ESIEE PARIS

IGI-3008

Tutoriel

Traduit et adapté de :

The Python Tutorial

`https://docs.python.org/3.4/tutorial/index.html`

Jean-Claude GEORGES

Année scolaire 2016–2017

Table des matières

Introduction

Python est un langage de programmation à la fois puissant et facile à prendre en main.

Il propose des structures de données de haut niveau performantes et permet d'aborder de manière simple, mais efficace, la programmation orientée objet.

Sa syntaxe élégante et le typage dynamique, joints à sa nature interprétée, font de Python un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur la plupart des plates-formes.

L'interpréteur Python et sa bibliothèque standard très fournie sont disponibles gratuitement (programmes sources et exécutables) pour toutes les grandes plates-formes à partir du site web Python **https://www.python.org/** et ils peuvent être distribués librement. Ce site contient également des liens vers de nombreux modules, programmes et outils Python, et vers de la documentation supplémentaire.

L'interpréteur Python est facilement extensible à l'aide de nouvelles fonctions ou structures de données, écrites en C ou en C++ (ou en d'autres langages compatibles avec C). Python est également adapté en tant que langage d'extension pour des applications personnalisables.

Ce tutoriel propose au lecteur une introduction informelle aux concepts et caractéristiques du langage Python. Il permet de prendre en main l'interpréteur Python à l'aide d'exemples pratiques. Et comme tous ces exemples sont autonomes, ce tutoriel peut également être lu hors-ligne.

On peut consulter avec profit (pages en anglais) :

- The Python Standard Library pour une description complète des objets et modules de base ;
- The Python Language Reference pour une définition formelle du langage ;
- Extending and Embedding the Python Interpreter pour écrire des extensions à Python ;
- Python/C API Reference Manual pour lier des programmes Python et des programmes C/C++.

Il existe également de nombreux livres (ou sites) décrivant Python en profondeur [1].

Ce tutoriel ne prétend pas couvrir toutes les caractéristiques de Python, ni même les plus utilisées. Il en présente plutôt les éléments les plus remarquables, et vous donnera une bonne idée du style et de la philosophie Python. Après l'avoir lu, vous serez capable d'écrire des modules et des programmes, et vous serez prêt à en apprendre davantage sur les modules de la bibliothèque Python décrits dans The Python Standard Library

Il est également conseillé de parcourir le glossaire (Glossary).

1. Quelques livres ou liens :
   - *Introduction to Computer Science Using Python : A Computational Problem-Solving Focus*, Charles Dierbach, (Wiley) : de loin le meilleur, mais trop cher (220 €...)
   - *Apprendre à programmer avec Python*, Gérard Swinnen (Eyrolles) : le plus abordable (et en français...), basé sur *How to Think Like a Computer Scientist*.
     Disponible en fichier pdf libre : http://inforef.be/swi/download/apprendre_python3_5.pdf
   - *How to Think Like a Computer Scientist : Learning with Python*, Allen Downey, Jeff Elkner and Chris Meyers (Green Tea Press) : un classique, très pédagogique.
     Disponible en version interactive : http://interactivepython.org/runestone/static/thinkcspy/index.html
   - *Beginning Python : From Novice to Professional*, Magnus Lie Hetland (APress) : excellent livre pour les débutants.
   - *Learning Python, 5th Edition*, Mark Lutz (O'Reilly Media) : bon livre, mais titre trompeur. Ne convient pas à des programmeurs débutants.
   - https://www.codingame.com/home : pour s'améliorer en programmation. Une multitude d'exercices (plusieurs dizaines de langages, confrontation avec d'autres utilisateurs,).
   - http://www.checkio.org/ : jeu dans lequel vous devez écrire des programmes. Lorsque vous avez résolu un problème, vous avez accès aux solutions des autres joueurs, ce qui est très formateur.
   - http://www.pythonchallenge.com/ : jeu d'énigmes, dans lequel chaque niveau peut être résolu avec un programme Python.
   - http://www.france-ioi.org/ : site d'entraînement à la programmation et l'algorithmique (en français)
   - http://www.pythontutor.com/visualize.html : permet d'exécuter pas à pas (en visualisant la mémoire) vos (petits) programmes Python.

Chapitre 1

Pour vous ouvrir l'appétit

Si vous utilisez souvent un ordinateur, il se peut que vous exécutiez des tâches que vous aimeriez automatiser. Par exemple, vous voudriez faire des *chercher-remplacer* dans de nombreux fichiers de texte, ou renommer et classer de façon complexe un grand nombre de photographies numériques. Peut-être voudriez-vous écrire une petite base de données personnalisée, ou une application graphique spécialisée, ou même un simple jeu.

Si vous êtes un développeur informatique professionnel, vous devez peut-être travailler avec des bibliothèques de programmes écrits en C, C++ ou Java, mais vous trouvez que le cycle de développement habituel *écriture/compilation/tests* est trop lent. Ou alors vous devez écrire un jeu de tests pour les fonctions de ces bibliothèques et vous trouvez que son écriture est une tâche fastidieuse. Ou encore vous avez écrit un programme susceptible d'utiliser un autre langage, et vous ne voulez pas reconcevoir et reprogrammer votre application dans ce nouveau langage.

Python est le langage qu'il vous faut.

Vous pourriez écrire un *shell script* Unix ou des fichiers *batch* Windows pour une partie de ces tâches, mais si les shell scripts sont bien adaptés pour déplacer des fichiers ou modifier des données textuelles, ils ne conviennent pas pour des applications graphiques ou des jeux. Vous pourriez écrire un programme en C, C++ ou Java, mais cela prend un temps énorme de développement rien que pour avoir le premier prototype. Python est plus simple d'utilisation, disponible pour Windows, Mac OS X et Unix, et vous aidera à réaliser plus rapidement votre travail.

Python est simple d'utilisation, mais c'est un vrai langage de programmation, proposant beaucoup plus de structures et d'assistance pour des programmes importants que les shell scripts ou les fichiers batch. D'autre part, Python offre plus de possibilités de tests que C, et, comme c'est un *langage de haut niveau*, il intègre des types de données de haut niveau, comme des tableaux dynamiques ou des mémoires associatives. Grâce à ses types de données plus génériques, Python peut s'appliquer dans de plus nombreux domaines que Awk ou même Perl, tout en permettant de réaliser un grand nombre de tâches aussi simplement que dans ces langages.

Python permet de découper votre programme en modules réutilisables par d'autres programmes Python. Il intègre une grande quantité de modules standard, utilisables en tant que base de vos programmes, ou comme exemples pour apprendre à programmer avec Python. Certains de ces modules fournissent des fonctionnalités comme les entrées/sorties sur fichier, les appels système, les *sockets* ou même des interfaces vers des outils de créations d'applications graphiques comme Tk.

Python est un langage interprété, ce qui permet de gagner du temps dans le développement de programmes, car ni compilation, ni édition de liens ne sont nécessaires. L'interpréteur peut être lancé dans une session interactive, ce qui facilite l'expérimentation de fonctionnalités du langage, en écrivant des programmes *jetables* ou en testant des fonctions durant le développement. Python peut également servir de calculatrice.

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

- les types de données de haut niveau permettent l'expression d'opérations compliquées avec une seule instruction ;
- le regroupement d'instructions se fait par indentation au lieu de parenthésage ouvrant et fermant ;
- les déclarations de variables ou d'arguments ne sont pas nécessaires.

Python est *extensible* : si vous savez programmer en C, vous pourrez facilement ajouter de nouvelles fonctions ou de nouveaux modules qui s'intégreront à l'interpréteur, soit pour permettre de réaliser des opérations critiques à vitesse maximale, soit pour lier des programmes Python à des bibliothèques de fonctions disponibles uniquement sous forme binaire (comme une bibliothèque graphique spécifique à un fournisseur). Lorsque vous serez vraiment un pro, vous pourrez relier l'interpréteur Python à une application écrite en C et l'utiliser comme une extension ou un langage de commande pour cette application.

Notez au passage que le nom du langage a pour origine Monty Python's Flying Circus, une émission de

télévision sur la BBC, et n'a aucun rapport avec les reptiles. Faire référence aux sketches des Monty Python dans vos documentations est non seulement autorisé, mais recommandé !

Maintenant qu'on vous a mis l'eau à la bouche avec Python, vous allez vouloir l'examiner plus en détail. Et comme le meilleur moyen d'apprendre un langage, c'est de le pratiquer, ce tutoriel vous invite à jouer avec l'interpréteur Python au fur et à mesure de votre lecture.

Le chapitre suivant vous expliquera les règles d'utilisation de l'interpréteur. Ce sont des informations un peu ennuyeuses, mais elles sont essentielles pour pouvoir entrer les exemples à venir.

La suite de ce tutoriel présentera des caractéristiques diverses et variées du langage Python et de sa philosophie, en commençant par des expressions, des instructions et des types de données simples, puis en introduisant les fonctions et les modules, pour finir par aborder des concepts avancés comme les exceptions et les classes définies par l'utilisateur.

Chapitre 2

Utilisation de l'interpréteur Python

2.1 Lancer l'interpréteur

Sous Unix, l'interpréteur Python est installé en général sous le nom complet **`/usr/local/bin/python3.4`** ; l'ajout de **`/usr/local/bin`** dans votre chemin de recherche Unix permettra de le lancer en entrant dans un terminal la commande suivante [1] :

```
$ python3.4
```

Comme le répertoire de l'interpréteur est une option choisie lors de l'installation, il se peut qu'il se situe à un autre endroit ; vérifiez auprès de votre gourou Python ou votre administrateur système (par exemple, **`/usr/local/python`** est un autre emplacement souvent choisi.)

Sur une machine Windows, l'installation de Python se fait habituellement dans **`C:\Python34`**, bien que vous puissiez modifier ce nom de répertoire lorsque vous exécutez l'installateur. Pour ajouter ce répertoire à votre chemin d'accès aux fichiers, vous pouvez entrer la commande suivante dans un terminal DOS :

```
C:\> set path=\%path\%;C:\python34
```

La frappe du caractère de fin de fichier (**`Control-D`** sous Unix, **`Control-Z`** sous Windows) à l'invite de commande principale fait quitter l'interpréteur avec 0 comme valeur de retour. Si cela ne fonctionne pas, vous pouvez quitter l'interpréteur avec la commande **`quit()`**.

Les fonctionnalités d'édition de ligne de l'interpréteur comprennent l'édition interactive, un système d'historique et le complètement (*completion*) automatique de code sur les systèmes prenant en charge l'édition de ligne. Le moyen le plus rapide de tester si l'édition de la ligne de commande est disponible est d'entrer **`Control-P`** sur la première invite de commande. Si vous entendez un bip, c'est que vous disposez de l'édition de ligne de commande (voir le chapitre 14 *Édition de ligne de commande et historique* pour une introduction aux rôles des touches). Si rien ne se passe, ou si **`Control-P`** apparaît à l'écran, l'édition de la ligne de commande est impossible ; il ne vous reste que la touche *retour arrière* (**`backspace`**) pour supprimer les derniers caractères de la ligne.

L'interpréteur Python fonctionne de la même façon que le *shell* Unix : lorsqu'il est invoqué avec l'entrée standard connectée à un terminal, il lit et exécute les commandes interactivement ; lorsqu'il est invoqué avec un argument nom de fichier ou avec un fichier en tant qu'entrée standard, il lit et exécute un *script* (la suite des instructions de ce fichier).

Une deuxième manière de lancer l'interpréteur Python est **`python -c `*`commande`*` [`*`arguments`*`] ...`**, qui exécute les instructions de la *commande*, comme dans l'option **`-c`** du shell. Comme les instructions Python contiennent souvent des espaces ou d'autres caractères spéciaux, il est généralement recommandé de mettre la *commande* entre simples quotes.

Certain modules Python peuvent également s'utiliser en tant que scripts. Il faut pour cela lancer **`python -m `*`module`*` [`*`arguments`*`] ...`**, qui exécute le fichier source du *module* comme si vous l'aviez lancé directement depuis la ligne de commande.

Lorsqu'un fichier de script est utilisé, il est parfois intéressant de lancer le script, puis d'entrer dans une session interactive. Cela peut être fait en passant l'option **`-i`** avant le nom du script.

Toutes les options de la ligne de commande sont détaillées dans le document *Command line and environment*.

1. Sous Unix, l'interpréteur Python3.X n'est pas installé par défaut sous le nom **`python`**, afin de ne pas entrer en conflit avec un exécutable Python2.x déjà installé.

2.1.1 Passage d'arguments

Quand ils sont connus de l'interpréteur, le nom du script et les arguments qui le suivent sont stockés dans une liste de chaînes de caractères et affectés à la variable **argv** du module **sys**. Il est possible d'accéder à cette liste en exécutant **import sys**. Cette liste contient toujours au moins un élément.

- si ni script, ni arguments ne sont fournis, **sys.argv[0]** est une chaîne vide.
- si le nom du script est **'-'** (pour l'entrée standard), **sys.argv[0]** prend la valeur **'-'**.
- si **-c *commande*** est utilisé, **sys.argv[0]** prend la valeur **'-c'**.
- si **-m *module*** est utilisé, **sys.argv[0]** prend la valeur du nom complet du module.

Les options situées après **-c *commande*** ou **-m *module*** ne sont pas consommés par la partie de l'interpréteur Python qui analyse les options, mais mises dans **sys.argv**, laissant le soin à la commande ou au module de les gérer.

2.1.2 Mode interactif

Lorsque les commandes sont lues depuis un terminal (clavier-écran), l'interpréteur est dit en *mode interactif*. Il affiche son invite de commande primaire, en général trois signes *supérieur à* (**>>>**) ; pour les lignes de suite, il affiche son invite de commande secondaire, par défaut trois points (**...**). Avant la toute première invite, l'interpréteur affiche un message de bienvenue, précisant son numéro de version et une note de copyright :

```
$ python3.4
Python 3.4 (default, Mar 16 2014, 09:25:04)
{[}GCC 4.8.2{]} on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

ou

```
C:\Users\jcg>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les lignes de suites s'avèrent nécessaires pour entrer une construction de code sur plusieurs lignes. Par exemple, jetez un œil à la documentation de l'instruction **if** :

```
>>> la_terre_est_plate = True
>>> if la_terre_est_plate:
...     print("Attention, tu vas tomber !")
...
Attention, tu vas tomber !
```

Pour en savoir plus sur le mode interactif, voir 16.1 *Mode interactif*.

2.2 L'interpréteur et son environnement

2.2.1 Encodage du texte des programmes sources

Par défaut, les fichiers sources Python sont considérés comme encodés en UTF-8. Avec cet encodage, les caractères de la plupart des langues du monde peuvent être utilisés simultanément dans des littéraux chaînes de caractères, des identificateurs ou des commentaires. – bien que la bibliothèque standard n'utilise que des caractères ASCII pour ses identificateurs, convention que tout code portable devrait suivre. Pour pouvoir afficher tous ces caractères correctement, votre éditeur doit reconnaître que le fichier est encodé en UTF-8 et doit utiliser une police qui prend en charge tous les caractères du fichier.

Il est toutefois possible de spécifier un encodage différent pour les fichiers sources. Pour permettre à l'interpréteur Python de le gérer, mettez un commentaire spécial juste après la ligne **#!** pour préciser l'encodage du fichier :

```
# -*- coding: nom_de_l_encodage -*-
```

Après cette déclaration, tout ce qui suit dans le fichier sera considéré comme étant encodé avec *encodage* au lieu de UTF-8. On peut trouver La liste des encodages possibles dans *The Python Library Reference*, à la section **codecs**.

Par exemple, si votre éditeur préféré ne prend pas en charge les fichiers encodés en UTF-8 et insiste pour utiliser d'autres encodages, écrivez :

```
# -*- coding: cp-1252 -*-
```

et continuez à utiliser les caractères de l'encodage Windows-1252 dans vos fichiers sources. Le commentaire concernant les encodages spéciaux doit se trouver à la *première ou deuxième* ligne du fichier.

Chapitre 3

Introduction informelle à Python

Dans les exemples suivants, les saisies et les affichages de distinguent par la présence ou l'absence des invites de commande (voir `>>>` et `...`) : pour reproduire un exemple, vous devez entrer tout ce qui est indiqué après l'invite, lorsqu'elle est présente ; les lignes qui ne commencent pas par une invite sont les affichages de l'interpréteur. Notez qu'une invite secondaire seule sur une ligne indique que vous devez entrer une ligne vide (pour indiquer la fin d'une commande sur plusieurs lignes).

Beaucoup d'exemples de ce manuel, même ceux entrés dans une session interactive, contiennent des commentaires. Les commentaires Python commencent par un caractère *croisillon* (*hash character*, `#`), et se continuent jusqu'à la fin de la ligne courante. Un commentaire peut commencer en début de ligne, ou après des espaces ou du code, mais pas dans une chaîne de caractères entre quotes. Un croisillon dans une chaîne de caractères est simplement un croisillon. Comme les commentaires servent à clarifier le code et qu'ils ne sont pas interprétés par Python, il n'est pas nécessaire de le recopier si vous entrez les exemples sur machine.

Quelques exemples de commentaires [1] :

```
>>> # ceci est le premier commentaire
>>> spam = 1  # et là le deuxième
>>>           # ... et maintenant un troisième !
>>> texte = "# Ceci n'est pas un commentaire (le croisillon est entre quotes)."
```

3.1 Utiliser Python comme une calculatrice

3.1.1 Nombres

L'interpréteur peut servir de calculatrice : on peut entrer une expression après l'invite et la valeur calculée de l'expression s'affichera. La syntaxe d'une expression est simple : les opérateurs `+`, `-`, `*` et `/` jouent le même rôle que dans la plupart des autres langages (par exemple, Pascal ou C) ; les parenthèses `()` sont utilisées pour regrouper des termes. Par exemple :

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # la division retourne toujours un nombre en virgule flottante
1.6
```

Les nombres entiers (comme `2`, `4` ou `20`) sont de type `int`, ceux avec une partie décimale (comme `5.0`, `1.6`) sont de type `float`. Nous reviendrons de manière plus approfondie sur les types numériques.

La division (`/`) retourne toujours un nombre flottant. Pour effectuer une division euclidienne (voir *floor division*) et obtenir un résultat entier (en supprimant la partie décimale du résultat *réel*), on utilise l'opérateur `//` ; pour calculer le reste, on utilise l'opérateur `%` :

```
>>> 17 / 3  # la division classique retourne un flottant
5.666666666666667
>>> 17 // 3  # la division euclidienne supprime la partie décimale
5
>>> 17 % 3  # l'opérateur % retourne le reste de la division
2
>>> 5 * 3 + 2  # quotient * diviseur + reste
```

1. Pour le sketch à l'origine de l'utilisation du terme *spam*, voir la vidéo SPAM

```
17
```

L'opérateur `**` est utilisé pour calculer les puissances [2] :

```
>>> 5 ** 2 # 5 au carré
25
>>> 2 ** 7 # 2 à la puissance 7
128
```

Le signe *égal* (`=`) permet d'affecter une valeur à une variable. L'affectation ne produit pas d'affichage :

```
>>> largeur = 21
>>> hauteur = 29.7
>>> largeur * hauteur # aire d'une feuille A4 en centimètres carrés
623.6999999999999
```

Si une variable n'est pas *définie* (n'a pas été affectée), toute tentative d'utilisation se soldera par une erreur :

```
>>> n # tentative d'utiliser une variable non définie
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python prend en charge les nombres à virgule flottante ; les opérations faisant intervenir des types numériques différents convertissent les opérandes entiers en nombres à virgule flottante :

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

En mode interactif, la dernière valeur affichée est affectée à une variable de nom `_` (l'équivalent de *Ans* d'une calculatrice). Vous pouvez donc, si vous utilisez Python comme calculette, enchaîner les calculs. Par exemple :

```
>>> prix = 28.50
>>> tva = 5.5 / 100
>>> prix * tva
1.5675000000000001
>>> prix + _
30.0675
>>> round(_, 2)
30.07
```

La variable `_` ne devrait être utilisée qu'en consultation. Ne lui affectez jamais de valeur, vous créeriez une variable locale de même nom qui cacherait la variable prédéfinie et ses propriétés *magiques*.

En plus des `int` et des `float`, Python permet d'utiliser d'autres types numériques, comme les décimaux (`Decimal`) et les fractions (`Fraction`). Python intègre également les complexes (*complex numbers*) ; le suffixe `j` ou `J` après un nombre en chiffres permet de spécifier la partie imaginaire (comme dans `3 + 5j`).

3.1.2 Chaînes de caractères

En plus des nombres, Python permet également de manipuler des chaînes de caractères, qui peuvent être entrées de plusieurs façons : les littéraux chaînes (constantes chaînes de caractères) sont délimités par des *quotes* (`'...'`) ou des *double quotes* (`"..."`), pour un résultat identique [3]. Le caractère *backslash* (\) est utilisé pour annuler le rôle délimitant des quotes :

```
>>> 'spam eggs'  #simples quotes
'spam eggs'
>>> 'n\'est-ce pas'  # utiliser \' pour intégrer une simple quote...
"n'est-ce pas"
>>> "N'est-ce pas"  # ... ou encadrer par des doubles quotes
"N'est-ce pas"
>>> '"Oui," dit-il.'
'"Oui," dit-il.'
>>> "\"Oui,\" dit-il."
'"Oui," dit-il.'
```

2. Comme `**` est plus prioritaire que `-`, `-3**2` est interprété comme `-(3**2)` dont la valeur calculée est `-9`. Pour obtenir la valeur `9`, il faut écrire `(-3)**2`.

3. Contrairement à d'autres langages, les caractères spéciaux comme `\n` ont la même signification, qu'ils soient placés entre simples quotes (`'...'`) ou double (`"..."`) quotes. La seule différence entre les deux est qu'il n'est pas nécessaire d'échapper `"` entre des simples quotes (mais il faut le faire pour `\'`) et vice versa.

```
>>> '"N\'est-ce pas ?" dit-elle.'
'"N\'est-ce pas ?" dit-elle.'
```

En mode interactif, la chaîne résultat est affichée entre quotes et les caractères spéciaux sont *échappés* avec des *backslashes*. Bien que cela puisse les faire apparaître comme différentes des chaînes entrées (les quotes délimitrices peuvent changer), les deux chaînes sont équivalentes. Une chaîne est toujours affichée entre simple quotes, sauf si elle contient une simple quote et pas de double quote, auquel cas elle est affichée entre double quotes. La fonction **`print()`** génère un affichage plus lisible, en supprimant les délimiteurs et en affichant les caractères spéciaux :

```
>>> '"N\'est-ce pas ?" dit-elle.'
'"N\'est-ce pas ?" dit-elle.'
>>> print('"N\'est-ce pas ?" dit-elle.')
"N'est-ce pas ?" dit-elle.
>>> s = 'Première ligne.\nDeuxième ligne.'  # \n : passage à la ligne
>>> s  # sans print(), \n s'affiche
'Première ligne.\nDeuxième ligne.'
>>> print(s)  # avec print(), \n produit un retour ligne
Première ligne.
Deuxième ligne.
```

Si on ne veut pas que les caractères précédés par **`\`** soient interprétés comme des caractères spéciaux, on peut utiliser des chaînes brutes (*raw strings*) en ajoutant un **`r`** devant la première quote :

```
>>> print('C:\classe\notes')
C:\classe
otes
>>> print(r'C:\classe\notes')
C:\classe\notes
```

Les littéraux chaînes peuvent s'étendre sur plusieurs lignes. Une façon de le faire est de les déliminer par des triple-quotes : **`"""..."""`** ou **`'''...'''`**. Les fins de lignes sont automatiquement insérées dans la chaîne, mais il est possible de l'empêcher en ajoutant un **`\`** en fin de ligne. L'exemple suivant :

```
>>> a = """\
... Usage: appli_geniale [OPTIONS]
...      -h                        Affiche ce message
...      -c  nom_serveur           Serveur auquel se connecter
... """
```

génère avec **`print()`** l'affichage suivant (notez que le premier saut de ligne n'est pas inclus) :

```
>>> print(a)
Usage: appli_geniale [OPTIONS]
     -h                        Affiche ce message
     -c  nom_serveur           Serveur auquel se connecter
```

Les chaînes peuvent être concaténées (mises bout à bout) à l'aide de l'opérateur **`+`** et répétées avec **`*`** :

```
>>> # 2 fois 'oua', suivi de 'ron'
>>> 2 * 'oua'  + 'ron'
'ouaouaron'
```

Deux ou plusieurs *littéraux chaînes* (c'est-à-dire entre quotes) voisins sont automatiquement concaténés.

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

Cela ne fonctionne que pour les littéraux, pas pour les variables ou les expressions :

```
>>> préfixe = 'Py'
>>> préfixe 'thon'  # pas de concaténation variable et constante
  File "<stdin>", line 1
    préfixe 'thon'  # pas de concaténation variable et constante
                ^
SyntaxError: invalid syntax
>>> (2 * 'oua') 'ron'
  File "<stdin>", line 1
    (2 * 'oua') 'ron'
                ^
SyntaxError: invalid syntax
```

Pour concaténer des variables, ou une variable et un littéral, il faut utiliser l'opérateur **`+`** :

```
>>> préfixe + 'thon'
'Python'
```

Cette possibilité est particulièrement intéressante pour écrire par morceaux de longues chaînes :

```
>>> texte = ('On met plusieurs lignes '
...             'entre parenthèses '
...             'pour les relier')
>>> texte
'On met plusieurs lignes entre parenthèses pour les relier'
```

Les chaînes peuvent être indexées (*indicées*) en commençant la numérotation à 0. Il n'y a pas comme en C de type *caractère seul* (un caractère isolé est simplement une chaîne de longueur égale à un) :

```
>>> mot = 'Python'
>>> mot[0]  # caractère en position 0
'P'
>>> mot[5]  # caractère en position 5
'n'
```

Les indices négatifs permettent de compter de droite à gauche :

```
>>> mot[-1]  # dernier caractère
'n'
>>> mot[-2]  # avant-dernier caractère
'o'
>>> mot[-6]
'P'
```

Notez que comme -0 est égal à 0, les indices négatifs commencent à -1.

En plus de l'indexation, la découpe (*slicing*) est également disponible. Alors que l'indexation permet d'obtenir des caractères simples, la découpe permet d'obtenir des sous-chaînes :

```
>>> mot[0:2]  # caractères de la position 0 (incluse) à 2 (exclue)
'Py'
>>> mot[2:5]  # caractères de la position 2 (incluse) à 5 (exclue)
'tho'
```

Notez que le caractère de début est systématiquement inclus dans la découpe, et que le caractère de fin est exclu. Cela permet d'assurer que **`s[:i] + s[i:]`** est toujours égal à **`s`** :

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

Les indices de découpe ont des valeurs par défaut intéressantes : si le premier indice est omis, on part du début de la chaîne, si le deuxième indice est omis, on termine en fin de la chaîne.

```
>>> mot[:2]  # du début jusqu'à la position 2 (exclue)
'Py'
>>> mot[4:]  # de la position 4 (incluse) jusqu'à la fin
'on'
>>> mot[-2:] # de l'avant-dernier (inclus) jusqu'à la fin
'on'
```

Un moyen de se rappeler comment les indices de découpe fonctionnent est de considérer que ces indices marquent les séparations entre caractères, avec le bord gauche du premier caractère numéroté 0. Le bord droit du dernier caractère d'une chaîne de longueur *n* aura alors comme index *n*. Par exemple :

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

Dans le tableau ci-dessus, la première ligne de nombres donne la position des index de 0 à 6 dans la chaîne ; la deuxième ligne donne les index négatifs correspondants. Ainsi, la découpe de *i* à *j* sera constituée de tous les caractères entre le bord *i* et le bord *j*.

