` (qui représente l'entrée standard) est passé comme nom de script, `sys.argv[0]` contient `'-'`. Quand `-c` *commande* est utilisé, `sys.argv[0]` contient `'-c'`. Enfin, quand `-m` *module* est utilisé, le nom complet du module est assigné à `sys.argv[0]`. Les options trouvées après `-c` *commande* ou `-m` *module* ne sont pas lues comme options de l'interpréteur Python mais laissées dans `sys.argv` pour être utilisées par le module ou la commande.

2.1.2 Mode interactif

Lorsque des commandes sont lues depuis un tty, l'interpréteur est dit être en *mode interactif*. Dans ce mode, il demande la commande suivante avec le *prompt primaire*, en général trois signes plus-grand-que (`>>>`) ; pour les lignes de continuation, il affiche le *prompt secondaire*, par défaut trois points (`...`). L'interpréteur affiche un message de bienvenue indiquant son numéro de version et une notice de copyright avant d'afficher le premier prompt :

```
$ python3.11
Python 3.11 (default, April 4 2021, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les lignes de continuation sont nécessaires pour entrer une construction multi-lignes. Par exemple, regardez cette instruction `if` :

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Pour plus d'informations sur le mode interactif, voir *Mode interactif*.

2.2 L'interpréteur et son environnement

2.2.1 Encodage du code source

Par défaut, Python considère que ses fichiers sources sont encodés en UTF-8. Dans cet encodage, les caractères de la plupart des langues peuvent être utilisés à la fois dans les chaînes de caractères, identifiants et commentaires. Notez que la bibliothèque standard n'utilise que des caractères ASCII dans ses identifiants et que nous considérons que c'est une bonne habitude que tout code portable devrait suivre. Pour afficher correctement tous ces caractères, votre éditeur doit reconnaître que le fichier est en UTF-8 et utiliser une police qui comprend tous les caractères utilisés dans le fichier.

Pour annoncer un encodage différent de l'encodage par défaut, une ligne de commentaire particulière doit être ajoutée en tant que *première* ligne du fichier. Sa syntaxe est la suivante :

```
# -*- coding: encoding -*-
```

où *encoding* est un des `codecs` géré par Python.

Par exemple, pour déclarer un encodage *Windows-1252*, la première ligne de votre code source doit être :

```
# -*- coding: cp1252 -*-
```

Une exception à la règle *première ligne* est lorsque la première ligne est un *shebang UNIX*. Dans ce cas, la déclaration de l'encodage doit être placée sur la deuxième ligne du fichier. Par exemple :

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Notes

CHAPITRE 3

Introduction informelle à Python

Dans les exemples qui suivent, les entrées et sorties se distinguent par la présence ou l'absence d'invite (»> et ...) : pour reproduire les exemples, vous devez taper tout ce qui est après l'invite, au moment où celle-ci apparaît ; les lignes qui n'affichent pas d'invite sont les sorties de l'interpréteur. Notez qu'une invite secondaire affichée seule sur une ligne dans un exemple indique que vous devez entrer une ligne vide ; ceci est utilisé pour terminer une commande multi-lignes.

Beaucoup d'exemples de ce manuel, même ceux saisis à l'invite de l'interpréteur, incluent des commentaires. Les commentaires en Python commencent avec un caractère croisillon, #, et s'étendent jusqu'à la fin de la ligne. Un commentaire peut apparaître au début d'une ligne ou à la suite d'un espace ou de code, mais pas à l'intérieur d'une chaîne de caractères littérale. Un caractère croisillon à l'intérieur d'une chaîne de caractères est juste un caractère croisillon. Comme les commentaires ne servent qu'à expliquer le code et ne sont pas interprétés par Python, ils peuvent être ignorés lorsque vous tapez les exemples.

Quelques exemples :

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Utilisation de Python comme une calculatrice

Essayons quelques commandes Python simples. Démarrez l'interpréteur et attendez l'invite primaire, >>>. Ça ne devrait pas être long.

3.1.1 Les nombres

L'interpréteur agit comme une simple calculatrice : vous pouvez lui entrer une expression et il vous affiche la valeur. La syntaxe des expressions est simple : les opérateurs +, -, * et / fonctionnent comme dans la plupart des langages (par exemple, Pascal ou C) ; les parenthèses peuvent être utilisées pour faire des regroupements. Par exemple :

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

Les nombres entiers (comme `2`, `4`, `20`) sont de type `int`, alors que les décimaux (comme `5.0`, `1.6`) sont de type `float`. Vous trouvez plus de détails sur les types numériques plus loin dans ce tutoriel.

Les divisions (`/`) donnent toujours des `float`. Utilisez l'opérateur `//` pour effectuer une *division entière* et donc obtenir un résultat entier. Pour obtenir le reste d'une division entière, utilisez l'opérateur `%` :

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

En Python, il est possible de calculer des puissances avec l'opérateur `**` [1] :

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

Le signe égal (=) est utilisé pour affecter une valeur à une variable. Dans ce cas, aucun résultat n'est affiché avant l'invite suivante :

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Si une variable n'est pas « définie » (si aucune valeur ne lui a été affectée), son utilisation produit une erreur :

1. Puisque `**` est prioritaire sur `-`, `-3 ** 2` est interprété `-(3 ** 2)` et vaut donc `-9`. Pour éviter cela et obtenir `9`, utilisez des parenthèses : `(-3) ** 2`.

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Les nombres à virgule flottante sont tout à fait admis (NdT : Python utilise le point . comme séparateur entre la partie entière et la partie décimale des nombres, c'est la convention anglo-saxonne) ; les opérateurs avec des opérandes de types différents convertissent l'opérande de type entier en type virgule flottante :

```
>>> 4 * 3.75 - 1
14.0
```

En mode interactif, la dernière expression affichée est affectée à la variable _. Ainsi, lorsque vous utilisez Python comme calculatrice, cela vous permet de continuer des calculs facilement, par exemple :

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Cette variable doit être considérée comme une variable en lecture seule par l'utilisateur. Ne lui affectez pas de valeur explicitement — vous créeriez ainsi une variable locale indépendante, avec le même nom, qui masquerait la variable native et son fonctionnement magique.

En plus des `int` et des `float`, il existe les `Decimal` et les `Fraction`. Python gère aussi les nombres complexes, en utilisant le suffixe `j` ou `J` pour indiquer la partie imaginaire (tel que `3+5j`).

3.1.2 Chaînes de caractères

En plus des nombres, Python sait aussi manipuler des chaînes de caractères, qui peuvent être exprimées de différentes manières. Elles peuvent être écrites entre guillemets simples (`'...'`) ou entre guillemets (`"..."`) sans distinction [2]. \ peut être utilisé pour protéger un guillemet :

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

En mode interactif, l'interpréteur affiche les chaînes de caractères entre guillemets. Les guillemets et autres caractères spéciaux sont protégés avec des barres obliques inverses (*backslash* en anglais). Bien que cela puisse être affiché différemment de ce qui a été entré (les guillemets peuvent changer), les deux formats sont équivalents. La chaîne est affichée

2. Contrairement à d'autres langages, les caractères spéciaux comme `\n` ont la même signification entre guillemets (`"..."`) ou entre guillemets simples (`'...'`). La seule différence est que, dans une chaîne entre guillemets, il n'est pas nécessaire de protéger les guillemets simples et vice-versa.

entre guillemets si elle contient un guillemet simple et aucun guillemet, sinon elle est affichée entre guillemets simples. La fonction `print()` affiche les chaînes de manière plus lisible, en retirant les guillemets et en affichant les caractères spéciaux qui étaient protégés par une barre oblique inverse :

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

Si vous ne voulez pas que les caractères précédés d'un `\` soient interprétés comme étant spéciaux, utilisez les *chaînes brutes* (*raw strings* en anglais) en préfixant la chaîne d'un `r` :

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Il existe une petite subtilité concernant les chaînes brutes : une chaîne brute ne peut pas se terminer par un nombre impair de caractères `\` ; voir l'entrée FAQ pour plus d'informations et des solutions de contournement.

Les chaînes de caractères peuvent s'étendre sur plusieurs lignes. Utilisez alors des triples guillemets, simples ou doubles : `'''...'''` ou `"""..."""`. Les retours à la ligne sont automatiquement inclus, mais on peut l'empêcher en ajoutant `\` à la fin de la ligne. L'exemple suivant :

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

produit l'affichage suivant (notez que le premier retour à la ligne n'est pas inclus) :

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

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

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Plusieurs chaînes de caractères, écrites littéralement (c'est-à-dire entre guillemets), côte à côte, sont automatiquement concaténées.

```
>>> 'Py' 'thon'
'Python'
```

Cette fonctionnalité est surtout intéressante pour couper des chaînes trop longues :

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Cela ne fonctionne cependant qu'avec les chaînes littérales, pas avec les variables ni les expressions :

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

Pour concaténer des variables, ou des variables avec des chaînes littérales, utilisez l'opérateur + :

```
>>> prefix + 'thon'
'Python'
```

Les chaînes de caractères peuvent être indexées (ou indicées, c'est-à-dire que l'on peut accéder aux caractères par leur position), le premier caractère d'une chaîne étant à la position 0. Il n'existe pas de type distinct pour les caractères, un caractère est simplement une chaîne de longueur 1 :

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Les indices peuvent également être négatifs, on compte alors en partant de la droite. Par exemple :

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Notez que, comme `-0` égale `0`, les indices négatifs commencent par `-1`.

En plus d'accéder à un élément par son indice, il est aussi possible de « trancher » (*slice* en anglais) une chaîne. Accéder à une chaîne par un indice permet d'obtenir un caractère, *trancher* permet d'obtenir une sous-chaîne :

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

Les valeurs par défaut des indices de tranches ont une utilité ; le premier indice vaut zéro par défaut (c.-à-d. lorsqu'il est omis), le deuxième correspond par défaut à la taille de la chaîne de caractères

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

Notez que le début est toujours inclus et la fin toujours exclue. Cela assure que `s[:i] + s[i:]` est toujours égal à `s` :

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Pour mémoriser la façon dont les tranches fonctionnent, vous pouvez imaginer que les indices pointent *entre* les caractères, le côté gauche du premier caractère ayant la position 0. Le côté droit du dernier caractère d'une chaîne de *n* caractères a alors pour indice *n*. Par exemple :

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

La première ligne de nombres donne la position des indices 0…6 dans la chaîne ; la deuxième ligne donne l'indice négatif correspondant. La tranche de *i* à *j* est constituée de tous les caractères situés entre les bords libellés *i* et *j*, respectivement.

Pour des indices non négatifs, la longueur d'une tranche est la différence entre ces indices, si les deux sont entre les bornes. Par exemple, la longueur de `word[1:3]` est 2.

Utiliser un indice trop grand produit une erreur :

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Cependant, les indices hors bornes sont gérés silencieusement lorsqu'ils sont utilisés dans des tranches :

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Les chaînes de caractères, en Python, ne peuvent pas être modifiées. On dit qu'elles sont *immuables*. Affecter une nouvelle valeur à un indice dans une chaîne produit une erreur :

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Si vous avez besoin d'une chaîne différente, vous devez en créer une nouvelle :

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

La fonction native `len()` renvoie la longueur d'une chaîne :

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Voir aussi :

textseq Les chaînes de caractères sont des exemples de *types séquences* ; elles acceptent donc les opérations classiques prises en charge par ces types.

string-methods Les chaînes de caractères gèrent un large éventail de méthodes de transformations basiques et de recherche.

f-strings Des chaînes littérales qui contiennent des expressions.

formatstrings Informations sur le formatage des chaînes avec la méthode `str.format()`.

old-string-formatting Description détaillée des anciennes méthodes de mise en forme, appelées lorsque les chaînes de caractères sont à gauche de l'opérateur `%`.

3.1.3 Listes

Python connaît différents types de données *combinés*, utilisés pour regrouper plusieurs valeurs. Le plus souple est la *liste*, qui peut être écrit comme une suite, placée entre crochets, de valeurs (éléments) séparées par des virgules. Les éléments d'une liste ne sont pas obligatoirement tous du même type, bien qu'à l'usage ce soit souvent le cas.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Comme les chaînes de caractères (et toute autre type de *sequence*), les listes peuvent être indicées et découpées :

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Toutes les opérations par tranches renvoient une nouvelle liste contenant les éléments demandés. Cela signifie que l'opération suivante renvoie une copie (superficielle) de la liste :

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Les listes gèrent aussi les opérations comme les concaténations :

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Mais à la différence des chaînes qui sont *immuables*, les listes sont *muables* : il est possible de modifier leur contenu :

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Il est aussi possible d'ajouter de nouveaux éléments à la fin d'une liste avec la méthode `append()` (les méthodes sont abordées plus tard) :

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Des affectations de tranches sont également possibles, ce qui peut même modifier la taille de la liste ou la vider complètement :

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

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

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Il est possible d'imbriquer des listes (c.-à-d. créer des listes contenant d'autres listes). Par exemple :

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Premiers pas vers la programmation

Bien entendu, on peut utiliser Python pour des tâches plus compliquées que d'additionner deux et deux. Par exemple, on peut écrire le début de la suite de Fibonacci comme ceci :

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

Cet exemple introduit plusieurs nouvelles fonctionnalités.

— La première ligne contient une *affectation multiple* : les variables `a` et `b` se voient affecter simultanément leurs nouvelles valeurs 0 et 1. Cette méthode est encore utilisée à la dernière ligne, pour démontrer que les expressions sur la partie droite de l'affectation sont toutes évaluées avant que les affectations ne soient effectuées. Ces expressions en partie droite sont toujours évaluées de la gauche vers la droite.

— La boucle `while` s'exécute tant que la condition (ici : `a < 10`) reste vraie. En Python, comme en C, tout entier différent de zéro est vrai et zéro est faux. La condition peut aussi être une chaîne de caractères, une liste, ou en fait toute séquence ; une séquence avec une valeur non nulle est vraie, une séquence vide est fausse. Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs de comparaison standards sont écrits comme en C : < (inférieur), > (supérieur), == (égal), <= (inférieur ou égal), >= (supérieur ou égal) et != (non égal).

— Le *corps* de la boucle est *indenté* : l'indentation est la méthode utilisée par Python pour regrouper des instructions. En mode interactif, vous devez saisir une tabulation ou des espaces pour chaque ligne indentée. En pratique, vous aurez intérêt à utiliser un éditeur de texte pour les saisies plus compliquées ; tous les éditeurs de texte dignes de ce nom disposent d'une fonction d'auto-indentation. Lorsqu'une expression composée est saisie en mode interactif, elle doit être suivie d'une ligne vide pour indiquer qu'elle est terminée (car l'analyseur ne peut pas deviner que vous venez de saisir la dernière ligne). Notez bien que toutes les lignes à l'intérieur d'un bloc doivent être indentées au même niveau.

— La fonction `print()` écrit les valeurs des paramètres qui lui sont fournis. Ce n'est pas la même chose que d'écrire l'expression que vous voulez afficher (comme nous l'avons fait dans l'exemple de la calculatrice), en raison de la manière qu'a `print` de gérer les paramètres multiples, les nombres décimaux et les chaînes. Les chaînes sont affichées sans apostrophe et une espace est insérée entre les éléments de telle sorte que vous pouvez facilement formater les choses, comme ceci :

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

Le paramètre nommé *end* peut servir pour enlever le retour à la ligne ou pour terminer la ligne par une autre chaîne :

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Notes

CHAPITRE 4

D'autres outils de contrôle de flux

En plus de l'instruction `while` qui vient d'être présentée, Python dispose des instructions de contrôle de flux classiques que l'on trouve dans d'autres langages, mais toujours avec ses propres tournures.

4.1 L'instruction `if`

L'instruction `if` est sans doute la plus connue. Par exemple :

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Il peut y avoir un nombre quelconque de parties `elif` et la partie `else` est facultative. Le mot clé `elif` est un raccourci pour *else if*, et permet de gagner un niveau d'indentation. Une séquence `if` ... `elif` ... `elif` ... est par ailleurs équivalente aux instructions `switch` ou `case` disponibles dans d'autres langages.

Pour les comparaisons avec beaucoup de constantes, ainsi que les tests d'appartenance à un type ou de forme d'un attribut, l'instruction `match` décrite plus bas peut se révéler utile (voir *L'instruction match*).

4.2 L'instruction `for`

L'instruction `for` que propose Python est un peu différente de celle que l'on peut trouver en C ou en Pascal. Au lieu de toujours itérer sur une suite arithmétique de nombres (comme en Pascal), ou de donner à l'utilisateur la possibilité de définir le pas d'itération et la condition de fin (comme en C), l'instruction `for` en Python itère sur les éléments d'une séquence (qui peut être une liste, une chaîne de caractères…), dans l'ordre dans lequel ils apparaissent dans la séquence. Par exemple :

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Écrire du code qui modifie une collection tout en itérant dessus peut s'avérer délicat. Il est généralement plus simple de boucler sur une copie de la collection ou de créer une nouvelle collection :

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '□□□': 'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3 La fonction `range()`

Si vous devez itérer sur une suite de nombres, la fonction native `range()` est faite pour cela. Elle génère des suites arithmétiques :

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

Le dernier élément fourni en paramètre ne fait jamais partie de la liste générée ; `range(10)` génère une liste de 10 valeurs, dont les valeurs vont de 0 à 9. Il est possible de spécifier une valeur de début et une valeur d'incrément différentes (y compris négative pour cette dernière, que l'on appelle également parfois le « pas ») :

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Pour itérer sur les indices d'une séquence, on peut combiner les fonctions `range()` et `len()` :

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Cependant, dans la plupart des cas, il est plus pratique d'utiliser la fonction `enumerate()`. Voyez pour cela *Techniques de boucles*.

Une chose étrange se produit lorsqu'on affiche un *range* :

```
>>> range(10)
range(0, 10)
```

L'objet renvoyé par `range()` se comporte presque comme une liste, mais ce n'en est pas une. Cet objet génère les éléments de la séquence au fur et à mesure de l'itération, sans réellement produire la liste en tant que telle, économisant ainsi de l'espace.

On appelle de tels objets des *itérables*, c'est-à-dire des objets qui conviennent à des fonctions ou constructions qui s'attendent à quelque chose duquel elles peuvent tirer des éléments, successivement, jusqu'à épuisement. Nous avons vu que l'instruction `for` est une de ces constructions, et un exemple de fonction qui prend un itérable en paramètre est `sum()` :

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Plus loin nous voyons d'autres fonctions qui donnent des itérables ou en prennent en paramètre. De plus amples détails sur `list()` sont donnés dans *Structures de données*.

4.4 Les instructions `break`, `continue` et les clauses `else` au sein des boucles

L'instruction `break`, comme en C, interrompt la boucle `for` ou `while` la plus profonde.

Les boucles peuvent également disposer d'une instruction `else` ; celle-ci est exécutée lorsqu'une boucle se termine alors que tous ses éléments ont été traités (dans le cas d'un `for`) ou que la condition devient fausse (dans le cas d'un `while`), mais pas lorsque la boucle est interrompue par une instruction `break`. L'exemple suivant, qui effectue une recherche de nombres premiers, en est une démonstration :

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Oui, ce code est correct. Regardez attentivement : l'instruction `else` est rattachée à la boucle `for`, et **non** à l'instruction `if`.)

Lorsqu'elle utilisée dans une boucle, la clause `else` est donc plus proche de celle associée à une instruction `try` que de celle associée à une instruction `if` : la clause `else` d'une instruction `try` s'exécute lorsqu'aucune exception n'est déclenchée, et celle d'une boucle lorsqu'aucun `break` n'intervient. Pour plus d'informations sur l'instruction `try` et le traitement des exceptions, consultez la section *Gestion des exceptions*.

L'instruction `continue`, également empruntée au C, fait passer la boucle à son itération suivante :

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 L'instruction `pass`

L'instruction `pass` ne fait rien. Elle peut être utilisée lorsqu'une instruction est nécessaire pour fournir une syntaxe correcte, mais qu'aucune action ne doit être effectuée. Par exemple :

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

On utilise couramment cette instruction pour créer des classes minimales :

```
>>> class MyEmptyClass:
...     pass
...
```

Un autre cas d'utilisation du `pass` est de réserver un espace en phase de développement pour une fonction ou un traitement conditionnel, vous permettant ainsi de construire votre code à un niveau plus abstrait. L'instruction `pass` est alors ignorée silencieusement :

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6 L'instruction `match`

L'instruction `match` confronte la valeur d'une expression à plusieurs filtres successifs donnés par les instructions `case`. L'instruction `match` peut faire penser au `switch` que l'on trouve dans les langages C, Java, JavaScript et autres, mais elle ressemble plus au filtrage par motif des langages Rust et Haskell. Seul le premier filtre qui correspond est exécuté et elle permet aussi d'extraire dans des variables des composantes de la valeur, comme les éléments d'une séquence ou les attributs d'un objet.

Dans sa plus simple expression, une instruction `match` compare une valeur à des littéraux :

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Remarquez l'emploi du signe souligné _ dans le dernier bloc, qui est normalement un nom de variable spécial. Ici, c'est un filtre *attrape-tout*, c'est-à-dire qu'il accepte toutes les valeurs. Si aucun des filtres dans les `case` ne fonctionne, aucune des branches indentées sous les `case` n'est exécutée.

On peut combiner plusieurs littéraux en un seul filtre avec le signe `|`, qui se lit OU :

```
case 401 | 403 | 404:
    return "Not allowed"
```

Les filtres peuvent prendre une forme similaire aux affectations multiples, et provoquer la liaison de variables :

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Observez bien cet exemple ! Le premier filtre contient simplement deux littéraux. C'est une sorte d'extension des filtres littéraux. Mais les deux filtres suivants mélangent un littéral et un nom de variable. Si un tel filtre réussit, il provoque l'af-

fectation à la variable. Le quatrième filtre est constitué de deux variables, ce qui le fait beaucoup ressembler à l'affectation multiple `(x, y) = point`.

Si vous structurez vos données par l'utilisation de classes, vous pouvez former des filtres avec le nom de la classe suivi d'une liste d'arguments. Ces filtres sont semblables à l'appel d'un constructeur, et permettent de capturer des attributs :

```
class Point:
    x: int
    y: int

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Un certain nombre de classes natives, notamment les classes de données, prennent en charge le filtrage par arguments positionnels en définissant un ordre des attributs. Vous pouvez ajouter cette possibilité à vos propres classes en y définissant l'attribut spécial `__match_args__`. Par exemple, le mettre à `("x", "y")` rend tous les filtres ci-dessous équivalents (en particulier, tous provoquent la liaison de l'attribut `y` à la variable `var`) :

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Une méthode préconisée pour lire les filtres est de les voir comme une extension de ce que l'on peut placer à gauche du signe = dans une affectation. Cela permet de visualiser quelles variables sont liées à quoi. Seuls les noms simples, comme `var` ci-dessus, sont des variables susceptibles d'être liées à une valeur. Il n'y a jamais de liaison pour les noms qualifiés (avec un point, comme dans `truc.machin`), les noms d'attributs (tels que `x=` et `y=` dans l'exemple précédent) et les noms de classes (identifiés par les parenthèses à leur droite, comme `Point`).