Pour les indices non-négatifs, la longueur de la découpe est la différence des indices, s'ils sont tous deux dans les limites de la chaîne. Par exemple, la longueur de **`mot[1:3]`** est 2.

Utiliser un index trop grand conduit à une erreur :

```
>>> mot[42]  # le mot n'a que 6 caractères
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Cependant, des indices de découpe hors limite sont gérés élégamment lors de ces découpes :

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

Les chaînes Python ne peuvent pas être modifiées — elles sont dites figées (voir *immutable*). Aussi, une tentative d'affectation à une position d'index conduira à une erreur :

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

Si une chaîne différente s'avère nécessaire, il faut en créer une nouvelle :

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

La fonction prédéfinie `len()` retourne la longueur de la chaîne :

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

Voir aussi :

`str` – type séquence de caractères (*Text Sequence Type — str*)

Les chaînes sont des exemples de types *séquence*, et disposent des opérateurs classiques des séquences.

Méthodes de chaînes (*String Methods*)

Les chaînes disposent d'un grand nombre de méthodes permettant les transformations et les recherches de bases.

Formatage de chaînes (*String Formatting*)

Description des formatages de chaînes avec `str.format()`.

Formater dans le style printf (*printf-style String Formatting*)

Description du vieux style de formatage, lorsque les chaînes et les chaînes Unicode étaient les opérandes gauche de l'opérateur `%`.

3.1.3 Listes

Python possède plusieurs types de données *composées*, permettant de regrouper diverses valeurs. Le plus utilisé de ces types est la *liste*, qui s'écrit comme une suite entre crochets de valeurs (items) séparées par des virgules. Les listes peuvent être composés d'items de différents types, mais en général, ils ont tous le même type.

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

Comme les chaînes de caractères, et tous les autres objets de type séquence (voir *sequence*), les listes peuvent être indexées et découpées :

```
>>> carrés[0]  # l'indexation retourne l'élément
1
>>> carrés[-1]
25
>>> carrés[-3:]  # la découpe (slicing) retourne une nouvelle liste
[9, 16, 25]
```

Toutes les opérations de découpe retournent une nouvelle liste contenant les éléments demandés. Ainsi, la découpe suivante retourne une nouvelle liste, copie superficielle (*shallow copy*) de la liste entière :

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

Les listes permettent également la concaténation :

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

Contrairement aux chaînes, qui sont figées (voir *immutable*), les listes sont modifiables (voir *mutable*), c'est-à-dire qu'il est possible de changer leur contenu :

```
>>> cubes = [1, 8, 27, 65, 125]  # il y a une faute
>>> 4 ** 3  # le cube de 4, c'est 64, pas 65 !
64
>>> cubes[3] = 64  # remplace la mauvaise valeur
>>> cubes
[1, 8, 27, 64, 125]
```

On peut également ajouter de nouveaux items en fin de liste, en utilisant la *méthode* **`append()`** (nous reviendrons sur les méthodes plus tard) :

```
>>> cubes.append(216)  # ajoute le cube de 6
>>> cubes.append(7 ** 3)  # et le cube de 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Les affectations sont possibles sur les découpes de listes, ce qui permet même de modifier leur taille ou de les vider entièrement :

```
>>> lettres = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> lettres
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # remplace quelques valeurs
>>> lettres[2:5] = ['C', 'D', 'E']
>>> lettres
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # maintenant,les supprime
>>> lettres[2:5] = []
>>> lettres
['a', 'b', 'f', 'g']
>>> # vide la liste en remplaçant tous les éléments par une liste vide
>>> lettres[:] = []
>>> lettres
[]
```

La fonction prédéfinie **`len()`** s'applique également aux listes :

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

Il est possible d'imbriquer les listes (créant ainsi 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 sûr, Python permet d'effectuer des tâches plus complexes que d'additionner deux et deux. Par exemple, nous pouvons afficher le début de la suite de Fibonacci comme suit :

```
>>> # La suite de Fibonacci :
>>> # la somme des deux derniers éléments définit le suivant
>>> a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Cet exemple présente plusieurs nouvelles caractéristiques.

- La première ligne comporte une *affectation multiple* : les variables **a** and **b** reçoivent simultanément les nouvelles valeurs 0 et 1. Sur la dernière ligne, on réutilise cette possibilité, et on voit que les expressions à droite du signe **=** sont évaluées avant que les affectations n'aient lieu. Ces expressions de droite sont évaluées de gauche à droite.
- La boucle **while** exécute ses instructions tant que la condition (ici, **b < 10**) est vraie. En Python, comme en C, toute valeur non nulle est vraie ; zéro est faux. La condition peut être également une chaîne ou une liste, ou toute séquence ; tout ce qui est de longueur différente de 0 est vrai, toute séquence vide est fausse. Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs standard de comparaison sont écrits comme en C : **<** (inférieur à), **>** (supérieur à), **==** (égal à), **<=** (inférieur ou égal à), **>=** (supérieur ou égal à) et **!=** (différent de).
- Le *corps* de la boucle est *indenté* : l'indentation est la façon dont Python regroupe les instructions. Sur l'invite de commande interactive, il faut saisir une tabulation ou des espaces pour chaque ligne indentée. En pratique, vous écrirez des programmes plus complexes en utilisant un éditeur de texte ; tous les éditeurs de texte décents possèdent une possibilité d'indentation automatique. Lorsqu'une instruction composée est entrée en mode interactif, elle doit être suivie par une ligne vide, pour signifier la fin de l'instruction (car l'analyseur de Python ne peut pas deviner que vous venez de taper la dernière instruction du groupe). Notez que chaque ligne dans un groupe doit être indentée de la même largeur.
- La fonction **print()** affiche les valeurs des arguments qui lui sont transmis. Elle diffère de l'affichage d'expressions que nous avons utilisé jusqu'ici, car elle permet de gérer plusieurs arguments, des valeurs en virgule flottante et des chaînes. Les chaînes sont affichées sans leurs quotes, et des espaces sont insérées entre les items, ce qui permet de faire des affichages élégants, comme ici :

  ```
  >>> i = 256*256
  >>> print('La valeur de i est', i)
  La valeur de i est 65536
  ```

  Le mot clé *end* peut être utilisé comme argument, pour éviter un retour à la ligne après chaque affichage, ou pour terminer l'affichage avec une chaîne différente :

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

Chapitre 4

Contrôle des instructions

Aux côtés de l'instruction **while** que l'on vient de voir, Python permet d'utiliser des instructions de contrôle classiques, communes à d'autres langages, avec quelques particularités.

4.1 La conditionnelle if

La plus connue des instructions de contrôle est sans doute l'instruction **if**. Par exemple :

```
>>> x = int(input("Entrez une valeur entière : "))
Entrez une valeur entière : 42
>>> if x < 0:
...     x = 0
...     print('Nombre négatif interdit. Modifié en 0')
... elif x == 0:
...     print('Zéro')
... elif x == 1:
...     print('Unique')
... else:
...     print('Plusieurs')
...
Plusieurs
```

On peut trouver zéro, une ou plusieurs parties **elif**, et la partie **else** est optionnelle. Le mot-clef '**elif**' est un raccourci pour '**else if**', et permet d'éviter un trop grand nombre d'indentations. Une suite **if**...**elif**...**elif**... permet de remplacer les instructions **switch** ou **case** disponibles dans d'autres langages.

4.2 La boucle for

L'instruction **for** de Python diffère de celles de C ou de Pascal. Plutôt que d'itérer systématiquement sur une progression arithmétique (comme en Pascal), ou de permettre au programmeur de définir à la fois l'étape de progression et la condition d'arrêt (comme en C), la boucle Python **for** parcourt les items d'une séquence (liste, chaîne, etc.), dans l'ordre d'apparition dans la séquence. Par exemple (et sans jeu de mot [1]).

```
>>> # prendre des mesures
>>> liste_de_mots = ['cat', 'window', 'defenestrate']
>>> for mot in liste_de_mots:
...     print(mot, len(mot))
...
cat 3
window 6
defenestrate 12
```

S'il est nécessaire de modifier la séquence sur laquelle on itère pendant l'éxécution de la boucle (par exemple pour dupliquer certains items), il est recommandé de faire d'abord une copie de la séquence. En effet, itérer sur une séquence ne fait pas une copie de la séquence. La syntaxe de découpe (*slice*) est particulièrement utile dans ce cas :

```
>>> for mot in liste_de_mots[:]:  # boucle sur une découpe copie de la liste
...     if len(mot) > 6:
...         liste_de_mots.insert(0, mot)
...
```

1. *defenestrate : v. To remove a Windows operating system from a computer.* https://www.wordnik.com/words/defenestrate

```
>>> liste_de_mots
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 La fonction `range()`

Pour itérer sur une suite de nombres, la fonction prédéfinie `range` est bien pratique ; elle génère les éléments d'une suite arithmétique :

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

La limite donnée ne fait pas partie des nombres générés ; `range(10)` génère 10 valeurs, les nombres permettant d'indexer une séquence de longueur 10 (de 0 à 9). Il est possible d'initialiser le *début* avec un autre nombre que 0, ou de spécifier un autre incrément (même négatif) que 1 :

```
>>> range(5, 10)
    de 5 à 9
>>> range(0, 10, 3)
    0, 3, 6, 9
>>> range(-10, -100, -30)
    -10, -40, -70
```

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

```
>>> a = ['Il', 'était', 'une', 'bergère']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Il
1 était
2 une
3 bergère
```

Toutefois, dans la plupart de ces genres de cas, il est plus indiqué d'utiliser la fonction `enumerate`, voir 5.6 *Techniques de boucles*.

Il arrive quelque chose de bizarre si on essaie d'afficher un *range* :

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

Un objet retourné par la fonction `range()` se comporte presque comme une liste, mais ce n'est pas une liste. C'est un objet *générateur* qui, lorsqu'on itère sur lui, retourne un à un les items de la séquence désirée, mais ne crée pas la liste complète, ce qui permet d'économiser de la mémoire.

On dit d'un tel objet qu'il est *itérable*, c'est-à-dire utilisable comme fournisseur pour des fonctions ou des constructions qui attendent quelque chose qui puisse leur donner des items un par un, jusqu'à épuisement des stocks. Nous avons vu que l'instruction `for` est un tel *itérateur*. La fonction `list()` en est un autre : elle crée des listes à partir d'itérables :

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Nous verrons plus loin d'autres fonctions qui retournent des itérables ou acceptent des itérables en argument.

4.4 Les instructions `break` et `continue` et la clause `else` sur les boucles

L'instruction `break`, comme en C, permet de sortir d'une boucle `for` ou `while` (la plus interne en cas de boucles imbriquées).

Les instructions de boucles peuvent se terminer par une clause `else` ; elle sera exécutée lorsque la boucle se terminera *normalement* (fin de la liste dans une boucle `for` ou condition devenue fausse dans une boucle

while), mais n'est pas éxécutée si la boucle se termine par un **break**. Ceci est montré dans l'exemple suivant qui recherche des nombres premiers :

```
>>> for n in range(2001, 2020, 2):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'est égal à', x, '*', n//x)
...             break
...     else:
...         # sortie de boucle sans avoir trouvé de diviseur
...         print(n, 'est un nombre premier')
...
2001 est égal à 3 * 667
2003 est un nombre premier
2005 est égal à 5 * 401
2007 est égal à 3 * 669
2009 est égal à 7 * 287
2011 est un nombre premier
2013 est égal à 3 * 671
2015 est égal à 5 * 403
2017 est un nombre premier
2019 est égal à 3 * 673
```

(Oui, ce code est correct. Regardez attentivement : la clause **else** appartient à la boucle **for**, pas à la conditionnelle **if**.)

Lorsqu'elle est utilisée avec une boucle, la clause **else** ressemble davantage à la clause **else** d'une instruction **try** qu'à celle d'une instruction **if** : une clause **else** d'une instruction **try** s'éxécute si aucune exception n'est rencontrée, et une clause **else** d'une boucle s'éxécute si aucun **break** n'est rencontré. pour en savoir plus sur l'instruction **try** et les exceptions, voir 8.3 *Gérer les exceptions*.

L'instruction **continue**, également inspirée de C, continue au prochain tour de boucle :

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Nombre pair trouvé :", num)
...         continue
...     print("Nombre trouvé :", num)
...
Nombre pair trouvé : 2
Nombre trouvé : 3
Nombre pair trouvé : 4
Nombre trouvé : 5
Nombre pair trouvé : 6
Nombre trouvé : 7
Nombre pair trouvé : 8
Nombre trouvé : 9
```

4.5 L'instruction **pass**

L'instruction **pass** ne fait rien. On l'utilise lorsque la syntaxe exige une instruction, mais qu'aucune action n'est à exécuter. Par exemple :

```
>>> while True:
...     pass  # Attente active d'une interruption clavier
...
```

On l'utilise souvent pour créer des classes minimales :

```
>>> class MaClasseVide:
...     pass
...
```

Un autre endroit où l'instruction **pass** est utilisée est un *marque-page* pour une fonction ou un bloc d'instructions sur laquel vous allez devoir travailler plus tard, et qui vous permettra de continuer à réfléchir à un niveau plus abstrait. L'instruction **pass** est tout simplement ignorée :

```
>>> def initlog(*args):
...     pass   # À faire plus tard !!
...
```

4.6 Définitions de fonctions

On peut écrire une fonction qui affiche les éléments d'une suite de Fibonacci inférieurs à une certaine valeur :

```
>>> def fib(n):    # écrit la suite de Fibonacci jusqu'à n
...     """Affiche la suite de Fibonacci jusqu'à n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Appel à la fonction que l'on vient de définir:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Le mot-clef **def** introduit une *définition* de fonction. Il doit être suivi par un nom de fonction et une suite de paramètres formels séparés par des virgules et entre parenthèses. Les instructions formant le corps de la fonction commencent à la ligne suivante et doivent être indentées.

La première instruction dans le corps de la fonction peut être une constante chaîne de caractères ; c'est la documentation de la fonction (*docstring*) (pour en savoir plus sur les *dostrings*, voir la section 4.7.6 *Chaînes de documentation*). Il existe des outils qui utilisent les *dostrings* pour générer automatiquement des documentations en ligne ou à imprimer, ou pour permettre à l'utilisateur de naviguer dans le programme ; c'est une bonne pratique que d'intégrer des *docstrings* dans le code que l'on écrit, prenez l'habitude de le faire.

L'*exécution* d'une fonction introduit une nouvelle table de symboles pour les variables locales à la fonction. Plus précisément, toute affectation de variable dans la fonction enregistre la valeur dans la table des symboles locaux ; si une variable est référencée, Python la recherche déjà la table des symboles locaux, puis dans les tables des symboles locaux des fonctions englobantes, puis dans la la table des symboles globaux et enfin dans la tables des noms prédéfinis. Ainsi, les variables globales ne peuvent être affectées directement dans le corps d'une fonction (à moins qu'elles ne soient explicitement déclarées dans une instruction **global**) ; toutefois, elles peuvent être référencées.

Les paramètres effectifs (arguments) d'un appel de fonction sont enregistrés dans la table des symboles locaux de la fonction lors de l'appel ; donc, les arguments sont toujours *passés par valeur*, (la *valeur* est toujours une *référence* à l'objet, pas la valeur de l'objet).[2] Lorsqu'une fonction appelle une autre fonction, une nouvelle table des symboles locaux est créée pour cet appel.

Une définition de fonction enregistre un nouveau nom dans la table courante des symboles locaux. La valeur associée à ce nom de fonction est reconnue par l'interpréteur comme étant de type fonction définie par l'utilisateur (*user-defined function*). Cette valeur peut être affectée à un autre nom, qui peut alors être utilisé comme fonction. Cela peut servir de mécanisme général de renommage :

```
>>> fib
<function fib at 0x0000000002F03048>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Si vous pratiquez d'autres langages, vous pouvez objecter que **fib** n'est pas une fonction, mais une procédure, car elle ne retourne pas de valeur. En fait, même les fonctions sans **return** retournent une valeur quelque peu rébarbative. Cette valeur est appelée **None** (c'est un nom prédéfini). Normalement l'interpréteur n'affiche pas **None** si c'est la seule valeur à devoir être affichée. On peut toutefois le voir en utilisant **print()** :

```
>>> fib(0)

>>> print(fib(0))

None
```

Il est facile d'écrire une fonction qui retourne une liste de nombres de Fibonacci, plutôt que de les afficher :

```
>>> def fib2(n): # retourne la suite de Fibonacci jusqu'à n
...     """Retourne la liste des nombres de Fibonacci jusqu'à n."""
...     résultat = []
...     a, b = 0, 1
...     while a < n:
...         résultat.append(a)    # voir ci-dessous
```

2. En fait, *passage par référence* serait une meilleure description, puisque si un objet modifiable est transmis, l'appelant verra les modifications effectuées par l'appelé sur cet objet (items insérés dans une liste, par exemple).

```
...         a, b = b, a+b
...     return résultat
...
>>> f100 = fib2(100)    # appel
>>> f100                # affiche le résultat
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Comme d'habitude, cet exemple expose de nouvelles caractéristiques de Python :

- L'instruction **`return`** retourne un résultat d'une fonction. Un **`return`** non suivi d'une expression retourne **`None`**. Sortir d'une fonction par la fin retourne également **`None`**.
- L'instruction **`résultat.append(a)`** appelle une méthode de l'objet liste **`résultat`**. Une méthode est une fonction "appartenant" à un objet et qui est nommée **`obj.nomDeMéthode`**, où **`obj`** est un objet (qui peut être une expression) et **`nomDeMéthode`** est le nom d'une méthode définie par le type de l'objet. Les types différents définissent des méthodes différentes. Des méthodes de différents types peuvent porter le même nom sans risque d'ambiguïté (il est possible de définir vos propres types et méthodes, en utilisant les *classes*, voir 9 *Classes*). La méthode **`append()`** de l'exemple est définie pour les objets de type liste (**`list`**) ; elle ajoute un nouvel élément en fin de liste. Dans l'exemple, c'est le même résultat que si l'on avait écrit **`résultat = résultat + [a]`**, mais c'est plus efficace.

4.7 Plus loin dans les définitions de fonctions

Il est également possible de définir des fonctions avec un nombre variable d'arguments. Trois différentes façons de le faire existent, qui peuvent être combinées.

4.7.1 Arguments avec valeur par défaut

La façon la plus utile est de spécifier une valeur par défaut pour un ou plusieurs arguments. La fonction ainsi créée peut être appelée avec moins d'arguments que ne précise sa définition. Par exemple :

```
def oui_ou_non(message, nb_essais=4, requête='oui ou non, SVP !'):
    while True:
        ok = input(message)
        if ok in ('o', 'oui'):
            return True
        if ok in ('n', 'non'):
            return False
        nb_essais = nb_essais - 1
        if nb_essais < 0:
            raise OSError('utilisateur buté')
        print(requête)
```

Cette fonction peut être appelée de plusieurs manières :

- en ne donnant que l'argument obligatoire : **`oui_ou_non('Quitter. Êtes-vous sûr?')`**
- en donnant un argument optionnel : **`oui_ou_non('Écraser le fichier?', 2)`**
- ou même en les précisant tous : **`oui_ou_non('Écraser le fichier?', 2, 'Répondez oui ou non!')`**

Cet exemple introduit également le mot-clef **`in`**, qui teste si une séquence contient ou non une valeur.

Les valeurs par défaut sont évaluées au moment de la définition de la fonction dans la portée courante, et donc les lignes suivantes afficheront 5 (et non 6 comme on pourrait le croire) :

```
>>> i = 5
>>> def f(arg=i):
...     print(arg)
...
>>> i = 6
>>> f()
5
```

Attention : La valeur par défaut n'est évaluée qu'une seule fois. C'est particulièrement important lorsque cette valeur est un objet modifiable comme une liste, un dictionnaire ou une instance de classe. Par exemple, la fonction suivante accumule les arguments passés lors des appels successifs :

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

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

```
print(f(3))
```

L'affichage sera :

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

Si vous ne voulez pas que la valeur par défaut soit partagée entre les appels successifs, vous devrez écrire la fonction de cette manière :

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

4.7.2 Arguments par mots-clefs

Les fonctions peuvent également être appelées en utilisant des arguments par mots-clefs (voir *keyword argument*) sous la forme **`mot_clef=valeur`**. Par exemple, la fonction suivante[3] :

```
>>> def perroquet(voltage, état='raide mort !', déplacement='une patte',
...               sorte='perroquet bleu de Norvège'):
...     print("- Ce perroquet ne bougerait pas", déplacement)
...     print("  même si on lui envoyait une décharge de", voltage, "volts.")
...     print("- Quel magnifique plumage, ce", sorte, end='.\n')
...     print("- Mais il est", état, "!")
...
```

accepte un argument obligatoire (**`voltage`**) et trois arguments optionnels (**`état`**, **`déplacement`** et **`sorte`**). Cette fonction peut être appelée de l'une ou l'autre des manières suivantes :

```
>>> perroquet(1000)                                    # 1 argument positionnel
- Ce perroquet ne bougerait pas une patte
  même si on lui envoyait une décharge de 1000 volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est raide mort ! !

>>> perroquet(voltage=1000)                            # 1 argument mot-clé
- Ce perroquet ne bougerait pas une patte
  même si on lui envoyait une décharge de 1000 volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est raide mort ! !

>>> perroquet(voltage=1000000, déplacement="d'un mm")   # 2 arguments mots-clés
- Ce perroquet ne bougerait pas d'un mm
  même si on lui envoyait une décharge de 1000000 volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est raide mort ! !

>>> perroquet(déplacement="d'un mm", voltage=1000000)   # 2 arguments mots-clés
- Ce perroquet ne bougerait pas d'un mm
  même si on lui envoyait une décharge de 1000000 volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est raide mort ! !

>>> perroquet("deux mille", "perdu corps et âme",
...           "un oeil")                                # 3 arguments positionnels
- Ce perroquet ne bougerait pas un oeil
  même si on lui envoyait une décharge de deux mille volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est perdu corps et âme !

>>> perroquet("mille", état=("en train de manger "
...                       "les pissenlits par la racine"))# 1 positionnel, 1 mot-clé
- Ce perroquet ne bougerait pas une patte
  même si on lui envoyait une décharge de mille volts.
- Quel magnifique plumage, ce perroquet bleu de Norvège.
- Mais il est en train de manger les pissenlits par la racine !
```

Par contre, les appels suivants sont invalides :

3. Pour l'origine de l'exemple, voir la vidéo Dead Parrot

```
>>> perroquet()                    # argument obligatoire manquant
>>> perroquet(voltage=5.0, 'mort')  # argument positionnel après argument mot-clé
>>> perroquet(110, voltage=220)     # deux valeurs pour le même argument
>>> perroquet(acteur='John Cleese') # argument mot-clé inconnu
```

Lors d'un appel de fonction, les arguments mots-clés doivent suivre les arguments positionnels. Tous les arguments mots-clés doivent correspondre à l'un des paramètres spécifiés dans le définition de la fonction (par exemple, **`acteur`** n'est pas un argument valide pour la fonction **`perroquet`**), et leur ordre est sans importance. C'est valable aussi pour les arguments obligatoires (par exemple, **`perroquet(voltage=1000)`** est valide). Aucun argument ne peut recevoir de valeur plus d'une fois. Voici un exemple qui provoque une erreur à cause de cette restriction :

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

Lorsque le dernier paramètre est de la forme **`**nom`**, il reçoit un dictionnaire (voir *Mapping Types – dict*) contenant des arguments mots-clés, sauf s'ils correspondent à un paramètre formel. Ceci peut être combiné avec l'utilisation d'un paramètre formel de la forme **`*nom`** (décrit dans la sous-section suivante) qui lui reçoit un tuple contenant les arguments positionnels suivant la liste des paramètres formels. (**`*nom`** doit précéder **`**nom`**.) Par exemple, si on définit cette fonction [4] :

```
>>> def fromagerie(fromage, *arguments, **motsclés):
...     print("- Avez-vous du", fromage, "?")
...     print("- Désolé, nous n'avons plus de", fromage)
...     for arg in arguments:
...         print(arg)
...     print("-" * 40)
...     clés = sorted(motsclés.keys())
...     for clé in clés:
...         print(clé, ":", motsclés[clé])
...
```

Elle pourra être appelée par exemple comme çeci :

```
>>> fromagerie("camembert", "Il est très coulant, monsieur.",
...            "Il est vraiment très, TRÈS coulant, monsieur.",
...            fromager="Michael Palin",
...            client="John Cleese",
...            sketch="Cheese Shop Sketch")
- Avez-vous du camembert ?
- Désolé, nous n'avons plus de camembert
Il est très coulant, monsieur.
Il est vraiment très, TRÈS coulant, monsieur.
----------------------------------------
client : John Cleese
fromager : Michael Palin
sketch : Cheese Shop Sketch
```

Notez que la liste des noms des arguments mots-clés est créée en triant le retour de la méthode **`keys()`** avant d'afficher son contenu ; si on ne l'avait pas fait, l'ordre d'affichage aurait été indéfini.

4.7.3 Arguments récupérés dans un tuple

Enfin, la manière la moins fréquente est de définir une fonction pouvant être appelée avec un nombre variable d'arguments. Ces arguments seront enveloppés dans un tuple (voir 5.3 *Tuples et séquences*). Avant les arguments en nombre variable, zéro or plusieurs arguments *normaux* peuvent être présents.

```
def ecriture_multiple(fichier, séparateur, *arguments):
    fichier.write(séparateur.join(arguments))
```

Normalement, ces arguments en quantité variable doivent être les derniers dans la suite des paramètres formels, car ils prendront en charge tous les arguments restants passés à la fonction. Tous les paramètres formels apparaissant après le paramètre **`*args`** sont des arguments mots-clés seulement (*keyword-only*), ce qui signifie qu'ils ne peuvent être utilisés que par mot-clé (et pas par position).