On peut imbriquer les filtres autant que de besoin. Ainsi, on peut lire une courte liste de points comme ceci :

```
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

Un filtre peut comporter un `if`, qui introduit ce que l'on appelle une garde. Si le filtre réussit, la garde est alors testée et, si elle s'évalue à une valeur fausse, l'exécution continue au bloc `case` suivant. Les variables sont liées avant l'évaluation de la garde, et peuvent être utilisées à l'intérieur :

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Voici d'autres caractéristiques importantes de cette instruction :

— comme dans les affectations multiples, les filtres de *n*-uplet et de liste sont totalement équivalents et autorisent tous les types de séquences. Exception importante, ils n'autorisent pas les itérateurs ni les chaînes de caractères ;

— les filtres de séquence peuvent faire intervenir l'affectation étoilée : `[x, y, *reste]` ou `(x, y, *reste)` ont le même sens que dans une affectation avec =. Le nom de variable après l'étoile peut aussi être l'attrape-tout `_`. Ainsi, `(x, y, *_)` est un filtre qui reconnaît les séquences à deux éléments ou plus, en capturant les deux premiers et en ignorant le reste ;

— Il existe des filtres d'association. Par exemple, le filtre `{"bande_passante": b, "latence": l}` extrait les valeurs des clés `"bande_passante"` et `"latence"` dans un dictionnaire. Contrairement aux filtres de séquence, les clés absentes du filtre sont ignorées. L'affectation double-étoilée (`**reste`) fonctionne aussi (cependant, `**_` serait redondant et n'est donc pas permis) ;

— on peut capturer la valeur d'une partie d'un filtre avec le mot-clé `as`, par exemple :

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

Ce filtre, lorsqu'il est comparé à une séquence de deux points, réussit et capture le second dans la variable `p2` ;

— la plupart des littéraux sont comparés par égalité. Néanmoins, les singletons `True`, `False` et `None` sont comparés par identité ;

— les filtres peuvent contenir des noms qui se réfèrent à des constantes. Ces noms doivent impérativement être qualifiés (contenir un point) pour ne pas être interprétés comme des variables de capture :

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or 'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

Pour plus d'explications et d'exemples, lire la **PEP 636** (en anglais), qui est écrite sous forme de tutoriel.

4.7 Définir des fonctions

On peut créer une fonction qui écrit la suite de Fibonacci jusqu'à une limite imposée :

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Le mot-clé `def` introduit une *définition* de fonction. Il doit être suivi du nom de la fonction et d'une liste, entre parenthèses, de ses paramètres. L'instruction qui constitue le corps de la fonction débute à la ligne suivante et doit être indentée.

La première instruction d'une fonction peut, de façon facultative, être une chaîne de caractères littérale ; cette chaîne de caractères sera alors la chaîne de documentation de la fonction, appelée *docstring* (consultez la section *Chaînes de documentation* pour en savoir plus). Il existe des outils qui utilisent ces chaînes de documentation pour générer automatiquement une documentation en ligne ou imprimée, ou pour permettre à l'utilisateur de naviguer de façon interactive dans le code ; prenez-en l'habitude, c'est une bonne pratique que de documenter le code que vous écrivez.

L'exécution d'une fonction introduit une nouvelle table de symboles utilisée par les variables locales de la fonction. Plus précisément, toutes les affectations de variables effectuées au sein d'une fonction stockent les valeurs dans la table des symboles locaux ; en revanche, les références de variables sont recherchées dans la table des symboles locaux, puis dans les tables des symboles locaux des fonctions englobantes, puis dans la table des symboles globaux et finalement dans la table des noms des primitives. Par conséquent, bien qu'elles puissent être référencées, il est impossible d'affecter une valeur à une variable globale ou à une variable d'une fonction englobante (sauf pour les variables globales désignées dans une instruction `global` et, pour les variables des fonctions englobantes, désignées dans une instruction `nonlocal`).

Les paramètres effectifs (arguments) d'une fonction sont introduits dans la table des symboles locaux de la fonction appelée, au moment où elle est appelée ; par conséquent, les passages de paramètres se font *par valeur*, la *valeur* étant toujours une *référence* à un objet et non la valeur de l'objet lui-même [1]. Lorsqu'une fonction appelle une autre fonction, ou s'appelle elle-même par récursion, une nouvelle table de symboles locaux est créée pour cet appel.

Une définition de fonction associe un nom de fonction à un objet fonction dans l'espace de noms actuel. Pour l'interpréteur, l'objet référencé par ce nom est une fonction définie par l'utilisateur. Plusieurs noms peuvent faire référence à une même fonction, ils peuvent alors tous être utilisés pour appeler la fonction :

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Si vous venez d'autres langages, vous pouvez penser que `fib` n'est pas une fonction mais une procédure, puisqu'elle ne renvoie pas de résultat. En fait, même les fonctions sans instruction `return` renvoient une valeur, quoique ennuyeuse. Cette valeur est appelée `None` (c'est le nom d'une primitive). Écrire la valeur `None` est normalement supprimé par l'interpréteur lorsqu'il s'agit de la seule valeur qui doit être écrite. Vous pouvez le constater, si vous y tenez vraiment, en utilisant `print()` :

```
>>> fib(0)
>>> print(fib(0))
None
```

Il est facile d'écrire une fonction qui renvoie une liste de la série de Fibonacci au lieu de l'afficher :

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
```

(suite sur la page suivante)

1. En fait, *appels par référence d'objets* serait sans doute une description plus juste dans la mesure où, si un objet muable est passé en argument, l'appelant verra toutes les modifications qui lui auront été apportées par l'appelé (insertion d'éléments dans une liste…).

(suite de la page précédente)

```
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Cet exemple, comme d'habitude, illustre de nouvelles fonctionnalités de Python :

— l'instruction `return` provoque la sortie de la fonction en renvoyant une valeur. `return` sans expression en paramètre renvoie `None`. Arriver à la fin d'une fonction renvoie également `None` ;

— l'instruction `result.append(a)` appelle une *méthode* de l'objet `result` qui est une liste. Une méthode est une fonction qui « appartient » à un objet et qui est nommée `obj.methodname`, où `obj` est un objet (il peut également s'agir d'une expression) et `methodname` est le nom d'une méthode que le type de l'objet définit. Différents types définissent différentes méthodes. Des méthodes de différents types peuvent porter le même nom sans qu'il n'y ait d'ambigüité (vous pouvez définir vos propres types d'objets et leurs méthodes en utilisant des *classes*, voir *Classes*). La méthode `append()` utilisée dans cet exemple est définie pour les listes ; elle ajoute un nouvel élément à la fin de la liste. Dans cet exemple, elle est l'équivalent de `result = result + [a]`, en plus efficace.

4.8 Davantage sur la définition des fonctions

Il est également possible de définir des fonctions avec un nombre variable d'arguments. Trois syntaxes peuvent être utilisées, éventuellement combinées.

4.8.1 Valeur par défaut des arguments

La forme la plus utile consiste à indiquer une valeur par défaut pour certains arguments. Ceci crée une fonction qui peut être appelée avec moins d'arguments que ceux présents dans sa définition. Par exemple :

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Cette fonction peut être appelée de plusieurs façons :

— en ne fournissant que les arguments obligatoires : `ask_ok('Do you really want to quit?')` ;

— en fournissant une partie des arguments facultatifs : `ask_ok('OK to overwrite the file?', 2)` ;

— en fournissant tous les arguments : `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`.

Cet exemple présente également le mot-clé `in`. Celui-ci permet de tester si une séquence contient une certaine valeur.

Les valeurs par défaut sont évaluées lors de la définition de la fonction dans la portée de la *définition*, de telle sorte que

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

affiche `5`.

Avertissement important : la valeur par défaut n'est évaluée qu'une seule fois. Ceci fait une différence lorsque cette valeur par défaut est un objet muable tel qu'une liste, un dictionnaire ou des instances de la plupart des classes. Par exemple, la fonction suivante accumule les arguments qui lui sont passés au fil des appels successifs :

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

affiche

```
[1]
[1, 2]
[1, 2, 3]
```

Si vous ne voulez pas que cette valeur par défaut soit partagée entre des appels successifs, vous pouvez écrire la fonction de cette façon :

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.8.2 Les arguments nommés

Les fonctions peuvent également être appelées en utilisant des *arguments nommés* sous la forme `kwarg=value`. Par exemple, la fonction suivante :

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepte un argument obligatoire (`voltage`) et trois arguments facultatifs (`state`, `action` et `type`). Cette fonction peut être appelée de n'importe laquelle des façons suivantes :

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
```

(suite sur la page suivante)

(suite de la page précédente)

```
parrot('a million', 'bereft of life', 'jump')          # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')   # 1 positional, 1 keyword
```

mais tous les appels qui suivent sont incorrects :

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Dans un appel de fonction, les arguments nommés doivent suivre les arguments positionnés. Tous les arguments nommés doivent correspondre à l'un des arguments acceptés par la fonction (par exemple, `actor` n'est pas un argument accepté par la fonction `parrot`), mais leur ordre n'est pas important. Ceci inclut également les arguments obligatoires (`parrot(voltage=1000)` est également correct). Aucun argument ne peut recevoir une valeur plus d'une fois, comme l'illustre cet exemple incorrect du fait de cette restriction :

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Quand un dernier paramètre formel est présent sous la forme `**name`, il reçoit un dictionnaire (voir typesmapping) contenant tous les arguments nommés à l'exception de ceux correspondant à un paramètre formel. Ceci peut être combiné à un paramètre formel sous la forme `*name` (décrit dans la section suivante) qui lui reçoit un *n-uplet* contenant les arguments positionnés au-delà de la liste des paramètres formels (`*name` doit être présent avant `**name`). Par exemple, si vous définissez une fonction comme ceci :

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Elle pourrait être appelée comme ceci :

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

et, bien sûr, elle affiche :

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
```

(suite sur la page suivante)

(suite de la page précédente)

```
client : John Cleese
sketch : Cheese Shop Sketch
```

Notez que Python garantit que l'ordre d'affichage des arguments est le même que l'ordre dans lesquels ils sont fournis lors de l'appel à la fonction.

4.8.3 Paramètres spéciaux

Par défaut, les arguments peuvent être passés à une fonction Python par position, ou explicitement en les nommant. Pour la lisibilité et la performance, il est logique de restreindre la façon dont les arguments peuvent être transmis afin qu'un développeur n'ait qu'à regarder la définition de la fonction pour déterminer si les éléments sont transmis par position seule, par position ou nommé, ou seulement nommé.

Voici à quoi ressemble une définition de fonction :

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

où / et * sont facultatifs. S'ils sont utilisés, ces symboles indiquent par quel type de paramètre un argument peut être transmis à la fonction : position seule, position ou nommé, et seulement nommé. Les paramètres par mot-clé sont aussi appelés paramètres nommés.

Les arguments positionnels-ou-nommés

Si / et * ne sont pas présents dans la définition de fonction, les arguments peuvent être passés à une fonction par position ou par nommés.

Paramètres positionnels uniquement

En y regardant de plus près, il est possible de marquer certains paramètres comme *positionnels uniquement*. S'ils sont marqués comme *positionnels uniquement*, l'ordre des paramètres est important, et les paramètres ne peuvent pas être transmis en tant que « arguments nommés ». Les paramètres « positionnels uniquement » sont placés avant un /. Le / est utilisé pour séparer logiquement les paramètres « positionnels uniquement » du reste des paramètres. S'il n'y a pas de / dans la définition de fonction, il n'y a pas de paramètres « positionnels uniquement ».

Les paramètres qui suivent le / peuvent être *positionnels-ou-nommés* ou *nommés-uniquement*.

Arguments nommés uniquement

Pour marquer les paramètres comme *uniquement nommés*, indiquant que les paramètres doivent être passés avec l'argument comme mot-clé, placez un * dans la liste des arguments juste avant le premier paramètre *uniquement nommé*.

Exemples de fonctions

Considérons l'exemple suivant de définitions de fonctions en portant une attention particulière aux marqueurs / et * :

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

La première définition de fonction, `standard_arg`, la forme la plus familière, n'impose aucune restriction sur la convention d'appel et les arguments peuvent être passés par position ou nommés :

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

La deuxième fonction `pos_only_arg` restreint le passage aux seuls arguments par position car il y a un / dans la définition de fonction :

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

La troisième fonction `kwd_only_args` n'autorise que les arguments nommés comme l'indique le * dans la définition de fonction :

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

Et la dernière utilise les trois conventions d'appel dans la même définition de fonction :

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword␣
↪arguments: 'pos_only'
```

Enfin, considérons cette définition de fonction qui a une collision potentielle entre l'argument positionnel `name` et `**kwds` qui a `name` comme mot-clé :

```
def foo(name, **kwds):
    return 'name' in kwds
```

Il n'y a pas d'appel possible qui renvoie `True` car le mot-clé `'name'` est toujours lié au premier paramètre. Par exemple :

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Mais en utilisant `/` (arguments positionnels seulement), c'est possible puisqu'il permet d'utiliser `name` comme argument positionnel et `'name'` comme mot-clé dans les arguments nommés :

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

En d'autres termes, les noms des paramètres seulement positionnels peuvent être utilisés sans ambiguïté dans `**kwds`.

Récapitulatif

Le cas d'utilisation détermine les paramètres à utiliser dans la définition de fonction :

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Quelques conseils :

- utilisez les paramètres positionnels si vous voulez que le nom des paramètres soit masqué à l'utilisateur. Ceci est utile lorsque les noms de paramètres n'ont pas de signification réelle, si vous voulez faire respecter l'ordre des arguments lorsque la fonction est appelée ou si vous avez besoin de prendre certains paramètres positionnels et mots-clés arbitraires ;
- utilisez les paramètres nommés lorsque les noms ont un sens et que la définition de la fonction est plus compréhensible avec des noms explicites ou si vous voulez empêcher les utilisateurs de se fier à la position de l'argument qui est passé ;
- dans le cas d'une API, utilisez les paramètres seulement positionnels pour éviter de casser l'API si le nom du paramètre est modifié dans l'avenir.

4.8.4 Listes d'arguments arbitraires

Pour terminer, l'option la moins fréquente consiste à indiquer qu'une fonction peut être appelée avec un nombre arbitraire d'arguments. Ces arguments sont intégrés dans un *n*-uplet (voir *n-uplets et séquences*). Avant le nombre variable d'arguments, zéro ou plus arguments normaux peuvent apparaître

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalement, ces arguments `variadiques` sont les derniers paramètres, parce qu'ils agrègent toutes les valeurs suivantes. Tout paramètre placé après le paramètre `*arg` ne pourra être utilisé que comme argument nommé, pas comme argument positionnel

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.8.5 Séparation des listes d'arguments

La situation inverse intervient lorsque les arguments sont déjà dans une liste ou un *n*-uplet mais doivent être séparés pour un appel de fonction nécessitant des arguments positionnés séparés. Par exemple, la primitive `range()` attend des arguments *start* et *stop* distincts. S'ils ne sont pas disponibles séparément, écrivez l'appel de fonction en utilisant l'opérateur `*` pour séparer les arguments présents dans une liste ou un *n*-uplet :

```
>>> list(range(3, 6))            # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

De la même façon, les dictionnaires peuvent fournir des arguments nommés en utilisant l'opérateur `**` :

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'␣
↪demised !
```

4.8.6 Fonctions anonymes

Avec le mot-clé `lambda`, vous pouvez créer de petites fonctions anonymes. En voici une qui renvoie la somme de ses deux arguments : `lambda a, b: a+b`. Les fonctions lambda peuvent être utilisées partout où un objet fonction est attendu. Elles sont syntaxiquement restreintes à une seule expression. Sémantiquement, elles ne sont que du sucre syntaxique pour une définition de fonction normale. Comme les fonctions imbriquées, les fonctions lambda peuvent référencer des variables de la portée englobante :

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

L'exemple précédent utilise une fonction anonyme pour renvoyer une fonction. Une autre utilisation classique est de donner une fonction minimaliste directement en tant que paramètre :

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.8.7 Chaînes de documentation

Voici quelques conventions concernant le contenu et le format des chaînes de documentation.

Il convient que la première ligne soit toujours courte et résume de manière concise l'utilité de l'objet. Afin d'être bref, nul besoin de rappeler le nom de l'objet ou son type, qui sont accessibles par d'autres moyens (sauf si le nom est un verbe qui décrit une opération). La convention veut que la ligne commence par une majuscule et se termine par un point.

S'il y a d'autres lignes dans la chaîne de documentation, la deuxième ligne devrait être vide, pour la séparer visuellement du reste de la description. Les autres lignes peuvent alors constituer un ou plusieurs paragraphes décrivant le mode d'utilisation de l'objet, ses effets de bord, etc.

L'analyseur de code Python ne supprime pas l'indentation des chaînes de caractères littérales multi-lignes, donc les outils qui utilisent la documentation doivent si besoin faire cette opération eux-mêmes. La convention suivante s'applique : la première ligne non vide *après* la première détermine la profondeur d'indentation de l'ensemble de la chaîne de documentation (on ne peut pas utiliser la première ligne qui est généralement accolée aux guillemets d'ouverture de la chaîne de caractères et dont l'indentation n'est donc pas visible). Les espaces « correspondant » à cette profondeur d'indentation sont alors supprimées du début de chacune des lignes de la chaîne. Aucune ligne ne devrait présenter un niveau d'indentation inférieur mais, si cela arrive, toutes les espaces situées en début de ligne doivent être supprimées. L'équivalent des espaces doit être testé après expansion des tabulations (normalement remplacées par 8 espaces).

Voici un exemple de chaîne de documentation multi-lignes :

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
```

(suite sur la page suivante)

(suite de la page précédente)

```
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8.8 Annotations de fonctions

Les annotations de fonction sont des métadonnées optionnelles décrivant les types utilisés par une fonction définie par l'utilisateur (voir les **PEP 3107** et **PEP 484** pour plus d'informations).

Les *annotations* sont stockées dans l'attribut `__annotations__` de la fonction, sous la forme d'un dictionnaire, et n'ont aucun autre effet. Les annotations sur les paramètres sont définies par deux points (`:`) après le nom du paramètre suivi d'une expression donnant la valeur de l'annotation. Les annotations de retour sont définies par `->` suivi d'une expression, entre la liste des paramètres et les deux points de fin de l'instruction `def`. L'exemple suivant a un paramètre requis, un paramètre optionnel et la valeur de retour annotés :

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.9 Aparté : le style de codage

Maintenant que vous êtes prêt à écrire des programmes plus longs et plus complexes, il est temps de parler du *style de codage*. La plupart des langages peuvent être écrits (ou plutôt *formatés*) selon différents styles ; certains sont plus lisibles que d'autres. Rendre la lecture de votre code plus facile aux autres est toujours une bonne idée et adopter un bon style de codage peut énormément vous y aider.

En Python, la plupart des projets adhèrent au style défini dans la **PEP 8** ; elle met en avant un style de codage très lisible et agréable à l'œil. Chaque développeur Python se doit donc de la lire et de s'en inspirer autant que possible ; voici ses principaux points notables :

- utilisez des indentations de 4 espaces et pas de tabulation.
  4 espaces constituent un bon compromis entre une indentation courte (qui permet une profondeur d'imbrication plus importante) et une longue (qui rend le code plus facile à lire). Les tabulations introduisent de la confusion et doivent être proscrites autant que possible ;
- faites en sorte que les lignes ne dépassent pas 79 caractères, au besoin en insérant des retours à la ligne.
  Vous facilitez ainsi la lecture pour les utilisateurs qui n'ont qu'un petit écran et, pour les autres, cela leur permet de visualiser plusieurs fichiers côte à côte ;
- utilisez des lignes vides pour séparer les fonctions et les classes, ou pour scinder de gros blocs de code à l'intérieur de fonctions ;
- lorsque c'est possible, placez les commentaires sur leurs propres lignes ;
- utilisez les chaînes de documentation ;
- utilisez des espaces autour des opérateurs et après les virgules, mais pas juste à l'intérieur des parenthèses : `a = f(1, 2) + g(3, 4)` ;
- nommez toujours vos classes et fonctions de la même manière ; la convention est d'utiliser une notation `UpperCamelCase` pour les classes, et `minuscules_avec_trait_bas` pour les fonctions et méthodes.

Utilisez toujours `self` comme nom du premier argument des méthodes (voyez *Une première approche des classes* pour en savoir plus sur les classes et les méthodes) ;

— n'utilisez pas d'encodage exotique dès lors que votre code est censé être utilisé dans des environnements internationaux. Par défaut, Python travaille en UTF-8. Pour couvrir tous les cas, préférez le simple ASCII ;

— de la même manière, n'utilisez que des caractères ASCII pour vos noms de variables s'il est envisageable qu'une personne parlant une autre langue lise ou doive modifier votre code.

Notes

CHAPITRE 5

Structures de données

Ce chapitre reprend plus en détail quelques points déjà décrits précédemment et introduit également de nouvelles notions.

5.1 Compléments sur les listes

Le type liste dispose de méthodes supplémentaires. Voici toutes les méthodes des objets liste :

`list.`**`append`**`(`*x*`)`

Ajoute un élément à la fin de la liste. Équivalent à `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`

Étend la liste en y ajoutant tous les éléments de l'itérable. Équivalent à `a[len(a):] = iterable`.

`list.`**`insert`**`(`*i*, *x*`)`

Insère un élément à la position indiquée. Le premier argument est la position de l'élément avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste et `a.insert(len(a), x)` est équivalent à `a.append(x)`.

`list.`**`remove`**`(`*x*`)`

Supprime de la liste le premier élément dont la valeur est égale à *x*. Une exception `ValueError` est levée s'il n'existe aucun élément avec cette valeur.

`list.`**`pop`**`(`[*i*]`)`

Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, `a.pop()` enlève et renvoie le dernier élément de la liste (les crochets autour du *i* dans la signature de la méthode indiquent que ce paramètre est facultatif et non que vous devez placer des crochets dans votre code ! Vous retrouverez cette notation fréquemment dans le Guide de Référence de la Bibliothèque Python).

`list.`**`clear`**`()`

Supprime tous les éléments de la liste. Équivalent à `del a[:]`.

`list.index(`*x*[, *start*[, *end*]]`)`

Renvoie la position du premier élément de la liste dont la valeur égale *x* (en commençant à compter les positions à partir de zéro). Une exception `ValueError` est levée si aucun élément n'est trouvé.

Les arguments optionnels *start* et *end* sont interprétés de la même manière que dans la notation des tranches et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'indice renvoyé est calculé relativement au début de la séquence complète et non relativement à *start*.

`list.count(`*x*`)`

Renvoie le nombre d'éléments ayant la valeur *x* dans la liste.

`list.sort(`*, *key=None*, *reverse=False*`)`

Ordonne les éléments dans la liste (les arguments peuvent personnaliser l'ordonnancement, voir `sorted()` pour leur explication).

`list.reverse()`

Inverse l'ordre des éléments dans la liste.

`list.copy()`

Renvoie une copie superficielle de la liste. Équivalent à `a[:]`.

L'exemple suivant utilise la plupart des méthodes des listes :

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Vous avez probablement remarqué que les méthodes qui ne font que modifier la liste, comme `insert`, `remove` ou `sort`, n'affichent pas de valeur de retour (elles renvoient `None`) [1]. C'est un principe respecté par toutes les structures de données muables en Python.

Vous avez peut-être remarqué aussi que certaines données ne peuvent pas être ordonnées ni comparées. Par exemple, la liste `[None, 'hello', 10]` ne peut pas être ordonnée parce que les entiers ne peuvent pas être comparés aux chaînes de caractères et *None* ne peut pas être comparé à d'autres types. En outre, il existe certains types qui n'ont pas de relation d'ordre définie. Par exemple, `3+4j < 5+7j` n'est pas une comparaison valide.

1. D'autres langages renvoient l'objet modifié, ce qui permet de chaîner les méthodes comme ceci : `d->insert("a")->remove("b")->sort();`.

5.1.1 Utilisation des listes comme des piles

Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti » ou LIFO pour *last-in, first-out* en anglais). Pour ajouter un élément sur la pile, utilisez la méthode `append()`. Pour récupérer l'objet au sommet de la pile, utilisez la méthode `pop()` sans indicateur de position. Par exemple :

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Utilisation des listes comme des files

Il est également possible d'utiliser une liste comme une file, où le premier élément ajouté est le premier récupéré (« premier entré, premier sorti » ou FIFO pour *first-in, first-out*) ; toutefois, les listes ne sont pas très efficaces pour réaliser ce type de traitement. Alors que les ajouts et suppressions en fin de liste sont rapides, les insertions ou les retraits en début de liste sont lents (car tous les autres éléments doivent être décalés d'une position).

Pour implémenter une file, utilisez plutôt la classe `collections.deque` qui a été conçue spécialement pour réaliser rapidement les opérations d'ajout et de retrait aux deux extrémités. Par exemple :

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Listes en compréhension

Les listes en compréhension fournissent un moyen de construire des listes de manière très concise. Une application classique consiste à construire une liste dont les éléments sont les résultats d'une opération appliquée à chaque élément d'une autre séquence ; une autre consiste à créer une sous-séquence des éléments respectant une condition donnée.