4. Pour l'origine de l'exemple, voir la vidéo Cheese Shop

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("Terre", "Mars", "Vénus")
'Terre/Mars/Vénus'
>>> concat("Terre", "Mars", "Vénus", sep=".")
'Terre.Mars.Vénus'
```

4.7.4 Déballer une liste d'arguments

La situation s'inverse lorsque les arguments sont déjà dans une liste ou un tuple, mais qu'ils doivent en être extraits pour un appel de fonction nécessitant une des arguments positionnels séparés. Par exemple, la fonction prédéfinie `range()` attend des arguments séparés *début* et *fin*. S'ils sont dans une liste (ou un tuple), on écrit l'appel de fonction avec l'opérateur `*`, qui extraira les éléments de la liste ou du tuple (déballage ou *unpacking*) :

```
>>> list(range(3, 6))            # appel normal, arguments séparés
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # appel avec les arguments extraits d'une liste
[3, 4, 5]
```

De la même manière, les dictionnaires peuvent fournir des arguments par mots-clés grâçe à l'opérateur `**` :

```
>>> def perroquet(voltage, état='raide mort', déplacement='une patte'):
...     print("- Ce perroquet ne bougerait pas", déplacement)
...     print("  même si on lui envoyait une décharge de", voltage, "volts.")
...     print("  Il est", état, "!")
...
>>> d = {"voltage": "quatre mille", "état": "tout ce qu'il y a de plus mort",
...     "déplacement": "un oeil"}
>>> perroquet(**d)
- Ce perroquet ne bougerait pas un oeil
  même si on lui envoyait une décharge de quatre mille volts.
  Il est tout ce qu'il y a de plus mort !
```

4.7.5 Lambda-expressions

On peut créer des fonctions anonymes grâce au mot-clé `lambda`. Par exemple, cette fonction `lambda a, b: a + b` retourne la somme de ses deux arguments. Les lambda-fonctions peuvent être utilisées partout où des objets fonction sont requis. Toutefois, elles sont restreintes à n'être formées que d'une seule expression. Sémantiquement, ce n'est qu'une façon d'écrire différemment une définition normale de fonction. Comme pour des définitions de fonctions imbriquées, les lambda-fonctions peuvent référencer des variables dans leur portée.

```
>>> def additionneur(n):
...     return lambda x: x + n
...
>>> f = additionneur(42)
>>> f(0)
42
>>> f(2015)
2057
```

L'exemple ci-dessus utilise une lambda fonction pour retourner une fonction. Une autre utilisation est quand il s'agit de passer une fonction en tant qu'argument :

```
>>> couples = [(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre')]
>>> couples.sort(key=lambda couple: couple[1]) # trie par ordre alphabétique des nombres
>>> couples
[(2, 'deux'), (4, 'quatre'), (3, 'trois'), (1, 'un')]
```

4.7.6 Chaînes de documentation

Voici quelques conventions sur les contenus et le formatage des chaînes de documentation.

La première ligne doit consister en un court résumé du rôle de l'objet. Pour des raisons de concision, il est inutile de préciser le nom ou le type de l'objet, car ils sont certainement disponibles par d'autres moyens (sauf s'il se trouve que le nom de l'objet est un verbe décrivant le rôle de la fonction). Cette ligne doit commencer par une majuscule et se terminer par un point.

S'il y a plusieurs lignes dans la chaîne de documentation, la deuxième ligne sera laissée vide, créant une séparation visible entre le résumé et le reste de la description. Les lignes suivantes seront constituées d'un ou plusieurs paragraphes décrivant les conventions d'appels, les effets de bord, etc.

L'analyseur syntaxique ne retire pas les indentations dans les littéraux chaînes multi-lignes, aussi les outils qui traitent la documentation doivent les retirer eux-mêmes, si besoin est. Ceci doit être fait en respectant la convention suivante. Le première ligne non vide **après** la première ligne détermine le niveau d'indentation pour toute la chaîne de documentation (on n'utilise pas la toute première ligne car elle est en général collée à la triple quote ouvrante, sans indentation). Les espaces "équivalents" à ce niveau d'indentation doivent être retirés du début de chaque ligne de la chaîne. Il ne devrait pas y avoir de lignes moins indentées, mais s'il y en a, tous les espaces doivent être retirés. Les équivalences d'espacement doivent être testées après développement des tabulation (en général, 8 espaces).

Voici un exemple de *docstring* multi-lignes :

```
>>> def ma_fonction():
...     """Ne fait rien, mais documente ce rien.
...
...     Non, réellement, ne fait rien du tout.
... """
...     pass
...
>>> print(ma_fonction.__doc__)
Ne fait rien, mais documente ce rien.

    Non, réellement, ne fait rien du tout.
```

4.7.7 Annotations de fonctions

Les annotations de fonctions sont des méta données d'information optionnelles sur les types utilisés dans les fonctions utilisateurs (voir PEP 484 pour plus d'information).

Les annotations sont stockées dans l'attribut `__annotations__` de la fonction sous la forme d'un dictionnaire et n'ont aucun effet ailleurs dans la fonction. Les annotations de paramètres se notent par un deux-points (`:`) suivant le nom du paramètre, suivi d'une expression spécifiant la valeur de l'annotation. Les annotations de retour se notent par un `->`, suivi par une expression, après la liste des paramètres et avant le deux-points (`:`) final de l'instruction `def`. L'exemple suivant annote deux arguments (un positionnel et un mot-clé) et le retour :

```
>>> def f(jambon: str, oeufs: str = 'oeufs') -> str:
...     print("Annotations :\n\t", f.__annotations__)
...     print("Arguments :\n\t", jambon, oeufs)
...     return jambon + ' et ' + oeufs
...
>>> f('spam')
Annotations :
     {'oeufs': <class 'str'>, 'return': <class 'str'>, 'jambon': <class 'str'>}
Arguments :
     spam oeufs
'spam et oeufs'
```

4.8 Intermède : style de codage

Maintenant que vous êtes prêt à écrire des programmes Python plus longs et plus complexes, il est temps de parler de *style de codage*. La plupart des langages permettent d'utiliser différents styles d'écriture (ou plus précisément de formatage) ; certains sont plus lisibles que d'autres. Rendre votre code plus lisible par d'autres est toujours une bonne idée, et adopter un agréable style de codage vous y aidera énormément.

Concernant Python, PEP 8 apparaît comme le style de codage auquel la plupart des projets adhèrent ; il promeut un style très lisible et agréable à l'œil. Tout programmeur Python devrait le lire un jour ; voici les points les plus importants qui en ont été tirés pour vous :

- Utilisez des indentations de 4 espaces (pas de tabulation).

  4 espaces constituent un bon compromis entre une petite indentation (permettant une plus grande profondeur d'imbrication) et une grande indentation (plus facile à lire). Les tabulations rendent les choses plus confuses, et il vaut mieux ne pas les utiliser.

- Coupez les lignes pour qu'elles ne dépassent pas 79 caractères.

  C'est utile pour les utilisateurs de petits écrans et cela permet d'avoir plusieurs codes côte-à-côte sur de grands écrans.

- Utilisez des lignes vides pour séparer fonctions et classes, et les grands blocs de code à l'intérieur des fonctions.
- Si possible, placez les commentaires sur des lignes à part.
- Utilisez les *docstrings*.
- Mettez des espaces autour des opérateurs et après les virgules, mais pas dans les parenthèses : `a = f(1, 2) + g(3, 4)`.
- Nommez vos classes et fonctions de manière cohérente ; la convention est d'utiliser :
  - la `CasseDuChameau` (*`CamelCase`*) pour les classes
  - la `casse_minuscule_avec_tirets_de_soulignements` (*`lower_case_with_underscores`*) pour les fonctions et les méthodes.

  Utilisez toujours `self` comme nom de premier argument de méthode (voir 9.3 *Un premier coup d'œil aux classes* pour en savoir plus sur les classes et les méthodes).
- N'utilisez pas d'encodages exotiques si votre code est destiné à être utilisé à l'international. L'encodage UTF-8 (par défaut sous Python) ou même l'encodage ASCII conviendront le mieux dans tous les cas.
- De même, n'utilisez pas des caractères non-ASCII dans les identificateurs, s'il y a la moindre chance que vos programmes puissent être lus ou maintenus par des gens ne parlant pas votre langue.

Chapitre 5

Structures de données

Ce chapitre décrit plus en détail des structures de données déjà abordées, et en présente de nouvelles.

5.1 Aller plus loin avec les listes

D'autres méthodes existent pour les listes. Voici toutes les méthodes des objets `list` :

`list.append(x)`
Ajoute un item en fin de liste. Équivalent à `a[len(a):] = [x]`.

`list.extend(L)`
Ajoute tous les éléments de L en fin de liste. Équivalent à `a[len(a):] = L`.

`list.insert(i, x)`
Insère un élément à la position indiquée. Le premier argument est l'indice de l'élément avant lequel l'insertion se fera. Ainsi, `a.insert(0, x)` insère en début de liste, et `a.insert(len(a), x)` équivaut à `a.append(x)`.

`list.remove(x)`
Supprime le premier élément de la liste dont la valeur est *x*. Une erreur se produit s'il n'y en a pas.

`list.pop([i])`
Supprime dans la liste l'élément situé à la position indiquée, et retourne cet élément. Si aucun argument n'est donné, `a.pop()` supprime et retourne le dernier élément de la liste (les crochets autour de *i* dans la signature de la méthode indique que le paramètre est optionnel, ne les entrez pas dans l'appel de la méthode. Cette notation est fréquemment utilisée dans *Python Library Reference*.)

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

`list.index(x)`
Retourne l'index dans la liste du premier élément dont la valeur est *x*. Une erreur se produit s'il n'y en a pas.

`list.count(x)`
Retourne le nombre d'occurrences de *x* dans la liste.

`list.sort()`
Trie *en place* les éléments de la liste.

`list.reverse()`
Inverse *en place* les éléments de la liste.

`list.copy()`
Retourne une copie *superficielle* de la liste. Équivalent à `a[:]`.

Un exemple utilisant la plupart de ces méthodes :

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
```

```
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

Vous avez sans doute remarqué que les méthodes comme **`insert`**, **`remove`** ou **`sort`** ne font que modifier la liste et ne retournent pas de valeur – elles retournent la valeur **`None`**[1]. C'est un principe de conception valable pour tous les objets modifiables dans Python.

5.1.1 Utiliser des listes en tant que piles

Les méthodes de liste permettent d'utiliser très facilement une liste en tant que pile, structure de données dans laquelle le dernier élément ajouté est le premier élément retiré ("*last-in, first-out*"). Pour empiler un élément, on utilise **`append()`**. Pour dépiler un élément, on utilise **`pop()`** (sans spécifier d'index). Par exemple :

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

5.1.2 Utiliser des listes en tant que files

Il est aussi possible d'utiliser des listes en tant que files, dans lesquelles le premier élément ajouté est le premier élément retiré ("*first-in, first-out*") ; toutefois, les listes ne sont pas efficaces pour cela. Alors que les ajouts et retraits en fin de liste sont rapides, les insertions ou les retraits en début de liste sont très lentes (car tous les autres éléments doivent être décalés d'une place). Pour implémenter une file, on utilise le type **`deque`** du module **`collections`** (voir **`collections.deque`**) qui a été conçu pour faire des ajouts et des retraits efficaces aux deux extrémités. Par exemple :

```
>>> from collections import deque
>>> file_d_attente = deque(["Eric", "John", "Michael"])
>>> file_d_attente.append("Terry")           # Terry arrive
>>> file_d_attente.append("Graham")          # Graham arrive
>>> file_d_attente.popleft()                 # Le premier arrivé s'en va
'Eric'
>>> file_d_attente.popleft()                 # Le deuxième s'en va
'John'
>>> file_d_attente                           # Encore en attente dans l'ordre d'arrivée
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Listes en compréhension

L'écriture de listes en compréhension est un moyen pratique d'initialiser de nouvelles listes. Dans de nombreuses applications, il faut créer des listes pour lesquelles chaque élément est le résultat d'une opération appliquée à chaque élément d'une séquence ou d'un itérable, ou créer une sous-liste d'éléments vérifiant un certain critère.

Supposons par exemple que nous ayons besoin d'une liste de carrés, comme :

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

```
>>> carrés = []
>>> for x in range(10):
...     carrés.append(x**2)
...
>>> carrés
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Notons que ce code crée (ou modifie) la variable **x** qui continue d'exister après la boucle. Nous pouvons cependant calculer la liste des carrés sans effet de bord :

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

ou, de manière équivalente :

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

qui est plus concise et plus lisible.

Une liste en compréhension est une expression suivie d'une clause **for**, puis de une ou plusieurs clauses **for** ou **if**, le tout entre crochets. Le résultat est une nouvelle liste dont les éléments sont les évaluations de l'expression dans le contexte des clauses **for** et **if** qui la suivent. Par exemple, cette liste en compréhension associe les éléments de deux listes, s'ils sont différents :

```
>>> [(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 produit le même résultat que :

```
>>> 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 **for** et du **if** est le même dans les deux cas.

Si l'expression est un tuple (par exemple **(x, y)** dans l'exemple précédent, elle doit être parenthésée.

Un autre exemple [2] :

```
>>> vecteur = [-4, -2, 0, 2, 4]
>>> # crée une nouvelle liste avec les valeurs multipliées par 2
>>> [x*2 for x in vecteur]
[-8, -4, 0, 4, 8]
>>> # filtre la liste en excluant les nombres négatifs
>>> [x for x in vecteur if x >= 0]
[0, 2, 4]
>>> # applique une fonction à tous les éléments
>>> [abs(x) for x in vecteur]
[4, 2, 0, 2, 4]
>>> # appel d'une méthode sur chaque élément
>>> fruits_frais = ['  banane', '  mûroise ', 'fruit de la passion  ']
>>> [arme.strip() for arme in fruits_frais] # suppr. espace avant/après les noms
['banane', 'mûroise', 'fruit de la passion']
>>> # crée une liste de couples (nombre, carré)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # le couple doit être parenthésé, sinon une erreur se produit
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
            ^
SyntaxError: invalid syntax
>>> # aplatisement d'une liste (liste en compréhension utilisant deux 'for')
>>> vecteur = [[1,2,3], [4,5,6], [7,8,9]]
>>> [nombre for élément in vecteur for nombre in élément]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Les listes en compréhension peuvent être composées à partir d'expressions complexes ou de fonctions imbriquées :

2. Pour l'origine de l'exemple, voir la vidéo Self Defence Against Fresh Fruit

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 8)]
['3.1', '3.14', '3.142', '3.1416', '3.14159', '3.141593', '3.1415927']
```

5.1.4 Listes en compréhension imbriquées

L'expression initiale d'une liste en compréhension peut être toute sorte d'expression, y compris une liste en compréhension. Considérons l'exemple suivant d'une matrice 3×4 implémentée sous la forme d'une liste de 3 listes de longueur 4 :

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

La liste en compréhension suivante crée la matrice transposée :

```
>>> [[ligne[col] for ligne in matrice] for col in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Comme indiqué dans la section précédente, la liste en compréhension imbriquée est évaluée dans le contexte du `for` qui la suit ; l'exemple est donc équivalent à :

```
>>> transposée = []
>>> for col in range(4):
...     transposée.append([ligne[col] for ligne in matrice])
...
>>> transposée
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

qui, lui-même, est équivalent à :

```
>>> transposée2 = []
>>> for col in range(4):
...     # les 3 lignes suivantes implémentent la liste en compréhension interne
...     ligne_transposée = []
...     for ligne in matrice:
...         ligne_transposée.append(ligne[col])
...     transposée2.append(ligne_transposée)
...
>>> transposée2
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Dans la *vraie vie*, on préférera toutefois les utilitaires prédéfinis aux instructions compliquées comme ci-dessus. La fonction `zip()` conviendra tout à fait pour ce cas :

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

Voir 4.7.4 *Déballer une liste d'arguments* pour plus de détails sur l'astérisque dans l'exemple ci-dessus.

5.2 L'instruction `del`

Il est possible de supprimer un élément d'une liste en précisant son index plutôt que sa valeur grâce à l'instruction `del`. Elle diffère de la méthode `pop()` qui retourne une valeur. L'instruction `del` permet également de supprimer des découpes (*slices*) d'une liste ou d'en supprimer tous les éléments (nous l'avons déjà fait en affectant une liste vide à la découpe copie `l[:]`). 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 s'utiliser également pour supprimer complètement la variable :

```
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Après cette instruction, toute référence à `a` produira une erreur (tant qu'une autre valeur ne lui sera pas affectée). Nous verrons d'autres utilisations de `del` plus tard.

5.3 Tuples et séquences

Nous avons vu que les listes et les chaines partagent beaucoup de caractéristiques , comme les opérations d'indexation ou de découpe. Ce sont deux exemples de types de données *séquences* (voir *Sequence Types — list, tuple, range*). Comme Python est un langage toujours en évolution, d'autres types de séquences peuvent lui être ajoutés dans le futur. Il existe un autre type de donnée séquence : le *tuple*.

Un tuple consiste en une suite de valeurs, séparées par des virgules, par exemple :

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples peuvent être imbriqués :
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Les tuples sont figés (immutable)
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # mais ils peuvent contenir des objets modifiables :
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Comme vous pouvez le voir, les tuples en sortie sont affichés entre parenthèses, ce qui fait que les tuples imbriqués sont interprétés correctement ; on peut les écrire avec ou sans parenthèses, bien que les parenthèses soient souvent nécessaire pour lever les ambiguïtés (si le tuple fait partie d'une plus grande expression). Il n'est pas possible de réaffecter les éléments d'un tuple, toutefois il est possible de créer des tuples contenant des objets modifiables, par exemple des listes.

Bien que les tuples semblent similaires aux listes, ils sont souvent utilisés dans des situations différentes et pour des objectifs différents. Les tuples sont figés (*immutable*), et contiennent généralement une suite d'éléments hétérogènes accédés par déballage (*unpacking*) (voir plus loin dans cette section) ou par index (ou même par attribut dans le cas de `namedtuples`). Les listes sont modifiables (*mutable*), et leurs éléments sont généralement de même type et accédés par itération sur la liste.

Un problème spécifique aux tuples est la création de tuples contenant 0 ou 1 élément : la syntaxe Python au prix de quelques bizarreries permet de le faire. Les tuples vides sont construits par une paire de parenthèses vide ; un tuple d'un élément est construit en faisant suivre la valeur par une virgule (encadrer une valeur par des parenthèses ne fonctionne pas). C'est laid, mais ça marche. Par exemple :

```
>>> vide = ()
>>> singleton = 'bonjour',    # <-- notez la virgule finale
>>> len(vide)
0
>>> len(singleton)
1
>>> singleton
('bonjour',)
```

L'instruction `t = 12345, 54321, 'hello!'` est un exemple d'emballage de tuple (*tuple packing*) : les valeurs `12345`, `54321` and `'hello!'` sont emballées dans le même tuple. L'opération inverse est également possible :

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

Appelé déballage de séquence (*sequence unpacking*), cette opération fonctionne pour toute séquence

située à droite du signe `=`. Elle nécessite qu'il y ait à gauche du signe `=` autant de variables qu'il n'y a d'éléments dans la séquence. Notez que l' affectation multiple (`a, b = b, a % b`) n'est qu'une combinaison d'emballage/déballage (`t = b, a % b`, puis `a, b = t`).

5.4 Ensembles

Python fournit également un type de données pour les *ensembles*. Un ensemble est une collection non ordonnée d'éléments sans répétition. Les ensembles sont principalement utilisés pour les tests d'appartenance et l'élimination des doublons. Ils permettent également les opérations mathématiques comme l'union, l'intersection, la différence et la différence symétrique.

Les accolades ou la fonction `set()` sont utilisées pour créer des ensembles. Note : pour créer un ensemble vide, il faut utiliser `set()` et non `{}` ; cette dernière syntaxe crée un dictionnaire vide, structure de données dont nous parlerons dans la section suivante.

Voici une courte démonstration :

```
>>> panier = {'pomme', 'orange', 'pomme', 'poire', 'orange', 'banane'}
>>> print(panier)                      # montre l'élimination des doublons
{'orange', 'banane', 'pomme', 'poire'}
>>> 'orange' in panier                 # le test d'appartenance est très efficace
True
>>> 'digitaire' in panier
False

>>> # Démonstration d'opérations ensemblistes sur les lettres de deux mots
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # dans  a
{'d', 'r', 'a', 'b', 'c'}
>>> a - b                              # dans a mais pas dans b
{'d', 'r', 'b'}
>>> a | b                              # dans a ou dans b
{'r', 'b', 'd', 'z', 'c', 'm', 'a', 'l'}
>>> a & b                              # dans a et dans b
{'a', 'c'}
>>> a ^ b                              # dans a ou dans b, mais pas dans les deux
{'r', 'b', 'z', 'd', 'm', 'l'}
```

Comme pour les *Listes en compréhension* (voir 5.1.3), il est possible d'écrire des ensembles en compréhension :

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

5.5 Dictionnaires

Un autre type de données prédéfini très utile est le *dictionnaire* (voir *Mapping Types – dict*). On les rencontre parfois dans d'autres langages sous l'appellation "mémoire associative" ou "tableau associatif". Contrairement aux séquences, indexées par des nombres, les dictionnaires sont indexés par des clés, qui doivent appartenir à un type figé (*immutable*) ; par exemple, les chaînes et les nombres peuvent être des clés. Les tuples également, à condition qu'ils ne contiennent que des chaînes, des nombres ou des tuples ; si un tuple contient un objet modifiable, que ce soit directement ou indirectement, il ne peut pas servir de clé. Les listes ne peuvent pas servir de clés, car elles sont modifiables par affectation indexée, affectation de découpe ou des méthodes comme `append()` et `extend()`.

On peut considérer un dictionnaire comme un ensemble non ordonnée de couples *clé : valeur*, avec l'obligation pour les clés d'être uniques (dans le dictionnaire). Une paire d'accolades `{}` crée un dictionnaire vide. Une suite de couples *clé : valeur* séparées par des virgules entre accolades initialise un dictionnaire ; c'est également la façon dont les dictionnaires sont affichés.

Les principales utilisations d'un dictionnaire sont le stockage d'une valeur associée à une clé, et la récupération de la valeur en fournissant la clé. Il est également possible de supprimer un couple *clé : valeur* grâce à `del`. Si l'on stocke une valeur en lui associant une clé déjà utilisé, l'ancienne valeur associée à cette clé est perdue. Essayer de récupérer une valeur associé à une clé qui n'existe pas produit une erreur.

Invoquer `list(d.keys())` sur un dictionnaire retourne la liste de toutes les clés utilisées dans ce dictionnaire, dans un ordre arbitraire (si vous la voulez triée, invoquez `sorted(d.keys())`[3]). Pour tester

3. Appeler `d.keys()` retournera un objet "vue" du dictionnaire (*dictionary view*). Il autorise des opérations comme les tests d'appartenance ou les itérations, mais son contenu n'est pas indépendant du dictionnaire original, ce n'est qu'une vue.

l'appartenance d'une clé au dictionnaire, on utilise le mot-clé **in**.

Voici un petit exemple d'utilisation de dictionnaire :

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

Le constructeur **dict()** crée un dictionnaire directement à partir d'une séquence de couples *(clé, valeur)* :

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

De plus, l'écriture de dictionnaire en compréhension à partir d'expressions *clé : valeur* est également possible :

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

Lorsque les clés sont de simples chaînes, il et parfois plus simple de spécifier les couples en tant qu'arguments motss-clés !

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

5.6 Techniques de boucles

En parcourant un dictionnaire, la clé et la valeur peuvent être récupérées simultanément en utilisant la méthode **items()**.

```
>>> chevaliers = {'gallahad': 'le pur', 'robin': 'le brave'}
>>> for nom, surnom in chevaliers.items():
...     print(nom.capitalize(), surnom)
...
Robin le brave
Gallahad le pur
```

En parcourant une séquence, l'index et la valeur correspondante peuvent être récupérés simultanément en utilisant la fonction **enumerate()**.

```
>>> for i, v in enumerate(['am', 'stram', 'gram']):
...     print(i, v)
...
0 am
1 stram
2 gram
```

Pour parcourir simultanément deux ou plusieurs séquences, les éléments correspondants de chaque séquence peuvent être associés grâce à la fonction **zip()** [4] .

```
>>> questions = ['ton nom', 'ta quête', 'ta couleur préférée']
>>> réponses = ['Lancelot', 'Le Sacré Graal', 'Le bleu']
>>> for q, r in zip(questions, réponses):
```

4. Pour l'origine de l'exemple, voir la vidéo The Bridge of Death

```
...     print("- Dis-moi {0}. \n- {1}.".format(q, r))
...
- Dis-moi ton nom.
- Lancelot.
- Dis-moi ta quête.
- Le Sacré Graal.
- Dis-moi ta couleur préférée.
- Le bleu.
```

Pour parcourir une séquence à rebours, on spécifie tout d'abord la séquence dans le sens normal, puis on appelle la fonction `reversed()`.

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

Pour parcourir une séquence dans l'ordre, on utilise la fonction `sorted()` qui retourne une liste triée sans modifier l'original.

```
>>> panier = ['pomme', 'orange', 'pomme', 'poire', 'orange', 'banane']
>>> for f in sorted(set(panier)):
...     print(f)
...
banane
orange
poire
pomme
```

Pour modifier la séquence sur laquelle on itère pendant l'exécution de la boucle (par exemple pour dupliquer certains items), il est recommandé de faire d'abord une copie de la séquence. En effet, itérer sur une séquence ne fait pas une copie de la séquence. La syntaxe de découpe (*slice*) est particulièrement utile dans ce cas :

```
>>> liste_de_mots = ['cat', 'window', 'defenestrate']
>>> for mot in liste_de_mots[:]:  # boucle sur une découpe, copie de la liste
...     if len(mot) > 6:
...         liste_de_mots.insert(0, mot)
...
>>> liste_de_mots
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.7 Plus sur les conditions

Les conditions utilisées dans les instructions `while` et `if` peuvent contenir n'importe quels opérateurs, en place des comparaisons.

Les opérateurs de comparaison `in` et `not in` testent si une valeur est présente (absente) dans une séquence. Les operateurs `is` et `is not` comparent si oui (ou non) deux objets sont bien le même objet ; cela n'a de sens que pour des objets modifiables comme les listes. Tous les opérateurs de comparaison ont la même priorité, plus faible que celle des opérateurs numériques.

Les comparaisons peuvent s'enchaîner : par exemple, `a < b == c` teste si `a` est inférieur à `b`, puis si `b` est égal à `c`.

Les comparaisons peuvent se combiner en utilisant les opérateurs booléens `and` et `or`, et le résultat d'une comparaison (ou de toute autre expression booléenne) peut être nié avec `not`. Ces opérateurs ont une plus faible priorité que les opérateurs de comparaison ; entre eux, `not` a la plus forte priorité et `or` la plus faible, ainsi `A and not B or C` est équivalent à `(A and (not B)) or C`. Comme toujours, les parenthèses permettent de spécifier l'expression désirée.

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

Il est possible d'utiliser le résultat d'une comparaison ou d'une expression booléenne dans une affectation de variable. Par exemple :

```
>>> chaine1, chaine2, chaine3 = '', 'Trondheim', 'Hammer Dance'
>>> chaine_non_nulle = chaine1 or chaine2 or chaine3
>>> chaine_non_nulle
'Trondheim'
```

Notez que contrairement au langage C, les affectations ne sont pas autorisées dans les expressions. Les programmeurs C ronchonneront, mais cela permet d'éviter l'erreur classique rencontrée dans les programmes C : entrer `=` dans une expression alors qu'on veut entrer `==`.

5.8 Comparer des séquences ou d'autres types

Les objets séquence peuvent être comparés à d'autres objets s'ils ont le même type. La comparaison utilise l'ordre *lexicographique* : tout d'abord, les deux premiers éléments sont comparés, et s'ils sont différents, cette différence détermine le résultat de la comparaison ; s'ils sont égaux, les deux éléments suivants sont comparés, et ainsi de suite, jusqu'à ce que l'une des deux séquences soit épuisée. Si deux éléments à comparer sont eux-mêmes des séquences de même type, on reprend récursivement la comparaison lexicographique. Si tous les éléments de même rang des deux séquences sont égaux, les séquences sont considérées comme égales. Si une séquence est une sous-séquence initiale de l'autre, la plus courte est considérée comme inférieure. L'ordre lexicographique pour les chaînes utilise les valeurs Unicode pour comparer les caractères. Quelques exemples de comparaison entre séquences de même type :

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

Notez que la comparaison d'objets de types différents avec `<` ou `>` est légale, à condition que ces objets possèdent les méthodes appropriées de comparaison. Par exemple, les types numériques sont comparés selon leur valeur numérique (0 est égal à 0.0, etc). Sinon, plutôt que de retourner une valeur arbitraire, l'interpréteur déclenchera une exception `TypeError`.

Chapitre 6

Modules

Si vous quittez l'interpréteur Python, puis si vous le relancez, les définitions que vous aviez faites (fonctions et variables) sont perdues. Il vaut mieux, si vous voulez écrire un programme plus conséquent, utiliser un éditeur de texte pour préparer vos instructions dans un fichier et lancer l'interpréteur avec ce fichier en entrée : c'est ce qu'on appelle créer un *script*. Au fur et à mesure que votre programme prendra de l'importance, il faudra même le découper en plusieurs fichiers, ce qui facilitera la maintenance. Vous voudrez peut-être aussi intégrer dans plusieurs programmes une fonction utilitaire que vous avez déjà écrite, sans avoir à recopier sa définition dans chacun d'eux.

Pour réaliser cela, Python vous permet de stocker les définitions dans un fichier et de les utiliser dans un script ou dans une session interactive de l'interpréteur. Un tel fichier est appelé un *module* ; les définitions d'un module peuvent être *importées* dans un autre module ou dans le module principal (*main module*) (l'ensemble des variables auxquelles vous pouvez accéder dans un script en cours d'exécution ou en mode calculatrice).

Un module est un fichier contenant des définitions et des instructions. Le nom du fichier est le nom du module suffixé par `.py`. À l'intérieur du module, son nom est disponible dans une variable globale de type chaîne appelée `__name__`. Par exemple, à l'aide de votre éditeur de texte favori, créez dans le répertoire courant un fichier `fibo.py` contenant :

`fibo.py`

```
# module nombres de Fibonacci

def fib(n):    # écrit la suite de Fibonacci jusqu'à n (exclu)
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a + b
    print()

def fib2(n): # retourne la suite de Fibonacci jusqu'à n (exclu)
    résultat = []
    a, b = 0, 1
    while b < n:
        résultat.append(b)
        a, b = b, a + b
    return résultat
```

Puis lancez l'interpréteur Python et importez ce module grâce à la commande suivante :

```
>>> import fibo
```

Cette instruction ajoute à la table des symboles courante le nom du module `fibo`, mais pas les noms des fonctions définies dans ce module. On peut accéder aux fonctions en utilisant l'identificateur `fibo` :

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

Si vous pensez utiliser souvent une fonction, vous pouvez l'affecter à un nom local :

```
>>> fib = fibo.fib
>>> fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

6.1 En savoir plus sur les modules

Un module peut contenir des instructions exécutables, et des définitions de fonctions. Ces instructions sont destinées à l'initialisation du module. Elles ne sont exécutées qu'une fois, la *première* fois que le nom du module est recontré dans une instruction `import` [1] (elles sont également exécutées si le fichier est lancé en tant que script).

Chaque module possède sa propre table des symboles privés, utilisée en tant que table des symboles globaux par toutes les fonctions définies dans le module. Ainsi, l'auteur d'un module peut utiliser des variables globales dans le module sans se préoccuper d'éventuels conflits avec les variables globales de l'utilisateur. D'autre part, si vous êtes sûr de ce que vous faites, vous pouvez modifier les variables globales du module en utilisant la même notation que celle utilisée pour accéder à une fonction. à savoir `nom_module.nom_variable`.

Les modules peuvent importer d'autres modules. Il est d'usage (sans être obligatoire) de placer toutes les instructions `import` en début de module (ou éventuellement de script,) Les noms des modules importés sont placés dans la tables des symboles du module importateur.

Une variante de l'instruction `import` permet d'importer directement des noms d'un module dans la table des symboles du module importateur. Par exemple :

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

Attention, dans ce cas le nom du module n'est pas ajouté à la table des symboles locaux (dans l'exemple, `fibo` n'est pas défini).

Il existe même une variante permettant d'importer tous les noms définis par un module :

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

Ici, tous les noms sont importés sauf ceux qui commencent par un tiret de soulignement (*underscore*, `'_'`). Dans la plupart des cas, les programmeurs Python n'utilisent pas cette facilité, parce qu'elle introduit un nombre indéterminé de noms, risquant de masquer d'autres noms préalablement définis.

Notez qu'en général cette pratique d'importer `*` à partir d'un module ou d'un paquetage est découragée, car elle produit un code difficilement lisible. Toutefois, cela reste acceptable pour économiser la saisie lors d'une session interactive.

> **Note:** Pour des raisons d'efficacité, chaque module n'est importé qu'une seule fois par session. Pour cette raison, si vous changez vos modules, vous devez relancer l'interpréteur – ou, s'il s'agit juste d'un module que vous voulez tester interactivement, utilisez `imp.reload()`, par exemple `import imp; imp.reload(nom_du_module)`.

6.1.1 Exécuter des modules en tant que scripts

Lorsque vous lancez un module Python avec

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

le code du module sera exécuté, comme si vous l'aviez importé, mais avec l'attribut `__name__` affecté par la valeur `"__main__"`. Cela signifie qu'en ajoutant ce code à la fin de du module `code.py` :

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

vous pouvez rendre votre fichier utilisable aussi bien en tant que script qu'en tant que module importable, car le code utilisant la ligne de commande n'est exécuté que si le module est lancé en tant que programme principal (*main*) :

```
$ python fibo.py 50

1 1 2 3 5 8 13 21 34
```

Si le module est importé, ce code n'est pas exécuté :

1. En fait, les définitions de fonction sont également des instructions qui sont exécutées ; l'exécution d'une définition de fonction au niveau du module ajoute le nom de cette fonction à la table des symboles globaux du module.

```
>>> import fibo
```

Cette possibilité est souvent utilisé pour fournir une interface commode à un module, ou dans le but d'effectuer des tests (lancer le module en tant que script exécute une séquence de tests).

6.1.2 Comment Python cherche les modules

Lorsqu'un module de nom `spam` est importé, l'interpréteur recherche s'il existe un module prédéfini portant ce nom. S'il n'en trouve pas, il cherche alors un fichier de nom `spam.py` dans une liste de chemins d'accès aux modules donnée par la variable `sys.path`. Cette variable est initialisée avec ces emplacements :

- le répertoire contenant le script lancé ou le répertoire courant quand aucun fichier n'est précisé lorsque Python est lancé ;
- `PYTHONPATH` : une liste de noms complets de répertoires, avec la même syntaxe que la variable système `PATH` ;
- des valeurs par défaut (noms de répertoires contenant des modules), qui dépendent de l'installation.

Note : Sur les systèmes prenant en charge les liens symboliques (*symlinks*), le répertoire contenant le script d'entrée est calculé après que le lien symbolique a été suivi. En d'autres termes, le répertoire contenant le lien symbolique n'est pas ajouté au chemin de recherche des modules (*module search path*).

Après l'initialisation, les programmes Python peuvent modifier `sys.path`. Le répertoire contenant le script est placé en début de liste de recherche, avant le chemin des modules de la bibliothèque standard. Cela signifie que les modules de ce répertoire seront importés à la place des modules de même nom de la bibliothèque standard. C'est en général une erreur, à moins que ce remplacement soit voulu. Voir la section 6.2 *Modules standard* pour plus d'information.

6.1.3 Fichiers Python "compilés"

Pour accélérer le chargement des modules, Python enregistre une version compilée de chaque module dans le répertoire `__pycache__` sous le nom ***module.version.pyc***, avec ***version*** encodant le format du fichier compilé (en général, le numéro de version de Python). Par exemple, avec la version 3.3 de CPython, la version compilée de `spam.py` serait enregistrée sous le nom `__pycache__/spam.cpython-33.pyc`. Cette convention de nommage permet la coexistence de modules compilés avec diverses versions de Python.

Python compare la date de dernière modification du programme source et celle du programme compilé pour voir s'il est nécessaire de le recompiler. C'est un processus entièrement automatique, ce qui fait que les modules compilés sont indépendants de la plate-forme (la même bibliothèque peut être partagée entre systèmes de différentes architectures).

Python ne teste pas le cache dans deux circonstances. Premièrement, il recompile systématiquement et n'enregistre pas le module compilé d'un module chargé directement depuis la ligne de commande. Deuxièmement, il ne teste pas le cache s'il n'y a pas de module source. Pour mettre en œuvre une distribution sans source (compilée uniquement), le module compilé doit se trouver dans le répertoire source, et il ne doit pas y avoir de module source.

Quelques astuces pour experts :

- Vous pouvez utiliser les options `-O` ou `-OO` dans la commande Python pour diminuer la taille du module compilé. L'option `-O` supprime les instructions `assert`, l'option `-OO` supprime les instructions `assert` et les chaînes `__doc__` . Comme certains programmes peuvent les utiliser, n'utilisez ces options que si vous savez ce que vous faites. Les modules "optimisés" ont une extension `.pyo` au lieu de `.pyc` et ils sont en général plus petits. Les futures versions peuvent changer les effets de cette optimisation.
- Un programme ne s'exécute pas plus vite s'il est lu à partir d'un fichier `.pyc` ou `.pyo` ; il n'y a que son chargement qui est accéléré.
- Le module `compileall` crée des fichiers `.pyc` (ou `.pyo` si `-O` est utilisé) pour tous les modules d'un répertoire.
- Plus de détails, y compris un organigramme des décisions, se trouvent ici : PEP 3147.

6.2 Modules standard

Python est fourni avec une bibliothèque de modules standard, décrits dans *The Python Library Reference*. Certains modules sont intégrés dans l'interpréteur ; ils donnent accès à des opérations qui ne font pas partie du langage au sens strict, mais ils sont néanmoins prédéfinis, soit pour des raisons d'efficacité, soit pour permettre l'utilisation de fonctions du système d'exploitation, comme les appels système. L'ensemble de ces modules dépend des options de configuration ainsi que de la plate-forme sur laquelle Python est installé. Ainsi, par

exemple, le module `winreg` n'est présent que sur les systèmes Windows. En particulier, un module mérite d'être étudié de plus près : le module `sys`, qui est intégré à chaque interpréteur Python. Les variables `sys.ps1` et `sys.ps2` définissent les chaînes utilisées en invites de commande primaire et secondaire :

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'À vos ordres : '
À vos ordres : print('Beurk !')
Beurk !
À vos ordres : 23+19
42
```

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 l'ensemble des chemins d'accès aux modules. Elle est initialisée à un chemin par défaut extraite de la variable d'environnement `PYTHONPATH`, or à une valeur prédéfinie si `PYTHONPATH` n'est pas définie. On la peut modifier grâce aux opérations standard sur les listes :

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

6.3 La fonction `dir()`

La fonction prédéfinie `dir()` permet de savoir quels noms sont définis dans un module. Elle retourne une liste triée de chaînes de caractères :

```
>>> import fibo, sys
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__', '__interactivehook__',
 '__loader__', '__name__', '__package__', '__plen', '__spec__', '__stderr__', '__stdin__',
 '__stdout__', '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'api_version', 'argv', 'base_exec_prefix',
 'base_prefix', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'getallocatedblocks', 'getcheckinterval', 'getdefaultencoding',
 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'getswitchinterval', 'gettrace', 'getwindowsversion', 'hash_info', 'hexversion',
 'implementation', 'int_info', 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules',
 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'setcheckinterval',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr', 'stdin',
 'stdout', 'thread_info', 'version', 'version_info', 'warnoptions', 'winver']
```

Sans arguments, `dir()` liste les noms définis par l'utilisateur :

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

Notez que la liste contient tous types de noms : variables, modules, fonctions, etc.

`dir()` ne liste pas les noms des fonctions et variables prédéfinies. si on les veut, elle sont définies dans le module standard `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 Packages

Les *packages* sont un moyen de structurer l'espace de noms d'un module Python, par l'utilisation de la notation pointée dans les noms de modules (*dotted module names*). Par exemple, le nom de module **`A.B`** désigne un sous-module **`B`** dans un package appelé **`A`**. Tout comme l'utilisation de modules permet aux auteurs de différents modules de ne pas se soucier des conflits entre noms de variables, l'utilisation de points dans les noms de modules permet aux auteurs de packages multi-modules comme NumPy ou Python Imaging Library de ne pas se soucier des conflits entre noms de modules.

Supposons que vous vouliez créer une collection de modules (un *package*) pour une gestion uniformisée de fichiers et de données sons. Il existe de nombreux formats de fichiers sons (généralement reconnus par leurs extensions : **`.wav`**, **`.aiff`**, **`.au`**), aussi vous aurez besoin de créer et de maintenir un ensemble croissant de modules pour les conversions entre les différents formats de fichiers. Il y aura également de nombreuses opérations différentes que vous voudrez utiliser sur des données sons (comme mixer, ajouter de l'écho, appliquer une fonction d'égaliseur, créer un effet stéréo artificiel), ce qui fait que vous allez écrire un nombre indéterminé de modules pour réaliser ces opérations. Voici une structure possible pour votre package (sous forme d'une arborescence de système de fichiers) :

```
sons/                          Package de premier niveau
      __init__.py              Initialise le package sons
      formats/                 Sous-package conversions de formats
              __init__.py
              wav_lire.py
              wav_érire.py
              aiff_lire.py
              aiff_écrire.py
              au_lire.py
              au_écrire.py
              ...
      effets/                 Sous-package effets sonores
              __init__.py
              écho.py
              ambiophonie.py
              rebours.py
              ...
      filtres/                  Sous-package filtres
              __init__.py
              égaliseur.py
              codeur_vocal.py
              karaoke.py
              ...
```

Pour importer un package, Python parcourt les répertoires listés dans **`sys.path`** pour trouver le sous-répertoire du package.

La présence d'un fichier **`__init__.py`** est obligatoire pour que Python considère qu'un répertoire contient un package ; cette précaution est nécessaire pour empêcher qu'un répertoire de nom commun, comme **`string`** soit considéré comme un package, et cache des modules valides apparaissant plus tard dans le chemin de recherche des modules. Dans les cas les plus simples, il suffit que **`__init__.py`** soit un fichier vide, mais il peut aussi contenir du code d'initialisation du package ou définir la variable **`__all__`** décrite plus loin.

Les utilisateurs d'un package peuvent en importer des modules individuels, par exemple :

```
import sons.effets.écho
```

Cette instruction charge le sous-module **sons.effets.écho**. Il doit être référencé par son nom complet.

```
sons.effets.écho.écho_filtre(entrée, sortie, délai=0.7, essais=4)
```

Une façon alternative d'importer le sous-module :

```
from sons.effets import écho
```

Cette instruction charge également le sous-module module **écho**, mais il peut (et doit) être maintenant référencé sans son préfixe, il peut donc être utilisé comme suit :

```
écho.écho_filtre(entrée, sortie, délai=0.7, essais=4)
```

Une autre possibilité est d'importer la fonction ou variable désirée directement :

```
from sons.effets.écho import écho_filtre
```

Là encore, le sous-module **écho** est accédé, mais maintenant la fonction **écho_filtre()** est disponible directement :

```
écho_filtre(entrée, sortie, délai=0.7, essais=4)
```

Notez que l'utilisation de l'instruction **from *package* import *item***, l'*item* peut être un sous-module (ou sous-package) du package, ou tout autre nom défini dans le package comme une fonction, une classe ou une variable. L'instruction **import** teste d'abord si l'*item* est défini dans le package ; si non, il suppose que c'est un module et tente de le charger. S'il ne le trouve pas, il déclenche une exception **ImportError**.

Au contraire, en utilisant la syntaxe **import *item.subitem.subsubitem***, chaque *item* sauf le dernier doit être un package ; le dernier item peut être un module ou un package, mais **ne peut pas être** une classe, une fonction ou une variable définie dans l'item précédent.

6.4.1 Importer * d'un package

Maintenant, que se passe-t-il quand l'utilisateur écrit **from sons.effets import *** ? On pourrait espérer que cette instruction parcourt le système de fichiers, trouve quels sous-modules sont présents dans le package et les importe tous. Cela prendrait beaucoup de temps et importer des sous-modules qui risqueraient d'avoir des effets de bord non désirés, qui ne devraient se produire que si le module était importé explicitement.

La seule solution pour l'auteur d'un package est de fournir un index explicite pour le package. L'instruction **import** suit la convention suivante : si le code du fichier **__init__.py** du package définit une liste de nom **__all__**, elle est considérée comme une liste des noms des modules devant être chargés lorsque l'instruction **from *package* import *** est rencontrée. C'est à l'auteur du package de maintenir cette liste à jour lorsqu'une nouvelle version du package est publiée. Les auteurs de packages peuvent aussi décider de ne pas mettre en œuvre cette possibilité, s'ils ne voient pas l'utilité d'importer * depuis leur package. Par exemple, le fichier **sons/effets/__init__.py** pourrait contenir le code suivant :

```
__all__ = ["écho", "ambiophonie", "rebours"]
```

Cela signifie que l'instruction **from sons.effets import *** importera les trois sous-modules indiqués à partir du package **sons.effets**.

Si **__all__** n'est pas défini, l'instruction **from sons.effets import *** n'importera pas tous les sous-modules du package **sons.effets** dans l'espace de noms en cours ; elle assurera simplement que le package **sons.effets** a été importé (éventuellement en exécutant le code d'initialisation de **__init__.py**), puis importera tous les noms définis dans le package, y compris les noms définis et les modules explicitement chargés dans **__init__.py**. Seront également inclus tous les sous-modules du package explicitement chargés par des instructions **import** préalables. Considérons ce code :

```
import sons.effets.écho
import sons.effets.ambiophonie
from sons.effets import *
```

Dans cet exemple, les modules **écho** et **ambiophonie** sont importés dans l'espace des noms courant car définis dans le package **sons.effets** lorsque l'instruction **from...import** est exécutée. (cela marche aussi quand **__all__** est défini).

Bien que certains modules soient conçus pour n'exporter que les noms autotisés lorsque vous utilisez `import *`, cette pratique n'est pas encouragée dans la production de code.

Souvenez-vous, il n'y a rien de mal à utiliser `from package import sous_module_spécifique` ! En fait, c'est même la notation recommandée sauf si le module importateur utilise des sous-modules de même nom dans différent packages.

6.4.2 Référence intra-package

Lorsque les packages sont structurés en sous-packages (comme avec le package `sons` de l'exemple), vous pouvez faire des importations en utilisant des références absolues pour les sous-modules de packages parents. Par exemple, si le module `sons.filtres.codeur_vocal` nécessite l'importation du module `écho` du package `sons.effets`, il peut utiliser `from sons.effets import écho`.

Vous pouvez également faire des importations en utilisant des références realtives, avec la forme `from module import name` de l'instruction `import`. Ces importations utilisent les points initiaux pour indiquer le package courant et le package parent impliqués dans l'importation relative. À partir du module `ambliophonie` par exemple, on peut écrire :

```
from . import écho
from .. import formats
from .. filtres import égaliseur
```

Notez que ces importations relatives sont basées sur le nom du module courant. Comme le nom du module principal est toujours `"__main__"`, un module destiné à être le module principal d'une application Python doit toujours utiliser des importations absolues.

6.4.3 Packages multi-répertoires

Les packages possèdent un attribut spécial supplémentaire, `__path__`. Il est initialisé par une liste contenant le nom du répertoire où se situe le fichier `__init__.py` du module, avant que le code s'y trouvant soit exécuté. Cette variable peut être modifiée ; une modification affectera les recherches ultérieures de modules et sous-packages contenus dans ce package

Bien que cette possibilité soit rarement nécessaire, elle peut être utilisée pour augmenter l'ensemble des modules accessibles dans un package.

Chapitre 7

Entrées et sorties

Il existe plusieurs manières de présenter les sorties d'un programme : les données peuvent être affichées sous une forme lisible, ou écrites dans un fichier pour une utilisation ultérieure. Ce chapitre présente quelques-unes de ces possibilités.

7.1 Formatage raffiné de sortie

Nous avons vu jusqu'ici deux façons d'écrire des valeurs : les *instructions d'expression* et la fonction `print()` (un troisième moyen est la méthode `write()` des objects fichiers (*file objects*) ; le fichier de sortie standard peut être référencé par `sys.stdout`. Voir *The Library Reference* pour plus d'information).

Il arrivera souvent que vous désiriez plus de contrôle sur le formatage de vos sorties que le simple affichage des données séparées par des espaces. Il y a deux façons de formater vos sorties ; le premier est de tout gérer par vous-même ; en utilisant les découpes et les concaténations, vous pouvez créer toutes les mises en page que vous pouvez imaginer. Le type chaîne de caractères (`str`) possède des méthodes qui permettent de justifier des textes sur des largeurs données ; elles seront présentées brièvement. la deuxième façon est d'utiliser la méthode `str.format()`.

Le module `string` contient la classe `Template` qui offre un autre moyen de transformer des valeurs en chaînes.

Toutefois, une question demeure : comment convertir des valeurs quelconques en chaînes de caractères ? Par chance, Python permet de convertir une valeur de n'importe quel type en une chaîne : passer cette valeur en argument de la fonction `repr()` ou de la fonction `str()`.

La fonction `str()` permet de retourner des représentations de valeurs lisibles par un humain, alors que `repr()` génère des représentations lisibles par l'interpréteur (ou qui générera une `SyntaxError` si la syntaxe est incorrecte). Pour les objets qui n'ont pas de représentation particulière destinée à l'humain, `str()` retournera la même valeur que `repr()`. Des valeurs, comme les nombres, ou des structures, comme les listes ou les dictionnaires, ont la même représentation, que vous utilisiez `str()` ou `repr()`. Toutefois, les chaînes de caractères ont deux représentations distinctes.

Quelques exemples :

```
>>> s = 'Salut la compagnie !'
>>> str(s)
'Salut la compagnie !'
>>> repr(s)
"'Salut la compagnie !'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'La valeur de x est ' + repr(x) + ', et celle de y est ' + repr(y) + '...'
>>> print(s)
La valeur de x est 32.5, et celle de y est 40000...
>>> # repr() sur une chaîne lui ajoute des quotes et des backslashes:
>>> hello = 'Salut la compagnie !\n'
>>> repr(hello)
"'Salut la compagnie !\\n'"
>>> # l'argment de repr() peut être n'importe quel objet Python :
>>> repr((x, y, ('spam', 'oeufs')))
"(32.5, 40000, ('spam', 'oeufs'))"
```

Suivent deux façons d'afficher une table de carrés et de cubes :

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Notez l'usage de 'end' dans la ligne précédente
...     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
 9  81  729
10 100 1000
>>> 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
```

Notez que dans le premier exemple, une espace est ajoutée entre les colonnes de par la façon dont `print()` fonctionne : elle ajoute toujours des espaces entre ses arguments.

Cet exemple montre l'utilisation de la méthode `str.rjust(largeur)` sur les objets chaînes de caractères, qui justifie à droite la chaîne dans une zone de largeur donnée, en la remplissant à gauche avec des espaces. Il existe des méthodes similaires : `str.ljust(largeur)` (justification à gauche) et `str.center(largeur)` (centrage). Ces méthodes n'affichent rien, elles se contentent de retourner une nouvelle chaîne. Si la chaîne fournie est trop longue, elles ne la tronquent pas, mais la retournent inchangée ; cela abîmera votre mise en page, mais c'est généralement mieux que la troncature, qui vous affichera une valeur fausse (si vous désirez réellement une troncature, vous pouvez utiliser une découpe, comme dans `x.ljust(n)[:n]`).

Une autre méthode, `str.zfill(largeur)`, fait précéder une chaîne numérique de zéros. Elle sait gérer les signes plus et moins :

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

L'utilisation basique de la méthode `str.format()` ressemble à ça :

```
>>> print('Nous somme les {} qui disent "{}!"'.format('chevaliers', 'Ni'))
Nous somme les chevaliers qui disent "Ni!"
```

Les accolades et les caractères qu'elles encadrent, appelés zones de formatage (*format fields*) sont remplacés par les objets passés en argument de la méthode `str.format()`. Un nombre dans les accolades s'utilise pour spécifier la position de l'objet dans la liste des arguments passés à la méthode `str.format()` :

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

Si des arguments mot-clés sont utilisés dans la méthode `str.format()`, leurs valeurs sont référencées par le nom de l'argument.

```
>>> print('Ce {plat} est {adjectif}.'.format(plat='spam', adjectif='vraiment infect'))
Ce spam est vraiment infect.
```

Les arguments positionnels et mot-clés peuvent être combinés :

```
print("L'histoire de {0}, {1} et {autre}.".format('Bill', 'Manfred', autre='Georg'))
```

Les options `'!a'` (applique `ascii()`), `'!s'` (applique `str()`) et `'!r'` (applique `repr()`) permettent de

convertir les valeurs avent le formattage :

```
>>> print("!s -> {0!s}".format("tête"))
!s -> tête
>>> print("!r -> {0!r}".format("tête"))
!r -> 'tête'
>>> print("!a -> {0!a}".format("tête"))
!a -> 't\xeate'
```

Le caractère deux-points (`':'`), suivi d'un spécificateur de format peut se situer après le nom de la zone, ce qui permet un meilleur contrôle sur le formatage de la valeur. L'exemple suivant arrondit Pi à trois chiffres après la virgule.