Par exemple, supposons que l'on veuille créer une liste de carrés, comme :

```
>>> squares = []
>>> for x in range(10):
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Notez que cela crée (ou remplace) une variable nommée `x` qui existe toujours après l'exécution de la boucle. On peut calculer une liste de carrés sans effet de bord avec :

```
squares = list(map(lambda x: x**2, range(10)))
```

ou, de manière équivalente :

```
squares = [x**2 for x in range(10)]
```

ce qui est plus court et lisible.

Une liste en compréhension consiste à placer entre crochets une expression suivie par une clause `for` puis par zéro ou plus clauses `for` ou `if`. Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux :

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

et c'est équivalent à :

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Notez que l'ordre des instructions `for` et `if` est le même dans ces différents extraits de code.

Si l'expression est un *n*-uplet (c'est-à-dire `(x, y)` dans cet exemple), elle doit être mise entre parenthèses

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Les listes en compréhension peuvent contenir des expressions complexes et des fonctions imbriquées :

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 Listes en compréhensions imbriquées

La première expression dans une liste en compréhension peut être n'importe quelle expression, y compris une autre liste en compréhension.

Voyez l'exemple suivant d'une matrice de 3 par 4, implémentée sous la forme de 3 listes de 4 éléments :

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Cette compréhension de liste transpose les lignes et les colonnes :

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Comme nous l'avons vu dans la section précédente, la liste en compréhension à l'intérieur est évaluée dans le contexte de l'instruction `for` qui la suit, donc cet exemple est équivalent à :

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

lequel est lui-même équivalent à :

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Dans des cas concrets, il est toujours préférable d'utiliser des fonctions natives plutôt que des instructions de contrôle de flux complexes. La fonction `zip()` fait dans ce cas un excellent travail :

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Voir *Séparation des listes d'arguments* pour plus de détails sur l'astérisque de cette ligne.

5.2 L'instruction `del`

Il existe un moyen de retirer un élément d'une liste à partir de sa position au lieu de sa valeur : l'instruction `del`. Elle diffère de la méthode `pop()` qui, elle, renvoie une valeur. L'instruction `del` peut également être utilisée pour supprimer des tranches d'une liste ou la vider complètement (ce que nous avions fait auparavant en affectant une liste vide à la tranche). Par exemple :

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` peut aussi être utilisée pour supprimer des variables :

```
>>> del a
```

À partir de là, référencer le nom `a` est une erreur (au moins jusqu'à ce qu'une autre valeur lui soit affectée). Vous trouverez d'autres utilisations de la fonction `del` dans la suite de ce tutoriel.

5.3 *n*-uplets et séquences

Nous avons vu que les listes et les chaînes de caractères ont beaucoup de propriétés en commun, comme l'indiçage et les opérations sur des tranches. Ce sont deux exemples de *séquences* (voir typesseq). Comme Python est un langage en constante évolution, d'autres types de séquences y seront peut-être ajoutés. Il existe également un autre type standard de séquence : le *n*-uplet (*tuple* en anglais).

Un *n*-uplet consiste en différentes valeurs séparées par des virgules, par exemple :

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
```

(suite sur la page suivante)

(suite de la page précédente)

```
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Comme vous pouvez le voir, les *n*-uplets sont toujours affichés entre parenthèses, de façon à ce que des *n*-uplets imbriqués soient interprétés correctement ; ils peuvent être saisis avec ou sans parenthèses, même si celles-ci sont souvent nécessaires (notamment lorsqu'un *n*-uplet fait partie d'une expression plus longue). Il n'est pas possible d'affecter de valeur à un élément d'un *n*-uplet ; par contre, il est en revanche possible de créer des *n*-uplets contenant des objets muables, comme des listes.

Si les *n*-uplets peuvent sembler similaires aux listes, ils sont souvent utilisés dans des cas différents et pour des raisons différentes. Les *n*-uplets sont *immuables* et contiennent souvent des séquences hétérogènes d'éléments auxquelles on accède par « dissociation » (*unpacking* en anglais, voir plus loin) ou par indice (ou même par attribut dans le cas des `namedtuples`). Les listes sont souvent *muables* et contiennent des éléments généralement homogènes auxquels on accède en itérant sur la liste.

La construction de *n*-uplets ne contenant aucun ou un seul élément est un cas particulier : la syntaxe a quelques tournures spécifiques pour le gérer. Un *n*-uplet vide se construit avec une paire de parenthèses vides ; un *n*-uplet avec un seul élément se construit en faisant suivre la valeur par une virgule (placer cette valeur entre parenthèses ne suffit pas). Pas très joli, mais efficace. Par exemple :

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

L'instruction `t = 12345, 54321, 'hello!'` est un exemple *d'agrégation de *n*-uplet** (*tuple packing* en anglais) : les valeurs `12345`, `54321` et `hello!` sont agrégées ensemble dans un *n*-uplet. L'opération inverse est aussi possible :

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

Ceci est appelé, de façon plus ou moins appropriée, une *dissociation de séquence* (*sequence unpacking* en anglais) et fonctionne pour toute séquence placée à droite de l'expression. Cette dissociation requiert autant de variables dans la partie gauche qu'il y a d'éléments dans la séquence. Notez également que cette affectation multiple est juste une combinaison entre une agrégation de *n*-uplet et une dissociation de séquence.

5.4 Ensembles

Python fournit également un type de donnée pour les *ensembles*. Un ensemble est une collection non ordonnée sans éléments en double. Un ensemble permet de réaliser des tests d'appartenance ou des suppressions de doublons de manière simple. Les ensembles savent également effectuer les opérations mathématiques telles que les unions, intersections, différences et différences symétriques.

On crée des ensembles en appelant avec des accolades ou avec la fonction `set()`. Notez que `{}` ne crée pas un ensemble vide, mais un dictionnaire (une structure de données dont nous allons parler dans la séquence suivante) vide ; utilisez `set()` pour ce cas.

Voici une brève démonstration :

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Tout comme pour les *listes en compréhension*, il est possible d'écrire des ensembles en compréhension :

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5 Dictionnaires

Un autre type de donnée très utile, natif dans Python, est le *dictionnaire* (voir typesmapping). Ces dictionnaires sont parfois présents dans d'autres langages sous le nom de « mémoires associatives » ou de « tableaux associatifs ». À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des *clés*, qui peuvent être de n'importe quel type immuable ; les chaînes de caractères et les nombres peuvent toujours être des clés. Des *n*-uplets peuvent être utilisés comme clés s'ils ne contiennent que des chaînes, des nombres ou des *n*-uplets ; si un *n*-uplet contient un objet muable, de façon directe ou indirecte, il ne peut pas être utilisé comme une clé. Vous ne pouvez pas utiliser des listes comme clés, car les listes peuvent être modifiées en place en utilisant des affectations par position, par tranches ou via des méthodes comme `append()` ou `extend()`.

Le plus simple est de voir un dictionnaire comme un ensemble de paires *clé-valeur* au sein duquel les clés doivent être uniques. Une paire d'accolades crée un dictionnaire vide : `{}`. Placer une liste de paires clé-valeur séparées par des

virgules à l'intérieur des accolades ajoute les valeurs correspondantes au dictionnaire ; c'est également de cette façon que les dictionnaires sont affichés.

Les opérations classiques sur un dictionnaire consistent à stocker une valeur pour une clé et à extraire la valeur correspondant à une clé. Il est également possible de supprimer une paire clé-valeur avec `del`. Si vous stockez une valeur pour une clé qui est déjà utilisée, l'ancienne valeur associée à cette clé est perdue. Si vous tentez d'extraire une valeur associée à une clé qui n'existe pas, une exception est levée.

Exécuter `list(d)` sur un dictionnaire `d` renvoie une liste de toutes les clés utilisées dans le dictionnaire, dans l'ordre d'insertion (si vous voulez qu'elles soient ordonnées, utilisez `sorted(d)`). Pour tester si une clé est dans le dictionnaire, utilisez le mot-clé `in`.

Voici un petit exemple utilisant un dictionnaire :

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Le constructeur `dict()` fabrique un dictionnaire directement à partir d'une liste de paires clé-valeur stockées sous la forme de *n*-uplets :

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

De plus, il est possible de créer des dictionnaires en compréhension depuis un jeu de clés et valeurs :

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Lorsque les clés sont de simples chaînes de caractères, il est parfois plus facile de définir les paires en utilisant des paramètres nommés :

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Techniques de boucles

Lorsque vous faites une boucle sur un dictionnaire, la clé et la valeur associée peuvent être récupérées en même temps en utilisant la méthode `items()`

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Lorsque vous faites une boucle sur une séquence, la position et la valeur associée peuvent être récupérées en même temps en utilisant la fonction `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Pour faire une boucle sur deux séquences ou plus en même temps, les éléments peuvent être associés en utilisant la fonction `zip()`

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Pour faire une boucle en sens inverse sur une séquence, commencez par spécifier la séquence dans son ordre normal, puis appliquez la fonction `reversed()`

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Pour faire une boucle sur une séquence de manière ordonnée, utilisez la fonction `sorted()` qui renvoie une nouvelle liste ordonnée sans altérer la source

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
```

(suite sur la page suivante)

(suite de la page précédente)

```
orange
orange
pear
```

L'utilisation de la fonction `set()` sur une séquence élimine les doublons. Combiner les fonctions `sorted()` et `set()` sur une séquence est la façon « canonique » d'itérer sur les éléments uniques d'une séquence dans l'ordre.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

Il est parfois tentant de modifier une liste pendant que l'on itère dessus. Il est souvent plus simple et plus sûr de créer une nouvelle liste à la place.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Plus d'informations sur les conditions

Les conditions utilisées dans une instruction `while` ou `if` peuvent contenir n'importe quel opérateur, pas seulement des comparaisons.

Les opérateurs de comparaison `in` et `not in` testent si une valeur appartient (ou pas) à un conteneur. Les opérateurs `is` et `is not` testent si deux objets sont vraiment le même objet. Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle des opérateurs numériques.

Les comparaisons peuvent être chaînées. Par exemple, `a < b == c` teste si `a` est inférieur à `b` et si, de plus, `b` égale `c`.

Les comparaisons peuvent être combinées en utilisant les opérateurs booléens `and` et `or`, le résultat d'une comparaison (ou de toute expression booléenne) pouvant être inversé avec `not`. Ces opérateurs ont une priorité inférieure à celle des opérateurs de comparaison ; entre eux, `not` a la priorité la plus élevée et `or` la plus faible, de telle sorte que `A and not B or C` est équivalent à `(A and (not B)) or C`. Comme d'habitude, les parenthèses permettent d'exprimer l'instruction désirée.

Les opérateurs booléens `and` et `or` sont appelés opérateurs *en circuit court* : leurs arguments sont évalués de la gauche vers la droite et l'évaluation s'arrête dès que le résultat est déterminé. Par exemple, si `A` et `C` sont vrais et `B` est faux, `A and B and C` n'évalue pas l'expression `C`. Lorsqu'elle est utilisée en tant que valeur et non en tant que booléen, la valeur de retour d'un opérateur en circuit court est celle du dernier argument évalué.

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

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Notez qu'en Python, à la différence du C, une affectation à l'intérieur d'une expression doit être faite explicitement avec l'opérateur morse `:=`. Cela évite des erreurs fréquentes que l'on rencontre en C, lorsque l'on tape = alors que l'on voulait faire un test avec ==.

5.8 Comparer des séquences avec d'autres types

Des séquences peuvent être comparées avec d'autres séquences du même type. La comparaison utilise un ordre *lexicographique* : les deux premiers éléments de chaque séquence sont comparés et, s'ils diffèrent, cela détermine le résultat de la comparaison ; s'ils sont égaux, les deux éléments suivants sont comparés à leur tour et ainsi de suite jusqu'à ce que l'une des séquences soit épuisée. Si deux éléments à comparer sont eux-mêmes des séquences du même type, alors la comparaison lexicographique est effectuée récursivement. Si tous les éléments des deux séquences sont égaux, les deux séquences sont alors considérées comme égales. Si une séquence est une sous-séquence de l'autre, la séquence la plus courte est celle dont la valeur est inférieure. La comparaison lexicographique des chaînes de caractères utilise le code Unicode des caractères. Voici quelques exemples de comparaisons entre séquences de même type :

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

Comparer des objets de type différents avec < ou > est autorisé si les objets ont des méthodes de comparaison appropriées. Par exemple, les types numériques sont comparés via leur valeur numérique, donc `0` égale `0,0`, etc. Dans les autres cas, au lieu de donner un ordre imprévisible, l'interpréteur lève une exception `TypeError`.

Notes

CHAPITRE 6

Modules

Lorsque vous quittez et entrez à nouveau dans l'interpréteur Python, tout ce que vous avez déclaré dans la session précédente est perdu. Afin de rédiger des programmes plus longs, vous devez utiliser un éditeur de texte, préparer votre code dans un fichier et exécuter Python avec ce fichier en paramètre. Cela s'appelle créer un *script*. Lorsque votre programme grandit, vous pouvez séparer votre code dans plusieurs fichiers. Ainsi, il vous est facile de réutiliser des fonctions écrites pour un programme dans un autre sans avoir à les copier.

Pour gérer cela, Python vous permet de placer des définitions dans un fichier et de les utiliser dans un script ou une session interactive. Un tel fichier est appelé un *module* et les définitions d'un module peuvent être importées dans un autre module ou dans le module *main* (qui est le module qui contient vos variables et définitions lors de l'exécution d'un script au niveau le plus haut ou en mode interactif).

Un module est un fichier contenant des définitions et des instructions. Son nom de fichier est le nom du module suffixé de `.py`. À l'intérieur d'un module, son propre nom est accessible par la variable `__name__`. Par exemple, prenez votre éditeur favori et créez un fichier `fibo.py` dans le répertoire courant qui contient :

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Maintenant, ouvrez un interpréteur et importez le module en tapant :

```
>>> import fibo
```

Les noms des fonctions définies dans `fibo` ne sont pas ajoutés directement dans l'*espace de nommage* courant (voir *Portées et espaces de nommage en Python* pour plus de détails), seul le nom de module `fibo` est ajouté. L'appel des fonctions se fait donc *via* le nom du module :

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Si vous avez l'intention d'utiliser souvent une fonction, il est possible de lui assigner un nom local :

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Les modules en détail

Un module peut contenir aussi bien des instructions que des déclarations de fonctions. Ces instructions permettent d'initialiser le module. Elles ne sont exécutées que la *première* fois que le nom d'un module est trouvé dans un `import` [1] (elles sont aussi exécutées lorsque le fichier est exécuté en tant que script).

Chaque module possède son propre espace de nommage, utilisé comme espace de nommage global par toutes les fonctions définies par le module. Ainsi l'auteur d'un module peut utiliser des variables globales dans un module sans se soucier de collisions de noms avec des variables globales définies par l'utilisateur du module. Cependant, si vous savez ce que vous faites, vous pouvez modifier une variable globale d'un module avec la même notation que pour accéder aux fonctions : `nommodule.nomelement`.

Des modules peuvent importer d'autres modules. Il est courant, mais pas obligatoire, de ranger tous les `import` au début du module (ou du script). Les noms des modules importés, s'ils sont placés au début d'un module (en dehors des fonctions et des classes) sont ajoutés à l'espace de nommage global du module.

Il existe une variante de l'instruction `import` qui importe les noms d'un module directement dans l'espace de nommage du module qui l'importe, par exemple :

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Cela n'insère pas le nom du module depuis lequel les définitions sont récupérées dans l'espace de nommage local (dans cet exemple, `fibo` n'est pas défini).

Il existe même une variante permettant d'importer tous les noms qu'un module définit :

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

1. En réalité, la déclaration d'une fonction est elle-même une instruction ; son exécution ajoute le nom de la fonction dans l'espace de nommage global du module.

Tous les noms ne commençant pas par un tiret bas (_) sont importés. Dans la grande majorité des cas, les développeurs n'utilisent pas cette syntaxe puisqu'en important un ensemble indéfini de noms, des noms déjà définis peuvent se retrouver masqués.

Notez qu'en général, importer * d'un module ou d'un paquet est déconseillé. Souvent, le code devient difficilement lisible. Son utilisation en mode interactif est acceptée pour gagner quelques secondes.

Si le nom du module est suivi par `as`, alors le nom suivant `as` est directement lié au module importé.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Dans les faits, le module est importé de la même manière qu'avec `import fibo`, la seule différence est qu'il sera disponible sous le nom de `fib`.

C'est aussi valide en utilisant `from`, et a le même effet :

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note : pour des raisons d'efficacité, chaque module n'est importé qu'une fois par session de l'interpréteur. Par conséquent, si vous modifiez vos modules, vous devez redémarrer l'interpréteur — ou, si c'est juste un module que vous voulez tester interactivement, utilisez `importlib.reload()`, par exemple `import importlib ; importlib.reload(modulename)`.

6.1.1 Exécuter des modules comme des scripts

Lorsque vous exécutez un module Python avec

```
python fibo.py <arguments>
```

le code du module est exécuté comme si vous l'aviez importé mais son `__name__` vaut `"__main__"`. Donc, en ajoutant ces lignes à la fin du module :

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

vous pouvez rendre le fichier utilisable comme script aussi bien que comme module importable, car le code qui analyse la ligne de commande n'est lancé que si le module est exécuté comme fichier « main » :

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Si le fichier est importé, le code n'est pas exécuté :

```
>>> import fibo
>>>
```

C'est typiquement utilisé soit pour proposer une interface utilisateur pour un module, soit pour lancer les tests sur le module (exécuter le module en tant que script lance les tests).

6.1.2 Les dossiers de recherche de modules

Lorsqu'un module nommé par exemple `spam` est importé, il est d'abord recherché parmi les modules natifs. Les noms de ces modules sont listés dans `sys.builtin_module_names`. S'il n'est pas trouvé, l'interpréteur cherche un fichier nommé `spam.py` dans une liste de dossiers donnée par la variable `sys.path`. Par défaut, `sys.path` est initialisée à :

- le dossier contenant le script courant (ou le dossier courant si aucun script n'est donné) ;
- `PYTHONPATH` (une liste de dossiers, utilisant la même syntaxe que la variable shell `PATH`) ;
- La valeur par défaut, qui dépend de l'installation (incluant par convention un dossier `site-packages`, géré par le module `site`).

Vous trouverez plus de détails dans sys-path-init.

Note : sur les systèmes qui gèrent les liens symboliques, le dossier contenant le script courant est résolu après avoir suivi le lien symbolique du script. Autrement dit, le dossier contenant le lien symbolique n'est **pas** ajouté aux dossiers de recherche de modules.

Après leur initialisation, les programmes Python peuvent modifier leur `sys.path`. Le dossier contenant le script courant est placé au début de la liste des dossiers à rechercher, avant les dossiers de bibliothèques. Cela signifie qu'un module dans ce dossier, ayant le même nom qu'un module, sera chargé à sa place. C'est une erreur typique, à moins que ce ne soit voulu. Voir *Modules standards* pour plus d'informations.

6.1.3 Fichiers Python « compilés »

Pour accélérer le chargement des modules, Python cache une version compilée de chaque module dans un fichier nommé `module(version).pyc` (ou *version* représente le format du fichier compilé, typiquement une version de Python) dans le dossier `__pycache__`. Par exemple, avec CPython 3.3, la version compilée de *spam.py* serait `__pycache__/spam.cpython-33.pyc`. Cette règle de nommage permet à des versions compilées par des versions différentes de Python de coexister.

Python compare les dates de modification du fichier source et de sa version compilée pour voir si le module doit être recompilé. Ce processus est entièrement automatique. Par ailleurs, les versions compilées sont indépendantes de la plateforme et peuvent donc être partagées entre des systèmes d'architectures différentes.

Il existe deux situations où Python ne vérifie pas le cache : le premier cas est lorsque le module est donné par la ligne de commande (cas où le module est toujours recompilé, sans même cacher sa version compilée) ; le second cas est lorsque le module n'a pas de source. Pour gérer un module sans source (où seule la version compilée est fournie), le module compilé doit se trouver dans le dossier source et sa source ne doit pas être présente.

Astuces pour les experts :

- vous pouvez utiliser les options `-O` ou `-OO` lors de l'appel à Python pour réduire la taille des modules compilés. L'option `-O` supprime les instructions `assert` et l'option `-OO` supprime aussi les documentations `__doc__`. Cependant, puisque certains programmes ont besoin de ces `__doc__`, vous ne devriez utiliser `-OO` que si vous savez ce que vous faites. Les modules « optimisés » sont marqués d'un `opt-` et sont généralement plus petits. Les versions futures de Python pourraient changer les effets de l'optimisation ;
- un programme ne s'exécute pas plus vite lorsqu'il est lu depuis un `.pyc`, il est juste chargé plus vite ;
- le module `compileall` peut créer des fichiers `.pyc` pour tous les modules d'un dossier ;
- vous trouvez plus de détails sur ce processus, ainsi qu'un organigramme des décisions, dans la **PEP 3147**.

6.2 Modules standards

Python est accompagné d'une bibliothèque de modules standards, décrits dans la documentation de la Bibliothèque Python, plus loin. Certains modules sont intégrés dans l'interpréteur, ils proposent des outils qui ne font pas partie du langage mais qui font tout de même partie de l'interpréteur, soit pour le côté pratique, soit pour mettre à disposition des outils essentiels tels que l'accès aux appels système. La composition de ces modules est configurable à la compilation et dépend aussi de la plateforme cible. Par exemple, le module `winreg` n'est proposé que sur les systèmes Windows. Un module mérite une attention particulière, le module `sys`, qui est présent dans tous les interpréteurs Python. Les variables `sys.ps1` et `sys.ps2` définissent les chaînes d'invites principales et secondaires :

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

Ces deux variables ne sont définies que si l'interpréteur est en mode interactif.

La variable `sys.path` est une liste de chaînes qui détermine les chemins de recherche de modules pour l'interpréteur. Elle est initialisée à un chemin par défaut pris de la variable d'environnement `PYTHONPATH` ou d'une valeur par défaut interne si `PYTHONPATH` n'est pas définie. `sys.path` est modifiable en utilisant les opérations habituelles des listes :

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La fonction `dir()`

La fonction interne `dir()` est utilisée pour trouver quels noms sont définis par un module. Elle donne une liste de chaînes classées par ordre lexicographique :

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
```

(suite sur la page suivante)

(suite de la page précédente)

```
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Sans paramètre, `dir()` liste les noms actuellement définis :

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Notez qu'elle liste tous les types de noms : les variables, fonctions, modules, etc.

`dir()` ne liste ni les fonctions primitives, ni les variables internes. Si vous voulez les lister, elles sont définies dans le module `builtins` :

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4 Les paquets

Les paquets sont un moyen de structurer les espaces de nommage des modules Python en utilisant une notation « pointée ». Par exemple, le nom de module `A.B` désigne le sous-module `B` du paquet `A`. De la même manière que l'utilisation des modules évite aux auteurs de différents modules d'avoir à se soucier des noms de variables globales des autres, l'utilisation des noms de modules avec des points évite aux auteurs de paquets contenant plusieurs modules tel que NumPy ou Pillow d'avoir à se soucier des noms des modules des autres.

Imaginez que vous voulez construire un ensemble de modules (un « paquet ») pour gérer uniformément les fichiers contenant du son et des données sonores. Il existe un grand nombre de formats de fichiers pour stocker du son (généralement identifiés par leur extension, par exemple `.wav`, `.aiff`, `.au`), vous avez donc besoin de créer et maintenir un nombre croissant de modules pour gérer la conversion entre tous ces formats. Vous voulez aussi pouvoir appliquer un certain nombre d'opérations sur ces sons : mixer, ajouter de l'écho, égaliser, ajouter un effet stéréo artificiel, etc. Donc, en plus des modules de conversion, vous allez écrire une myriade de modules permettant d'effectuer ces opérations. Voici une structure possible pour votre paquet (exprimée sous la forme d'une arborescence de fichiers) :

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Lorsqu'il importe des paquets, Python cherche dans chaque dossier de `sys.path` un sous-dossier du nom du paquet.

The `__init__.py` files are required to make Python treat directories containing the file as packages. This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Les utilisateurs d'un module peuvent importer ses modules individuellement, par exemple :

```
import sound.effects.echo
```

charge le sous-module `sound.effects.echo`. Il doit alors être référencé par son nom complet.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Une autre manière d'importer des sous-modules est :

```
from sound.effects import echo
```

charge aussi le sous-module `echo` et le rend disponible sans avoir à indiquer le préfixe du paquet. Il peut donc être utilisé comme ceci :

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Une autre méthode consiste à importer la fonction ou la variable désirée directement :

```
from sound.effects.echo import echofilter
```

Le sous-module `echo` est toujours chargé mais ici la fonction `echofilter()` est disponible directement :

```
echofilter(input, output, delay=0.7, atten=4)
```

Notez que lorsque vous utilisez `from package import element`, `element` peut aussi bien être un sous-module, un sous-paquet ou simplement un nom déclaré dans le paquet (une variable, une fonction ou une classe). L'instruction `import` cherche en premier si `element` est défini dans le paquet ; s'il ne l'est pas, elle cherche à charger un module et, si elle n'en trouve pas, une exception `ImportError` est levée.

Au contraire, en utilisant la syntaxe `import element.souselement.soussouselement`, chaque `element` sauf le dernier doit être un paquet. Le dernier `element` peut être un module ou un paquet, mais ne peut être ni une fonction, ni une classe, ni une variable définie dans l'élément précédent.

6.4.1 Importer * depuis un paquet

Qu'arrive-t-il lorsqu'un utilisateur écrit `from sound.effects import *` ? Idéalement, on pourrait espérer que Python aille chercher tous les sous-modules du paquet sur le système de fichiers et qu'ils seraient tous importés. Cela pourrait être long et importer certains sous-modules pourrait avoir des effets secondaires indésirables ou, du moins, désirés seulement lorsque le sous-module est importé explicitement.

La seule solution, pour l'auteur du paquet, est de fournir un index explicite du contenu du paquet. L'instruction `import` utilise la convention suivante : si le fichier `__init__.py` du paquet définit une liste nommée `__all__`, cette liste est utilisée comme liste des noms de modules devant être importés lorsque `from package import *` est utilisé. Il est de la responsabilité de l'auteur du paquet de maintenir cette liste à jour lorsque de nouvelles versions du paquet sont publiées. Un auteur de paquet peut aussi décider de ne pas autoriser d'importer `*` pour son paquet. Par exemple, le fichier `sound/effects/__init__.py` peut contenir le code suivant :

```
__all__ = ["echo", "surround", "reverse"]
```

Cela signifie que `from sound.effects import *` importe les trois sous-modules explicitement désignés du paquet `sound.effects`.

Si `__all__` n'est pas définie, l'instruction `from sound.effects import *` n'importe *pas* tous les sous-modules du paquet `sound.effects` dans l'espace de nommage courant mais s'assure seulement que le paquet `sound.effects` a été importé (c.-à-d. que tout le code du fichier `__init__.py` a été exécuté) et importe ensuite les noms définis dans le paquet. Cela inclut tous les noms définis (et sous-modules chargés explicitement) par `__init__.py`. Sont aussi inclus tous les sous-modules du paquet ayant été chargés explicitement par une instruction `import`. Typiquement :

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

Dans cet exemple, les modules `echo` et `surround` sont importés dans l'espace de nommage courant lorsque `from... import` est exécuté parce qu'ils sont définis dans le paquet `sound.effects` (cela fonctionne aussi lorsque `__all__` est définie).

Bien que certains modules ont été pensés pour n'exporter que les noms respectant une certaine structure lorsque `import *` est utilisé, `import *` reste considéré comme une mauvaise pratique dans du code à destination d'un environnement de production.

Rappelez-vous que rien ne vous empêche d'utiliser `from paquet import sous_module_specifique` ! C'est d'ailleurs la manière recommandée, à moins que le module qui fait les importations ait besoin de sous-modules ayant le même nom mais provenant de paquets différents.

6.4.2 Références internes dans un paquet

Lorsque les paquets sont organisés en sous-paquets (comme le paquet `sound` par exemple), vous pouvez utiliser des importations absolues pour cibler des paquets voisins. Par exemple, si le module `sound.filters.vocoder` a besoin du module `echo` du paquet `sound.effects`, il peut utiliser `from sound.effects import echo`.

Il est aussi possible d'écrire des importations relatives de la forme `from module import name`. Ces importations relatives sont préfixées par des points pour indiquer leur origine (paquet courant ou parent). Depuis le module `surround`, par exemple vous pouvez écrire :

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Notez que les importations relatives se fient au nom du module actuel. Puisque le nom du module principal est toujours `"__main__"`, les modules utilisés par le module principal d'une application ne peuvent être importés que par des importations absolues.

6.4.3 Paquets dans plusieurs dossiers

Les paquets possèdent un attribut supplémentaire, `__path__`, qui est une liste initialisée avant l'exécution du fichier `__init__.py`, contenant le nom de son dossier dans le système de fichiers. Cette liste peut être modifiée, altérant ainsi les futures recherches de modules et sous-paquets contenus dans le paquet.

Bien que cette fonctionnalité ne soit que rarement utile, elle peut servir à élargir la liste des modules trouvés dans un paquet.

Notes

CHAPITRE 7

Les entrées/sorties

Il existe bien des moyens de présenter les sorties d'un programme ; les données peuvent être affichées sous une forme lisible par un être humain ou sauvegardées dans un fichier pour une utilisation future. Ce chapitre présente quelques possibilités.

7.1 Formatage de données

Jusqu'ici, nous avons rencontré deux moyens d'écrire des données : les *déclarations d'expressions* et la fonction `print()`. Une troisième méthode consiste à utiliser la méthode `write()` des fichiers, avec le fichier de sortie standard référencé en tant que `sys.stdout`. Voyez le Guide de Référence de la Bibliothèque Standard pour en savoir plus.

Souvent vous voudrez plus de contrôle sur le formatage de vos sorties et aller au delà d'un affichage de valeurs séparées par des espaces. Il y a plusieurs moyens de les formater.

— Pour utiliser *les expressions formatées*, commencez une chaine de caractère avec `f` ou `F` avant d'ouvrir vos guillemets doubles ou triples. Dans ces chaines de caractère, vous pouvez entrer des expressions Python entre les caractères `{` et `}` qui peuvent contenir des variables ou des valeurs littérales.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

— La méthode `str.format()` sur les chaines de caractères exige un plus grand effort manuel. Vous utiliserez toujours les caractères `{` et `}` pour indiquer où une variable sera substituée et donner des détails sur son formatage, mais vous devrez également fournir les informations à formater.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

— Enfin, vous pouvez construire des concaténations de tranches de chaînes vous-même, et ainsi créer n'importe quel agencement. Le type des chaînes a des méthodes utiles pour aligner des chaînes dans une largeur de taille fixe.

Lorsque qu'un affichage basique suffit, pour afficher simplement une variable pour en inspecter le contenu, vous pouvez convertir n'importe quelle valeur en chaîne de caractères en utilisant la fonction `repr()` ou la fonction `str()`.

La fonction `str()` est destinée à représenter les valeurs sous une forme lisible par un être humain, alors que la fonction `repr()` est destinée à générer des représentations qui puissent être lues par l'interpréteur (ou qui lèvera une `SyntaxError` s'il n'existe aucune syntaxe équivalente). Pour les objets qui n'ont pas de représentation humaine spécifique, `str()` renvoie la même valeur que `repr()`. Beaucoup de valeurs, comme les nombres ou les structures telles que les listes ou les dictionnaires, ont la même représentation en utilisant les deux fonctions. Les chaînes de caractères, en particulier, ont deux représentations distinctes.

Quelques exemples :

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

Le module `string` contient une classe `Template` qui permet aussi de remplacer des valeurs au sein de chaînes de caractères, en utilisant des marqueurs comme `$x`, et en les remplaçant par les valeurs d'un dictionnaire, mais sa capacité à formater les chaînes est plus limitée.

7.1.1 Les chaines de caractères formatées (*f-strings*)

Les chaines de caractères formatées (aussi appelées f-strings) vous permettent d'inclure la valeur d'expressions Python dans des chaines de caractères en les préfixant avec `f` ou `F` et écrire des expressions comme `{expression}`.

L'expression peut être suivie d'un spécificateur de format. Cela permet un plus grand contrôle sur la façon dont la valeur est rendue. L'exemple suivant arrondit pi à trois décimales après la virgule :

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Donner un entier après le `':'` indique la largeur minimale de ce champ en nombre de caractères. C'est utile pour faire de jolis tableaux

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
```

(suite sur la page suivante)

(suite de la page précédente)

```
Jack        ==>        4098
Dcab        ==>        7678
```

D'autres modificateurs peuvent être utilisés pour convertir la valeur avant son formatage. `'!a'` applique `ascii()`, `'!s'` applique `str()`, et `'!r'` applique `repr()` :

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

Le spécificateur = peut être utilisé pour développer une expression en « texte de l'expression, un signe égal, puis la représentation de l'expression évaluée » :

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

Lisez self-documenting expressions pour davantage d'information sur le spécificateur =. Pour une référence sur ces spécifications de formats, voir le guide de référence pour le formatspec.

7.1.2 La méthode de chaine de caractères `format()`

L'utilisation de base de la méthode `str.format()` ressemble à ceci :

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

Les accolades et les caractères à l'intérieur (appelés les champs de formatage) sont remplacés par les objets passés en paramètres à la méthode `str.format()`. Un nombre entre accolades se réfère à la position de l'objet passé à la méthode `str.format()`.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

Si des arguments nommés sont utilisés dans la méthode `str.format()`, leurs valeurs sont utilisées en se basant sur le nom des arguments

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Les arguments positionnés et nommés peuvent être combinés arbitrairement :

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
```

Si vous avez une chaîne de formatage vraiment longue que vous ne voulez pas découper, il est possible de référencer les variables à formater par leur nom plutôt que par leur position. Utilisez simplement un dictionnaire et la notation entre crochets `'[]'` pour accéder aux clés

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Vous pouvez obtenir le même résultat en passant le dictionnaire `table` comme des arguments nommés en utilisant la notation `**`

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

C'est particulièrement utile en combinaison avec la fonction native `vars()` qui renvoie un dictionnaire contenant toutes les variables locales.

À titre d'exemple, les lignes suivantes produisent un ensemble de colonnes alignées de façon ordonnée donnant les entiers, leurs carrés et leurs cubes :

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Pour avoir une description complète du formatage des chaînes de caractères avec la méthode `str.format()`, lisez : formatstrings.

7.1.3 Formatage de chaînes à la main

Voici le même tableau de carrés et de cubes, formaté à la main :

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
```

(suite sur la page suivante)

(suite de la page précédente)

```
 9  81  729
10 100 1000
```

(Remarquez que l'espace séparant les colonnes vient de la manière dont `print()` fonctionne : il ajoute toujours des espaces entre ses arguments.)

La méthode `str.rjust()` des chaînes de caractères justifie à droite une chaîne dans un champ d'une largeur donnée en ajoutant des espaces sur la gauche. Il existe des méthodes similaires `str.ljust()` et `str.center()`. Ces méthodes n'écrivent rien, elles renvoient simplement une nouvelle chaîne. Si la chaîne passée en paramètre est trop longue, elle n'est pas tronquée mais renvoyée sans modification ; cela peut chambouler votre mise en page mais c'est souvent préférable à l'alternative, qui pourrait mentir sur une valeur (et si vous voulez vraiment tronquer vos valeurs, vous pouvez toujours utiliser une tranche, comme dans `x.ljust(n)[:n]`).

Il existe une autre méthode, `str.zfill()`, qui comble une chaîne numérique à gauche avec des zéros. Elle comprend les signes plus et moins :

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Anciennes méthodes de formatage de chaînes

L'opérateur % (modulo) peut également être utilisé pour le formatage des chaînes de caractères. Pour `'string' % values`, les instances de `%` dans la chaine de caractères `string` sont remplacées par zéro ou plusieurs d'éléments de `values`. Cette opération est communément appelée interpolation de chaîne de caractères. Par exemple :

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Vous trouvez plus d'informations dans la section old-string-formatting.

7.2 Lecture et écriture de fichiers

La fonction `open()` renvoie un *objet fichier* et est le plus souvent utilisée avec deux arguments positionnels et un argument nommé : `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

Le premier argument est une chaîne contenant le nom du fichier. Le deuxième argument est une autre chaîne contenant quelques caractères décrivant la façon dont le fichier est utilisé. *mode* peut être `'r'` quand le fichier n'est accédé qu'en lecture, `'w'` en écriture seulement (un fichier existant portant le même nom sera alors écrasé) et `'a'` ouvre le fichier en mode ajout (toute donnée écrite dans le fichier est automatiquement ajoutée à la fin). `'r+'` ouvre le fichier en mode lecture/écriture. L'argument *mode* est optionnel, sa valeur par défaut est `'r'`.

Normalement, les fichiers sont ouverts en *mode texte*, c'est-à-dire que vous lisez et écrivez des chaînes de caractères depuis et dans ce fichier, suivant un encodage donné via *encoding*. Si *encoding* n'est pas spécifié, l'encodage par défaut dépend de la plateforme (voir `open()`). UTF-8 étant le standard moderne de facto, `encoding="utf-8"` est recommandé à moins que vous ne sachiez que vous devez utiliser un autre encodage. L'ajout d'un `'b'` au mode ouvre le fichier en

mode binaire. Les données en mode binaire sont lues et écrites sous forme d'objets `bytes`. Vous ne pouvez pas spécifier *encoding* lorsque vous ouvrez un fichier en mode binaire.

En mode texte, le comportement par défaut, à la lecture, est de convertir les fin de lignes spécifiques à la plateforme (`\n` sur Unix, `\r\n` sur Windows, etc.) en simples `\n`. Lors de l'écriture, le comportement par défaut est d'appliquer l'opération inverse : les `\n` sont convertis dans leur équivalent sur la plateforme courante. Ces modifications effectuées automatiquement sont normales pour du texte mais détérioreraient des données binaires contenues dans un fichier de type `JPEG` ou `EXE`. Soyez particulièrement attentifs à ouvrir ces fichiers binaires en mode binaire.

C'est une bonne pratique d'utiliser le mot-clé `with` lorsque vous traitez des fichiers. Vous fermez ainsi toujours correctement le fichier, même si une exception est levée. Utiliser `with` est aussi beaucoup plus court que d'utiliser l'équivalent avec des blocs `try...finally` :

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

Si vous n'utilisez pas le mot clé `with`, vous devez appeler `f.close()` pour fermer le fichier, et ainsi immédiatement libérer les ressources qu'il utilise.

Avertissement : Appeler `f.write()` sans utiliser le mot clé `with` ni appeler `f.close()` **pourrait** mener à une situation où les arguments de `f.write()` ne seraient pas complètement écrits sur le disque, même si le programme se termine avec succès.

Après la fermeture du fichier, que ce soit *via* une instruction `with` ou en appelant `f.close()`, toute tentative d'utilisation de l'objet fichier échoue systématiquement.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Méthodes des objets fichiers

Les derniers exemples de cette section supposent qu'un objet fichier appelé `f` a déjà été créé.

Pour lire le contenu d'un fichier, appelez `f.read(taille)`, cette dernière lit une certaine quantité de données et la renvoie sous forme de chaîne (en mode texte) ou d'objet *bytes* (en mode binaire). *taille* est un argument numérique facultatif. Lorsque *taille* est omis ou négatif, la totalité du contenu du fichier sera lue et renvoyée ; c'est votre problème si le fichier est deux fois plus grand que la mémoire de votre machine. Sinon, au maximum *taille* caractères (en mode texte) ou *taille* octets (en mode binaire) sont lus et renvoyés. Si la fin du fichier est atteinte, `f.read()` renvoie une chaîne vide (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` lit une seule ligne du fichier ; un caractère de fin de ligne (`\n`) est laissé à la fin de la chaîne. Il n'est omis que sur la dernière ligne du fichier si celui-ci ne se termine pas un caractère de fin de ligne. Ceci permet de rendre

la valeur de retour non ambigüe : si `f.readline()` renvoie une chaîne vide, c'est que la fin du fichier a été atteinte, alors qu'une ligne vide est représentée par `'\n'` (une chaîne de caractères ne contenant qu'une fin de ligne).

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Pour lire ligne à ligne, vous pouvez aussi boucler sur l'objet fichier. C'est plus efficace en termes de gestion mémoire, plus rapide et donne un code plus simple :

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Pour construire une liste avec toutes les lignes d'un fichier, il est aussi possible d'utiliser `list(f)` ou `f.readlines()`.

`f.write(chaine)` écrit le contenu de *chaine* dans le fichier et renvoie le nombre de caractères écrits.

```
>>> f.write('This is a test\n')
15
```

Les autres types doivent être convertis, soit en une chaîne (en mode texte), soit en objet *bytes* (en mode binaire) avant de les écrire :

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` renvoie un entier indiquant la position actuelle dans le fichier, mesurée en octets à partir du début du fichier lorsque le fichier est ouvert en mode binaire, ou un nombre obscur en mode texte.

Pour modifier la position dans le fichier, utilisez `f.seek(décalage, origine)`. La position est calculée en ajoutant *décalage* à un point de référence ; ce point de référence est déterminé par l'argument *origine* : la valeur 0 pour le début du fichier, 1 pour la position actuelle et 2 pour la fin du fichier. *origine* peut être omis et sa valeur par défaut est 0 (Python utilise le début du fichier comme point de référence).

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Sur un fichier en mode texte (ceux ouverts sans `b` dans le mode), seuls les changements de position relatifs au début du fichier sont autorisés (sauf une exception : se rendre à la fin du fichier avec `seek(0, 2)`) et les seules valeurs possibles pour le paramètre *décalage* sont les valeurs renvoyées par `f.tell()`, ou zéro. Toute autre valeur pour le paramètre *décalage* produit un comportement indéfini.

Les fichiers disposent de méthodes supplémentaires, telles que `isatty()` et `truncate()` qui sont moins souvent utilisées ; consultez la Référence de la Bibliothèque Standard pour avoir un guide complet des objets fichiers.

7.2.2 Sauvegarde de données structurées avec le module `json`

Les chaînes de caractères peuvent facilement être écrites dans un fichier et relues. Les nombres nécessitent un peu plus d'effort, car la méthode `read()` ne renvoie que des chaînes. Elles doivent donc être passées à une fonction comme `int()`, qui prend une chaîne comme `'123'` en entrée et renvoie sa valeur numérique 123. Mais dès que vous voulez enregistrer des types de données plus complexes comme des listes, des dictionnaires ou des instances de classes, le traitement lecture/écriture à la main devient vite compliqué.

Plutôt que de passer son temps à écrire et déboguer du code permettant de sauvegarder des types de données compliqués, Python permet d'utiliser JSON (*JavaScript Object Notation*), un format répandu de représentation et d'échange de données. Le module standard appelé `json` peut transformer des données hiérarchisées Python en une représentation sous forme de chaîne de caractères. Ce processus est nommé *sérialiser*. Reconstruire les données à partir de leur représentation sous forme de chaîne est appelé *déserialiser*. Entre sa sérialisation et sa dé-sérialisation, la chaîne représentant les données peut avoir été stockée ou transmise à une autre machine.

Note : Le format JSON est couramment utilisé dans les applications modernes pour échanger des données. Beaucoup de développeurs le maîtrisent, ce qui en fait un format de prédilection pour l'interopérabilité.

Si vous avez un objet `x`, vous pouvez voir sa représentation JSON en tapant simplement :

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

Une variante de la fonction `dumps()`, nommée `dump()`, sérialise simplement l'objet donné vers un *text file*. Donc si `f` est un *text file* ouvert en écriture, il est possible de faire :

```
json.dump(x, f)
```

Pour reconstruire l'objet, si `f` est cette fois un *binary file* ou un *text file* ouvert en lecture :

```
x = json.load(f)
```

Note : Les fichiers JSON doivent être encodés en UTF-8. Utilisez `encoding="utf-8"` lorsque vous ouvrez un fichier JSON en tant que *fichier texte*, que ce soit en lecture ou en écriture.

Cette méthode de sérialisation peut sérialiser des listes et des dictionnaires. Mais sérialiser d'autres types de données requiert un peu plus de travail. La documentation du module `json` explique comment faire.

Voir aussi :

Le module `pickle`

Au contraire de *JSON*, *pickle* est un protocole permettant la sérialisation d'objets Python arbitrairement complexes. Il est donc spécifique à Python et ne peut pas être utilisé pour communiquer avec d'autres langages. Il est aussi, par défaut, une source de vulnérabilité : dé-sérialiser des données au format *pickle* provenant d'une source malveillante et particulièrement habile peut mener à exécuter du code arbitraire.

CHAPITRE 8

Erreurs et exceptions

Jusqu'ici, les messages d'erreurs ont seulement été mentionnés. Mais si vous avez essayé les exemples vous avez certainement vu plus que cela. En fait, il y a au moins deux types d'erreurs à distinguer : les *erreurs de syntaxe* et les *exceptions*.

8.1 Les erreurs de syntaxe

Les erreurs de syntaxe, qui sont des erreurs d'analyse du code, sont peut-être celles que vous rencontrez le plus souvent lorsque vous êtes encore en phase d'apprentissage de Python :

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

L'analyseur indique la ligne incriminée et affiche une petite « flèche » pointant vers le premier endroit de la ligne où l'erreur a été détectée. L'erreur est causée (ou, au moins, a été détectée comme telle) par le symbole placé *avant* la flèche. Dans cet exemple la flèche est sur la fonction `print()` car il manque deux points (`':'`) juste avant. Le nom du fichier et le numéro de ligne sont affichés pour vous permettre de localiser facilement l'erreur lorsque le code provient d'un script.

8.2 Exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des *exceptions* et ne sont pas toujours fatales : nous apprendrons bientôt comment les traiter dans vos programmes. La plupart des exceptions toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(suite sur la page suivante)

(suite de la page précédente)

```
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : les types indiqués dans l'exemple sont `ZeroDivisionError`, `NameError` et `TypeError`. Le texte affiché comme type de l'exception est le nom de l'exception native qui a été déclenchée. Ceci est vrai pour toutes les exceptions natives mais n'est pas une obligation pour les exceptions définies par l'utilisateur (même si c'est une convention bien pratique). Les noms des exceptions standards sont des identifiants natifs (pas des mots-clef réservés).

Le reste de la ligne fournit plus de détails en fonction du type de l'exception et de ce qui l'a causée.

La partie précédente du message d'erreur indique le contexte dans lequel s'est produite l'exception, sous la forme d'une trace de pile d'exécution. En général, celle-ci contient les lignes du code source ; toutefois, les lignes lues à partir de l'entrée standard ne sont pas affichées.

Vous trouvez la liste des exceptions natives et leur signification dans bltin-exceptions.

8.3 Gestion des exceptions

Il est possible d'écrire des programmes qui prennent en charge certaines exceptions. Regardez l'exemple suivant, qui demande une saisie à l'utilisateur jusqu'à ce qu'un entier valide ait été entré, mais permet à l'utilisateur d'interrompre le programme (en utilisant `Control-C` ou un autre raccourci que le système accepte) ; notez qu'une interruption générée par l'utilisateur est signalée en levant l'exception `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

L'instruction `try` fonctionne comme ceci :

- premièrement, la *clause try* (instruction(s) placée(s) entre les mots-clés `try` et `except`) est exécutée.
- si aucune exception n'intervient, la clause `except` est sautée et l'exécution de l'instruction `try` est terminée.
- si une exception intervient pendant l'exécution de la clause `try`, le reste de cette clause est sauté. Si le type d'exception levée correspond à un nom indiqué après le mot-clé `except`, la clause `except` correspondante est exécutée, puis l'exécution continue après le bloc `try`/`except`.
- si une exception intervient et ne correspond à aucune exception mentionnée dans la clause `except`, elle est transmise à l'instruction `try` de niveau supérieur ; si aucun gestionnaire d'exception n'est trouvé, il s'agit d'une *exception non gérée* et l'exécution s'arrête avec un message comme indiqué ci-dessus.