```
>>> import math
>>> print('La valeur de PI est environ {0:.3f}'.format(math.pi))
La valeur de PI est environ 3.142
```

Un nombre entier suivant le `':'` rendra la zone au moins aussi large que cet entier (en nombre de caractères). On utilise cette fonctionnalité pour réaliser de beaux alignements :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nom, poste in table.items():
...     print('{0:10} ==> {1:10d}'.format(nom, poste))
...
Dcab       ==>       7678
Sjoerd     ==>       4127
Jack       ==>       4098
```

Si vous avez une très longue chaîne de format que vous ne voulez pas couper, il serait commode de pouvoir référencer les variables à formater par leur nom plutôt que par leur position. On peut le faire en fournissant simplement le dictionnaire et en utilisant les 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
```

On peut également le faire en passant le dictionnaire en tant qu'argument par mot-clé grâce à 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
```

Ceci est particulièrement utile en association avec la fonction prédéfinie `vars()`, qui retourne un dictionnaire contenant toutes les variables locales.

Pour une information complète sur les formatages de chaînes avec `str.format()`, voir *Format String Syntax*.

7.1.1 Formatage de chaînes à l'ancienne

L'opérateur `%` peut également être utilisé pour le formatage de chaînes. Il interprète son opérande gauche comme un format similaire à celui de `printf()` du langage C à appliquer à son opérande droit, et retourne la chaîne résultant de ce formatage. Par exemple :

```
>>> import math
>>> print('La valeur de PI est environ %5.3f.' % math.pi)
La valeur de PI est environ 3.142.
```

On peut trouver plus d'information dans la documentation *printf-style String Formatting*.

7.2 Lire et écrire dans des fichiers

La fonction `open()` retourne un objet fichier (*file object*) ; elle est habituellement invoquée avec deux arguments : `open(nom_fichier, mode)`.

```
>>> f = open('workfile', 'w')
```

Le premier argument contient le nom du fichier à utiliser. Le deuxième argument est aussi une chaîne contenant des caractères spécifiant la façon dont le fichier va être utilisé. *mode* peut être `'r'` pour ouvrir le fichier en mode lecture (le fichier ne sera que consulté), `'w'` pour ouvrir le fichier en mode écriture (on crée un fichier dans lequel on ne peut qu'écrire ; si un fichier de même nom existe déjà, il sera effacé) ou `'a'` pour ouvrir le fichier en mode adjonction (les données écrites dans le fichier le seront systématiquement en fin de fichier). `'r+'` ouvre le fichier à la fois en lecture et en écriture. L'argument *mode* est optionnel ; `'r'` est la valeur par défaut.

Habituellement, les fichiers sont ouverts en mode *texte*, ce qui signifie que l'on y lit et écrit des chaînes de caractères, encodées dans un encodage spécifique (par défaut, UTF-8). `'b'` ajouté à l'argument *mode* ouvre le fichier en mode *binaire* : les données seront lues et écrites sous forme d'objets binaires (*bytes objects*). Ce mode devrait être utilisé pour tous les fichiers contenant autre chose que du texte.

En mode texte, la lecture convertit les fins de lignes spécifiques aux plates-formes (`\n` on Unix, `\r\n` on Windows) en `\n`. L'écriture convertit les occurrences de `\n` en fins de lignes spécifiques aux plates-formes. Cette modification faite en coulisse convient bien aux fichiers textes, mais corrompt les données binaires comme dans les fichiers `JPEG` ou `EXE`. Soyez très attentifs et utilisez le mode binaire pour travailler sur ce genre de fichiers.

7.2.1 Méthodes sur les objets fichiers

Les exemples suivants feront tous l'hypothèse qu'un objet fichier de nom `f` a déjà été créé.

Pour lire le contenu d'un fichier, on invoque `f.read(taille)`, qui lit une certaine quantité de données et les retourne en tant que chaîne ou séquence d'octets. *taille* est un argument numérique optionnel. Si l'argument *taille* n'est pas précisé ou est négatif, tout le contenu du fichier sera lu et retourné ; c'est votre problème si votre fichier est deux fois plus grand que la mémoire de votre ordinateur. Sinon, au plus *taille* caractères ou octets seront lu et retournés. Si la fin de votre fichier a été atteinte, `f.read()` retournera une chaîne vide (`''`).

```
>>> f.read()
"C'est le fichier tout entier.\n"
>>> f.read()
''
```

Pour les fichiers texte, `f.readline()` lit une ligne du fichier ; un caractère fin de ligne (`'\n'`) est systématiquement laissé en fin de chaîne, sauf pour le dernière ligne du fichier si celle-ci ne contient pas de caractère fin de ligne. Cela permet de lever toute ambiguïté sur la valeur de retour ; si `f.readline()` retourne une chaîne vide, la fin de fichier a été atteinte, alors qu'une ligne vide est représentée par `'\n'`, une chaîne ne contenant qu'un caractère fin de ligne.

```
>>> f.readline()
"C'est la première ligne du fichier.\n"
>>> f.readline()
'Deuxième ligne du fichier\n'
>>> f.readline()
''
```

Pour lire une à une les lignes d'un fichier texte, on peut boucler sur lui. Cela économise de la mémoire, c'est rapide, et le code reste simple :

```
>>> for ligne in f:
...     print(ligne, end='')
...
C'est la première ligne du fichier.
Deuxième ligne du fichier
```

Pour créer une liste de toutes les lignes du fichier, on peut utiliser `list(f)` or `f.readlines()`.

`f.write(chaîne)` écrit le contenu de la *chaîne* dans le fichier, et retourne le nombre de caractères écrits.

```
>>> f.write("C'est un test\n")
14
```

Pour écrire autre chose que du texte, il faut déjà le convertir en texte :

```
>>> value = ('la RÉPONSE', 42)
>>> s = str(value)
>>> f.write(s)
18
```

`f.tell()` retourne un entier donnant la position courante dans le fichier : un nombre d'octets depuis le début du fichier en mode binaire, un nombre opaque (uniquement utilisable par `f.seek()`) en mode texte).

Pour changer la position courante dans l'objet fichier, on utilise `f.seek(décalage, à_partir_de)`. La position est calculée en ajoutant *décalage* à un point de référence, choisi selon la valeur de l'argument *à_partir_de* argument. Si *à_partir_de* vaut 0, on part du début de fichier, s'il vaut 1, on part de la position courante et s'il vaut 2, on part de la fin de fichier. *à_partir_de* peut être omis (0 est la valeur par défaut, faisant partir du début de fichier).

```
>>> f = open('fichier_essai', 'wb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Se positionne sur le 6e octet du fichier (le no 5)
5
>>> f.read(1)     # Lit l'octet courant
b'5'
>>> f.seek(-3, 2) # Recule de trois places à partir de la fin.
13
>>> f.read(1)
b'd'
```

Pour les fichiers texte (ceux ouverts sans `'b'` dans l'argument *mode*), seuls les positionnements à partir du début de fichier sont autorisés (la seule exception étant le positionnement en toute fin de fichier avec `seek(0, 2)`) et les seules valeurs de *décalage* valides sont celles retournées par un appel à `f.tell()`, ou 0. Toute autre valeur de décalage aura un comportement non défini.

Lorsque vous en avez terminé avec un fichier, invoquez `f.close()` pour le fermer et libérer les ressources allouées par le système à l'ouverture. Après l'exécution de `f.close()`, toute tentative d'accéder à l'objet fichier échouera.

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

Une bonne habitude est celle d'utiliser le mot-clé `with` pour travailler avec des objets fichier. Cela a pour avantage de fermer proprement le fichier après sa suite d'instructions, même si une exception est levée en cours. C'est aussi plus court que d'écrire les bloc équivalents `try`-`finally` :

```
>>> with open('fichier_essai', 'r') as f:
...     données_lues = f.read()
>>> f.closed
True
```

Les objets fichier possèdent quelques méthodes supplémentaires comme `isatty()` et `truncate()` plus rarement utilisées ; consultez *the Library Reference* pour un guide complet sur les objets fichier (*file objects*).

7.3 Sauvegarde de données structurées avec `json`

Les chaînes peuvent être facilement écrites ou lues depuis un fichier. Les nombres demandent un petit effort supplémentaire, car la méthode `read()` ne retourne que des chaînes, qui doivent être passées à une fonction comme `int()`, laquelle prend en argument une chaîne comme `'123'` et retourne sa valeur numérique `123`. Pour sauvegarder des valeurs de types plus compliqué, comme des listes imbriquées ou des dictionnaires, analyser et sérialiser à la main devient compliqué.

Plutôt que d'obliger les utilisateurs à éternellement écrire et déboguer du code permettant de sauvegarder des données de types compliqués, Python vous permet d'utiliser le format d'échange de données répandu appelé JSON (JavaScript Object Notation). Le module standard `json` peut prendre des hiérarchies de données et les convertit en représentation chaîne ; ce processus s'appelle la sérialisation (*serializing*). Reconstruire les données à partir de leur représentation chaîne s'appelle la désérialisation (*deserializing*). Entre la sérialisation et la désérialisation, la chaîne représentant l'objet peut être stockée dans un fichier ou une donnée, ou envoyée par une connexion réseau ou distante à une autre machine.

> **Note:** le format JSON est très utilisé par les applications modernes pour permettre l'échange de données. De nombreux programmeurs y sont déjà habitués, ce qui en fait un choix recommandé pour l'interopérabilité.

On peut visualiser le représentation chaîne d'un objet `x` en une seule ligne de code :

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

```
'[1, "simple", "list"]'
```

Une variante de la fonction `dumps()` est la fonction `dump()`, qui sérialise un objet vers un objet fichier texte (*text file*). Ainsi, si `f` est un objet fichier texte ouvert en écriture, nous pouvons écrire ceci :

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

Pour décoder l'objet sauvegardé, si `f` est un objet fichier texte ouvert en lecture :

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

Les techniques de base de sérialisation permettent de gérer listes et dictionnaires, mais la sérialisation d'instances de classes quelconques avec JSON nécessite des efforts supplémentaires. La documentation pour le module `json` contient toutes les explications permettant de le faire.

Voir aussi : `pickle` - the pickle module

Contrairement à *JSON* (voir 7.3), *pickle* est un protocole permettant la sérialisation de toutes sortes d'objets Python. Toutefois, comme il est spécifique à Python, il ne peut pas être utilisé pour communiquer avec des applications écrites dans d'autres langages. Il est de plus non sécurisé par défaut : la désérialisation de données *pickle* en provenance d'une source non fiable peut exécuter un code arbitraire, si la donnée a été préparée par un programmeur malveillant.

Chapitre 8

Erreurs et exceptions

Jusqu'à présent, nous avons simplement mentionné les messages d'erreurs, et, si vous avez essayé d'entrer quelques exemples, vous en avez certainement rencontrés. Il y a au moins deux sortes d'erreurs : les erreurs de syntaxe (*syntax errors*) et les exceptions.

8.1 Les erreurs de syntaxe

Les erreurs de syntaxe, également appelées erreurs de l'analyseur syntaxique (*parsing errors*), sont certainement les erreurs les plus fréquemment rencontrées lorsque l'on apprend Python.

```
>>> if 2 + 2 == 4 print('Le monde est bien fait')
  File "<stdin>", line 1
    if 2 + 2 == 4 print('Le monde est bien fait')
                  ^
SyntaxError: invalid syntax
```

En cas d'erreur, l'analyseur syntaxique reproduit la ligne en cause et affiche une petite flèche indiquant l'endroit de la ligne où l'erreur a été détectée. L'erreur est causée (ou au moins détectée) par le texte *précédant* la flèche : dans l'exemple, l'erreur est détectée à l'appel de la fonction `print()`, car il manque un deux-points (`':'`) avant l'appel. Le nom du fichier et le numéro de ligne sont affichés, pour que vous puissiez savoir où regarder si le programme provient d'un script.

8.2 Exceptions

Même si une instruction ou une expression est syntaxiquement correcte, elle peut provoquer une erreur lorsqu'on tente de l'exécuter. Les erreurs détectées à l'exécution sont appelées des *exceptions* et ne sont pas forcément fatales : vous apprendrez bientôt comment les gérer dans des programmes Python. Toutefois, la plupart des exceptions ne sont pas gérées par les programmes, et conduisent à des messages d'erreurs comme ci-dessous :

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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't convert 'int' object to str implicitly
```

La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions sont de plusieurs types et le type d'exception est affiché dans le message : dans l'exemple, les types sont `ZeroDivisionError`, `NameError` et `TypeError`. la chaîne affichée pour indiquer le type d'exception est le nom de l'exception prédéfinie qui s'est produite. Ceci est vrai pour toutes les exceptions prédéfinies, mais pas forcément pour les exceptions définies par l'utilisateur (bien que ce soit une convention recommandée). Les noms d'exception standard sont des identificateurs prédéfinis (pas des mots-clés réservés du langage).

Le reste de la ligne fournit des détails sur le type d'exception et ce qui l'a causée.

Les premières lignes du message d'erreur montrent le contexte d'exécution lors de l'erreur, sous forme d'une pile d'appels. En général, c'est une liste de lignes des programmes sources de la pile des appels : toutefois, les lignes lues depuis l'entrée standard ne sont pas affichées.

La documentation *Built-in Exceptions* détaille les exceptions prédéfinies et leurs significations.

8.3 Gérer les exceptions

Il est possible d'écrire des programmes qui gèrent certaines exceptions spécifiées. L'exemple suivant boucle sur une demande à l'utilisateur d'entrer un nombre, jusqu'à ce qu'il entre un entier valide, mais permet à l'utilisateur d'interrompre le programme (avec `Control-C` ou une touche d'interruption autorisée par le système) ; notez que l'interruption utilisateur est signalée par la levée d'une exception `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Entre un nombre entier : "))
...         break
...     except ValueError:
...         print("Même pas un vrai nombre ! Essaye encore...")
...
```

L'instruction `try` fonctionne comme suit :

- tout d'abord, la clause *try* (les instructions entre les mots-clés `try` et `except`) sont exécutées ;
- si aucune exception ne se produit, la clause *except* est ignorée et l'exécution de l'instruction `try` se termine ;
- si une exception se produit pendant l'exécution de la clause *try*, la suite de la clause est ignorée ; puis, si le type de l'exception correspond à l'exception précisée après le mot-clé `except`, la clause *except* est exécutée et l'exécution reprend après l'instruction `try` ;
- si une exception se produit et qu'elle ne correspond pas à l'exception précisée après le mot-clé `except`, elle est transmise à l'instruction englobante `try` ; si aucun gestionnaire n'est trouvé, c'est une *exception non gérée* et l'exécution s'interrompt avec un message d'erreur comme vu précédemment.

Une instruction `try` peut contenir plus d'une clause *except* , pour spécifier des gestions de différentes exceptions. Un gestionnaire au plus sera exécuté. Un gestionnaire ne s'occupe que des exceptions se produisant dans la *clause try* correspondante, pas de celles qui se produisent dans d'autres gestionnaires de la même instruction `try`. Une *clause except* peut spécifier plusieurs exceptions dans un tuple parenthésé, par exemple :

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

La dernière clause *except* peut ne pas mentionner de noms d'exceptions, pour servir de joker. À n'utiliser qu'avec précaution, car cette manière de faire peut très facilement masquer une réelle erreur de programmation ! Elle peut aussi afficher un message d'erreur et relancer l'exception (ce qui permet à l'appelant de gérer lui-même l'exception) :

```
import sys

try:
    f = open('mon_fichier.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Erreur système : {0}".format(err))
except ValueError:
    print("Impsossible de convertir la donnée lue en un entier.")
except:
    print("Erreur non prévue : ", sys.exc_info()[0])
    raise
```

L'instruction `try ... except` peut être suivie d'une clause *else* facultative, qui, si elle est présente, doit se trouver après toutes les clauses *except*. Elle est utilisée pour y mettre du code qui doit être eécuté si la clause *try* n'a pas levé d'exception. Par exemple :

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print(arg, ': ouverture impossible')
    else:
```

```
        print(arg, 'possède', len(f.readlines()), 'lignes.')
        f.close()
```

L'utilisation de la clause **else** est une meilleure façon de faire que d'ajouter du code à la clause **try**, car elle évite de traiter accidentellement une exception qui n'aurait pas été levée dans le code protégé par l'instruction **try ... except**.

Lorsqu'une exception se produit, elle peut avoir une valeur associée, appelée *argument d'exception*. La présence et le type de l'argument dépend du type de l'exception.

La clause *except* peut spécifier une variable introduite pas **as** après le nom de l'exception. La variable est liée à une instance de l'exception avec les arguments stockés dans **instance.args**. Par commodité, l'instance d'exception définit une méthode **__str__()** , ce qui fait que les arguments peuvent être affichés directement, sans avoir à référencer ***var*.args**. On peut également instancier une exception avant de la lancer, et lui ajouter tous les attributs que l'on veut :

```
>>> try:
...     raise Exception('spam', 'œufs')
... except Exception as instance:
...     print(type(instance))   # l'instance d'exception
...     print(instance.args)    # les arguments (stockés dans args)
...     print(instance)         # __str__ permet d'afficher directement args
...                             # mais peut être surchargée
...                             # dans les  sous-classes d'exception
...     x, y = instance.args # déballage de args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'œufs')
('spam', 'œufs')
x = spam
y = œufs
```

Si une exception possède des arguments, ils sont affichés dans la dernière partie (**detail**) du message d'erreur affiché pour une exception non gérée.

Les gestionnaires d'exception ne gèrent pas uniquement les exceptions se produisant directement dans la clause *try*, mais aussi celles se produisant dans les fonctions appelées, même indirectement. Par exemple :

```
>>> def ça_va_planter():
...     x = 1/0
...
>>> try:
...     ça_va_planter()
... except ZeroDivisionError as erreur:
...     print("Gestion d'une erreur à l'exécution :", erreur)
...
Gestion d'une erreur à l'exécution : division by zero
```

8.4 Lancer des exceptions

L'instruction **raise** permet au programmeur de forcer le déclenchement d'une exception. Par exemple :

```
>>> raise NameError("C'est moi qui l'ai lancée !")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: C'est moi qui l'ai lancée !
```

L'unique argument de **raise** indique l'exception à lancer. cela peut être soit une instance d'exception, soit une classe exception (héritant de la classe **Exception**).

Si vous avez simplement besoin de savoir si une exception a été lancée, mais si vous ne comptez pas la gérer, une forme simplifiée de l'instruction **raise** permet de relancer l'exception :

```
>>> try:
...     raise NameError("C'est moi qui l'ai lancée !")
... except NameError:
...     print('une exception passe !')
...     raise
...
une exception passe !
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
NameError: C'est moi qui l'ai lancée !
```

8.5 Exceptions définies par l'utilisateur

Les programmes peuvent disposer de leurs propres exceptions en créant une nouvelle classe d'exception (voir le chapitre 9 *Classes* pour plus d'information sur les classes). Les exceptions doivent dériver de la classe `Exception`), directement ou indirectement. Par exemple :

```
>>> class MonErreur(Exception):
...     def __init__(self, valeur):
...         self.valeur = valeur
...     def __str__(self):
...         return repr(self.valeur)
...
>>> try:
...     raise MonErreur(2*2)
... except MonErreur as e:
...     print("Mon exception s'est produite, valeur  :" , e.valeur)
...
Mon exception s'est produite, valeur  : 4
>>> raise MonErreur('Oups !')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__console__.MonErreur: 'Oups !'
```

Dans cet exemple, la méthode `__init__()` de la classe of `Exception` a été redéfinie, en créant simplement l'attribut *valeur*. Ceci remplace le comportement par défaut qui crée l'attribut *args*.

Les classes d'exceptions définies par l'utilisateur peuvent faire tout ce qu'une autre classe peut faire, mais généralement, elles restent simples, se contentant de fournir quelques attributs permettant au gestionnaire de disposer d'informations sur l'erreur. Lorsque l'on crée un module susceptible de provoquer plusieurs erreurs distinctes, il est d'usage de créer une classe de base pour les exceptions du modules et de la sous-classer pour créer les classes d'exceptions spécifiques pour les différentes condtions d'erreur :

```
class Erreur(Exception):
    """Classe de base pour les exceptions de ce module."""
    pass

class ErreurDeSaisie(Erreur):
    """Exception levée pour les erreurs dans les entrées .

    Attributs:
        expression -- expression d'entrée dans laquelle l'erreur est apparue
        message -- explication de l'erreur
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class ErreurDeTransition(Erreur):
    """Levée lorsqu'une opération tente une transition d'état non autorisée.

    Attributs:
        précédent -- état en début de transition
        suivant -- nouvel état à atteindre
        message -- explication des raisons de l'impossibilité du changement d'état
    """

    def __init__(self, précédent, suivant, message):
        self.précédent = précédent
        self.suivant = suivant
        self.message = message
```

La plupart des exceptions ont des noms se terminant par "Error", comme les noms des exceptions standard [1].

Beaucoup de modules standard définissent leurs propres exceptions pour reporter les erreurs qui peuvent se produire dans les fonctions qu'ils définissent. Vous trouverez plus d'information sur les classes dans le chapitre 9 *Classes*.

1. Ce n'est pas le cas dans cette traduction, pour bien différencier les noms utilisateur des noms prédéfinis

8.6 Définir des actions de nettoyage

L'instruction **try** possède une clause supplémentaire optionnelle **finally** destinée à définir les actions de "nettoyage" à exécuter en toutes circonstances. Par exemple :

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Adieu, monde cruel !')
...
Adieu, monde cruel !
KeyboardInterrupt
```

Une clause *finally* est toujours exécutée avant de quitter l'instruction **try**, qu'une exception se soit produite ou non. Lorsqu'une exception s'est produite dans la clause *try* et n'a pas été gérée dans une clause *except* (ou si elle s'est produite dans une clause *except* ou une clause *else*), elle est relancée après que la clause *finally* a été exécutée. La clause *finally* est également exécutée "en sortant" quand n'importe quelle clause de l'instruction **try** est quittée, par un **break**, un **continue** ou un **return**. Un exemple plus compliqué :

```
>>> def divise(x, y):
...     try:
...         résultat = x / y
...     except ZeroDivisionError:
...         print("division par zéro !")
...     else:
...         print("le résultat est ", résultat)
...     finally:
...         print("on est dans la clause finally")
...
>>> divise(2, 1)
le résultat est  2.0
on est dans la clause finally

>>> divise(2, 0)
division par zéro !
on est dans la clause finally

>>> divise("2", "1")
on est dans la clause finally
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divise
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Comme vous pouvez le voir, la clause *finally* est exécutée dans tous les cas. L'erreur **TypeError** provoquée par la division de deux chaînes n'est pas gérée par une clause *except* et donc relancée après l'exécution de la clause *finally*. Dans les applications réelles, la clause *finally* est utile pour libérer des ressources extérieures (comme des fichiers ou des connexions réseau), que l'utilisation des ces ressources ait pu se faire ou non.

8.7 Actions de nettoyage prédéfinies

Certains objets définissent des actions de nettoyage standard, à mettre en œuvre lorsque l'objet n'est plus utilisé, que les opérations sur cet objet aient réussi ou échoué. Voyez l'exemple suivant, qui essaie d'ouvrir un fichier et d'afficher son contenu.