Une instruction `try` peut comporter plusieurs clauses `except` pour permettre la prise en charge de différentes exceptions. Mais un seul gestionnaire, au plus, sera exécuté. Les gestionnaires ne prennent en charge que les exceptions qui interviennent dans la clause `try` correspondante, pas dans d'autres gestionnaires de la même instruction `try`. Mais une même clause `except` peut citer plusieurs exceptions sous la forme d'un *n*-uplet entre parenthèses, comme dans cet exemple :

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Une classe dans une clause `except` est compatible avec une exception si elle est de la même classe ou d'une de ses classes dérivées. Mais l'inverse n'est pas vrai, une clause `except` spécifiant une classe dérivée n'est pas compatible avec une classe mère. Par exemple, le code suivant affiche B, C et D dans cet ordre :

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Notez que si les clauses `except` avaient été inversées (avec `except B` en premier), il aurait affiché B, B, B — la première clause `except` qui correspond est déclenchée.

Quand une exception intervient, une valeur peut lui être associée, que l'on appelle *l'argument* de l'exception. La présence de cet argument et son type dépendent du type de l'exception.

La clause `except` peut spécifier un nom de variable après le nom de l'exception. Cette variable est liée à l'instance d'exception avec les arguments stockés dans l'attribut `args`. Pour plus de commodité, l'instance de l'exception définit la méthode `__str__()` afin que les arguments puissent être affichés directement sans avoir à référencer `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

La sortie produite par `__str__()` de l'exception est affichée en dernière partie (« détail ») du message des exceptions qui ne sont pas gérées.

`BaseException` est la classe mère de toutes les exceptions. Une de ses sous-classes, `Exception`, est la classe mère de toutes les exceptions non fatales. Les exceptions qui ne sont pas des sous-classes de `Exception` ne sont normalement

pas gérées, car elles sont utilisées pour indiquer que le programme doit se terminer. C'est le cas de `SystemExit` qui est levée par `sys.exit()` et `KeyboardInterrupt` qui est levée quand l'utilisateur souhaite interrompre le programme.

`Exception` peut être utilisée pour intercepter (presque) tous les cas. Cependant, une bonne pratique consiste à être aussi précis que possible dans les types d'exception que l'on souhaite gérer et autoriser toutes les exceptions non prévues à se propager.

La manière la plus utilisée pour gérer une `Exception` consiste à afficher ou journaliser l'exception et ensuite la lever à nouveau afin de permettre à l'appelant de la gérer à son tour :

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

L'instruction `try` … `except` accepte également une clause `else` optionnelle qui, lorsqu'elle est présente, doit se placer après toutes les clauses `except`. Elle est utile pour du code qui doit être exécuté lorsqu'aucune exception n'a été levée par la clause `try`. Par exemple :

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Il vaut mieux utiliser la clause `else` plutôt que d'ajouter du code à la clause `try` car cela évite de capturer accidentellement une exception qui n'a pas été levée par le code initialement protégé par l'instruction `try` … `except`.

Les gestionnaires d'exceptions n'interceptent pas que les exceptions qui sont levées immédiatement dans leur clause `try`, mais aussi celles qui sont levées au sein de fonctions appelées (parfois indirectement) dans la clause `try`. Par exemple :

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Déclencher des exceptions

L'instruction `raise` permet au programmeur de déclencher une exception spécifique. Par exemple :

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Le seul argument à `raise` indique l'exception à déclencher. Cela peut être soit une instance d'exception, soit une classe d'exception (une classe dérivée de `BaseException`, telle que `Exception` ou une de ses sous-classes). Si une classe est donnée, elle est implicitement instanciée *via* l'appel de son constructeur, sans argument :

```
raise ValueError  # shorthand for 'raise ValueError()'
```

Si vous avez besoin de savoir si une exception a été levée mais que vous n'avez pas intention de la gérer, une forme plus simple de l'instruction `raise` permet de propager l'exception :

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5 Chaînage d'exceptions

Si une exception non gérée se produit à l'intérieur d'une section `except`, l'exception en cours de traitement est jointe à l'exception non gérée et incluse dans le message d'erreur :

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

Pour indiquer qu'une exception est la conséquence directe d'une autre, l'instruction `raise` autorise une clause facultative `from` :

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Cela peut être utile lorsque vous transformez des exceptions. Par exemple :

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

Cela permet également de désactiver le chaînage automatique des exceptions à l'aide de l'idiome `from None` :

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Pour plus d'informations sur les mécanismes de chaînage, voir bltin-exceptions.

8.6 Exceptions définies par l'utilisateur

Les programmes peuvent nommer leurs propres exceptions en créant une nouvelle classe d'exception (voir *Classes* pour en savoir plus sur les classes de Python). Les exceptions sont typiquement dérivées de la classe `Exception`, directement ou non.

Les classes d'exceptions sont des classes comme les autres, et peuvent donc utiliser toutes les fonctionnalités des classes. Néanmoins, en général, elles demeurent assez simples, et se contentent d'offrir des attributs qui permettent aux gestionnaires de ces exceptions d'extraire les informations relatives à l'erreur qui s'est produite.

La plupart des exceptions sont définies avec des noms qui se terminent par "Error", comme les exceptions standards.

Beaucoup de modules standards définissent leurs propres exceptions pour signaler les erreurs possibles dans les fonctions qu'ils définissent.

8.7 Définition d'actions de nettoyage

L'instruction `try` a une autre clause optionnelle qui est destinée à définir des actions de nettoyage devant être exécutées dans certaines circonstances. Par exemple :

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Si la clause `finally` est présente, la clause `finally` est la dernière tâche exécutée avant la fin du bloc `try`. La clause `finally` se lance que le bloc `try` produise une exception ou non. Les prochains points parlent de cas plus complexes lorsqu'une exception apparait :

- Si une exception se produit durant l'exécution de la clause `try`, elle peut être récupérée par une clause `except`. Si l'exception n'est pas récupérée par une clause `except`, l'exception est levée à nouveau après que la clause `finally` a été exécutée.
- Une exception peut se produire durant l'exécution d'une clause `except` ou `else`. Encore une fois, l'exception est reprise après que la clause `finally` a été exécutée.
- Si dans l'exécution d'un bloc `finally`, on atteint une instruction `break`, `continue` ou `return`, alors les exceptions ne sont pas reprises.
- Si dans l'exécution d'un bloc `try`, on atteint une instruction `break`, `continue` ou `return`, alors la clause `finally` s'exécute juste avant l'exécution de `break`, `continue` ou `return`.
- Si la clause `finally` contient une instruction `return`, la valeur retournée sera celle du `return` de la clause `finally`, et non la valeur du `return` de la clause `try`.

Par exemple :

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Un exemple plus compliqué :

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
```

(suite sur la page suivante)

(suite de la page précédente)

```
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Comme vous pouvez le voir, la clause `finally` est exécutée dans tous les cas. L'exception de type `TypeError`, déclenchée en divisant deux chaînes de caractères, n'est pas prise en charge par la clause `except` et est donc propagée après que la clause `finally` a été exécutée.

Dans les vraies applications, la clause `finally` est notamment utile pour libérer des ressources externes (telles que des fichiers ou des connexions réseau), quelle qu'ait été l'utilisation de ces ressources.

8.8 Actions de nettoyage prédéfinies

Certains objets définissent des actions de nettoyage standards qui doivent être exécutées lorsque l'objet n'est plus nécessaire, indépendamment du fait que l'opération ayant utilisé l'objet ait réussi ou non. Regardez l'exemple suivant, qui tente d'ouvrir un fichier et d'afficher son contenu à l'écran

```
for line in open("myfile.txt"):
    print(line, end="")
```

Le problème avec ce code est qu'il laisse le fichier ouvert pendant une durée indéterminée après que le code a fini de s'exécuter. Ce n'est pas un problème avec des scripts simples, mais peut l'être au sein d'applications plus conséquentes. L'instruction `with` permet d'utiliser certains objets comme des fichiers d'une façon qui assure qu'ils seront toujours nettoyés rapidement et correctement.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Après l'exécution du bloc, le fichier *f* est toujours fermé, même si un problème est survenu pendant l'exécution de ces lignes. D'autres objets qui, comme pour les fichiers, fournissent des actions de nettoyage prédéfinies l'indiquent dans leur documentation.

8.9 Levée et gestion de multiples exceptions non corrélées

Il existe des situations où il est nécessaire de signaler plusieurs exceptions qui se sont produites. C'est souvent le cas dans les programmes à multiples fils, lorsque plusieurs tâches échouent en parallèle. Mais il existe également d'autres cas où il est souhaitable de poursuivre l'exécution, de collecter plusieurs erreurs plutôt que de lever la première exception rencontrée.

L'idiome natif `ExceptionGroup` englobe une liste d'instances d'exceptions afin de pouvoir les lever en même temps. C'est une exception, et peut donc être interceptée comme toute autre exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |   File "<stdin>", line 3, in f
  | ExceptionGroup: there were problems
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

En utilisant `except*` au lieu de `except`, vous pouvez choisir de ne gérer que les exceptions du groupe qui correspondent à un certain type. Dans l'exemple qui suit, dans lequel se trouve imbriqué un groupe d'exceptions, chaque clause `except*` extrait du groupe des exceptions d'un certain type tout en laissant toutes les autres exceptions se propager vers d'autres clauses et éventuellement être réactivées.

```
>>> def f():
...     raise ExceptionGroup("group1",
...                          [OSError(1),
...                           SystemError(2),
...                           ExceptionGroup("group2",
...                                          [OSError(3), RecursionError(4)])])
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in f
  | ExceptionGroup: group1
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Notez que les exceptions imbriquées dans un groupe d'exceptions doivent être des instances, pas des types. En effet, dans la pratique, les exceptions sont normalement celles qui ont déjà été déclenchées et interceptées par le programme, en utilisant le modèle suivant :

```
>>> excs = []
... for test in tests:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10 Enrichissement des exceptions avec des notes

Quand une exception est créée pour être levée, elle est généralement initialisée avec des informations décrivant l'erreur qui s'est produite. Il existe des cas où il est utile d'ajouter des informations après que l'exception a été interceptée. Dans ce but, les exceptions ont une méthode `add_note(note)` qui reçoit une chaîne et l'ajoute à la liste des notes de l'exception. L'affichage de la pile de trace standard inclut toutes les notes, dans l'ordre dans lequel elles ont été ajoutées, après l'exception.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
>>>
```

Par exemple, lors de la collecte d'exceptions dans un groupe d'exceptions, il est probable que vous souhaitiez ajouter des informations de contexte aux erreurs individuelles. Dans ce qui suit, chaque exception du groupe est accompagnée d'une note indiquant quand cette erreur s'est produite.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |   File "<stdin>", line 2, in f
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

CHAPITRE 9

Classes

Les classes sont un moyen de réunir des données et des fonctionnalités. Créer une nouvelle classe crée un nouveau *type* d'objet et ainsi de nouvelles *instances* de ce type peuvent être construites. Chaque instance peut avoir ses propres attributs, ce qui définit son état. Une instance peut aussi avoir des méthodes (définies par la classe de l'instance) pour modifier son état.

La notion de classes en Python s'inscrit dans le langage avec un minimum de syntaxe et de sémantique nouvelles. C'est un mélange des mécanismes rencontrés dans C++ et Modula-3. Les classes fournissent toutes les fonctionnalités standards de la programmation orientée objet : l'héritage de classes autorise les héritages multiples, une classe dérivée peut surcharger les méthodes de sa ou ses classes mères et une méthode peut appeler la méthode d'une classe mère qui possède le même nom. Les objets peuvent contenir n'importe quel nombre ou type de données. De la même manière que les modules, les classes participent à la nature dynamique de Python : elles sont créées pendant l'exécution et peuvent être modifiées après leur création.

Dans la terminologie C++, les membres des classes (y compris les données) sont *publics* (sauf exception, voir *Variables privées*) et toutes les fonctions membres sont *virtuelles*. Comme avec Modula-3, il n'y a aucune façon d'accéder aux membres d'un objet à partir de ses méthodes : une méthode est déclarée avec un premier argument explicite représentant l'objet et cet argument est transmis de manière implicite lors de l'appel. Comme avec Smalltalk, les classes elles-mêmes sont des objets. Il existe ainsi une sémantique pour les importer et les renommer. Au contraire de C++ et Modula-3, les types natifs peuvent être utilisés comme classes mères pour être étendus par l'utilisateur. Enfin, comme en C++, la plupart des opérateurs natifs avec une syntaxe spéciale (opérateurs arithmétiques, indiçage, etc.) peuvent être redéfinis pour les instances de classes.

En l'absence d'une terminologie communément admise pour parler des classes, nous utilisons parfois des termes de Smalltalk et C++. Nous voulions utiliser les termes de Modula-3 puisque sa sémantique orientée objet est plus proche de celle de Python que C++, mais il est probable que seul un petit nombre de lecteurs les connaissent.

9.1 Objets et noms : préambule

Les objets possèdent une existence propre et plusieurs noms peuvent être utilisés (dans divers contextes) pour faire référence à un même objet. Ce concept est connu sous le nom d'alias dans d'autres langages. Il n'apparaît pas au premier coup d'œil en Python et il peut être ignoré tant qu'on travaille avec des types de base immuables (nombres, chaînes, *n*-uplets). Cependant, les alias peuvent produire des effets surprenants sur la sémantique d'un code Python mettant en jeu des objets muables comme les listes, les dictionnaires et la plupart des autres types. En général, leur utilisation est bénéfique au programme car les alias se comportent, d'un certain point de vue, comme des pointeurs. Par exemple, transmettre un objet n'a aucun coût car c'est simplement un pointeur qui est transmis par l'implémentation ; et si une fonction modifie un objet passé en argument, le code à l'origine de l'appel voit le changement. Ceci élimine le besoin d'avoir deux mécanismes de transmission d'arguments comme en Pascal.

9.2 Portées et espaces de nommage en Python

Avant de présenter les classes, nous devons aborder la notion de portée en Python. Les définitions de classes font d'habiles manipulations avec les espaces de nommage, vous devez donc savoir comment les portées et les espaces de nommage fonctionnent. Soit dit en passant, la connaissance de ce sujet est aussi utile aux développeurs Python expérimentés.

Commençons par quelques définitions.

Un *espace de nommage* est une table de correspondance entre des noms et des objets. La plupart des espaces de nommage sont actuellement implémentés sous forme de dictionnaires Python, mais ceci n'est normalement pas visible (sauf pour les performances) et peut changer dans le futur. Comme exemples d'espaces de nommage, nous pouvons citer les primitives (fonctions comme `abs()` et les noms des exceptions de base) ; les noms globaux dans un module ; et les noms locaux lors d'un appel de fonction. D'une certaine manière, l'ensemble des attributs d'un objet forme lui-même un espace de nommage. L'important à retenir concernant les espaces de nommage est qu'il n'y a absolument aucun lien entre les noms de différents espaces de nommage ; par exemple, deux modules différents peuvent définir une fonction `maximize` sans qu'il n'y ait de confusion. Les utilisateurs des modules doivent préfixer le nom de la fonction avec celui du module.

À ce propos, nous utilisons le mot *attribut* pour tout nom suivant un point. Par exemple, dans l'expression `z.real`, `real` est un attribut de l'objet `z`. Rigoureusement parlant, les références à des noms dans des modules sont des références d'attributs : dans l'expression `nommodule.nomfonction`, `nommodule` est un objet module et `nomfonction` est un attribut de cet objet. Dans ces conditions, il existe une correspondance directe entre les attributs du module et les noms globaux définis dans le module : ils partagent le même espace de nommage [1] !

Les attributs peuvent être en lecture seule ou modifiables. S'ils sont modifiables, l'affectation à un attribut est possible. Les attributs de modules sont modifiables : vous pouvez écrire `nommodule.la_reponse = 42`. Les attributs modifiables peuvent aussi être effacés avec l'instruction `del`. Par exemple, `del nommodule.la_reponse` supprime l'attribut `la_reponse` de l'objet nommé `nommodule`.

Les espaces de nommage sont créés à différents moments et ont différentes durées de vie. L'espace de nommage contenant les primitives est créé au démarrage de l'interpréteur Python et n'est jamais effacé. L'espace de nommage globaux pour un module est créé lorsque la définition du module est lue. Habituellement, les espaces de nommage des modules durent aussi jusqu'à l'arrêt de l'interpréteur. Les instructions exécutées par la première invocation de l'interpréteur, qu'elles soient lues depuis un fichier de script ou de manière interactive, sont considérées comme faisant partie d'un module appelé `__main__`, de façon qu'elles possèdent leur propre espace de nommage (les primitives vivent elles-mêmes dans un module, appelé `builtins`).

L'espace des noms locaux d'une fonction est créé lors de son appel, puis effacé lorsqu'elle renvoie un résultat ou lève une exception non prise en charge (en fait, « oublié » serait une meilleure façon de décrire ce qui se passe réellement). Bien sûr, des invocations récursives ont chacune leur propre espace de nommage.

1. Il existe une exception : les modules disposent d'un attribut secret en lecture seule appelé `__dict__` qui renvoie le dictionnaire utilisé pour implémenter l'espace de nommage du module ; le nom `__dict__` est un attribut mais pas un nom global. Évidemment, si vous l'utilisez, vous brisez l'abstraction de l'implémentation des espaces de nommage. Il est donc réservé à des choses comme les débogueurs post-mortem.

La *portée* est la zone textuelle d'un programme Python où un espace de nommage est directement accessible. « Directement accessible » signifie ici qu'une référence non qualifiée à un nom est cherchée dans l'espace de nommage.

Bien que les portées soient déterminées de manière statique, elles sont utilisées de manière dynamique. À n'importe quel moment de l'exécution, il y a au minimum trois ou quatre portées imbriquées dont les espaces de nommage sont directement accessibles :

- la portée la plus au centre, celle qui est consultée en premier, contient les noms locaux ;
- les portées des fonctions englobantes, qui sont consultées en commençant avec la portée englobante la plus proche, contiennent des noms non-locaux mais aussi non-globaux ;
- l'avant-dernière portée contient les noms globaux du module courant ;
- la portée englobante, consultée en dernier, est l'espace de nommage contenant les primitives.

Si un nom est déclaré comme global, alors toutes les références et affectations vont directement dans l'avant-dernière portée contenant les noms globaux du module. Pour pointer une variable qui se trouve en dehors de la portée la plus locale, vous pouvez utiliser l'instruction `nonlocal`. Si une telle variable n'est pas déclarée *nonlocal*, elle est en lecture seule (toute tentative de la modifier crée simplement une *nouvelle* variable dans la portée la plus locale, en laissant inchangée la variable du même nom dans sa portée d'origine).

Habituellement, la portée locale référence les noms locaux de la fonction courante. En dehors des fonctions, la portée locale référence le même espace de nommage que la portée globale : l'espace de nommage du module. Les définitions de classes créent un nouvel espace de nommage dans la portée locale.

Il est important de réaliser que les portées sont déterminées de manière textuelle : la portée globale d'une fonction définie dans un module est l'espace de nommage de ce module, quelle que soit la provenance de l'appel à la fonction. En revanche, la recherche réelle des noms est faite dynamiquement au moment de l'exécution. Cependant la définition du langage est en train d'évoluer vers une résolution statique des noms au moment de la « compilation », donc ne vous basez pas sur une résolution dynamique (en réalité, les variables locales sont déjà déterminées de manière statique) !

Une particularité de Python est que, si aucune instruction `global` ou `nonlocal` n'est active, les affectations de noms vont toujours dans la portée la plus proche. Les affectations ne copient aucune donnée : elles se contentent de lier des noms à des objets. Ceci est également vrai pour l'effacement : l'instruction `del x` supprime la liaison de `x` dans l'espace de nommage référencé par la portée locale. En réalité, toutes les opérations qui impliquent des nouveaux noms utilisent la portée locale : en particulier, les instructions `import` et les définitions de fonctions effectuent une liaison du module ou du nom de fonction dans la portée locale.

L'instruction `global` peut être utilisée pour indiquer que certaines variables existent dans la portée globale et doivent être reliées en local ; l'instruction `nonlocal` indique que certaines variables existent dans une portée supérieure et doivent être reliées en local.

9.2.1 Exemple de portées et d'espaces de nommage

Ceci est un exemple montrant comment utiliser les différentes portées et espaces de nommage, et comment `global` et `nonlocal` modifient l'affectation de variable :

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
```

(suite sur la page suivante)

(suite de la page précédente)

```
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Ce code donne le résultat suivant :

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Vous pouvez constater que l'affectation *locale* (qui est effectuée par défaut) n'a pas modifié la liaison de *spam* dans *scope_test*. L'affectation `nonlocal` a changé la liaison de *spam* dans *scope_test* et l'affectation `global` a changé la liaison au niveau du module.

Vous pouvez également voir qu'aucune liaison pour *spam* n'a été faite avant l'affectation `global`.

9.3 Une première approche des classes

Le concept de classe introduit un peu de syntaxe nouvelle, trois nouveaux types d'objets ainsi que quelques nouveaux éléments de sémantique.

9.3.1 Syntaxe de définition des classes

La forme la plus simple de définition d'une classe est la suivante :

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Les définitions de classes, comme les définitions de fonctions (définitions `def`), doivent être exécutées avant d'avoir un effet. Vous pouvez tout à fait placer une définition de classe dans une branche d'une instruction conditionnelle `if` ou encore à l'intérieur d'une fonction.

Dans la pratique, les déclarations dans une définition de classe sont généralement des définitions de fonctions mais d'autres déclarations sont permises et parfois utiles (nous revenons sur ce point plus tard). Les définitions de fonction à l'intérieur d'une classe ont normalement une forme particulière de liste d'arguments, dictée par les conventions d'appel aux méthodes (à nouveau, tout ceci est expliqué plus loin).

Quand une classe est définie, un nouvel espace de nommage est créé et utilisé comme portée locale --- Ainsi, toutes les affectations de variables locales entrent dans ce nouvel espace de nommage. En particulier, les définitions de fonctions y lient le nom de la nouvelle fonction.

À la fin de la définition d'une classe, un *objet classe* est créé. C'est, pour simplifier, une encapsulation du contenu de l'espace de nommage créé par la définition de classe. Nous revoyons les objets classes dans la prochaine section. La portée

locale initiale (celle qui prévaut avant le début de la définition de la classe) est ré-instanciée et l'objet de classe est lié ici au nom de classe donné dans l'en-tête de définition de classe (`ClassName` dans l'exemple).

9.3.2 Objets classes

Les objets classes prennent en charge deux types d'opérations : des références à des attributs et l'instanciation.

Les *références d'attributs* utilisent la syntaxe standard utilisée pour toutes les références d'attributs en Python : `obj.nom`. Les noms d'attribut valides sont tous les noms qui se trouvaient dans l'espace de nommage de la classe quand l'objet classe a été créé. Donc, si la définition de classe est de cette forme :

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

alors `MyClass.i` et `MyClass.f` sont des références valides à des attributs, renvoyant respectivement un entier et un objet fonction. Les attributs de classes peuvent également être affectés, de sorte que vous pouvez modifier la valeur de `MyClass.i` par affectation. `__doc__` est aussi un attribut valide, renvoyant la *docstring* appartenant à la classe : `"A simple example class"`.

L'*instanciation* de classes utilise la notation des fonctions. Considérez simplement que l'objet classe est une fonction sans paramètre qui renvoie une nouvelle instance de la classe. Par exemple (en considérant la classe définie ci-dessus) :

```
x = MyClass()
```

crée une nouvelle *instance* de la classe et affecte cet objet à la variable locale `x`.

L'opération d'instanciation (en "appelant" un objet classe) crée un objet vide. De nombreuses classes aiment créer des instances personnalisées correspondant à un état initial spécifique. À cet effet, une classe peut définir une méthode spéciale nommée `__init__()`, comme ceci :

```
def __init__(self):
    self.data = []
```

Quand une classe définit une méthode `__init__()`, l'instanciation de la classe appelle automatiquement `__init__()` pour la nouvelle instance de la classe. Donc, dans cet exemple, l'initialisation d'une nouvelle instance peut être obtenue par :

```
x = MyClass()
```

Bien sûr, la méthode `__init__()` peut avoir des arguments pour une plus grande flexibilité. Dans ce cas, les arguments donnés à l'opérateur d'instanciation de classe sont transmis à `__init__()`. Par exemple

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objets instances

Maintenant, que pouvons-nous faire avec des objets instances ? Les seules opérations comprises par les objets instances sont des références d'attributs. Il y a deux sortes de noms d'attributs valides, les attributs 'données' et les méthodes.

Les *attributs 'données'* correspondent à des "variables d'instance" en Smalltalk et aux "membres de données" en C++. Les attributs 'données' n'ont pas à être déclarés. Comme les variables locales, ils existent dès lors qu'ils sont assignés une première fois. Par exemple, si `x` est l'instance de `MyClass` créée ci-dessus, le code suivant affiche la valeur `16`, sans laisser de trace :

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

L'autre type de référence à un attribut d'instance est une *méthode*. Une méthode est une fonction qui "appartient à" un objet (en Python, le terme de méthode n'est pas unique aux instances de classes : d'autres types d'objets peuvent aussi avoir des méthodes. Par exemple, les objets listes ont des méthodes appelées `append`, `insert`, `remove`, `sort` et ainsi de suite. Toutefois, dans la discussion qui suit, sauf indication contraire, nous utilisons le terme de méthode exclusivement en référence à des méthodes d'objets instances de classe).

Les noms de méthodes valides d'un objet instance dépendent de sa classe. Par définition, tous les attributs d'une classe qui sont des objets fonctions définissent les méthodes correspondantes de ses instances. Donc, dans notre exemple, `x.f` est une référence valide à une méthode car `MyClass.f` est une fonction, mais pas `x.i` car `MyClass.i` n'en est pas une. Attention cependant, `x.f` n'est pas la même chose que `MyClass.f` --- Il s'agit d'un *objet méthode*, pas d'un objet fonction.

9.3.4 Objets méthode

Le plus souvent, une méthode est appelée juste après avoir été liée :

```
x.f()
```

Dans l'exemple de la classe `MyClass`, cela renvoie la chaîne de caractères `hello world`. Toutefois, il n'est pas nécessaire d'appeler la méthode directement : `x.f` est un objet méthode, il peut être gardé de côté et être appelé plus tard. Par exemple :

```
xf = x.f
while True:
    print(xf())
```

affiche `hello world` jusqu'à la fin des temps.

Que se passe-t-il exactement quand une méthode est appelée ? Vous avez dû remarquer que `x.f()` a été appelée dans le code ci-dessus sans argument, alors que la définition de la méthode `f()` spécifie bien qu'elle prend un argument. Qu'est-il arrivé à l'argument ? Python doit sûrement lever une exception lorsqu'une fonction qui requiert un argument est appelée sans -- même si l'argument n'est pas utilisé…

En fait, vous avez peut-être deviné la réponse : la particularité des méthodes est que l'objet est passé comme premier argument de la fonction. Dans notre exemple, l'appel `x.f()` est exactement équivalent à `MyClass.f(x)`. En général, appeler une méthode avec une liste de *n* arguments est équivalent à appeler la fonction correspondante avec une liste d'arguments créée en ajoutant l'instance de l'objet de la méthode avant le premier argument.

Si vous ne comprenez toujours pas comment les méthodes fonctionnent, un coup d'œil à l'implémentation vous aidera peut-être. Lorsque un attribut d'une instance est référencé et que ce n'est pas un attribut 'données', sa classe est recherchée. Si le nom correspond à un attribut valide et que c'est un objet fonction, un objet méthode est créé en générant un objet

abstrait qui regroupe (des pointeurs vers) l'objet instance et l'objet fonction qui vient d'être trouvé : c'est l'objet méthode. Quand l'objet méthode est appelé avec une liste d'arguments, une nouvelle liste d'arguments est construite à partir de l'objet instance et de la liste des arguments. L'objet fonction est alors appelé avec cette nouvelle liste d'arguments.

9.3.5 Classes et variables d'instance

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe :

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Comme nous l'avons vu dans *Objets et noms : préambule*, les données partagées *muable* (telles que les listes, dictionnaires, etc.) peuvent avoir des effets surprenants. Par exemple, la liste *tricks* dans le code suivant ne devrait pas être utilisée en tant que variable de classe car, dans ce cas, une seule liste est partagée par toutes les instances de *Dog* :

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Une conception correcte de la classe est d'utiliser une variable d'instance à la place :

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Remarques diverses

Si le même nom d'attribut apparaît à la fois dans une instance et dans une classe, alors la recherche d'attribut donne la priorité à l'instance :

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Les attributs 'données' peuvent être référencés par des méthodes comme par des utilisateurs ordinaires ("clients") d'un objet. En d'autres termes, les classes ne sont pas utilisables pour implémenter des types de données purement abstraits. En fait, il n'est pas possible en Python d'imposer de masquer des données — tout est basé sur des conventions (d'un autre côté, l'implémentation de Python, écrite en C, peut complètement masquer les détails d'implémentation et contrôler l'accès à un objet si nécessaire ; ceci peut être utilisé par des extensions de Python écrites en C).

Les clients doivent utiliser les attributs 'données' avec précaution --- ils pourraient mettre le désordre dans les invariants gérés par les méthodes avec leurs propres valeurs d'attributs. Remarquez que les clients peuvent ajouter leurs propres attributs 'données' à une instance d'objet sans altérer la validité des méthodes, pour autant que les noms n'entrent pas en conflit --- là aussi, adopter une convention de nommage peut éviter bien des problèmes.

Il n'y a pas de notation abrégée pour référencer des attributs 'données' (ou les autres méthodes !) depuis les méthodes. Nous pensons que ceci améliore en fait la lisibilité des méthodes : il n'y a aucune chance de confondre variables locales et variables d'instances quand on regarde le code d'une méthode.

Souvent, le premier argument d'une méthode est nommé `self`. Ce n'est qu'une convention : le nom `self` n'a aucune signification particulière en Python. Notez cependant que si vous ne suivez pas cette convention, votre code risque d'être moins lisible pour d'autres programmeurs Python et il est aussi possible qu'un programme qui fasse l'introspection de classes repose sur une telle convention.

Tout objet fonction qui est un attribut de classe définit une méthode pour des instances de cette classe. Il n'est pas nécessaire que le texte de définition de la fonction soit dans la définition de la classe : il est possible d'affecter un objet fonction à une variable locale de la classe. Par exemple :

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Maintenant, `f`, `g` et `h` sont toutes des attributs de la classe `C` et font référence à des fonctions objets. Par conséquent, ce sont toutes des méthodes des instances de `C` --- `h` est exactement identique à `g`. Remarquez qu'en pratique, ceci ne sert qu'à embrouiller le lecteur d'un programme.

Les méthodes peuvent appeler d'autres méthodes en utilisant des méthodes qui sont des attributs de l'argument `self` :

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Les méthodes peuvent faire référence à des noms globaux de la même manière que les fonctions. La portée globale associée à une méthode est le module contenant la définition de la classe (la classe elle-même n'est jamais utilisée en tant que portée globale). Alors qu'il est rare d'avoir une bonne raison d'utiliser des données globales dans une méthode, il y a de nombreuses utilisations légitimes de la portée globale : par exemple, les fonctions et modules importés dans une portée globale peuvent être utilisés par des méthodes, de même que les fonctions et classes définies dans cette même portée. Habituellement, la classe contenant la méthode est elle-même définie dans cette portée globale et, dans la section suivante, nous verrons de bonnes raisons pour qu'une méthode référence sa propre classe.

Toute valeur est un objet et a donc une *classe* (appelée aussi son *type*). Elle est stockée dans `objet.__class__`.

9.5 Héritage

Bien sûr, ce terme de "classe" ne serait pas utilisé s'il n'y avait pas d'héritage. La syntaxe pour définir une sous-classe est de cette forme :

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Le nom `BaseClassName` doit être défini dans une portée contenant la définition de la classe dérivée. À la place du nom d'une classe mère, une expression est aussi autorisée. Ceci peut être utile, par exemple, lorsque la classe est définie dans un autre module :

```
class DerivedClassName(modname.BaseClassName):
```

L'exécution d'une définition de classe dérivée se déroule comme pour une classe mère. Quand l'objet de la classe est construit, la classe mère est mémorisée. Elle est utilisée pour la résolution des références d'attributs : si un attribut n'est pas trouvé dans la classe, la recherche se poursuit en regardant dans la classe mère. Cette règle est appliquée récursivement si la classe mère est elle-même dérivée d'une autre classe.

Il n'y a rien de particulier dans l'instanciation des classes dérivées : `DerivedClassName()` crée une nouvelle instance de la classe. Les références aux méthodes sont résolues comme suit : l'attribut correspondant de la classe est recherché, en remontant la hiérarchie des classes mères si nécessaire, et la référence de méthode est valide si cela conduit à une fonction.

Les classes dérivées peuvent surcharger des méthodes de leurs classes mères. Comme les méthodes n'ont aucun privilège particulier quand elles appellent d'autres méthodes d'un même objet, une méthode d'une classe mère qui appelle une autre méthode définie dans la même classe peut en fait appeler une méthode d'une classe dérivée qui la surcharge (pour les programmeurs C++ : toutes les méthodes de Python sont en effet "virtuelles").

Une méthode dans une classe dérivée peut aussi, en fait, vouloir étendre plutôt que simplement remplacer la méthode du même nom de sa classe mère. L'appel direct à la méthode de la classe mère s'écrit simplement `BaseClassName.nomMethode(self, arguments)`. C'est parfois utile également aux clients (notez bien que ceci ne fonctionne que si la classe mère est accessible en tant que `BaseClassName` dans la portée globale).

Python définit deux fonctions primitives pour gérer l'héritage :

- utilisez `isinstance()` pour tester le type d'une instance : `isinstance(obj, int)` renvoie `True` seulement si `obj.__class__` est égal à `int` ou à une autre classe dérivée de `int` ;
- utilisez `issubclass()` pour tester l'héritage d'une classe : `issubclass(bool, int)` renvoie `True` car la classe `bool` est une sous-classe de `int`. Cependant, `issubclass(float, int)` renvoie `False` car `float` n'est pas une sous-classe de `int`.

9.5.1 Héritage multiple

Python gère également une forme d'héritage multiple. Une définition de classe ayant plusieurs classes mères est de cette forme :

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Dans la plupart des cas, vous pouvez vous représenter la recherche d'attributs dans les classes parentes comme étant : le plus profond d'abord, de gauche à droite, sans chercher deux fois dans la même classe si elle apparaît plusieurs fois dans la hiérarchie. Ainsi, si un attribut n'est pas trouvé dans `DerivedClassName`, il est recherché dans `Base1`, puis (récursivement) dans les classes mères de `Base1` ; s'il n'y est pas trouvé, il est recherché dans `Base2` et ses classes mères, et ainsi de suite.

Dans les faits, c'est un peu plus complexe que ça ; l'ordre de la recherche (*method resolution order, ou MRO* en anglais) change dynamiquement pour gérer des appels coopératifs à `super()`. Cette approche est connue sous le nom de la "appel de la méthode la plus proche" (*call-next-method* en anglais) dans d'autres langages avec héritage multiple. Elle est plus puissante que le simple appel à super que l'on trouve dans les langages à héritage simple.

L'ordre défini dynamiquement est nécessaire car tous les cas d'héritage multiple comportent une ou plusieurs relations en losange (où au moins une classe peut être accédée à partir de plusieurs chemins en partant de la classe la plus basse). Par exemple, puisque toutes les classes héritent de `object`, tout héritage multiple ouvre plusieurs chemins pour atteindre `object`. Pour qu'une classe mère ne soit pas appelée plusieurs fois, l'algorithme dynamique linéarise l'ordre de recherche d'une façon qui préserve l'ordre d'héritage, de la gauche vers la droite, spécifié dans chaque classe, qui appelle chaque

classe parente une seule fois, qui est monotone (ce qui signifie qu'une classe peut être sous-classée sans affecter l'ordre d'héritage de ses parents). Prises ensemble, ces propriétés permettent de concevoir des classes de façon fiable et extensible dans un contexte d'héritage multiple. Pour plus de détails, consultez http://www.python.org/download/releases/2.3/mro/.

9.6 Variables privées

Les membres "privés", qui ne peuvent être accédés que depuis l'intérieur d'un objet, n'existent pas en Python. Toutefois, il existe une convention respectée par la majorité du code Python : un nom préfixé par un tiret bas (comme `_spam`) doit être considéré comme une partie non publique de l'API (qu'il s'agisse d'une fonction, d'une méthode ou d'un attribut 'données'). Il doit être vu comme un détail d'implémentation pouvant faire l'objet de modifications futures sans préavis.

Dès lors qu'il y a un cas d'utilisation valable pour avoir des attributs privés aux classes (notamment pour éviter des conflits avec des noms définis dans des sous-classes), il existe un support (certes limité) pour un tel mécanisme, appelé *name mangling*. Tout identifiant de la forme `__spam` (avec au moins deux tirets bas en tête et au plus un à la fin) est remplacé textuellement par `_classname__spam`, où `classname` est le nom de la classe sans le ou les premiers tirets-bas. Ce "découpage" est effectué sans tenir compte de la position syntaxique de l'identifiant, tant qu'il est présent dans la définition d'une classe.

Ce changement de nom est utile pour permettre à des sous-classes de surcharger des méthodes sans casser les appels de méthodes à l'intérieur d'une classe. Par exemple :

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

L'exemple si dessus fonctionnerait même si `MappingSubclass` introduisait un identifieur `__update` puisqu'il a été remplacé avec `_Mapping__update` dans la classe `Mapping` et `_MappingSubclass__update` dans la classe `MappingSubclass` respectivement.

Notez que ces règles sont conçues avant tout pour éviter les accidents ; il reste possible d'accéder ou de modifier une variable considérée comme privée. Ceci peut même être utile dans certaines circonstances, comme au sein du débogueur.

Remarquez que le code que vous passez à `exec()`, `eval()` ne considère pas le nom de la classe appelante comme étant la classe courante ; le même effet s'applique à la directive `global` dont l'effet est, de la même façon, restreint au code compilé dans le même ensemble de byte-code. Les mêmes restrictions s'appliquent à `getattr()`, `setattr()` et `delattr()`, ainsi qu'aux références directes à `__dict__`.

9.7 Trucs et astuces

Il est parfois utile d'avoir un type de donnée similaire au *record* du Pascal ou au *struct* du C, qui regroupent ensemble quelques attributs « données » nommés. L'approche idiomatique correspondante en Python est d'utiliser des `dataclasses` :

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

À du code Python qui s'attend à recevoir un type de donnée abstrait spécifique, on peut souvent fournir une classe qui simule les méthodes de ce type. Par exemple, à une fonction qui formate des données extraites d'un objet fichier, vous pouvez lui passer comme argument une instance d'une classe qui implémente les méthodes `read()` et `readline()` en puisant ses données à partir d'un tampon de chaînes de caractères.

Les objets méthodes d'instances ont aussi des attributs : `m.__self__` est l'instance d'objet avec la méthode `m()` et `m.__func__` est l'objet fonction correspondant à la méthode.

9.8 Itérateurs

Vous avez maintenant certainement remarqué que l'on peut itérer sur la plupart des objets conteneurs en utilisant une instruction `for` :

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Ce style est simple, concis et pratique. L'utilisation d'itérateurs imprègne et unifie Python. En arrière plan, l'instruction `for` appelle la fonction `iter()` sur l'objet conteneur. Cette fonction renvoie un objet itérateur qui définit la méthode `__next__()`, laquelle accède aux éléments du conteneur un par un. Lorsqu'il n'y a plus d'élément, `__next__()` lève une exception `StopIteration` qui indique à la boucle de l'instruction `for` de se terminer. Vous pouvez appeler la méthode `__next__()` en utilisant la fonction native `next()`. Cet exemple montre comment tout cela fonctionne :

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Une fois compris les mécanismes de gestion des itérateurs, il est simple d'ajouter ce comportement à vos classes. Définissez une méthode `__iter__()` qui renvoie un objet disposant d'une méthode `__next__()`. Si la classe définit elle-même la méthode `__next__()`, alors `__iter__()` peut simplement renvoyer `self` :

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Générateurs

Les *générateurs* sont des outils simples et puissants pour créer des itérateurs. Ils sont écrits comme des fonctions classiques mais utilisent l'instruction `yield` lorsqu'ils veulent renvoyer des données. À chaque fois qu'il est appelé par `next()`, le générateur reprend son exécution là où il s'était arrêté (en conservant tout son contexte d'exécution). Un exemple montre très bien combien les générateurs sont simples à créer :

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Tout ce qui peut être fait avec des générateurs peut également être fait avec des itérateurs basés sur des classes, comme décrit dans le paragraphe précédent. Ce qui rend les générateurs si compacts, c'est que les méthodes `__iter__()` et `__next__()` sont créées automatiquement.

Une autre fonctionnalité clé est que les variables locales ainsi que le contexte d'exécution sont sauvegardés automatiquement entre les appels. Cela simplifie d'autant plus l'écriture de ces fonctions et rend leur code beaucoup plus lisible qu'avec une approche utilisant des variables d'instance telles que `self.index` et `self.data`.

En plus de la création automatique de méthodes et de la sauvegarde du contexte d'exécution, les générateurs lèvent automatiquement une exception `StopIteration` lorsqu'ils terminent leur exécution. La combinaison de ces fonctionnalités rend très simple la création d'itérateurs, sans plus d'effort que l'écriture d'une fonction classique.

9.10 Expressions et générateurs

Des générateurs simples peuvent être codés très rapidement avec des expressions utilisant la même syntaxe que les compréhensions de listes, mais en utilisant des parenthèses à la place des crochets. Ces expressions sont conçues pour des situations où le générateur est utilisé tout de suite dans une fonction. Ces expressions sont plus compactes mais moins souples que des définitions complètes de générateurs et ont tendance à être plus économes en mémoire que leur équivalent en compréhension de listes.

Exemples :

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

CHAPITRE 10

Survol de la bibliothèque standard

10.1 Interface avec le système d'exploitation

Le module `os` propose des dizaines de fonctions pour interagir avec le système d'exploitation :

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python311'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Veillez bien à utiliser `import os` plutôt que `from os import *`, sinon `os.open()` cache la primitive `open()` qui fonctionne différemment.

Les primitives `dir()` et `help()` sont des aides utiles lorsque vous travaillez en mode interactif avez des gros modules comme `os` :

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Pour la gestion des fichiers et dossiers, le module `shutil` expose une interface plus abstraite et plus facile à utiliser :

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Jokers sur les noms de fichiers

Le module `glob` fournit une fonction pour construire des listes de fichiers à partir de motifs :

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Paramètres passés en ligne de commande

Typiquement, les outils en ligne de commande ont besoin de lire les paramètres qui leur sont donnés. Ces paramètres sont stockés dans la variable `argv` du module `sys` sous la forme d'une liste. Par exemple, l'affichage suivant vient de l'exécution de `python demo.py one two three` depuis la ligne de commande :

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

Le module `argparse` fournit un mécanisme plus sophistiqué pour traiter les arguments de la ligne de commande. Le script suivant extrait un ou plusieurs noms de fichiers et un nombre facultatif de lignes à afficher :

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Lorsqu'il est exécuté avec la ligne de commande `python top.py --lines=5 alpha.txt beta.txt`, le script définit `args.lines` à `5` et `args.filenames` à `['alpha.txt', 'beta.txt']`.

10.4 Redirection de la sortie d'erreur et fin d'exécution

Le module `sys` a aussi des attributs pour *stdin*, *stdout* et *stderr*. Ce dernier est utile pour émettre des messages d'avertissement ou d'erreur qui restent visibles même si *stdout* est redirigé :

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Le moyen le plus direct de terminer un script est d'utiliser `sys.exit()`.

10.5 Recherche de motifs dans les chaînes

Le module `re` fournit des outils basés sur les expressions rationnelles permettant des opérations complexes sur les chaînes. C'est une solution optimisée, utilisant une syntaxe concise, pour rechercher des motifs complexes ou effectuer des remplacements complexes dans les chaînes :

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Lorsque les opérations sont simples, il est préférable d'utiliser les méthodes des chaînes. Elles sont plus lisibles et plus faciles à déboguer :

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Mathématiques

Le module `math` donne accès aux fonctions sur les nombres à virgule flottante (*float* en anglais) de la bibliothèque C :

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Le module `random` offre des outils pour faire des tirages aléatoires :

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

Le module `statistics` permet de calculer des valeurs statistiques basiques (moyenne, médiane, variance…) :

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

Le projet SciPy <https://scipy.org> contient beaucoup d'autres modules dédiés aux calculs numériques.

10.7 Accès à internet

Il existe beaucoup de modules permettant d'accéder à internet et gérer les protocoles réseaux. Les deux plus simples sont `urllib.request` qui permet de récupérer des données à partir d'une URL et `smtplib` pour envoyer des courriers électroniques :

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())         # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Notez que le deuxième exemple a besoin d'un serveur mail tournant localement.)

10.8 Dates et heures

Le module `datetime` propose des classes pour manipuler les dates et les heures de manière simple ou plus complexe. Bien que faire des calculs de dates et d'heures soit possible, la priorité de l'implémentation est mise sur l'extraction efficace des attributs pour le formatage et la manipulation. Le module gère aussi les objets dépendant des fuseaux horaires

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compression de données

Les formats d'archivage et de compression les plus communs sont directement gérés par les modules `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` et `tarfile`

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Mesure des performances

Certains utilisateurs de Python sont très intéressés par les performances de différentes approches d'un même problème. Python propose un outil de mesure répondant simplement à ces questions.

Par exemple, pour échanger deux variables, il peut être tentant d'utiliser l'empaquetage et le dépaquetage de *n*-uplets plutôt que la méthode traditionnelle. Le module `timeit` montre rapidement le léger gain de performance obtenu :

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

En opposition à `timeit` et sa granularité fine, `profile` et `pstats` fournissent des outils permettant d'identifier les parties les plus gourmandes en temps d'exécution dans des volumes de code plus grands.

10.11 Contrôle qualité

Une approche possible pour développer des applications de très bonne qualité est d'écrire des tests pour chaque fonction au fur et à mesure de son développement, puis d'exécuter ces tests fréquemment lors du processus de développement.

Le module `doctest` cherche des tests dans les chaînes de documentation. Un test ressemble à un simple copier-coller d'un appel et son résultat depuis le mode interactif. Cela améliore la documentation en fournissant des exemples tout en prouvant qu'ils sont justes :

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

Le module `unittest` requiert plus d'efforts que le module `doctest` mais il permet de construire un jeu de tests plus complet que l'on fait évoluer dans un fichier séparé :

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 Piles fournies

Python adopte le principe des "piles fournies". Vous pouvez le constater au travers des fonctionnalités évoluées et solides fournies par ses plus gros paquets. Par exemple :

- Les modules `xmlrpc.client` et `xmlrpc.server` permettent d'appeler des fonctions à distance quasiment sans effort. En dépit du nom des modules, aucune connaissance du XML n'est nécessaire.
- Le paquet `email` est une bibliothèque pour gérer les messages électroniques, incluant les MIME et autres encodages basés sur la **RFC 2822**. Contrairement à `smtplib` et `poplib` qui envoient et reçoivent des messages, le paquet *email* est une boite à outils pour construire, lire des structures de messages complexes (comprenant des pièces jointes) ou implémenter des encodages et protocoles.
- Le paquet `json` permet de lire et d'écrire du JSON, format d'encodage de données répandu. Le module `csv` gère la lecture et l'écriture de données stockées sous forme de valeurs séparées par des virgules dans des fichiers (*Comma-Separated Values* en anglais), format typiquement interopérable avec les bases de données et les feuilles de calculs. Pour la lecture du XML, utilisez les paquets `xml.etree.ElementTree`, `xml.dom` et `xml.sax`. Combinés, ces modules et paquets simplifient grandement l'échange de données entre les applications Python et les autres outils.
- Le module `sqlite3` est une abstraction de la bibliothèque SQLite, permettant de manipuler une base de données persistante, accédée et manipulée en utilisant une syntaxe SQL quasi standard.
- L'internationalisation est possible grâce à de nombreux paquets tels que `gettext`, `locale` ou `codecs`.

CHAPITRE 11

Survol de la bibliothèque standard -- Deuxième partie

Cette deuxième partie aborde des modules plus à destination des programmeurs professionnels. Ces modules sont rarement nécessaires dans de petits scripts.

11.1 Formatage de l'affichage

Le module `reprlib` est une variante de la fonction `repr()`, spécialisé dans l'affichage concis de conteneurs volumineux ou fortement imbriqués :

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

Le module `pprint` propose un contrôle plus fin de l'affichage des objets, aussi bien natifs que ceux définis par l'utilisateur, de manière à être lisible par l'interpréteur. Lorsque le résultat fait plus d'une ligne, il est séparé sur plusieurs lignes et est indenté pour rendre la structure plus visible :

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

Le module `textwrap` formate des paragraphes de texte pour tenir sur un écran d'une largeur donnée :

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
```

(suite sur la page suivante)

(suite de la page précédente)

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

Le module `locale` utilise une base de données des formats spécifiques à chaque région pour les dates, nombres, etc. L'attribut `grouping` de la fonction de formatage permet de formater directement des nombres avec un séparateur :

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Gabarits (*templates* en anglais)

Le module `string` contient une classe polyvalente : `Template`. Elle permet d'écrire des gabarits (*templates* en anglais) avec une syntaxe simple, dans le but d'être utilisable par des non-développeurs. Ainsi, vos utilisateurs peuvent personnaliser leur application sans la modifier.