```
for ligne in open("mon_fichier.txt"):
    print(ligne, end="")
```

Le problème avec ce code est qu'il laisse le fichier ouvert pour une durée indéterminée après son exécution. Ce n'est pas très grave dans un script simple, mais peut s'avérer catastrophique dans des applications plus importantes. L'instruction **with** autorise l'utilisation d'objets tels que les fichiers de façon à assurer qu'ils sont toujours nettoyés dès que possible et correctement.

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

Après l'exécution de l'instruction, le fichier f sera fermé, même si un problème est advenu pendant la lecture des lignes. Les objets qui, comme les fichiers, fournissent des actions de nettoyage l'indiquent dans leur documentation.

Chapitre 9

Classes

Comparé avec d'autres langages de programmation, le mécanisme de classes de Python intègres les classes avec un minimum d'ajouts syntaxique et sémantique. C'est un hybride des mécanismes de classes de C++ et de Modula-3. Les classes de Python fournissent toutes les caractéristiques standard de la Programmation Orientée Objet : le mécanisme d'héritage de classes autorise l'héritage multiple, une classe derivée peut redéfinir toute méthode de sa (ou ses) classe(s) de base, et une méthode peut appeler une méthode de même nom dans sa classe de base. Les objets peuvent contenir toutes quantités et tous types de données. Comme pour les modules, les classes tirent parti de la nature dynamique de Python : elles sont créées à l'exécution et peuvent être modifiées après leur création.

Selon la terminologie C++, les membres normaux des classes Python (y compris les données membres) sont *publics* (excepté voir plus loin 9.6 *Variables privées*), et toutes les fonctions membres sont *virtuelles*. Comme en Modula-3, il n'y pas de notation abrégée pour référencer les membres d'un objet dans ses méthodes : la fonction méthode est déclarée avec une premier argument explicite représentant l'objet, qui est fourni implicitement par l'appel. Comme dans Smalltalk, les classes sont elles-mêmes des objets. Cela fournit un modèle sémantique à l'importation et au renommage. Contrairement à C++ et à Modula-3, les types prédéfinis peuvent être utilisés commes classe de base pour des extensions utilisateurs. De plus, comme en C++, la plupart des opérateurs prédéfinis à syntaxe spéciale (opérateurs arithmétiques, d'indexation, etc.) peuvent être redéfinis pour agir sur des instances de classes.

À défaut de terminologie universellement acceptée pour parler de classes, on utilisera occasionnellement le vocabulaire Smalltalk et C++. Le vocabulaire Modula-3 aurait été plus adapté, car sa sémantique de l'orienté objet est plus proche de celle de Python, mais il est probable que que peu de lecteurs en ont entendu parler.

9.1 Un mot sur les noms et les objets

Les objets possèdent une seule nature, et de multiples noms (dans de multiples portées) peuvent être liés au même objet. C'est ce que l'on appelle le cumul de noms (*aliasing*) dans d'autres langages. Cela n'est généralement pas très apprécié au premier abord, et cela peut être complètement ignoré si on ne manipule que des objets figés (*immutable*) comme les nombres, les chaînes de caractères ou les tuples. Cependant, l'*aliasing* peut produire des effets surprenants sur la sémantique d'un code Python manipulant des objets modifiables (*mutable*) tels que des listes, des dictionnaires et la plupart des autres types. Cela bénéficie en général au programme, car les alias se comportent sous certains aspects comme des pointeurs. Par exemple, passer un objet est peu coûteux, car seul un pointeur est transmis de par l'implémentation ; et si une fonction modifie un objet passé en argument, l'appelant verra le changement – ce qui élimine le besoin de deux mécanismes différents de passage d'arguments, comme en Pascal.

9.2 Portées et espaces de noms dans Python

Avant de présenter les classes, abordons les règles de portées de Python. Les définitions de classes peuvent poser quelques problèmes délicats avec les espaces de noms, et il est nécessaire de connaître le fonctionnement des portées et des espaces de noms pour comprendre ce qui va suivre. Soit dit en passant, connaître ce sujet est utile pour tout programmeur Python.

Commençons par quelques définitions.

Un espace de noms (*namespace*) est une association de noms et d'objets. La plupart des espaces de noms sont configurés en Python sous forme de dictionnaires, mais cela n'est visible en aucune façon (excepté pour les performances) et cela peut changer dans le futur. Des exemples d'espaces de noms sont : l'ensemble des noms prédéfinis (contenant les noms des fonctions – comme `abs()`, des types et des exceptions prédéfinis) ; les noms globaux dans un module et les noms locaux dans une invocation de fonction. Dans un sens, l'ensemble

des attributs d'un objet forme lui aussi un espace de noms. La chose essentielle à savoir à propos des espaces de noms, c'est qu'il n'y a absolument aucune relation entre les noms de différents espaces de noms ; par exemple, deux modules différents peuvent tous deux définir une fonction `maximum` sans confusion possible – les utilisateurs de ces modules devant préfixer le nom de la fonction par le nom du module.

Dans la suite, le mot *attribut* sera utilisé pour tout nom suivant un point — par exemple, dans l'expression `z.real`, `real` est un attribut de l'objet `z`. À proprement parler, les références à des noms dans des modules sont des références d'attributs : dans l'expression `nom_module.nom_fonction`, `nom_module` est un objet module et `nom_fonction` en est un attribut. Dans ce cas, il se trouve qu'il y a une association directe entre les attributs du module et les noms globaux définis dans le module : ils partagent le même espace [1] !

Les attributs peuvent être en lecture seule ou en lecture/écriture (*writable*). Dans ce dernier cas, les affectations aux attributs sont autorisés. C'est le cas des attributs de modules : on peut écrire `nom_module.la_réponse = 42`. Les attributs inscriptibles peuvent également être supprimés avec l'instruction `del`. Par exemple, `del nom_module.la_réponse` supprimera l'attribut `la_réponse` de l'objet module de nom `nom_module`.

Les espaces de noms sont créés à des moments divers et ont des durées de vie différentes. L'espace de noms contenant les noms prédéfinis est créé au lancement de l'interpréteur Python, et n'est jamais supprimé. L'espace de noms global d'un module est créé lors de la lecture de la définition du module ; normalement, les espaces de noms des modules sont présents jusqu'à ce que l'on quitte l'interpréteur. Les instructions exécutées au premier niveau de l'interpréteur, qu'elles soient lues dans un fichier script ou interactivement, sont considérées comme faisant partie d'un module de nom `__main__`, aussi ont-elles leur propre espace de noms global (en fait, les noms prédéfinis vivent également dans un module, le module `builtins`).

L'espace de noms local à une fonction est créé lorsque la fonction est appelée et il est supprimé lorsque le fonction se termine, que ce soit par un `return` ou une exception non gérée par la fonction ("oublié" serait un meilleur terme que "supprimé" pour décrire ce qui se passe vraiment). Bien sûr, des appels récursifs possèdent chacun leur propre espace de noms local.

Une portée (*scope*) est une région de texte d'un programme Python dans laquelle un espace de noms est directement accessible. "Directement accessible" signifie qu'une référence non qualifiée à un nom tente de le retouver dans l'espace de noms.

Bien que les portées soient déterminées statiquement, elles sont utilisées dynamiquement. À tout moment durant l'exécution, il y au moins trois portées imbriquées pour lesquelles des espaces de noms sont directement accessibles :

- la portée la plus interne, dans laquelle l'exploration pour une recherche de nom commence, contient les noms locaux ;
- les portées des fonctions englobantes, qui sont explorées à partir de la plus interne, contiennent des noms non-locaux, mais aussi non-globaux ;
- l'avant dernière portée contient les noms globaux du module courant ;
- la portée la plus externe (la dernière à explorer) est l'espace de noms contenant les noms prédéfinis.

Si un nom est déclaré global, toutes les références et affectations commencent à la portée des noms globaux du module. Pour lier des variables en dehors de la portée la plus interne, l'instruction `nonlocal` peut être utilisée. Si une variable n'est pas déclarée `nonlocal`, elle est en lecture seule (toute tentative d'y écrire créera simplement une *nouvelle* variable locale dans la portée la plus interne, laissant inchangée la variable externe de même nom).

Le plus souvent, la portée locale référence les noms locaux présents dans le code de la fonction en cours. En dehors des fonctions, la portée locale référence le même espace de noms que la portée globale : celui du module. Les définitions de classes mettent en place un autre espace de noms dans la portée locale.

Il est important de réaliser que les portées sont déterminées par le texte : la portée globale d'une fonction définie dans un module est l'espace de noms de ce module, peu importe d'où et avec quel alias cette fonction a été appelée. Par contre, la recherche effective de noms se fait dynamiquement à l'exécution – toutefois, l'évolution du langage tend vers une résolution de noms statique, à la "compilation", aussi ne basez pas vos programmes sur la résolution dynamique (en fait, les variables locales sont déjà déterminées statiquement).

Une spécificité de Python est que, si aucune instruction `global` n'est présente, les affectations à des noms se font toujours dans la portée la plus interne. Les affectations ne copient pas les données - elles ne font que lier les noms aux objets. C'est la même chose pour les suppressions : l'instruction `del x` supprime la liaison de `x` dans l'espace de noms référencé par la portée locale. En fait, toutes les opérations introduisant de nouveaux noms utilise la portée locale : en particulier, l'instruction `import` et les définitions de fonction lient les noms de module ou de fonction dans la portée locale.

1. Excepté pour une chose. Les objets modules ont un attribut secret en lecture seule appelé `__dict__` qui retourne le dictionnaire utilisé pour implémenter l'espace de noms du module ; le nom `__dict__` est un attribut du module, mais pas un nom global. Évidemment, utiliser cette particularité viole l'abstraction de l'implémentation des espaces de noms, et ne devrait être réservé qu'aux débogages post-mortem.

L'instruction **global** peut s'utiliser pour indiquer qu'une variable particulière vit dans la portée globale et doit y être reliée ; l'instruction **nonlocal** indique qu'une variable particulière vit dans une portée englobante et doit y être reliée.

9.2.1 Exemple de portée et d'espace de noms

Voici un exemple démontrant comment référencer les diverses portées et espaces de noms, et comment les instructions **global** et **nonlocal** affectent les liaisons de variables :

```
def test_de_portée():
    def affectation_en_local():
        spam = "spam local"
    def affectation_en_nonlocal():
        nonlocal spam
        spam = "spam nonlocal"
    def affectation_en_global():
        global spam
        spam = "spam global"
    spam = "spam du test"
    affectation_en_local()
    print("Dans test_de_portée, après appel à affectation_en_local :", spam)
    affectation_en_nonlocal()
    print("Dans test_de_portée, après appel à affectation_en_nonlocal :", spam)
    affectation_en_global()
    print("Dans test_de_portée, après appel à affectation_en_global :", spam)

test_de_portée()
print("Dans la portée globale :", spam)
```

La sortie de cet exemple est :

```
Dans test_de_portée, après appel à affectation_en_local : spam du test
Dans test_de_portée, après appel à affectation_en_nonlocal : spam nonlocal
Dans test_de_portée, après appel à affectation_en_global : spam nonlocal
Dans la portée globale : spam global
```

Notez comment l'affectation en *local* (qui est le comportement par défaut) ne modifie pas la liaison de *spam* dans la portée du test. L'affectation en **nonlocal** modifie la liaison de *spam* dans la portée du test et l'affectation en **global** modifie la liaison de *spam* dans la portée (globale) du module.

Vous pouvez voir aussi qu'il n'y a pas de liaison préalable pour *spam* avant l'affectation en **global**.

9.3 Un premier coup d'œil aux classes

Les classes amènent un petit peu de syntaxe supplémentaire, introduisent trois nouveaux types d'objets et ajoutent un peu de sémantique.

9.3.1 Syntaxe de définition de classe

La forme la plus simple d'une définition de classe ressemble à cela :

```
class NomDeClasse:
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

Les définitions de classe, comme les définitions de fonction (instruction **def**) doivent être exécutées avant qu'elles ne puissent produire un effet (on pourrait en théorie placer une définition de classe dans une branche d'instruction **if** ou dans une fonction).

En pratique, les instructions dans une définition de classe sont presque toujours des définitions de fonction, mais d'autres instructions sont autorisées, et parfois bien utiles - nous y reviendrons plus loin. Les définitions de fonction dans une classe ont une forme particulière de liste d'arguments, dictée par la convention d'appel des méthodes - nous y reviendrons également.

Lorsqu'une définition de classe s'exécute, un nouvel espace de noms est créé, et utilisé en tant que portée locale - c'est à dire que toutes les affectations à des variables locales se feront dans ce nouvel espace de noms. En particulier, les définitions de fonction lieront le nom de la nouvelle fonction ici.

Lorsqu'une définition de classe se termine normalement (par la fin), un objet classe (*class object*) est créé. C'est basiquement une enveloppe autour du contenu de l'espace de noms créé par la définition de classe ; nous en apprendrons plus sur les objets classe dans la prochaine section. La portée locale originale (celle en cours juste avant le début de la définition de classe) est rétablie, et l'objet classe y est lié au nom de classe donné en début de définition (`NomDeClasse` dans l'exemple).

9.3.2 Objets classe

Les objets classe prennent en charge deux types d'opération : les références à des attributs et l'instanciation.

Les *références à des attributs* utilisent la syntaxe standard pour toute référence à un attribut en Python : `obj.nom`. Les noms d'attributs valides sont tous les noms qui de trouvaient dans l'espace de noms de la classe lorsque l'objet classe a été créé. Ainsi, si la définition était la suivante :

```
class MaClasse:
    """Un exemple simple de classe"""
    i = 12345
    def f(self):
        return 'Salut la compagnie !'
```

alors `MaClasse.i` et `MaClasse.f` sont des références d'attribut valides, retournant respectivement un objet entier et un objet fonction. Les attributs de classe peuvent également être modifiés, ainsi vous pouvez changer la valeur de `MaClasse.i` par une affectation. Un autre attribut valide est `__doc__`, qui retourne la *docstring* de la classe : `"Un exemple simple de classe"`.

L'instanciation de classe utilise la notation fonctionnelle. Faites simplement comme si l'objet classe était une fonction sans paramètre : l'appel retournera une nouvelle instance de la classe. Par exemple (avec la définition de classe précédente) :

```
x = MaClasse()
```

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

L'opération d'instanciation ("appeler" un objet classe) crée un objet vide. Beaucoup de classes préfèrent créer des instances initialisées de manière spécifique. Pour cela, une classe peut définir une méthode spéciale appelée `__init__()`, comme ici :

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

Lorsqu'une classe définit une méthode `__init__()`, l'instanciation de classe invoque automatiquement `__init__()` pour toute nouvelle instance de classe. Ainsi, dans cet exemple, une nouvelle instance initialisée peut être obtenue par :

```
x = MaClasse()
```

Bien sûr, la méthode `__init__()` peut avoir des arguments pour être plus flexible. Dans ce cas, les argument donnés à l'opérateur d'instanciation sont transmis à `__init__()`. Par exemple,

```
>>> class Complexe:
...     def __init__(self, partie_réelle, partie_imaginaire):
...         self.r = partie_réelle
...         self.i = partie_imaginaire
...
>>> x = Complexe(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objets instance

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

Les *attributs donnée* correspondent aux "variables d'instance" de Smalltalk, et aux "données membres" de C++. Les attributs donnée n'ont pas besoin d'être déclarés ; tout comme les variables locales, ils commencent à exister la première fois qu'ils sont affectés. Par exemple, si `x` est une instance de `MaClasse` créée ci-dessus, le bout de code suivant affichera la valeur `16`, sans laisser de trace :

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

L'autre sorte de référence d'attribut d'instance est la *méthode*. Une méthode est une fonction appartenant à un objet. En Python, le terme méthode n'est pas réservé aux instances de classe : d'autre types d'objets peuvent avoir des méthodes. Par exemple, les objets `list` ont des méthodes comme `append`, `insert`, `remove`, `sort`, etc. Toutefois, dans la suite, nous utiliserons le terme méthode uniquement pour signifier méthode d'objets instance, sauf précision contraire.

Les noms de méthodes valides sur un objet instance dépendent de la classe. Par définition, tous les attributs d'une classe qui sont des objets fonction définissent des méthodes correspondantes sur les instances de cette classe. Ainsi, dans notre exemple, `x.f` est une référence de méthode valide, car `MaClasse.f` est une fonction, mais `x.i` ne l'est pas, car `MaClasse.i` n'est pas une fonction. Mais `x.f` n'est pas le même objet que `MaClasse.f` — c'est un objet méthode, pas un objet fonction.

9.3.4 Objets méthode

Habituellement, une méthode est appelée juste après qu'elle soit liée :

```
x.f()
```

Dans l'exemple `MaClasse`, cela retournera la chaîne `'Salut la compagnie !'`. Toutefois, il n'est pas nécessaire d'appeler une méthode immédiatement : `x.f` est un objet méthode et il peut être stocké, puis appelé plus tard. Par exemple :

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

affichera `Salut la compagnie !` jusqu'à la fin des temps...

Que se passe-t-il exactement quand une méthode est appelée ? Vous avez peut-être remarqué que `x.f()` était appelé sans argument, bien que la définition de la fonction `f()` spécifiait un argument. Qu'est il advenu à cet argument ? Python déclenche sûrement une exception quand une fonction déclarée avec un argument est appelée sans argument - même si l'argument n'était pas utilisé dans le corps de la fonction...

En fait, vous avez peut-être deviné la réponse : le dispositif spécial, avec les méthodes, est que l'objet est passé en premier argument de la fonction. Dans notre exemple, l'appel `x.f()` est exactement équivalent à un appel `MaClasse.f(x)`. En général, appeler une méthode sur un objet instance avec une liste d'arguments revient à appeler la fonction correspondante avec une nouvelle liste d'arguments créée en insérant l'objet de la méthode en tête de liste.

Si vous ne comprenez toujours pas comment les méthodes fonctionnent, regardez l'implémentation peut clarifier la situation. Lorsqu'un attribut d'instance qui n'est pas une donnée est référencé, sa classe est recherchée. Si le nom de l'attribut est associé à un attribut de classe qui est un objet fonction, un objet méthode est créé qui enveloppe l'objet instance et l'objet fonction dans un objet abstrait : l'objet méthode. Lorsque l'objet méthode est appelé avec une liste d'arguments, une nouvelle liste d'arguments est construite avec l'objet instance et la liste d'arguments. L'objet fonction est appelé avec cette nouvelle liste d'arguments.

9.3.5 Variables de classe et variables d'instance

En général, les variables d'instance servent à des données uniques à chaque instance et les variables de classe servent pour des attributs et méthodes partagées par toutes les instances de la classe :

```
>>> class Chien:
...     famille = 'canidé'    # variable de classe, partagée par toutes les instances
...     def __init__(self, nom):
...         self.nom = nom    # variable d'instance unique pour chaque instance
...
>>> d = Chien('Milou')
>>> e = Chien('Idéfix')
>>> d.famille                 # partagée par tous les chiens
'canidé'
>>> e.famille                 # partagée par tous les chiens
'canidé'
>>> d.nom                     # unique pour d
'Milou'
```

```
>>> e.nom                    # unique pour e
'Idéfix'
```

Comme déjà dit dans la section 9.1 *Un mot sur les noms et les objets*, les données partagées peuvent apporter quelques surprises si ce sont des objets modifiables (mutable) comme des listes ou des dictionnaires. Par exemple, la liste *tours* dans le code suivant ne devrait pas être utilisée en tant que variable de classe, car c'est une liste unique partagée par toutes les instances de *Chien* :

```
>>> class Chien:
...     tours = []             # mauvais usage d'une variable de classe
...     def __init__(self, nom):
...         self.nom = nom
...     def ajoute_tour(self, tour):
...         self.tours.append(tour)
...
>>> d = Chien('Milou')
>>> e = Chien('Idéfix')
>>> d.ajoute_tour('roule')
>>> e.ajoute_tour('fais le mort')
>>> d.tours                # partagé par tous les chiens : était-ce voulu ?
['roule', 'fais le mort']
```

Une conception correcte de la classe aurait dû utiliser une variable d'instance :

```
>>> class Chien:
...     def __init__(self, nom):
...         self.nom = nom
...         self.tours = []    # une liste vide pour chaque chien
...     def ajoute_tour(self, tour):
...         self.tours.append(tour)
...
>>> d = Chien('Milou')
>>> e = Chien('Idéfix')
>>> d.ajoute_tour('roule')
>>> e.ajoute_tour('fais le mort')
>>> d.tours
['roule']
>>> e.tours
['fais le mort']
```

9.4 Quelques remarques

Les attributs donnée redéfinissent les attributs méthode de même nom ; pour éviter les conflits accidentels de noms, qui peuvent causer des bugs difficiles à trouver dans de grands programmes, il est judicieux d'utiliser une technique de nommage permettant de minimiser les risques de conflits. Des conventions possibles peuvent être : mettre en majuscule les noms de méthodes, préfixer les attributs de données par un caractère spécial (par exemple un tiret de soulignement), ou utiliser des verbes pour les méthodes et des noms pour les données.

Les attributs donnée peuvent être référencés par des méthodes, mais aussi par des utilisateurs ("clients") d'un objet. En d'autres termes, les classes Python ne peuvent pas être utilisées pour construire des types de données abstraits. En fait, rien en Python ne permet de cacher des données - tout est basé sur des conventions (par contre, l'implémentation de Python, écrite en C, peut cacher complètement les détails d'implémentation et contrôler l'accès à un objet si nécessaire ; cela peut également être utilisé pour les extensions écrites en C).

Les clients doivent utiliser les attributs donnée avec précaution - ils risque d'outrepasser des invariants maintenus par les méthodes qui contrôlent leurs attributs donnée. Notez que les clients peuvent ajouter leurs propres attributs donnée à un objet instance sans affecter la validité des méthodes, tant qu'ils évitent les conflits de noms - là encore, une convention de nommage peut éviter de nombreux maux de tête.

Il n'y a pas de raccourci pour référencer les attributs donnée (ou d'autres méthodes !) à l'intérieur des méthodes (cela améliore la lisibilité des méthodes : il n'y aucun risque de confondre variables locales et variables d'instance en parcourant le code d'une méthode).

Souvent, le premier argument d'une méthode est appelé **`self`**. Ce n'est qu'une convention : le nom **`self`** ne joue aucun rôle particulier en Python. Notez toutefois que si vous ne suivez pas cette convention, votre code sera plus difficilement lisible par d'autres programmeurs Python et qu'il est également concevable que des visualisateurs de classe utilisent cette convention.

Chaque objet fonction qui est un attribut de classe définit une méthode pour les instances de cette classe. Il n'est pas nécessaire que la définition de la fonction soit incluse textuellement dans la définition de classe : affecter un objet fonction à une variable locale de la classe marche aussi. Par exemple :

```
# Fonction définie en dehors de la classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'Salut la compagnie !'
    h = g
```

Maintenant **f**, **g** et **h** sont tous des attributs de la classe **C** qui réfèrent des objets fonction, et qui sont par conséquent des méthodes pour les instances de **C** — **h** étant exactement équivalent à **g**. Notez que cette pratique ne sert en général qu'à embrouiller le lecteur d'un programme.

Les méthodes peuvent appeler d'autres méthodes en utilisant les attributs méthode de l'argument **self** :

```
class Sac:
    def __init__(self):
        self.contenu = []
    def ajoute(self, x):
        self.contenu.append(x)
    def ajoutes_en_deux(self, x):
        self.ajoute(x)
        self.ajoute(x)
```

Les méthodes peuvent référencer des noms globaux de la même façon que des fonctions ordinaires. La portée globale associée à une méthode est le module contenant sa définition (une classe n'est jamais utilisée en tant que module global). Bien qu'il y ait rarement de bonnes raisons pour utiliser des données globales dans une méthode, il existe beaucoup de cas d'usage d'une portée globale : d'abord, les fonctions et modules importés dans la portée globale peuvent être utilisées par les méthodes, comme les fonctions et les classes qui y sont définies. Habituellement, la classe contenant la méthode est elle-même définie dans cette portée globale, et nous verrons dans la prochaine section de bonnes raisons pour qu'une méthode fasse référence à sa propre classe.

Toute valeur est un objet, et donc possède une classe (également appelée son *type*) Il est stocké dans l'attribut ***object*.__class__**.

9.5 Héritage

Évidemment, une spécificité d'un langage ne pourrait pas porter le nom de "classe" si ce langage ne permettait pas l'héritage. La syntaxe d'une définition de classe dérivée ressemble à cela :

```
class NomDeClasseDérivée(NomDeClasseDeBase):
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

Le nom **NomDeClasseDeBase** doit être défini dans une portée contenant la définition de la classe dérivée. Au lieu du nom d'une classe de base, d'autres expressions arbitraires sont autorisées. Cela peut être utile lorsque, par exemple, la classe de base est définie dans un autre module.

```
class NomDeClasseDérivéee(nom_de_module.NomDeClasseDeBase):
```

L'exécution d'une définition de classe dérivée s'effectue de la même façon que pour une classe de base. Lorsque l'objet classe est construit, la classe de base est mémorisée, ce qui permettra de résoudre les références d'attributs : si un attribut requis n'est pas trouvé dans la classe, la recherche se poursuit dans la classe de base. cette règle s'applique récursivement si la classe de basse elle-même est dérivée d'une autre classe.

Il n'y a rien de spécial concernant l'instanciation de classes dérivées : **NomDeClasseDérivéee()** crée une nouvelle instance de la classe. Les références de méthodes sont résolues comme suit : l'attribut de la classe correspondante est cherché, en remontant dans la chaîne des classes de base si nécessaire, et la référence de méthode est valide si la recherche rapporte un objet fonction.

Les classes dérivées peuvent redéfinir des méthodes de leurs classes de base. Comme les méthodes n'ont pas de privilèges spéciaux lorsqu'elles appellent d'autres méthodes du même objet, une méthode d'une classe de base qui appelle une autre méthode définie dans la même classe peut finir par appeler une méthode d'une classe dérivée qui l'aurait redéfinie (selon la terminologie C++, toutes les méthodes de Python sont *virtuelles*).

Une méthode redéfinie dans une classe dérivée peut en fait vouloir étendre plutôt que remplacer la

méthode de la classe de base de mêm nom. Il y a un moyen simple de forcer l'appel à la méthode de la classe de base : invoquez simplement `NomDeClasseDeBase.nom_de_méthode(self, arguments)`. Cela peut également servir occasionnellement à des clients (notez que cela ne fonctionne que si le nom de la classe de base est accessible dans la portée globale).

Python possède deux fonctions prédéfinies servant dans le domaine de l'héritage :

- `isinstance()` pour tester le type d'une instance : `isinstance(obj, int)` sera vrai (*True*) seulement si `obj.__class__` vaut `int` ou une classe dérivée de `int`.
- `issubclass` pour tester l'héritage de classes : `issubclass(bool, int)` est vrai car `bool` est une sous-classe de `int`. Toutefois, `issubclass(float, int)` est faux car `float` n'est pas une sous-classe de `int`.

9.5.1 Héritage multiple

Python permet également une forme d'héritage multiple. Une définition de classe dérivée de plusieurs classes ressemble à cela :

```
class NomDeClasseDérivée(Base1, Base2, Base3):
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

La plupart du temps, pour les cas les plus simples, vous pouvez vous représenter la recherche d'attributs dans une hiérarchie de classes comme une recherche en profondeur, puis en largeur, sans chercher deux fois dans la même classe s'il existe un cycle dans la hiérarchie. Ainsi, si un attribut n'est pas trouvé dans `NomDeClasseDérivée`, il est cherché dans `Base1`, puis (récursivement) dans ses classes de bases, et, s'il n'est toujours pas trouvé, dans `Base2`, etc.

En fait, c'est un petit peu plus compliqué que ça ; l'ordonnancement de résolution des méthodes (*method resolution order*, *MRO*) change dynamiquement pour supporter les appels coopératifs à `super()`. Cette approche est connue dans d'autres langages à héritage multiple sous le nome de *call-next-method* et est plus puissante que l'appel à `super` que l'on trouve dans les langages à héritage simple.

L'ordonnancement dynamique est nécessaire car tous les cas d'héritage multiple présentent une ou plusieurs relations en losange (lorsqu'au moins une des classes parentes peut être atteinte par plusieurs chemins à partir de la classe dérivée). Par exemple, toutes les classes héritent de la classe `object`, donc tout héritage multiple conduira à plusieurs chemins pour atteindre `object`. Pour empêcher les classes de bases d'être testées plusieurs fois, l'algorithme dynamique linéarise l'ordonnancement de recherche d'une façon qui préserve l'ordre de gauche à droite spécifié dans chaque classe, qui n'appelle chaque parent qu'une seule fois et qui est monotone (c'est à dire qu'une classe peut être sous-classée sans modifier l'ordre de précédence de ses parents). Combinées ensemble, ces propriétés rendent possible la conception de classes fiables et extensibles supportant l'héritage multiple. Pour plus de détails, voir `https://www.python.org/download/releases/2.3/mro/`.

9.6 Variables privées

Les variables d'instances privées (qui ne peuvent être accédées qu'à partir de l'intérieur d'un objet), n'existent pas en Python. Toutefois, une convention est suivie dans la plupart des programmes Python : un nom préfixé par un tiret de soulignement (par exemple `_spam`) devrait être considéré comme une partie non publique de l'interface de programmation (*API*), que ce soit une méthode, une fonction ou une donnée. Il devrait être considéré comme un détail d'implémentation pouvant être modifié sans sommation.

Comme il existe des cas pour lesquels des membres de classe privés s'avèrent utiles, (ne serait ce que pour éviter des conflits de noms entre noms définis dans des sous-classes), Python fournit un support limité pour un tel mécanisme, appelé distorsion de nom (*name mangling*). Tout identificateur de la forme `__spam` (au moins deux tirets de soulignement au début, au plus un à la fin) est remplacé textuellement par `_NomDeClasse__spam`, où `NomDeClasse` est le nom de la classe courante sans d'éventuels tirets de soulignements initiaux. Cette transformation est faite sans s'occuper de la position syntaxique de l'identificateur, à condition qu'il se trouve dans la définition d'une classe.

La distorsion de noms est utile pour permettre aux sous-classes de redéfinir des méthodes, sans poser de problèmes pour les appels de méthodes intraclasse. Par exemple :

```
>>> class ListeSimple: #liste de données simples
...     def __init__(self, iterable): # initialisation par une suite de données
...         self.items_list = []
...         for x in iterable:
```

```
...             self.__ajoute(x) # que même pour les sous-classes utilisant cet __init__()
...                              # c'est __ajoute() qui sera appelé. Essayez de supprimer
...                              # les __ et comprenez le message d'erreur
...     def ajoute(self, x): # ajoute la donnée x à la  liste
...         self.items_list.append(x)
...     __ajoute = ajoute   # copie privée de la méthode ajoute()
...
>>> class ListeDeCouples(ListeSimple): # liste de données couples
...     # une nouvele signature pour ajoute()
...     # __init__() n'a pas besoin d'être redéfinie
...     def ajoute(self, x, y): # ajoute le couple (x,y) à la  liste
...         self.items_list.append((x,y))
...
>>> simple = ListeSimple([1,2,3])
>>> double = ListeDeCouples([("01","Ain"),("02", "Aisne"),("03","Allier")])
>>> simple.ajoute(4) # un paramètre pour les listes simples
>>> double.ajoute("04", "Alpes-de-Haute-Provence") # deux paramètres pour les listes couples
>>> for l in (simple, double,  ):
...     for x in l.items_list:
...         print(x)
...
1
2
3
4
('01', 'Ain')
('02', 'Aisne')
('03', 'Allier')
('04', 'Alpes-de-Haute-Provence')
```

Notez que les règles de distorsion sont conçues principalement pour éviter les accidents ; il est toujours possible d'accéder ou de modifier une variable considérée comme privée. Cela peut être même utile dans certaines circonstances, comme dans un débogage.

Notez que le code passé à **`exec()`** ou **`eval()`** ne considèrent pas le nom de la classe appelante comme la classe courante ; c'est analogue à l'effet d'une instruction **`global`**, qui est limité au code compilé en un seul ensemble. La même restriction s'applique à **`getattr()`**, **`setattr()`** et **`delattr()`**, comme au référencement direct de **`__dict__`**.

9.7 Un peu de tout

Il peut être utile de temps à autre de disposer d'un type de données similaire au "record" du langage Pascal ou au "struct" du langage C, rassemblant quelques données nommées. Une définition de classe *vide* le fait convenablement :

```
class Employé:
    pass

x = Employé() # Crée un enregistrement Employé vide

# Renseigne les champs de l'enregistrement
x.nom = 'Eric Idle'
x.groupe = 'monty python'
x.salaire = 1000
```

Un morceau de code Python qui attend un type particulier de données peut souvent recevoir à la place une classe qui émule les méthodes de ce type. Par exemple, si vous avez une fonction qui formate des données en provenance d'un objet fichier, vous pouvez définir une classe avec les méthodes **`read()`** et **`readline()`** qui prend les données dans une chaîne de caractères et les passe en argument.

Les objets méthode d'instance ont eux aussi des attributs : **`m.__self__`** est l'objet instance associé à la méthode **`m()`**, et **`m.__func__`** est l'objet fonction correspondant à cette méthode.

9.8 Les exceptions sont aussi des classes

Les exceptions définies par l'utilisateur sont également des classes. En utilisant ce mécanisme, il est possible de créer des hiérarchies d'exceptions extensibles.

Il y a deux formes valides sémantiquement pour l'instruction **`raise`** :

```
raise Classe
```

```
raise instance
```

Dans la deuxième forme, **instance** doit être une instance de **BaseException** ou d'une de ses classes dérivées. La première forme est un raccourci pour :

```
raise Classe()
```

Une classe dans une clause *except* est compatible avec une exception si c'est la même classe ou une classe de base (mais pas dans l'autre sens, une clause *except* contenant une classe dérivée n'est pas compatible avec une classe de base). Par exemple, le code suivant affichera B, C, D dans cet ordre :

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

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

Notez que si les clauses *except* étaient inversées (avec **except B** en premier), ce code aurait affiché B, B, B car la première clause *except* qui convient est déclenchée.

Lorsqu'un message d'erreur est affiché à cause d'une exception non gérée, le nom de la classe exception s'affiche, suivi d'un deux-points, d'une espace et finalement de l'instance convertie en chaîne par la fonction prédéfinie **str()**.

9.9 Itérateurs

Vous avez certainement remarqué que la plupart des objets conteneurs peuvent être parcourus en utilisant une instruction **for** :

```
>>> for element in [1, 2, 3]:
...     print(element, end=' ')
...
1 2 3
>>> for element in (1, 2, 3):
...     print(element, end = ' ')
...
1 2 3
>>> for key in {'un':1, 'deux':2}:
...     print(key, end = ' ')
...
deux un
>>> for char in "123":
...     print(char, end = ' ')
...
1 2 3
>>> with open("mon_fichier.txt",encoding="UTF8") as f:
...     for line in f:
...         print(line, end='')
...
voici la première ligne
et maintenant la deuxième ligne
et enfin, la dernière ligne
```

Ce style de parcours est clair, concis et commode. L'utilisation d'itérateurs se répand dans Python et unifie les parcours d'objets divers. En coulisses, l'instruction **for** appelle **iter** sur l'objet conteneur. Cette fonction retourne un objet itérateur qui définit la méthode **__next__()** qui accède un par un aux éléments du conteneur. Lorsqu'il n'y a plus d'éléments, **__next__()** déclenche une exception **StopIteration** qui indique à la boucle **for** de se terminer. Vous pouvez appeler la méthode **__next__()** à l'aide de la fonction prédéfinie **next** ; l'exemple suivant montre comment tout cela fonctionne :

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x0000000002EE0240>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En ayant vu ce qui se passe dans le protocole *itérateur*, il devient facile d'ajouter un comportement d'itérateur à vos classes. Définissez une méthode `__iter__()` qui retourne un objet possédant une méthode `__next__()`. Si la classe définit déjà une méthode `__next__()`, il suffit à `__iter__()` de retourner `self` :

```
>>> class Rebours:
...     """Itérateur pour parcourir une séquence à rebours."""
...     def __init__(self, données):
...         self.données = données
...         self.index = len(données)
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.index == 0:
...             raise StopIteration
...         self.index = self.index - 1
...         return self.données[self.index]
...
>>> rev = Rebours('spam')
>>> iter(rev)
<__console__.Rebours object at 0x0000000002EE0CF8>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.10 Générateurs

Les *générateurs* sont des outils simples et puissants pour créer des itérateurs. Ils sont écrits comme des fonctions normales, mais utilisent l'instruction `yield` pour retourner des données. Chaque fois que `next()` est invoquée sur lui, le générateur reprend son exécution à l'endroit où il l'avait laissée (il mémorise toutes les données et la dernière instruction exécutée). L'exemple suivant montre la facilité de création d'un générateur :

```
>>> def rebours(données):
...     for index in range(len(données)-1, -1, -1):
...         yield données[index]
...
>>> for char in rebours('golf'):
...     print(char)
...
f
l
o
g
```

Tout ce qui peut être fait avec des générateurs peut aussi l'être avec des itérateurs de classes décrits dans la section précédente. Ce qui rend les générateurs si compacts est que les méthodes `__iter__()` et `__next__()` sont créées automatiquement.

Une autre caractéristique clé est que les variables locales et l'état d'exécution sont automatiquement mémorisés entre deux appels. Cela rend la fonction plus simple à écrire et à comprendre qu'une approche utilisant des variables d'instances comme `self.index` and `self.données`.

En plus de la création automatique de méthodes et de la mémorisation de l'état du programme, lorsqu'un générateur se termine, il lève automatiquement une exception `StopIteration`. Combinées entre elles, ces caractéristiques permettent de créer des itérateurs aussi facilement qu'une fonction normale.

9.11 Expressions génératrices

De simples générateurs peuvent être codés encore plus succinctement sous forme d'expressions utilisant une syntaxe similaire à celle des listes en compréhension, mais entre parenthèses au lieu de crochets. Ces expressions sont proposées pour les situations où le générateur serait utilisé directement par une fonction englobante. Les expressions génératrices sont plus compactes mais moins polyvalentes que les définitions de vrais générateurs. Elles sont plus économes en mémoire que les listes en compréhension équivalentes.

Exemples :

```
>>> sum(i*i for i in range(10))                 # somme des carrés
285

>>> vecteur1 = [10, 20, 30]
>>> vecteur2 = [7, 5, 3]
>>> sum(x*y for x,y in zip(vecteur1, vecteur2)) # produit scalaire
260

>>> from math import pi, sin
>>> table_sinus = {x: sin(x*pi/180) for x in range(0, 91)}
>>> print(table_sinus[45])
0.7071067811865476

>>> with open("mon_fichier.txt",encoding="UTF8") as f:
...     mots_uniques = set(mot  for ligne in f  for mot in ligne.split())
...
>>> print(mots_uniques)
{'deuxième', 'maintenant', 'la', 'ligne', 'première', 'et', 'dernière', 'enfin,', 'voici'}

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Chapitre 10

Visite de la bibliothèque standard

10.1 Interface système

Le module **os** propose des douzaines de fonctions pour interagir avec le système d'exploitation :

```
>>> import os
>>> os.getcwd()      # retourne le répertoire courant
'C:\\Users\\JCG\\cours\\cours'
>>> os.chdir('pythontex-files-essai')   # change le répertoire courant
>>> os.system('mkdir tmp')   # exécute la commande système mkdir
1
```

Assurez vous d'entrer la commande **import os** plutôt que **from os import ***. Cela empêchera la fonction **os.open()** de cacher la fonction prédéfinie **open()** qui agit de manière très différente.

Les fonctions prédéfinies **dir()** et **help()** sont d'un grand secours pour mieux savoir comment travailler avec des modules importants comme **os** :

```
>>> import os
>>> dir(os)
<retourne une liste de toutes les fonctions du module>
>>> help(os)
<retourne une copieuse page (1400 lignes !) d'aide créée à partir de la docstring du module>
```

Pour un usage quotidien et des tâches de gestion de répertoire, le module **shutil** fournit une interface de haut niveau plus facile à utiliser :

```
>>> import shutil
>>> shutil.copyfile('fichier.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Caractères joker pour fichiers

Le module **glob** fournit une fonction pour créer des listes de noms de fichiers pour des recherches incluant des caractères joker :

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Arguments de la ligne de commande

Les scripts utilitaires courants utilisent souvent les arguments de la ligne de commande. Ces arguments sont mémorisés dans l'attribut **argv** du module **sys** en tant que liste. Par exemple, avec le fichier

demo.py

```
import sys
print(sys.argv)
```

L'affichage suivant résulte de l'invocation de `python demo.py un deux trois` sur la ligne de commande :

```
['demo.py', 'un', 'deux', 'trois']
```

Le module `getopt` traite `sys.argv` selon les conventions de la fonction Unix `getopt`. Une commande plus puissante et plus flexible est proposée par le module `argparse`.

10.4 Redirection de la sortie d'erreurs et terminaison de programme

Le module `sys` possède également des attributs `stdin`, `stdout`, et `stderr`. Ce dernier est utile pour émettre des messages d'avertissement ou d'erreurs, et les laisser visibles, même si `stdout` a été redirigé :

```
>>> sys.stderr.write("Attention ! fichier journal non trouvé, création d'un nouveau\n")
Attention ! fichier journal non trouvé, création d'un nouveau
```

Le moyen le plus direct de mettre fin à un script est d'invoquer `sys.exit()`.

10.5 Correspondance de motifs de chaînes

Le module `re` propose des outils d'expressions régulières pour le traitement avancé de chaînes. Pour des correspondances de motif (*pattern matching*) ou des manipulations complexes, les expressions régulières offrent des solutions courtes et optimisées :

```
>>> import re
>>> re.findall(r'[a-z]*s\b', 'liste des mots qui se termine par s')
['des', 'mots', 's']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'supprime les les mots répétés plus d\'une fois fois')
"supprime les mots répétés plus d'une fois"
```

Quand ce sont des manipulations simples qui sont à faire, les méthodes de chaînes sont préférables car plus simples à lire et à déboguer :

```
>>> 'dermatologue à peau'.replace('peau', 'Pau')
'dermatologue à Pau'
```

10.6 Mathématiques

Le module `math` fournit l'accès aux fonctions sous-jacentes de la bibliothèque C de mathématiques en virgule flottante :

```
>>> import math
>>> math.cos(math.pi / 4)
0.7071067811865476
>>> math.log(1024, 2)
10.0
```

Le module `random` propose des outils pour faire des sélections pseudo-aléatoires :

```
>>> import random
>>> random.choice(['pomme', 'poire', 'banane']) #sélection d'un élément
'pomme'
>>> random.sample(range(100), 10)   # échantillon sans remplacement
[98, 35, 46, 67, 8, 41, 60, 70, 29, 31]
>>> random.random()    # flottant pseudo-aléatoire compris entre 0 et 1
0.9178630898981328
>>> random.randrange(6)    # entier pseudo-aléatoire choisi dans range(6)
0
```

Le module `statistics` propose des outils statistiques de base (moyenne, médiane, variance, etc.) sur des données numériques :

```
>>> 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 <http://scipy.org> propose beaucoup d'autres modules pour le calcul numérique.

10.7 Accès à Internet

Il existe un grand nombre de modules pour accéder à Internet ou manipuler des protocoles internet . Deux des plus simples sont : **urllib.request** pour récupérer des données à partir d'URL et **smtplib** pour envoyer du courrier élecronique :

```
>>> from urllib.request import urlopen
>>> # aller chercher l'heure en temps universel sur le site de la marine US
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8')  # décodage de la donnée en texte.
...         if 'UTC' in line :  # cherche la ligne temps universel
...             print(line)

<BR>Aug. 02, 08:55:21 UTC        Universal Time

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('haruspice@sanzot.fr', 'jcesar@rome.gov',
... """To: jcesar@rome.gov
... From: haruspice@sanzot.fr
...
... Prends garde aux Ides de Mars.
... """)
>>> server.quit()
```

Notez que le second exemple nécessite un serveur de mail s'exécutant sur l'hôte local *localhost*.

10.8 Date et heure

Le module **datetime** fournit des classes permettant de manipuler les dates et les heures, de manière simple ou complexe. Bien que l'arithmétique sur la date et l'heure soit supportée, l'accent est surtout mis sur des récupérations efficaces de données pour des formatages ou des manipulations. Le module fournit également des objets connaissant les fuseaux horaires.

```
>>> # les dates sont construites et formatées facilement
>>> from datetime import date
>>> aujourdhui = date.today()
>>> aujourdhui
datetime.date(2015, 8, 2)
>>> aujourdhui.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'08-02-15. 02 Aug 2015 is a Sunday on the 02 day of August.'
>>> # pour les avoir en français
>>> import locale
>>> locale.setlocale(locale.LC_ALL,'')
'French_France.1252'
>>> aujourdhui.strftime("%d/%m/%y. Le %d %b %Y est un %A et le jour %d de %B.")
'02/08/15. Le 02 août 2015 est un dimanche et le jour 02 de août.'
>>> # les dates supprotent l'arithmétique des calendriers
>>> dateRF = date(1789, 7, 14)
>>> age = aujourdhui - dateRF
>>> age.days
82563
```

10.9 Compression de données

Les formats courants d'archivage et de compression sont directement supportés par des modules existants, par exemple : **zlib**, **gzip**, **bz2**, **lzma**, **zipfile**, et **tarfile**,

```
>>> import zlib
>>> s = b'si mon tonton tond ton tonton, ton tonton sera tondu'
>>> len(s)
52
```

```
>>> t = zlib.compress(s)
>>> len(t)
36
>>> zlib.decompress(t)
b'si mon tonton tond ton tonton, ton tonton sera tondu'
>>> zlib.crc32(s) # contrôle de redondance cyclique
3692511599
```

10.10 Mesure de performances

Certains utilisateurs Python veulent comparer les performances entre des approches différentes pour résoudre un même problème. Python fournit un outil de mesure qui répond immédiatement à ces questions.

Par exemple, il peut être intéressant de comparer la technique d'emballage/déballage de tuples et l'approche traditionnelle pour permuter deux variables. Le module **timeit** démontre rapidement l'avantage pour la nouvelle approche :

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.09303532472681726
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.04785480170033196
```

Si le module **timeit** permet de tester finement des petits bouts de code, les modules **profile** et **pstats** fournissent des outils pour déterminer les goulots d'étranglement dans de plus grands blocs de code.

10.11 Contrôle qualité

Une approche pour produire des logiciels de haute qualité est d'écrire des tests pour chaque fonction en cours d'écriture et d'exécuter fréquemment ces tests durant le processus de développement.

Le module **doctest** fournit un outil pour examiner un module et ses tests de validations intégrés dans les chaînes de documentation du programme. La construction de test se fait simplement en copiant-collant un appel typique et son résultat attendu dans la chaîne de documentation. Cela améliore la documentation en fournissant à l'utilisateur un exemple d'appel et cela permet au module **doctest** d'assurer la validité du code par sa documentation :

```
>>> def moyenne(valeurs):
...     """Calcule la moyenne arithmétique d'une liste de nombres.
...
...     >>> print(moyenne([20, 30, 70]))
...     40.0
...     """
...     return sum(valeurs) / len(valeurs)

>>> import doctest
>>> doctest.testmod()   # valide automatiquement les test intégrés
TestResults(failed=0, attempted=1)
```

Le module **unittest** est moins simple d'utilisation que le module **doctest**, mais il permet de maintenir un ensemble de tests plus complet dans un fichier séparé :

```
import unittest

class TestFonctionsStatistiques(unittest.TestCase):
    def test_moyenne(self):
        self.assertEqual(moyenne([20, 30, 70]), 40.0)
        self.assertEqual(round(moyenne([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            moyenne([])
        with self.assertRaises(TypeError):
            moyenne(20, 30, 70)
```

```
>>> unittest.main() # Invoque tous les tests si appelé de la ligne de commande
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

10.12 Piles fournies

Python a une philosophie "piles fournies (*batteries included*). Cela se voit de manière claire en parcourant les codes perfectionnés et robustes de ses packages les plus importants. Par exemple :

- Les modules **`xmlrpc.client`** et **`xmlrpc.server`** permettent de réaliser très simplement des appels de procédures distantes (*remote procedure calls, RPC*). Malgré leurs noms, aucune connaissance de XML n'est nécessaire pour les utiliser.
- Le package **`email`** est une bibliothèque de gestion de messages électroniques, incluant les messages MIME ou basés sur RFC 2822. Contrairement aux modules **`smtplib`** et **`poplib`** qui simplement envoient et reçoivent des messages, le package **`email`** contient une boîte à outils complète, permettant de construire ou décoder des structures de messages complexes (y compris les pièces jointes) et pour implémenter un encodage internet et des protocoles d'en-tête.
- Le package **`json`** fournit un support robuste pour analyser ce format très populaire d'échange de données. Le module **`csv`** permet de lire ou d'écrire les fichiers en format de valeurs délimitées par les virgules (*Comma-Separated Value, CSV*), couramment proposé par les bases de données et les feuilles de calcul. La gestion de XML est supportée par les packages **`xml.etree.ElementTree`**, **`xml.dom`** et **`xml.sax`**. Ensemble, ces modules et packages simplifient énormément les échanges de données entre application Python et d'autres outils.
- Le module **`sqlite3`** est une enveloppe pour la bibliothèque de base de données SQLite, fournissant une base de données persistante qui peut être mise à jour et accédée en utilisant une syntaxe SQL légèrement non-standard.
- L'internationalisation est supportée par plusieurs modules comme **`gettext`**, **`locale`**, et le package **`codecs`**.

Chapitre 11

Seconde visite de la bibliothèque standard

Cette seconde visite parcourt des modules plus avancés destinés à des usages professionnels. Ces modules sont rarement utilisés dans de petits scripts.

11.1 Formatage de sorties

Le module **reprlib** fournit une version de **repr()** adaptée pour des affichages abrégés de conteneurs de données de grande taille ou profondément imbriqués :

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

Le module **pprint** fournit un contrôle plus élaboré de l'affichage d'objets prédéfinis ou utilisateurs de façon lisible par l'interpréteur. Si le résultat dépasse une ligne, l'afficheur élégant (*pretty printer*) ajoute des passages à la ligne et des indentations pour présenter clairement la structure de la donnée :

```
>>> import pprint
>>> t = [[[['noir', 'cyan'], 'blanc', ['vert', 'rouge']], [['magenta', 'jaune'], 'bleu']]]
>>> pprint.pprint(t, width=30)
[[[['noir', 'cyan'],
   'blanc',
   ['vert', 'rouge']],
  [['magenta', 'jaune'],
   'bleu']]]
```

Le module **textwrap** formate des paragraphes de texte en lignes de taille inférieure à une largeur donnée :

```
>>> import textwrap
>>> doc = """La méthode wrap() ressemble à la méthode fill(), mis à part qu'elle retourne
... une liste de chaînes au lieu d'une grande chaîne incluant des retours à la ligne."""
>>> print(textwrap.fill(doc, width=40))
La méthode wrap() ressemble à la méthode
fill(), mis à part qu'elle retourne une
liste de chaînes au lieu d'une grande
chaîne incluant des retours à la ligne.
```

Le module **locale** accède une base de données de formats spécifiques à une culture. L'attribut de regroupement (*grouping attribute*) de la fonction **format()** fournit un moyen direct de formater les nombres avec des séparateurs décimaux ou de milliers :

```
>>> import locale
>>> x = 1234567.8
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252') # direction USA
'English_United States.1252'
>>> conv = locale.localeconv()          # récupère un dictionnaire de conventions
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> print(locale.format_string(" %s%.*f ",
...             (conv['currency_symbol'], conv['frac_digits'], x), grouping=True))
 $1,234,567.80
>>> locale.setlocale(locale.LC_ALL, 'French_France.1252') # direction France
'French_France.1252'
>>> conv = locale.localeconv()          # récupère un dictionnaire de conventions
```

```
>>> locale.format("%d", x, grouping=True) # '\0xa' correspond à l'espace insécable
'1\xa0234\xa0567'
>>> print(locale.format_string(    "%.*f %s", (
...             conv['frac_digits'], x, conv['currency_symbol']), grouping=True))
1 234 567,80 €
```

11.2 Modèles de chaînes

Le module **string** inclut une classe polyvalente **Template** avec une syntaxe simplifiée utilisable pour l'édition de texte par des utilisateurs finaux. Ceci permet aux utilisateurs de personnaliser leurs applications sans avoir à les modifier.

Le formatage utilise des noms d'emplacement formés d'un **$** suivi d'un identificateur Python valide (caractères alphanumériques et tirets de soulignement). Si l'emplacement est entre accolades, il peut être suivi d'autres caractères accolés directement. L'écriture **$$** crée un seul caractère **$** :

```
>>> from string import Template
>>> t = Template("Un don de $somme $$ pour ${prefixe}physique.")
>>> t.substitute(somme=1000, intitule="département", prefixe = "l'astro")
"Un don de 1000 $ pour l'astrophysique."
>>> t.substitute(somme=10000, intitule="laboratoire", prefixe = "la méta")
'Un don de 10000 $ pour la métaphysique.'
```

La méthode **substitute()** déclenche une exception **KeyError** lorsqu'un emplacement n'est pas fourni dans un dictionnaire ou un argument mot-clé. Pour des applications de type publi-postage, les données utilisateur peuvent être manquantes. La méthode **safe_substitute()** peut s'avérer utile dans ce cas : elle laisse les emplacements inchangés si une donnée manque :

```
>>> t = Template('$item : retour au $proprio.')
>>> d = dict(item='hirondelle à vide')
>>> t.substitute(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Anaconda3\lib\string.py", line 125, in substitute
    return self.pattern.sub(convert, self.template)
  File "C:\Anaconda3\lib\string.py", line 115, in convert
    val = mapping[named]
KeyError: 'proprio'
>>> t.safe_substitute(d)
'hirondelle à vide : retour au $proprio.'
```

Les sous-classes de **Template** peuvent spécifier un délimiteur spécialisé. Par exemple, un petit utilitaire de renommage peut choisir d'utiliser des caractères **%** comme emplacement pour la date courante, le numéro d'image, ou le format de fichier :

```
>>> import time, os.path
>>> photos = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class Renommage(Template):
...     delimiter = '%'
...
>>> fmt = "Venise_%n%f" # ou input('Entrez style renommage (%d-date %n-num %f-format) : ')
>>> t = Renommage(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, nom_fichier in enumerate(photos):
...     base, ext = os.path.splitext(nom_fichier)
...     nouveau_nom = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(nom_fichier, nouveau_nom)) # c'est ici qu'on renommerait
...
img_1074.jpg --> Venise_0.jpg
img_1076.jpg --> Venise_1.jpg
img_1077.jpg --> Venise_2.jpg
```

Une autre application de **Template** est de séparer la logique d'un programme des détails de divers formats de sortie. Cela rend possible des modèles de substitution personnalisés pour des fichiers XML, des rapports en mode texte ou en HTML.

11.3 Gérer des données binaires structurées

Le module **struct** fournit les fonctions **pack()** et **unpack()** pour gérer des formats d'enregistrements binaires de taille variable. L'exemple suivant montre comment boucler sur les informations d'en-têtes de fichier

ZIP sans utiliser le module **zipfile**. Les codes d'empaquetage (*pack codes*) **"H"** et **"I"** représentent des nombres respectivement sur deux et quatre octets. L'indicateur **"<"** précise qu'ils sont de taille standard et en ordre *little-endian* :

```
>>> import struct
>>> with open('cours.zip', 'rb') as f: # fichier zip contenant 6 fichiers exemples
...     données = f.read()
...     pos = 0
...     for i in range(3): # lecture des trois premiers en-têtes de fichier
...         pos += 14
...         champs = struct.unpack('<IIHH', données[pos: pos+16])
...         crc32, taille_comp, taille_décomp, taille_nom_fic, taille_suppl = champs
...         pos += 16
...         nom_fic = données[pos: pos+taille_nom_fic]
...         pos += taille_nom_fic
...         extra = données[pos: pos + taille_suppl]
...         print(nom_fic, hex(crc32), taille_comp, taille_décomp)
...         pos += taille_suppl + taille_comp     # positionnement sur le prochain en-tête
...
b'demo.py' 0x50ef4c9 27 27
b'fibo.py' 0xd90a1245 185 393
b'heritage.py' 0xca72c7a2 661 1705
```

11.4 *Multi-threading*

Le *threading* est une technique permettant de dissocier des tâches indépendantes les unes des autres. Les *threads* peuvent être utilisés pour améliorer les temps de réponse dans les applications qui, par exemple, attendent des entrées utilisateurs tandis que d'autres activités s'exécutent en tâche de fond. Un cas d'utilisation proche est l'exécution parallèle d'entrés/sorties avec des calculs dans un autre *thread*.

Le code suivant montre comment le module **threading** peut exécuter des tâches en fond pendant que le programme principal continue :

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, fic_source, fic_cible):
        threading.Thread.__init__(self)
        self.fic_source = fic_source
        self.fic_cible = fic_cible
    def run(self):
        f = zipfile.ZipFile(self.fic_cible, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.fic_source)
        f.close()
        print('Compression en tâche de fond de', self.fic_source, 'terminée')

tâche_de_fond = AsyncZip('mon_fichier.txt', 'mon_archive.zip')
tâche_de_fond.start()
print("Le programme principal continue de s'exécuter.")

tâche_de_fond.join()    # attente de la terminaison de la tâche de fond
print('Le programme a attendu la terminaison de la tâche de fond.')
```

La principale difficulté dans l'écriture d'applications *multi-threads* est la coordination des *threads* partageant des données ou d'autres ressources. Dans ce but, le module **threading** fournit des primitives de synchronisation : verrous, événements, variables de condition et sémaphores.