Le format utilise des marqueurs formés d'un `$` suivi d'un identifiant Python valide (caractères alphanumériques et tirets-bas). Entourer le marqueur d'accolades permet de lui coller d'autres caractères alphanumériques sans intercaler une espace. Écrire `$$` produit un simple `$` :

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

La méthode `substitute()` lève une exception `KeyError` lorsqu'un marqueur n'a pas été fourni, ni dans un dictionnaire, ni sous forme d'un paramètre nommé. Dans certains cas, lorsque la donnée à appliquer peut n'être fournie que partiellement par l'utilisateur, la méthode `safe_substitute()` est plus appropriée car elle laisse tels quels les marqueurs manquants :

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Les classes filles de `Template` peuvent définir leur propre délimiteur. Typiquement, un script de renommage de photos par lots peut choisir le symbole pourcent comme marqueur pour les champs tels que la date actuelle, le numéro de l'image

ou son format :

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Une autre utilisation des gabarits consiste à séparer la logique métier des détails spécifiques à chaque format de sortie. Il est ainsi possible de générer des gabarits spécifiques pour les fichiers XML, texte, HTML…

11.3 Traitement des données binaires

Le module `struct` expose les fonctions `pack()` et `unpack()` permettant de travailler avec des données binaires. L'exemple suivant montre comment parcourir un entête de fichier ZIP sans recourir au module `zipfile`. Les marqueurs `"H"` et `"I"` représentent des nombres entiers non signés, stockés respectivement sur deux et quatre octets. Le `"<"` indique qu'ils ont une taille standard et utilisent la convention petit-boutiste :

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4 Fils d'exécution

Des tâches indépendantes peuvent être exécutées de manière non séquentielle en utilisant des fils d'exécution (*threading* en anglais). Les fils d'exécution peuvent être utilisés pour améliorer la réactivité d'une application qui interagit avec l'utilisateur pendant que d'autres traitements sont exécutés en arrière-plan. Une autre utilisation typique est de séparer sur deux fils d'exécution distincts les entrées / sorties et le calcul.

Le code suivant donne un exemple d'utilisation du module `threading` exécutant des tâches en arrière-plan pendant que le programme principal continue de s'exécuter :

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

Le principal défi des applications avec plusieurs fils d'exécution consiste à coordonner ces fils qui partagent des données ou des ressources. Pour ce faire, le module `threading` expose quelques outils dédiés à la synchronisation comme les verrous (*locks* en anglais), les événements (*events* en anglais), les variables conditionnelles (*condition variables* en anglais) et les sémaphores (*semaphore* en anglais).

Bien que ces outils soient puissants, de petites erreurs de conception peuvent engendrer des problèmes difficiles à reproduire. Donc, l'approche classique pour coordonner des tâches est de restreindre l'accès d'une ressource à un seul fil d'exécution et d'utiliser le module `queue` pour alimenter ce fil d'exécution en requêtes venant d'autres fils d'exécution. Les applications utilisant des `Queue` pour leurs communication et coordination entre fils d'exécution sont plus simples à concevoir, plus lisibles et plus fiables.

11.5 Journalisation

Le module `logging` est un système de journalisation complet. Dans son utilisation la plus élémentaire, les messages sont simplement envoyés dans un fichier ou sur `sys.stderr` :

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Cela produit l'affichage suivant :

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Par défaut, les messages d'information et de débogage sont ignorés, les autres sont envoyés vers la sortie standard. Il est aussi possible d'envoyer les messages par courriel, datagrammes, en utilisant des connecteurs réseau ou vers un serveur HTTP. Des nouveaux filtres permettent d'utiliser des sorties différentes en fonction de la priorité du message : `DEBUG`, `INFO`, `WARNING`, `ERROR` et `CRITICAL`.

La configuration de la journalisation peut être effectuée directement dans le code Python ou peut être chargée depuis un fichier de configuration, permettant de personnaliser la journalisation sans modifier l'application.

11.6 Références faibles

Python gère lui-même la mémoire (par comptage des références pour la plupart des objets et en utilisant un *ramasse-miettes* (*garbage collector* en anglais) pour éliminer les cycles). La mémoire est libérée rapidement lorsque sa dernière référence est supprimée.

Cette approche fonctionne bien pour la majorité des applications mais, parfois, il est nécessaire de surveiller un objet seulement durant son utilisation par quelque chose d'autre. Malheureusement, le simple fait de le suivre crée une référence qui rend l'objet permanent. Le module `weakref` expose des outils pour suivre les objets sans pour autant créer une référence. Lorsqu'un objet n'est pas utilisé, il est automatiquement supprimé du tableau des références faibles et une fonction de rappel (*callback* en anglais) est appelée. Un exemple typique est le cache d'objets coûteux à créer :

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python311/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Outils pour les listes

Beaucoup de structures de données peuvent être représentées avec les listes natives. Cependant, d'autres besoins peuvent émerger pour des structures ayant des caractéristiques différentes, typiquement en termes de performance.

Le module `array` fournit un objet `array()` ne permettant de stocker que des listes homogènes mais d'une manière plus compacte. L'exemple suivant montre une liste de nombres stockés chacun sur deux octets non signés (marqueur `"H"`) plutôt que d'utiliser 16 octets comme l'aurait fait une liste classique :

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Le module `collections` fournit la classe `deque()`. Elle ressemble à une liste mais est plus rapide pour l'insertion ou l'extraction des éléments par la gauche et plus lente pour accéder aux éléments du milieu. Ces objets sont particulièrement adaptés pour construire des queues ou des algorithmes de parcours d'arbres en largeur (ou BFS, pour *Breadth First Search* en anglais) :

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

En plus de fournir des implémentations de listes alternatives, la bibliothèque fournit des outils tels que `bisect`, un module contenant des fonctions de manipulation de listes triées :

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Le module `heapq` permet d'implémenter des tas (*heap* en anglais) à partir de simples listes. La valeur la plus faible est toujours à la première position (indice 0). C'est utile dans les cas où l'application accède souvent à l'élément le plus petit mais sans vouloir classer entièrement la liste :

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Arithmétique décimale à virgule flottante

Le module `decimal` exporte la classe `Decimal` : elle est spécialisée dans le calcul de nombres décimaux représentés en virgule flottante. Par rapport à la classe native `float`, elle est particulièrement utile pour :

— les applications traitant de finance et autres utilisations nécessitant une représentation décimale exacte,
— le contrôle de la précision,
— le contrôle sur les arrondis pour répondre à des obligations légales ou réglementaires,
— suivre les décimales significatives, ou
— les applications pour lesquelles l'utilisateur attend des résultats identiques aux calculs faits à la main.

Par exemple, calculer 5 % de taxe sur une facture de 70 centimes donne un résultat différent en nombre à virgule flottante binaire et décimale. La différence devient significative lorsqu'on arrondit le résultat au centime près :

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

Le résultat d'un calcul donné par `Decimal` conserve les zéros non-significatifs. La classe conserve automatiquement quatre décimales significatives pour des opérandes à deux décimales significatives. La classe `Decimal` imite les mathématiques telles qu'elles pourraient être effectuées à la main, évitant les problèmes typiques de l'arithmétique binaire à virgule flottante qui n'est pas capable de représenter exactement certaines quantités décimales.

La représentation exacte de la classe `Decimal` lui permet de faire des calculs de modulo ou des tests d'égalité qui ne seraient pas possibles avec une représentation à virgule flottante binaire :

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Le module `decimal` permet de faire des calculs avec autant de précision que nécessaire :

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

CHAPITRE 12

Environnements virtuels et paquets

12.1 Introduction

Les programmes Python utilisent souvent des paquets et modules qui ne font pas partie de la bibliothèque standard. Ils nécessitent aussi, parfois, une version spécifique d'une bibliothèque, par exemple parce qu'un certain bogue a été corrigé ou encore que le programme a été implémenté en utilisant une version obsolète de l'interface de cette bibliothèque.

Cela signifie qu'il n'est pas toujours possible, pour une installation unique de Python, de couvrir tous les besoins de toutes les applications. Basiquement, si une application A dépend de la version 1.0 d'un module et qu'une application B dépend de la version 2.0, ces dépendances entrent en conflit et installer la version 1.0 ou 2.0 laisse une des deux applications incapable de fonctionner.

La solution est de créer un *environnement virtuel*, un dossier auto-suffisant qui contient une installation de Python pour une version particulière de Python ainsi que des paquets additionnels.

Différentes applications peuvent alors utiliser des environnements virtuels différents. Pour résoudre l'exemple précédent où il existe un conflit de dépendances, l'application A a son environnement virtuel avec la version 1.0 installée pendant que l'application B a un autre environnement virtuel avec la version 2.0. Si l'application B requiert que la bibliothèque soit mise à jour à la version 3.0, cela n'affecte pas l'environnement de A.

12.2 Création d'environnements virtuels

Le module utilisé pour créer et gérer des environnements virtuels s'appelle `venv`. `venv` installe en général la version de Python la plus récente dont vous disposez. Si plusieurs versions de Python sont sur votre système, vous pouvez choisir une version particulière en exécutant `python3.X` où X indique la version de votre choix.

Pour créer un environnement virtuel, décidez d'un dossier où vous voulez le placer et exécutez le module `venv` comme un script avec le chemin du dossier :

```
python -m venv tutorial-env
```

Cela crée le dossier `tutorial-env` (s'il n'existe pas) et des sous-dossiers contenant une copie de l'interpréteur Python et d'autres fichiers utiles.

Un répertoire habituel pour un environnement virtuel est `.venv`. Ce nom fait que le répertoire est généralement caché dans votre explorateur de fichiers, et donc non gênant, tout en lui donnant un nom qui explique pourquoi le répertoire existe. Il empêche également de rentrer en conflit avec les fichiers de définition de variable d'environnement `.env` que certains outils utilisent.

Une fois l'environnement virtuel créé, vous pouvez l'activer.

Sur Windows, lancez :

```
tutorial-env\Scripts\activate.bat
```

Sur Unix et MacOS, lancez :

```
source tutorial-env/bin/activate
```

(Ce script est écrit pour le shell **bash**. Si vous utilisez **csh** ou **fish**, utilisez les variantes `activate.csh` ou `activate.fish`.)

Activer l'environnement virtuel change le prompt de votre ligne de commande pour afficher le nom de l'environnement virtuel que vous utilisez. Cela modifie aussi l'environnement afin, lorsque vous tapez `python`, d'exécuter la version spécifique de Python installée dans l'environnement. Par exemple :

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

Pour désactiver un environnement virtuel, tapez :

```
deactivate
```

dans le terminal.

12.3 Gestion des paquets avec *pip*

You can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the Python Package Index. You can browse the Python Package Index by going to it in your web browser.

`pip` a plusieurs sous-commandes : `install`, `uninstall`, `freeze`, etc. Consultez le guide installing-index pour une documentation exhaustive sur `pip`.

Vous pouvez installer la dernière version d'un paquet en indiquant son nom :

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Vous pouvez installer une version spécifique d'un paquet en donnant le nom du paquet suivi de == et du numéro de version souhaitée :

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Si vous relancez cette commande, `pip` remarque que la version demandée est déjà installée et ne fait rien. Vous pouvez fournir un numéro de version différent pour récupérer cette version ou lancer `python -m pip install --upgrade` pour mettre à jour le paquet à la dernière version :

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` suivi d'un ou plusieurs noms de paquets les supprime de votre environnement virtuel.

`python -m pip show` affiche des informations à propos d'un paquet précis :

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` liste tous les paquets installés dans l'environnement virtuel :

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` produit une liste similaire des paquets installés mais l'affichage adopte un format que `pip install` peut lire. La convention habituelle est de mettre cette liste dans un fichier `requirements.txt` :

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Le fichier `requirements.txt` peut alors être ajouté dans un système de gestion de versions comme faisant partie de votre application. Les utilisateurs peuvent alors installer tous les paquets nécessaires à l'application avec `install -r` :

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` reconnait beaucoup d'autres options, documentées dans le guide installing-index. Lorsque vous avez écrit un paquet, si vous voulez le rendre disponible sur PyPI, lisez le guide distributing-index.

CHAPITRE 13

Pour aller plus loin

La lecture de ce tutoriel a probablement renforcé votre intérêt pour Python et vous devez être impatient de l'utiliser pour résoudre des vrais problèmes. Où aller pour en apprendre plus ?

Ce tutoriel fait partie de la documentation de Python, et la documentation de Python est vaste :

— library-index :
  Nous vous conseillons de naviguer dans le manuel, c'est une référence complète (quoique laconique) sur les types, fonctions et modules de la bibliothèque standard. La distribution standard de Python inclut *énormément* de code supplémentaire. Il existe des modules pour lire les courriels, récupérer des documents *via* HTTP, générer des nombres aléatoires, analyser les paramètres de la ligne de commande, compresser des données et beaucoup d'autres fonctions. Une balade dans la documentation de la bibliothèque vous donnera une idée de ce qui est disponible.
— installing-index explique comment installer des paquets écrits par d'autres utilisateurs de Python.
— reference-index contient une explication détaillée de la syntaxe et sémantique de Python. C'est une lecture fastidieuse, mais elle a sa place dans une documentation exhaustive.

D'autres ressources :

— https://www.python.org : le site principal pour Python. Il contient du code, de la documentation, des liens vers d'autres sites traitant de Python partout sur Internet.
— https://docs.python.org/fr/ offre un accès rapide à la documentation de Python en français.
— https://pypi.org (*The Python Package Index* en anglais, pour le "répertoire des paquets Python") : auparavant surnommé "La Fromagerie" [1] (*The Cheese Shop* en anglais), c'est un catalogue de modules Python disponibles au téléchargement, construit par les utilisateurs. Lorsque vous commencez à distribuer du code, vous pouvez l'inscrire ici afin que les autres puissent le trouver.
— https://code.activestate.com/recipes/langs/python/ : "The Python Cookbook" est un recueil assez imposant d'exemples de code, de modules et de scripts. Les contributions les plus remarquables y sont regroupées dans le livre "Python Cookbook" (O'Reilly & Associates, ISBN 0-596-00797-3).
— https://pyvideo.org regroupe des liens vers des vidéos relatives à Python, enregistrées lors de conférences ou de réunions de groupes d'utilisateurs.
— https://scipy.org : le projet "The Scientific Python" contient des modules pour manipuler et calculer rapidement sur des tableaux. Le projet héberge aussi divers paquets sur l'algèbre linéaire, les transformées de Fourier, des résolveurs non-linéaires, différentes distributions de nombres aléatoires, l'analyse statistique et d'autres choses du domaine scientifique.

1. « Cheese Shop » est un sketch de Monty Python : un client entre dans une fromagerie, mais peu importe le fromage que demande le client, le vendeur dit qu'il n'en a pas.

Pour poser des questions ou remonter des problèmes liés à Python, vous pouvez écrire sur le forum *comp.lang.python* ou les envoyer à la liste de diffusion <python-list@python.org>. Le forum et la liste de diffusion sont liées, un message publié sur l'un sera automatiquement transféré sur l'autre. Des centaines de messages y sont publiés chaque jour, posant (ou répondant à) des questions, suggérant de nouvelles fonctionnalités et annonçant des nouveaux modules. Les archives sont disponibles à https://mail.python.org/pipermail/.

Avant de publier un message, assurez-vous d'avoir lu la liste de la Foire Aux Questions (aussi appelée FAQ). La FAQ répond à beaucoup de questions fréquentes et contient probablement une solution à votre problème.

Notes

CHAPITRE 14

Édition interactive des entrées et substitution d'historique

Certaines versions de l'interpréteur Python prennent en charge l'édition de la ligne d'entrée courante et la substitution d'historique, similaires aux facilités que l'on trouve dans le shell Korn et dans le shell GNU Bash. Ces implémentations utilisent la bibliothèque GNU Readline, qui gère plusieurs styles d'édition. La bibliothèque a sa propre documentation, nous ne la dupliquons pas ici.

14.1 Complétion automatique et édition de l'historique

La complétion de noms de variables et de modules est automatiquement activée au démarrage de l'interpréteur. Ainsi, la touche `Tab` invoque la fonction de complétion ; la recherche s'effectue dans les noms d'instructions Python, les noms des variables locales et les noms de modules disponibles. Pour les expressions pointées telles que `string.a`, l'expression est évaluée jusqu'au dernier `'.'` avant de suggérer les options disponibles à partir des attributs de l'objet résultant de cette évaluation. Notez bien que ceci peut exécuter une partie du code de l'application si un objet disposant d'une méthode `__getattr__()` fait partie de l'expression. La configuration par défaut sauvegarde l'historique dans un fichier nommé `.python_history` dans votre dossier d'utilisateur. L'historique est ainsi conservé entre les sessions interactives successives.

14.2 Alternatives à l'interpréteur interactif

Cette facilité constitue un énorme pas en avant comparé aux versions précédentes de l'interpréteur. Toutefois, il reste des fonctions à implémenter comme l'indentation correcte sur les lignes de continuation (l'analyseur sait si une indentation doit suivre) ; le mécanisme de complétion devrait utiliser la table de symboles de l'interpréteur. Une commande pour vérifier (ou même suggérer) les correspondances de parenthèses, de guillemets, etc., serait également utile.

Une alternative améliorée de l'interpréteur interactif est développée depuis maintenant quelques temps : IPython. Il fournit la complétion, l'exploration d'objets et une gestion avancée de l'historique. Il peut également être personnalisé en profondeur et embarqué dans d'autres applications. Un autre environnement interactif amélioré similaire est bpython.

CHAPITRE 15

Arithmétique en nombres à virgule flottante : problèmes et limites

Les nombres à virgule flottante sont représentés, au niveau matériel, en fractions de nombres binaires (base 2). Par exemple, en représentation **décimale** le nombre `0.125` exprime 1/10 + 2/100 + 5/1000. De la même manière, en représentation **binaire** le nombre `0.001` exprime 0/2 + 0/4 + 1/8. Ces deux fractions de l'unité ont une valeur identique, la seule différence est que la première est une fraction dont la notation est en base 10, la seconde est une fraction dont la notation est en base 2.

Malheureusement, la plupart des fractions décimales ne peuvent pas avoir de représentation exacte en fractions binaires. Par conséquent, en général, les nombres à virgule flottante que vous donnez sont seulement approximés en fractions binaires pour être stockés dans la machine.

Le problème est plus simple à aborder en base 10. Prenons par exemple, la fraction 1/3. Vous pouvez l'approximer en une fraction décimale :

```
0.3
```

ou, mieux

```
0.33
```

ou, mieux

```
0.333
```

etc. Peu importe le nombre de décimales que vous écrivez, le résultat ne vaut jamais exactement 1/3, mais c'est une estimation s'en approchant toujours mieux.

De la même manière, peu importe combien de décimales en base 2 vous utilisez, la valeur décimale 0.1 ne peut pas être représentée exactement en fraction binaire. En base 2, 1/10 est le nombre périodique suivant

```
0.0001100110011001100110011001100110011001100110011...
```

En se limitant à une quantité finie de bits, on ne peut obtenir qu'une approximation. Sur la majorité des machines aujourd'hui, les nombres à virgule flottante sont approximés par une fraction binaire avec les 53 premiers bits comme numérateur et une puissance de deux au dénominateur. Dans le cas de 1/10, la fraction binaire est `3602879701896397 / 2 ** 55` qui est proche mais ne vaut pas exactement 1/10.

Du fait de la manière dont les flottants sont affichés par l'interpréteur, il est facile d'oublier que la valeur stockée est une approximation de la fraction décimale d'origine. Python n'affiche qu'une approximation décimale de la valeur stockée en binaire. Si Python devait afficher la vraie valeur décimale de l'approximation binaire stockée pour 0,1, il afficherait

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

C'est bien plus de décimales que ce qu'attendent la plupart des utilisateurs, donc Python affiche une valeur arrondie afin d'améliorer la lisibilité

```
>>> 1 / 10
0.1
```

Rappelez-vous simplement que, bien que la valeur affichée ressemble à la valeur exacte de 1/10, la valeur stockée est la représentation la plus proche en fraction binaire.

Il existe beaucoup de nombres décimaux qui partagent une même approximation en fraction binaire. Par exemple, `0.1`, `0.10000000000000001` et `0.1000000000000000055511151231257827021181583404541015625` ont tous pour approximation `3602879701896397 / 2 ** 55`. Puisque toutes ces valeurs décimales partagent la même approximation, chacune peut être affichée tout en respectant `eval(repr(x)) == x`.

Historiquement, le mode interactif de Python et la primitive `repr()` choisissaient la version avec 17 décimales significatives, `0.10000000000000001`. Python, depuis la version 3.1 (sur la majorité des systèmes) est maintenant capable de choisir la plus courte représentation et n'affiche que `0.1`.

Ce comportement est inhérent à la nature même de la représentation des nombres à virgule flottante dans la machine : ce n'est pas un bogue dans Python et ce n'est pas non plus un bogue dans votre code. Vous pouvez observer le même type de comportement dans tous les autres langages utilisant le support matériel pour le calcul des nombres à virgule flottante (bien que certains langages ne rendent pas visible la différence par défaut, ou pas dans tous les modes d'affichage).

Pour obtenir un affichage plus plaisant, les fonctions de formatage de chaînes de caractères peuvent limiter le nombre de décimales significatives affichées :

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Il est important de comprendre que tout cela n'est, au sens propre, qu'une illusion : vous demandez simplement à Python d'arrondir la valeur stockée réellement dans la machine à *l'affichage*.

Une autre conséquence du fait que 0,1 n'est pas exactement stocké 1/10 est que la somme de trois valeurs de 0,1 ne donne pas 0,3 non plus :

```
>>> .1 + .1 + .1 == .3
False
```

Aussi, puisque 0,1 ne peut pas être stocké avec une représentation plus proche de sa valeur exacte 1/10, comme 0,3 qui ne peut pas être plus proche de sa valeur exacte 3/10, arrondir au préalable avec la fonction `round()` n'aide en rien :

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Bien que les nombres ne peuvent se rapprocher plus de la valeur qu'on attend qu'ils aient, la fonction `round()` peut être utile à postériori pour arrondir deux valeurs inexactes et pouvoir les comparer :

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

L'arithmétique des nombres binaires à virgule flottante réserve beaucoup de surprises de ce genre. Le problème avec « 0.1 » est expliqué en détails ci-dessous, dans la section « Erreurs de représentation ». Voir The Perils of Floating Point pour une liste plus complète de ce genre de surprises.

Même s'il est vrai qu'il n'existe pas de réponse simple, ce n'est pas la peine de vous méfier outre mesure des nombres à virgule flottante ! Les erreurs, en Python, dans les opérations de nombres à virgule flottante sont dues au matériel sous-jacent et, sur la plupart des machines, sont de l'ordre de 1 sur 2**53 par opération. C'est plus que suffisant pour la plupart des tâches, mais vous devez garder à l'esprit que ce ne sont pas des opérations décimales et que chaque opération sur des nombres à virgule flottante peut souffrir d'une nouvelle erreur.

Bien que des cas pathologiques existent, pour la plupart des cas d'utilisations courants vous obtiendrez le résultat attendu à la fin en arrondissant simplement au nombre de décimales désirées à l'affichage avec `str()`. Pour un contrôle fin sur la manière dont les décimales sont affichées, consultez dans formatstrings les spécifications de formatage de la méthode `str.format()`.

Pour les cas requérant une représentation décimale exacte, le module `decimal` peut être utile : il implémente l'arithmétique décimale et peut donc être un choix adapté pour des applications nécessitant une grande précision.

Une autre forme d'arithmétique exacte est implémentée dans le module `fractions` qui se base sur les nombres rationnels (donc 1/3 peut y être représenté exactement).

Si vous êtes un utilisateur intensif des opérations sur les nombres à virgule flottante, nous vous conseillons de considérer le paquet *NumPy* ainsi que les paquets pour les opérations statistiques et mathématiques fournis par le projet SciPy. Consultez <https://scipy.org>.

Python fournit des outils qui peuvent être utiles dans les rares occasions où vous voulez réellement connaître la valeur exacte d'un nombre à virgule flottante. La méthode `float.as_integer_ratio()` donne la valeur du nombre sous forme de fraction :

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Puisque le ratio est exact, il peut être utilisé pour recréer la valeur originale sans perte :

```
>>> x == 3537115888337719 / 1125899906842624
True
```

La méthode `float.hex()` donne le nombre en hexadécimal (base 16), donnant ici aussi la valeur exacte stockée par la machine :

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Cette représentation hexadécimale petit être utilisée pour reconstruire, sans approximation, le *float* :

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Puisque cette représentation est exacte, elle est pratique pour échanger des valeurs entre différentes versions de Python (indépendamment de la machine) ou d'autres langages qui comprennent ce format (tels que Java et C99).

Une autre fonction utile est `math.fsum()`, elle aide à diminuer les pertes de précision lors des additions. Elle surveille les *décimales perdues* au fur et à mesure que les valeurs sont ajoutées au total. Cela peut faire une différence au niveau de la précision globale en empêchant les erreurs de s'accumuler jusqu'à affecter le résultat final :

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Erreurs de représentation

Cette section explique en détail l'exemple du « 0.1 » et montre comment vous pouvez effectuer une analyse exacte de ce type de cas par vous-même. Nous supposons que la représentation binaire des nombres flottants vous est familière.

Le terme *Erreur de représentation* (*representation error* en anglais) signifie que la plupart des fractions décimales ne peuvent être représentées exactement en binaire. C'est la principale raison pour laquelle Python (ou Perl, C, C++, Java, Fortran et beaucoup d'autres) n'affiche habituellement pas le résultat exact en décimal.

Pourquoi ? 1/10 n'est pas représentable de manière exacte en fraction binaire. Cependant, toutes les machines d'aujourd'hui (novembre 2000) suivent la norme IEEE-754 en ce qui concerne l'arithmétique des nombres à virgule flottante et la plupart des plateformes utilisent un « IEEE-754 double précision » pour représenter les *floats* de Python. Les « IEEE-754 double précision » utilisent 53 bits de précision donc, à la lecture, l'ordinateur essaie de convertir 0,1 dans la fraction la plus proche possible de la forme *J*/2***N* avec *J* un nombre entier d'exactement 53 bits. Pour réécrire

```
1 / 10 ~= J / (2**N)
```

en

```
J ~= 2**N / 10
```

en se rappelant que *J* fait exactement 53 bits (donc `>= 2**52` mais `< 2**53`), la meilleure valeur possible pour *N* est 56 :

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Donc 56 est la seule valeur possible pour *N* qui laisse exactement 53 bits pour *J*. La meilleure valeur possible pour *J* est donc ce quotient, arrondi :

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Puisque la retenue est plus grande que la moitié de 10, la meilleure approximation est obtenue en arrondissant par le haut :

```
>>> q+1
7205759403792794
```

Par conséquent la meilleure approximation possible pour 1/10 en « IEEE-754 double précision » est celle au-dessus de 2**56, soit :

```
7205759403792794 / 2 ** 56
```

Diviser le numérateur et le dénominateur par deux réduit la fraction à :

```
3602879701896397 / 2 ** 55
```

Notez que puisque l'arrondi a été fait vers le haut, le résultat est en réalité légèrement plus grand que 1/10 ; si nous n'avions pas arrondi par le haut, le quotient aurait été légèrement plus petit que 1/10. Mais dans aucun cas il ne vaut *exactement* 1/10 !

Donc l'ordinateur ne « voit » jamais 1/10 : ce qu'il voit est la fraction exacte donnée ci-dessus, la meilleure approximation utilisant les nombres à virgule flottante double précision de l'« IEEE-754 » :

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Si nous multiplions cette fraction par 10**55, nous pouvons observer les valeurs de ses 55 décimales de poids fort :

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

La valeur stockée dans l'ordinateur est donc égale à 0,1000000000000000055511151231257827021181583404541015625. Au lieu d'afficher toutes les décimales, beaucoup de langages (dont les vieilles versions de Python) arrondissent le résultat à la 17^{e} décimale significative :

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Les modules `fractions` et `decimal` rendent simples ces calculs :

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

CHAPITRE 16

Annexe

16.1 Mode interactif

16.1.1 Gestion des erreurs

Quand une erreur se produit, l'interpréteur affiche un message d'erreur et la trace d'appels. En mode interactif, il revient à l'invite de commande primaire ; si l'entrée provient d'un fichier, l'interpréteur se termine avec un code de sortie non nul après avoir affiché la trace d'appels (les exceptions gérées par une clause `except` dans une instruction `try` ne sont pas considérées comme des erreurs dans ce contexte). Certaines erreurs sont inconditionnellement fatales et provoquent la fin du programme avec un code de sortie non nul ; les incohérences internes et, dans certains cas, les pénuries de mémoire sont traitées de la sorte. Tous les messages d'erreur sont écrits sur le flux d'erreur standard ; l'affichage normal des commandes exécutées est écrit sur la sortie standard.

Taper le caractère d'interruption (généralement `Ctrl+C` ou `Supprimer`) au niveau de l'invite de commande primaire annule l'entrée et revient à l'invite [1]. Saisir une interruption tandis qu'une commande s'exécute lève une exception `KeyboardInterrupt` qui peut être gérée par une instruction `try`.

16.1.2 Scripts Python exécutables

Sur les systèmes Unix, un script Python peut être rendu directement exécutable, comme un script *shell*, en ajoutant la ligne

```
#!/usr/bin/env python3.5
```

(en supposant que l'interpréteur est dans le `PATH` de l'utilisateur) au début du script et en rendant le fichier exécutable. `'#!'` doivent être les deux premiers caractères du fichier. Sur certaines plateformes, cette première ligne doit finir avec une fin de ligne de type Unix (`'\n'`) et pas de type Windows (`'\r\n'`). Notez que le caractère croisillon, `'#'`, est utilisé pour initier un commentaire en Python.

Un script peut être rendu exécutable en utilisant la commande **`chmod`**.

1. Un problème avec GNU *Readline* peut l'en empêcher.

```
$ chmod +x myscript.py
```

Sur les systèmes Windows il n'y a pas de "mode exécutable". L'installateur Python associe automatiquement les fichiers en `.py` avec `python.exe` de telle sorte qu'un double clic sur un fichier Python le lance comme un script. L'extension peut aussi être `.pyw`. Dans ce cas, la console n'apparait pas.

16.1.3 Configuration du mode interactif

En mode interactif, il peut être pratique de faire exécuter quelques commandes au lancement de l'interpréteur. Configurez la variable d'environnement `PYTHONSTARTUP` avec le nom d'un fichier contenant les instructions à exécuter, à la même manière du `.profile` pour un *shell* Unix.

Ce fichier n'est lu qu'en mode interactif, pas quand Python lit les instructions depuis un fichier, ni quand `/dev/tty` est donné explicitement comme fichier source (pour tout le reste, Python se comporte alors comme dans une session interactive). Les instructions de ce fichier sont exécutées dans le même espace de nommage que vos commandes, donc les objets définis et modules importés peuvent être utilisés directement dans la session interactive. Dans ce fichier, il est aussi possible de changer les invites de commande `sys.ps1` et `sys.ps2`.

Si vous voulez exécuter d'autres fichiers du dossier courant au démarrage, vous pouvez le programmer dans le fichier de démarrage global, par exemple avec le code suivant : `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Et si vous voulez exécuter le fichier de démarrage depuis un script, vous devez le faire explicitement dans le script :

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 Modules de personnalisation

Python peut être personnalisé *via* les modules `sitecustomize` et `usercustomize`. Pour découvrir comment ils fonctionnent, vous devez d'abord trouver l'emplacement de votre dossier « site-packages » utilisateur. Démarrez Python et exécutez ce code :

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.5/site-packages'
```

Vous pouvez maintenant y créer un fichier `usercustomize.py` et y écrire ce que vous voulez. Il est toujours pris en compte par Python, peu importe le mode, sauf lorsque vous démarrez avec l'option `-s` qui désactive l'importation automatique.

`sitecustomize` fonctionne de la même manière mais est généralement créé par un administrateur et stocké dans le dossier site-packages global. Il est importé avant `usercustomize`. Pour plus de détails, consultez la documentation de `site`.

Notes

ANNEXE A

Glossaire

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

... Peut faire référence à :
- L'invite de commande utilisée par défaut dans l'interpréteur interactif lorsqu'on entre un bloc de code indenté, dans des délimiteurs fonctionnant par paires (parenthèses, crochets, accolades, triple guillemets), ou après un avoir spécifié un décorateur.
- La constante `Ellipsis`.

2to3 Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

2to3 est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. 2to3-reference.

classe 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 `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`.

Chaque `yield` suspend temporairement l'exécution, en gardant en mémoire l'endroit et l'état de l'exécution (ce qui inclut les variables locales et les *try* en cours). Lorsque l'exécution de l'itérateur de générateur asynchrone reprend avec un nouvel *awaitable* renvoyé par `__anext__()`, elle repart de là où elle s'était arrêtée. Voir les **PEP 492** et **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 Un *file object* capable de lire et d'écrire des *bytes-like objects*. Des fichiers binaires sont, par exemple, les fichiers ouverts en mode binaire (`'rb'`, `'wb'`, ou `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, les instances de `io.BytesIO` ou de `gzip.GzipFile`.

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

référence empruntée Dans l'API C de Python, une référence empruntée est une référence vers un objet qui n'affecte pas son compteur de référence. Elle devient invalide si l'objet est supprimé, par exemple au cours d'un passage du ramasse-miettes qui conduit à la disparition de la dernière *référence forte* vers l'objet.

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).

nombre complexe Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de `-1`, souvent écrite `i` en mathématiques ou `j` par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe `j`, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

gestionnaire de contexte Objet contrôlant l'environnement à l'intérieur d'un bloc `with` en définissant les méthodes `__enter__()` et `__exit__()`. Consultez la **PEP 343**.

variable de contexte Une variable qui peut avoir des valeurs différentes en fonction de son contexte. Cela est similaire au stockage par fil d'exécution (*Thread Local Storage* en anglais) dans lequel chaque fil d'exécution peut avoir une valeur différente pour une variable. Toutefois, avec les variables de contexte, il peut y avoir plusieurs contextes dans un fil d'exécution et l'utilisation principale pour les variables de contexte est de garder une trace des variables dans les tâches asynchrones concourantes. Voir `contextvars`.

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*.

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 N'importe quel objet définissant les méthodes `__get__()`, `__set__()`, ou `__delete__()`. Lorsque l'attribut d'une classe est un descripteur, son comportement spécial est déclenché lors de la recherche des attributs. Normalement, lorsque vous écrivez *a.b* pour obtenir, affecter ou effacer un attribut, Python recherche l'objet nommé *b* dans le dictionnaire de la classe de *a*. Mais si *b* est un descripteur, c'est la méthode de ce descripteur qui est alors appelée. Comprendre les descripteurs est requis pour avoir une compréhension approfondie de Python, ils sont la base de nombre de ses caractéristiques notamment les fonctions, méthodes, propriétés, méthodes de classes, méthodes statiques et les références aux classes parentes.

Pour plus d'informations sur les méthodes des descripteurs, consultez descriptors ou le guide pour l'utilisation des descripteurs.

dictionnaire Structure de donnée associant des clés à des valeurs. Les clés peuvent être n'importe quel objet possédant les méthodes `__hash__()` et `__eq__()`. En Perl, les dictionnaires sont appelés "*hash*".

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*) Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

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 Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, un connecteur réseau…). Les objets fichiers sont aussi appelés *file-like-objects* ou *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.

Depuis Python 3.3, il existe deux types de chercheurs : les *chercheurs dans les méta-chemins* à utiliser avec `sys.meta_path`; les *chercheurs d'entrée dans path* à utiliser avec `sys.path_hooks`.

Voir les **PEP 302**, **PEP 420** et **PEP 451** pour plus de détails.

division entière Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut `2`, contrairement à `11 / 4` qui vaut `2.75`. Notez que `(-11) // 4` vaut `-3` car l'arrondi se fait à l'entier inférieur. Voir la **PEP 328**.

fonction Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et function.

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*.

Chaque `yield` suspend temporairement l'exécution, en se rappelant l'endroit et l'état de l'exécution (y compris les variables locales et les *try* en cours). Lorsque l'itérateur de générateur reprend, il repart là où il en était (contrairement à une fonction qui prendrait un nouveau départ à chaque invocation).

expression génératrice Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une clause `for` définissant une variable de boucle, un intervalle et une clause `if` optionnelle. Toute cette expression génère des valeurs pour la fonction qui l'entoure :

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

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.

Les tentatives précédentes d'implémenter un interpréteur Python avec une granularité de verrouillage plus fine ont toutes échouées, à cause de leurs mauvaises performances dans le cas d'un processeur unique. Il est admis que corriger ce problème de performance induit mènerait à une implémentation beaucoup plus compliquée et donc plus coûteuse à maintenir.

***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 Un objet est *hachable* s'il a une empreinte (*hash*) qui ne change jamais (il doit donc implémenter une méthode `__hash__()`) et s'il peut être comparé à d'autres objets (avec la méthode `__eq__()`). Les objets hachables dont la comparaison par `__eq__` est vraie doivent avoir la même empreinte.

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 muables (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.

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 a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

interprété Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

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 Objet capable de renvoyer ses éléments un à un. Par exemple, tous les types de séquences (comme `list`, `str`, et `tuple`), quelques autres types comme `dict`, les *objets fichiers* ou tout objet d'une classe ayant une méthode `__iter__()` ou `__getitem__()` qui implémente la sémantique d'une *séquence*.

Les itérables peuvent être utilisés dans des boucles `for` et à beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()` …). Lorsqu'un itérable est passé comme argument à la fonction native `iter()`, celle-ci fournit en retour un itérateur sur cet itérable. Cet itérateur n'est valable que pour une seule passe sur le jeu de valeurs. Lors

de l'utilisation d'itérables, il n'est habituellement pas nécessaire d'appeler `iter()` ou de s'occuper soi-même des objets itérateurs. L'instruction `for` le fait automatiquement pour vous, créant une variable temporaire anonyme pour garder l'itérateur durant la boucle. Voir aussi *itérateur*, *séquence* et *générateur*.

itérateur Objet représentant un flux de donnée. Des appels successifs à la méthode `__next__()` de l'itérateur (ou le passer à la fonction native `next()`) donne successivement les objets du flux. Lorsque plus aucune donnée n'est disponible, une exception `StopIteration` est levée. À ce point, l'itérateur est épuisé et tous les appels suivants à sa méthode `__next__()` lèveront encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui-même, de façon à ce que chaque itérateur soit aussi itérable et puisse être utilisé dans la plupart des endroits où d'autres itérables sont attendus. Une exception notable est un code qui tente plusieurs itérations complètes. Un objet conteneur, (tel que `list`) produit un nouvel itérateur neuf à chaque fois qu'il est passé à la fonction `iter()` ou s'il est utilisé dans une boucle `for`. Faire ceci sur un itérateur donnerait simplement le même objet itérateur épuisé utilisé dans son itération précédente, le faisant ressembler à un conteneur vide.

Vous trouverez davantage d'informations dans typeiter.

Particularité de l'implémentation CPython : CPython n'est pas toujours cohérent sur le fait de demander ou non à un itérateur de définir `__iter__()`.

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.

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()` permet de récupérer l'encodage régional.

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

liste Un type natif de *sequence* dans Python. En dépit de son nom, une `list` ressemble plus à un tableau (*array* dans la plupart des langages) qu'à une liste chaînée puisque les accès se font en 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 Objet qui charge un module. Il doit définir une méthode nommée `load_module()`. Un chargeur est typiquement donné par un *chercheur*. Voir la **PEP 302** pour plus de détails et `importlib.ABC.Loader` pour sa *classe mère abstraite*.

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 muables` (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 L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir The Python 2.3 Method Resolution Order pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

module Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de 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`.

MRO Voir *ordre de résolution des méthodes*.

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

***n*-uplet nommé** 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
```

Certains *n-uplets nommés* sont des types natifs (comme les exemples ci-dessus). Sinon, un *n-uplet nommé* peut être créé à partir d'une définition de classe habituelle qui hérite de `tuple` et qui définit les champs nommés. Une telle classe peut être écrite à la main ou être créée avec la fonction `collections.namedtuple()`. Cette dernière méthode ajoute des méthodes supplémentaires qui ne seront pas trouvées dans celles écrites à la main ni dans les n-uplets nommés natifs.

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 Un *paquet* tel que défini dans la **PEP 421** qui ne sert qu'à contenir des sous-paquets. Les paquets-espace de nommage peuvent n'avoir aucune représentation physique et, plus spécifiquement, ne sont pas comme un *paquet classique* puisqu'ils n'ont pas de fichier `__init__.py`.

Voir aussi *module*.

portée imbriquée Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction 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 Ancien nom pour l'implémentation actuelle des classes, pour tous les objets. Dans les anciennes versions de Python, seules les nouvelles classes pouvaient utiliser les nouvelles fonctionnalités telles que `__slots__`, les descripteurs, les propriétés, `__getattribute__()`, les méthodes de classe et les méthodes statiques.

objet N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

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

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* Appelable dans la liste `sys.path_hook` qui donne un *chercheur d'entrée dans path* s'il sait où trouver des modules pour une *entrée dans path* donnée.

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 Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Les développeurs peuvent utiliser la fonction `sys.getrefcount()` pour obtenir le nombre de références à un objet donné.

paquet classique *paquet* traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.
Voir aussi *paquet-espace de nommage*.

__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 *itérable* qui offre un accès efficace à ses éléments par un indice sous forme de nombre entier via la méthode spéciale `__getitem__()` et qui définit une méthode `__len__()` donnant sa taille. Voici quelques séquences natives : `list`, `str`, `tuple`, et `bytes`. Notez que `dict` possède aussi une méthode `__getitem__()` et une méthode `__len__()`, mais il est considéré comme un *mapping* plutôt qu'une séquence, car ses accès se font par une clé arbitraire *immuable* plutôt qu'un nombre entier.
La classe abstraite de base `collections.abc.Sequence` définit une interface plus riche qui va au-delà des simples `__getitem__()` et `__len__()`, en ajoutant `count()`, `index()`, `__contains__()` et `__reversed__()`. Les types qui implémentent cette interface étendue peuvent s'enregistrer explicitement en utilisant `register()`.

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.

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`.

référence forte Dans l'API C de Python, une référence forte est une référence vers un objet qui incrémente son compteur de références lorsqu'elle est créée et le décrémente lorsqu'elle est effacée.

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*.

chaîne entre triple guillemets Chaîne qui est délimitée par trois guillemets simples (`'`) ou trois guillemets doubles (`"`). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

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.

Les indications de type sont facultatives et ne sont pas indispensables à l'interpréteur Python, mais elles sont utiles aux outils d'analyse de type statique et aident les IDE à compléter et à réusiner (*code refactoring* en anglais) le code.

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

À propos de ces documents

Ces documents sont générés à partir de sources en reStructuredText par Sphinx, un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page reporting-bugs qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

— Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
— le projet Docutils pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
— Fredrik Lundh pour son projet *Alternative Python Reference*, dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez Misc/ACKS dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !

ANNEXE C

Histoire et licence

C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir https://www.cwi.nl/) aux Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir https://www.cnri.reston.va.us/) de Reston, en Viriginie, d'où il diffusa plusieurs versions du logiciel.

En mai 2000, Guido et l'équipe de développement centrale de Python sont partis vers BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe de PythonLabs est partie vers Digital Creations (désormais Zope Corporation ; voir https://www.zope.com/). En 2001, la Python Software Foundation (PSF, voir https://www.python.org/psf/) voit le jour. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Zope Corporation en est un sponsor.

Toutes les versions de Python sont Open Source (voir https://www.opensource.org/ pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	2.1.1	2001-maintenant	PSF	oui

Note : Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non-compatibles GPL » ne le peuvent pas.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

Le logiciel Python et sa documentation sont distribués sous *la licence d'utilisation PSF*.

Depuis Python 3.8.6, les exemples, recettes et autres codes présents dans la documentation sont sous la double licence d'utilisation PSF et *la licence Zero-Clause BSD*.

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 PSF LICENSE AGREEMENT FOR PYTHON 3.11.3

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.11.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.11.3 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.11.3 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 3.11.3 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
   3.11.3.

4. PSF is making Python 3.11.3 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 3.11.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.3
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.3, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.11.3, 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   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
   ("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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 LICENCE BSD ZERO-CLAUSE POUR LE CODE DANS LA DOCUMENTATION DE PYTHON 3.11.3

```
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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Le module `_random` inclut du code construit à partir d'un téléchargement depuis http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. Voici mot pour mot les commentaires du code original :

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

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

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

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

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
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

Les modules `asynchat` et `asyncore` contiennent la note suivante :

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Le module `uu` contient la note suivante :

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Le module `test_epoll` contient la note suivante :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
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:
```

(suite sur la page suivante)

(suite de la page précédente)

```
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/

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

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et macOS peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
```

(suite sur la page suivante)

(suite de la page précédente)

```
    *    the documentation and/or other materials provided with the
    *    distribution.
    *
    * 3. All advertising materials mentioning features or use of this
    *    software must display the following acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
    *
    * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
    *    endorse or promote products derived from this software without
    *    prior written permission. For written permission, please contact
    *    openssl-core@openssl.org.
    *
    * 5. Products derived from this software may not be called "OpenSSL"
    *    nor may "OpenSSL" appear in their names without prior written
    *    permission of the OpenSSL Project.
    *
    * 6. Redistributions of any form whatsoever must retain the following
    *    acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
```

(suite sur la page suivante)

(suite de la page précédente)

```
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

Le module `pyexpat` est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
```

(suite sur la page suivante)

(suite de la page précédente)

```
"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

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 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.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

Le module `_decimal` est construit en incluant une copie de la bibliothèque *libmpdec*, sauf si elle est compilée avec `--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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
  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,
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 Audioop

The audioop module uses the code base in g771.c file of the SoX project :

```
Programming the AdLib/Sound Blaster
FM Music Chips
Version 2.0 (24 Feb 1992)
Copyright (c) 1991, 1992 by Jeffrey S. Lee
jlee@smylex.uucp
Warranty and Copyright Policy
This document is provided on an "as-is" basis, and its author makes
no warranty or representation, express or implied, with respect to
its quality performance or fitness for a particular purpose.  In no
event will the author of this document be liable for direct, indirect,
special, incidental, or consequential damages arising out of the use
or inability to use the information contained within.  Use of this
document is at your own risk.
This file may be used and copied freely so long as the applicable
copyright notices are retained, and no modifications are made to the
text of the document.  No money shall be charged for its distribution
beyond reasonable shipping, handling and duplication costs, nor shall
proprietary changes be made to this document so that it cannot be
distributed freely.  This document may not be included in published
material or commercial packages without the written consent of its
author.
```

ANNEXE D

Copyright

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

Index

Non alphabétique

A

B

C

D

E

P

R

S

T

V