Toutefois, bien que ce soient des outils puissants, de petites erreurs de conception peuvent conduire à des problèmes difficiles à reproduire. Aussi, l'approche favorite des programmeurs pour les tâches de coordination, est de concentrer tous les accès à une ressource dans un seul *thread* et d'utiliser le module **queue** pour alimenter ce *thread* avec des requêtes en provenance d'autres *threads*. Les applications utilisant des objets **Queue** pour la communication et la coordination *inter-thread* sont plus simples à concevoir, plus lisibles et plus fiables.

11.5 Journalisation

Le module **logging** propose un système de journalisation complet et flexible. Dans son utilisation de base, les messages de journal sont enregitrés dans un fichier ou envoyés sur **sys.stderr** :

```
import logging
logging.debug('Information de débogage')
logging.info("Message d'information")
```

```
logging.warning('Avertissement : fichier de configuration %s non trouvé', 'serveur.conf')
logging.error("Une erreur s'est produite")
logging.critical("Erreur critique -- fermeture de l'application")
```

Les sorties suivantes sont produites :

```
WARNING:root:Avertissement : fichier de configuration serveur.conf non trouvé
ERROR:root:Une erreur s'est produite
CRITICAL:root:Erreur critique -- fermeture de l'application
```

Par défaut, les messages d'information et de débogage sont supprimés et les sorties se font sur l'erreur standard. Les autres options de sortie comprennent l'envoi des messages par email, datagrammes, *sockets* ou vers un serveur HTTP. De nouveaux filtres permettent de sélectionner différents routages basés sur la priorité des messages : `DEBUG`, `INFO`, `WARNING`, `ERROR`, et `CRITICAL`.

Le système de journalisation peut être configuré directement à partir de Python ou chargé à partir d'un fichier de configuration éditable par l'utilisateur, permettant de personnaliser la journalisation sans modifier l'application.

11.6 Références faibles

Python gère automatiquement la mémoire (comptage de références pour la plupart des objets et récupération mémoire – *garbage collection* – pour éliminer les cycles. La mémoire est libérée peu de temps après que la dernière référence a été éliminée.

Cette approche fonctionne correctement pour la plupart des applications, mais il arrive qu'il soit nécessaire de suivre à la trace les objets tant qu'ils sont utilisés. Malheureusement, le seul fait d'observer un objet crée une référence sur lui qui le rend permanent. Le module `weakref` fournit des outils permettant de suivre des objets sans créer de référence. Lorsque l'objet n'est plus utilisé, il est automatiquement supprimé d'une table de références faibles et une fonction de rappel (*callback*) est déclenchée pour les objets référence faible. Les applications typiques comportent des caches pour objets coûteux à construire :

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, valeur):
...         self.valeur = valeur
...     def __repr__(self):
...         return str(self.valeur)
...
>>> a = A(10)                    # crée une référence
>>> d = weakref.WeakValueDictionary()
>>> d['primaire'] = a            # ne crée pas de référence
>>> d['primaire']                # retourne l'objet s'il est toujours "vivant"
10
>>> del a                        # supprime l'unique référence
>>> gc.collect()                 # force l'exécution de la récupération mémoire
0
>>> d['primaire']                # l'entré a été automatiquement supprimée
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python34/lib/weakref.py", line 125, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Utilitaires pour les listes

De nombreuses structures de données peuvent être implémentées par le type prédéfini `list`. Cependant, il est parfois nécessaire d'utiliser des implémentations alternatives avec d'autres compromis de performance.

Le module `array` fournit un objet `array()` se comportant comme une liste ne contenant que des données de même type et les mémorisant de manière plus compacte. L'exemple suivant montre un tableau de nombres non signés stockés sur deux octets (code de type `"H"`), plutôt que l'habituel stockage sur 16 octets pour les éléments de type `int` dans une liste normale :

```
>>> 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 un objet **deque()** se comportant comme une liste, mais avec des ajouts en fin de liste et des suppressions en tête de liste plus efficaces, mais des accès en pleine liste plus lents. Ces objets sont bien adaptés aux files (*queues*) et aux parcours d'arbre en largeur :

```
>>> from collections import deque
>>> d = deque(["tâche1", "tâche2", "tâche3"])
>>> d.append("tâche4")
>>> print("Gestion", d.popleft())
Gestion tâche1
```

```
noeuds_non_exploités = deque([noeud_initial])
def parcours_en_largeur(noeuds_non_exploités):
    noeud_courant = noeuds_non_exploités.popleft()
    for noeud_suivant in genère_mouvement(noeud_courant):
        if est_but(noeud_suivant):
            return noeud_suivant
        noeuds_non_exploités.append(noeud_suivant)
```

En plus de ces implémentations alternatives de listes, la bibliothèque standard offre d'autres outils comme le module **bisect** proposant des fonctions de manipulation de listes triées :

```
>>> import bisect
>>> meilleurs_scores = [(100, 'Eric'), (200, 'Michael'), (400, 'John'), (500, 'Terry')]
>>> bisect.insort(meilleurs_scores, (300, 'Graham'))
>>> meilleurs_scores
[(100, 'Eric'), (200, 'Michael'), (300, 'Graham'), (400, 'John'), (500, 'Terry')]
```

Le module **heapq** propose des fonctions permettant d'implémenter des tas (*heaps*) basées sur des listes ordinaires. L'entrée de valeur minimum est toujours stockée en position zéro, ce qui est utile pour des applications qui accèdent de manière répétée au plus petit élément, sans pour cela nécessiter un tri complet de la liste :

```
>>> from heapq import heapify, heappop, heappush
>>> données = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(données)                      # réordonne la liste en un tas
>>> heappush(données, -5)                 # ajoute un élément (conservant la structure de tas)
>>> [heappop(données) for i in range(3)]  # retire et fournit les trois plus petites entrées
[-5, 0, 1]
```

11.8 Arithmétique en virgule flottante décimale

Le module **decimal** propose offre le type de données **Decimal** pour l'arithmétique en virgule flottante décimale. Comparé au type prédéfini **float** (implémentation en virgule flottante binaire), cette classe est spécialement appropriée pour :

- les applications financières et celles qui nécessitent une représentation décimale exacte,
- contrôler la précision de manière fine,
- contrôler les arrondis pour respecter les dispositions légales ou réglementaires,
- contrôler le nombre de chiffres significatifs,
- les applications pour lesquelles l'utilisateur s'attend à des résultats identiques à ceux obtenus si les calculs étaient faits à la main.

Par exemple, la prise en compte d'une taxe de 5% sur un appel téléphonique à 70 centimes donne des résultats différents, selon que le calcul s'effectue en virgule flottante décimale ou en virgule flottante binaire. La différence est visible si les résultats sont arrondis au centime le plus proche :

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2) # décimal
Decimal('0.74')
>>> round(.70 * 1.05, 2) # binaire
0.73
```

Le résultat en **Decimal** maintient un zéro significatif, produisant automatiquement des résultats à quatre chiffres significatifs pour des multiplicandes à deux chiffres significatifs. **Decimal** simule les mathématiques faites à la main et permet d'éviter les problèmes qui peuvent survenir lorsque la représentation en virgule flottante binaire ne peut pas être exacte pour une quantité décimale.

Le représentation exacte permet à la classe **Decimal** d'effectuer des calculs modulaires et des tests d'égalité, impraticables en 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 également d'augmenter le nombre de chiffres significatifs des calculs :

```
>>> getcontext().prec = 60
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')
>>> Decimal.sqrt(Decimal(2))
Decimal('1.41421356237309504880168872420969807856967187537694807317668')
```

Chapitre 12

Environnements virtuels et packages

12.1 Introduction

Les applications Python utilisent souvent des packages et des modules ne faisant pas partie de la bibliothèque standard. Ces applications nécessitent parfois une version spécifique d'une bibliothèque, car elles requièrent qu'un bug particulier ait été corrigé ou elles ont été écrites en utilisant une ancienne version de l'interface de cette bibliothèque.

Cela signifie qu'il n'est pas toujours possible pour une installation Python de convenir aux contraintes de versions pour toutes les applications. Si l'application A demande la présence de la version 1.0 d'un certain module et une application B nécessite la version 2.0, alors les contraints sont incompatibles, et l'installation de l'une des versions rendra impossible l'exécution d'une des deux applications.

La solution à ce problème consiste à créer un environnement virtuel (*virtual environment*, souvent abrégé en *virtualenv*), une branche du système de fichiers autosuffisante, contenant une installation d'une certaine version de Python et de packages additionnels.

Différentes applications peuvent alors utiliser différents environnements virtuels. Pour résoudre les conflits de l'exemple précédent, l'application A peut disposer de son propre environnement virtuel avec la version 1.0 installée, alors que l'application B aura un autre environnement virtuel avec la version 2.0. Si le développement de l'application B nécessite un jour la version 3.0, cela n'affectera pas l'environnement de l'application A.

12.2 Création d'environnements virtuels

Le script utilisé pour créer et gérer des environnements virtuels se nomme **`pyvenv`**.

`pyvenv` installera généralement la version disponible la plus récente de Python ; le script est également installé avec un numéro de version : si vous disposez de plusieurs versions de Python sur votre système, vous pouvez sélectionner la version désirée en spécifiant par exemple **`pyvenv-3.4`** ou la version que vous désirez.

Pour créer un environnement virtuel, choisissez un répertoire dans lequel le placer et lancez **`pyvenv`** avec le nom de chemin du répertoire :

```
pyvenv mon-environnement
```

Cette commande créera le répertoire **`mon-environnement`** s'il n'existe pas, ainsi que des sous-répertoires contenant une copie de l'interpréteur Python, la bibliothèque standard, et des fichiers auxiliaires divers.

Une fois créé l'environnement virtuel, vous devez l'activer.

Sous Windows, lancez :

```
mon-environnement/Scripts/activate
```

Sous Unix ou MacOS, lancez :

```
source mon-environnement/bin/activate
```

Ce script est écrit pou le *bash shell*. Si vous utilisez les shells *csh* ou *fish*, utilisez les scripts dédiés **`activate.csh`** ou **`activate.fish`**.

L'activation de l'environnement virtuel modifie l'invite de commande du shell pour indiquer l'environnement utilisé. et modifiera l'environnement de sorte que l'exécution de **`python`** vous lancera cette installation particulière de Python. Par exemple :

```
-> source ~/envs/mon-environnement/bin/activate
(mon-environnement) -> python
Python 3.4.3+ (3.4:c7b9645a6f35+, May 22 2015, 09:31:25) ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python34.zip', ...,
'~/envs/mon-environnement/lib/python3.4/site-packages']
>>>
```

12.3 Gestion des packages avec `pip`

Une fois activé l'environnement virtuel, vous pouvez y installer, mettre à jour ou supprimer des packages grâce à un programme appelé **pip**. Par défaut, **`pip`** installe des packages en provenance du *Python Package Index*, **<https://pypi.python.org/pypi>**. Vous pouvez parcourir le *Python Package Index* en vous y connectant par Internet, ou vous pouvez utilisez les possibilité limitées de **`pip`** :

```
(mon-environnement) -> pip search astronomy
gammapy                - A Python package for gamma-ray astronomy
image_registration     - Image Registration Tools for extended images in astronomy.
astropyp               - Astronomy Pypeline Framework and FITS Viewer
astropy                - Community-developed python astronomy tools
gary                   - Galactic astronomy and gravitational dynamics.
palpy                  - PAL -- A Positional Astronomy Library
novas                  - The United States Naval Observatory NOVAS astronomy library
PyAstronomy            - A collection of astronomy related tools for Python.
python-casacore        - A wrapper around CASACORE, the radio astronomy library
astroobs               - Provides astronomy ephemeris to plan telescope observations
skyfield               - Elegant astronomy for Python
astroML                - tools for machine learning and data mining in Astronomy
```

`pip` possède plusieurs sous-commandes : "search", "install", "uninstall", "freeze", etc. Consultez le guide *Installing Python Modules* pour une documentation complète de **`pip`**.

Vous pouvez installer la dernière version d'un package en précisant son nom :

```
(mon-environnement) -> 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 également installer une version particulière du package en précisant son nom suivi de **`==`** et du numéro de version :

```
(mon-environnement) -> 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`** remarquera que la version demandée est déjà installée et ne fera rien. Vous pouvez fournir un autre numéro de version pour l'obtenir, ou lancer **`pip install --upgrade`** pour obtenir la dernière version de votre package :

```
(mon-environnement) -> 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
```

pip uninstall suivi d'un ou plusieurs noms de packages supprimera ces packages de l'environnement virtuel.

pip show affichera les informations sur le package spécifié :

```
(mon-environnement) -> 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/jcg/envs/mon-environnement/lib/python3.4/site-packages
Requires:
```

pip list affichera tous les packages installés dans l'environnement virtuel :

```
(mon-environnement) -> pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

pip freeze générera une liste similaire des packages installés, mais avec le format attendu par la commande **pip install**. Une convention répandue est d'enregistrer cette liste dans un fichier **requirements.txt** :

```
(mon-environnement) -> pip freeze > requirements.txt (mon-environnement) -> cat
requirements.txt novas==3.1.1.3 numpy==1.9.2 requests==2.7.0
```

Le fichier **requirements.txt** peut faire partie du contrôle de version et être intégré dans l'application. Les utilisateurs pourront alors installer tous les packages nécessaires avec **install -r** :

```
(mon-environnement) -> 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 comporte beaucoup plus d'options. Consultez le guide *Installing Python Modules* pour une documentation complète sur **pip**. Lorsque vous aurez écrit un package, si vous voulez le soumettre au *Python Package Index*, consultez le guide *Distributing Python Modules guide*.

Chapitre 13

Comment aller plus loin ?

La lecture de ce tutoriel a probablement renforcé votre intérêt pour Python - vous êtes sans doute impatient d'utiliser Python pour des problèmes réels. Où vous rendre pour en savoir plus ?

Ce tutoriel fait partie de la documentation Python. D'autres éléments de la documentation sont :

- *The Python Standard Library*

  Vous devriez parcourir ce manuel, qui fournit une référence complète, quoique concise, sur les types, fonctions et modules de la bibliothèque standard. La distribution standard de Python comprend énormément de code additionnel : des modules pour gérer les courriers électroniques sous Unix, récupérer des document par HTTP, générer des nombres aléatoires, analyser les options de la ligne de commande, écrire des programmes CGI, compresser des données, et beaucoup d'autres possibilités. Parcourir *The Python Standard Library* vous donnera des idées sur tout ce qui existe.
- *Installing Python Modules* indique comment installer des modules additionnels écrits par d'autres utilisateur Python.
- *The Python Language Reference* : une explication détaillée de la syntaxe et de la sémantique Python. Sa lecture est pas fastidieuse, mais elle est profitable pour avoir un guide complet du langage lui-même.

Encore quelques ressources Python :

- https://www.python.org : Le principal site Web Python. Il contient du code, de la documentation et des liens vers des pages Web liées à Python. Ce site Web est dupliqué dans diverses régions du monde (Europe, Japon, Australie, etc.) ; un site miroir peut être plus rapidement accessible que le site principal, selon votre localisation.
- https://docs.python.org : accès rapide à toute la documentation Python.
- https://pypi.python.org/pypi : *The Python Package Index*, surnommé auparavant *The Cheese Shop*, est un index des modules utilisateurs disponible en téléchargement. Une fois que vous avez écrit du code, vous pouvez l'enregistrer ici pour le mettre à disposition d'autre utilisateurs.
- http://code.activestate.com/recipes/langs/python/ : *The Python Cookbook* est une énorme compilation d'exemples de codes, de grands modules et de scripts utiles. Les contributions les plus notables sont réunies dans le livre intitulé également *Python Cookbook* (O'Reilly & Associates, ISBN 0-596-00797-3.)
- http://www.pyvideo.org compile des liens vers des vidéos liée à Python, filmées lors de conférences ou de rencontres d'utilisateurs.
- http://scipy.org : Le projet *Scientific Python* regroupe des modules pour le calcul rapide et la manipulations efficace des matrices, et de nombreux packages pour l'algèbre linéaire, les transformées de Fourier, les solveurs non linéaires, les distributions aléatoires, l'analyse statistique, etc.

Si vous avez des questions ou des problèmes liés à Python, vous pouvez les soumettre au forum *comp.lang.python*, ou les envoyer à la *mailing list* à l'adresse python-list@python.org.

Le forum et la *mailing list* sont reliés (les messages postés à l'un sont automatiquement transmis à l'autre). Il y a des centaines de messages par jour, posant (et répondant à) des questions, proposant de nouvelles fonctionnalités, ou annonçant la mise à disposition de nouveaux modules. Les archives sont disponibles à https://mail.python.org/pipermail/.

Avant de soummettre une question, vérifiez qu'elle n'est pas déjà dans la liste des *Frequently Asked Questions*. La FAQ répond à de nombreuses questions qui reviennent souvent, et contient peut-être déjà la solution à votre problème.

Chapitre 14

Édition de ligne de commande et historique

Certaines versions de l'interpréteur Python proposent les facilités d'édition sur la ligne de commande et un mécanisme d'historique, similaires aux fonctionnlités disponibles dans le *Korn shell* ou le *GNU Bash shell*. Elles sont implémentées en utilisant la bibliothèque GNU Readline, proposant plusieurs styles d'édition. Cette bibliothèque dispose de sa propre documentation, qui ne sera pas reproduite ici.

14.1 Complétion automatique et historique

La complétion des noms de variables et de modules est *automatiquement activée* au lancement de l'interpréteur, de manière à ce que la touche `Tab` appelle la fonction de complétion ; celle-ci parcourt les noms des instructions Python, des variables locales et des modules disponibles. Pour les expressions pointées (comme `string.a`), elle évalue l'expression jusqu'au `'.'` final, puis suggère des complétions à partir des attributs de l'objet résultant. Notez que cela risque d'exécuter du code défini dans l'application si un objet disposant d'une méthode `__getattr__()` fait partie de l'expression. La configuration par défaut sauvegarde également votre historique dans un fichier de nom `.python_history` dans votre répertoire utilisateur. L'historique sera donc à nouveau disponible lors de la prochaine session interactive de l'interpréteur.

14.2 Alternatives à l'interpréteur interactif

Ces facilités constituent un énorme pas en avant par rapport aux versions précédentes de l'interpréteur ; toutefois, ils subsiste quelques souhaits non encore pris en compte : il serait agréable que l'indentation correcte soit proposée sur les lignes de suite (l'analyseur syntaxique peut savoir si une indentation est nécessaire sur la ligne suivante). Le mécanisme de complétion devrait utiliser la table des symboles de l'interpréteur. Une commande pour tester (ou suggérer) les appariements de parenthèses, quotes, etc. serait également bienvenue.

Un interpréteur interactif amélioré existe depuis un certain temps : **IPython**, qui propose la complétion par tabulation, l'exploration d'objet et la gestion avancée de l'historique. Il peut également être complètement personnalisé et intégré dans d'autres applications. Un autre environnement interactif avancé est **bpython**.

Chapitre 15

Arithmétique en virgule flottante : problèmes et limitations

Les nombres à virgule sont représentés en machine en base deux (binaire). Par exemple, le nombre à virgule $0,125_{10}$ vaut $1/10+2/100+5/1000$. De même, le nombre à virgule $0,001_2$ vaut $0/2+0/4+1/8$. Ces deux nombres ont la même valeur, mais le premier est écrit en base dix, et le second en base deux.

Malheureusement, la plupart des nombres à virgules écrits en base dix ne peuvent être représentés exactement en base deux. En conséquence, les nombres à virgule (que vous entrez en base dix) ne sont en général qu'approximés par les nombres en base deux stockés dans la machine.

Le problème est plus facile à comprendre si l'on commence par la base dix. Considérons la fraction $1/3$. On peut l'approximer par un nombre en base dix : $0,3_{10}$ ou $0,33_{10}$ ou $0,333_{10}$ et ainsi de suite.

Quel que soit le nombre de chiffres que vous écrirez, le résultat ne sera jamais exactement $1/3$, mais une approximation de plus en plus précise de $1/3$.

De la même manière, quel que soit le nombre de chiffres binaires que vous écrirez, il est impossible de représenter exactement en base deux le nombre $0,1_{10}$. En base deux, la valeur $1/10$ est la répétition infinie des chiffres $0,0001100110011001100110011...{}_2$

Arrêtez l'écriture à n'importe quel nombre fini de bits, et vous obtiendrez une approximation. Sur la plupart des machines actuelles, les nombres à virgule sont approximés par une fraction en base deux dont le numérateur est le nombre formé par les 53 premiers bits significatifs du nombre et le dénominateur est une puissance de deux. Dans le cas de $1/10$, la fraction est $3602879701896397/2^{55}$ qui est très proche, mais pas égale, à la vraie valeur de $1/10$.

De nombreux utilisateurs ne se rendent pas compte de cette approximation, à cause de la manière dont les nombres sont affichés. Python n'affiche qu'une approximation décimale de la valeur décimale exacte de l'approximation binaire stockée dans la machine. Sur la plupart des machine, si Python affichait la valeur la valeur décimale exacte de l'approximation binaire stockée pour $0,1_{10}$, l'affichage serait :

```
>>> 0.1 # affichage fictif
0.1000000000000000055511151231257827021181583404541015625
```

C'est plus de chiffres que nécessaire pour la plupart des gens, aussi Python ne conserve qu'un nombre acceptable de chiffres en affichant une valeur arrondie :

```
>>> 0.1 # affichage réel
0.1
```

Souvenez-vous que même si l'affichage ressemble à la valeur exacte de $1/10$, la vraie valeur stockée en mémoire est la fraction binaire représentable la plus proche.

Curieusement, il s'en suit que de nombreux nombres décimaux ont la même fraction binaire représentable la plus proche. Par exemple :

$0,1_{10}$ $0,10000000000000001_{10}$ $0,1000000000000000055511151231257827021181583404541015625_{10}$ sont tous trois approximés par $3602879701896397/2^{55}$. Comme toutes ces valeurs décimales partagent la même approximation, chacune d'entre elle pourrait être utilisée pour l'affichage tout en préservant l'invariant `eval(repr(x)) == x`.

Historiquement, l'affichage (la fonction `repr()`) de Python choisissait la représentation à 17 chiffres significatifs `0.10000000000000001`. Depuis Python 3.1, Python (sur la plupart des systèmes) est maintenant capable de choisir la plus courte pour afficher simplement `0.1`.

Notez que cela est dû à la véritable nature de la représentation en virgule flottante binaire ; ce n'est pas un bug ni de Python, ni de votre code. Vous rencontrerez cet aspect des choses dans tous les langages intégrant

l'arithmétique en virgule flottante (bien que certains langages ne l'affichent pas, que ce soit par défaut ou dans tous les modes d'affichage).

Pour des affichages plus agréables, vous pouvez utiliser le formatage de chaînes pour ne conserver qu'un nombre limité de chiffres significatifs :

```
>>> format(math.pi, '.12g')  # 12 chiffres significatifs
'3.14159265359'

>>> format(math.pi, '.2f')   # 2 chiffres après le point décimal
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Il est important de réaliser que ce n'est qu'une illusion : vous n'arrondissez que l'affichage approximatif d'une valeur machine exacte.

Une illusion peut en engendrer une autre. Par exemple, comme $0,1_{10}$ ne vaut pas exactement $1/10$, additionner trois représentations de $0,1_{10}$ ne résultera pas en une représentation de $0,3_{10}$:

```
>>> .1 + .1 + .1 == .3
False
```

De plus, comme la représentation de $0,1_{10}$ ne peut pas s'approcher plus de la valeur exacte de $1/10$ et que celle de $0,3_{10}$ ne peut pas s'approcher plus de la valeur exacte de $3/10$, l'arrondi préalable avec la fonction `round()` ne sera p'aucun secours :

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Bien que les nombres ne puissent s'approcher plus de leur valeurs exactes, la fonction `round()` peut être utile pour arrondir **après** les calculs de façon à ce que des résultats avec des valeurs inexactes puissent être comparés.

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

L'arithmétique en virgule flottante binaire réserve bien d'autres surprises. Le problème de `0.1` est expliqué en détail plus loin, dans la section "Erreur de représentation". Voir aussi l'article *The Perils of Floating Point* pour une revue plus complète d'autres surprises courantes.

Comme il est dit à la fin de l'article, "il n'y a pas de réponses simples" (*there are no easy answers*). Toutefois, ne soyez pas trop paranoïaque ! Les erreurs dans les opérations sur les `float` en Python sont héritées de l'implémentation en machine de l'arithmétique virgule flottante, et sur la plupart des machines, elles sont de l'ordre de $1/2^{53}$ par opération. ; c'est plus que convenable pour la plupart des tâches, mais il faut garder à l'esprit que ce n'est pas de l'arithmétique décimale, et que chaque opération impliquant des `float` peut générer une nouvelle erreur d'arrondi.

Bien que des cas pathologiques existent, le plus souvent, dans les cas ordinaires d'utilisation de l'arithmétique virgule flottante, vous obtiendrez à la fin les résultats attendus si vous arrondissez simplement l'affichage de vos résultats finaux au nombre de décimales désiré. `str()` suffit en général, et pour un contrôle plus fin, regardez les spécifications de la méthode `str.format()` dans la documentation *Format String Syntax*.

Pour les usages requérant une représentation exacte en décimal, utilisez le module `decimal` qui implémente une arithmétique décimale appropriée pour les applications comptables ou de haute précision.

Une autre forme d'arithmétique exacte est proposée par le module `fractions` qui implémente une arithmétique basée sur les rationnels (des nombres comme $1/3$ y sont représentés de manière exacte).

Si vous êtes un utilisateur acharné d'opérations en virgule flottante, vous devriez regarder de plus près le package *Numerical Python* et d'autres packages pour les calculs mathématiques et statistiques proposés par le projet *SciPy*. Voir <http://scipy.org>.

Python fournit des outils pouvant vous aider dans les rares cas où vous voulez connaître la valeur exacte d'un `float`. La méthode `float.as_integer_ratio` exprime la valeur d'un `float` en tant que fraction :

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Comme le quotient est exact, il peut être utilisé pour recréer la valeur originale sans perte.

```
>>> x == 3537115888337719 / 1125899906842624
```

```
True
```

La méthode `float.hex()` exprime un `float` en hexadécimal (base 16), donnat encore une fois la valeur exacte stockée en mémoire :

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Cette représentation hexadécimale exacte peut être utilisée pour reconstruire exactement la valeur `float` de départ :

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Comme cette représentation est exacte, elle est commode transférer de manière fiable des valeurs entre différentes versions de Python (indépendant de la plate-forme) ou échanger des données avec d'autres langages supportant le même format (comme Java ou C99).

Un autre outil pratique est la fonction `math.fsum()` qui aide à atténuer la perte de précision durant une addition. Elle prend en compte les "chiffres perdus" lorsque des valeurs s'ajoutent durant une longue addition. Cela peut faire une différence dans la précision totale, de sorte que les erreurs ne s'accumulent pas autant que dans le résultat de base :

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Erreur de représentation

Cette section explique en détail l'exemple de $0,1_{10}$ et montre comment réaliser par vous-même une analyse exacte de cas similaires. Une connaissance de base de la représentation en virgule flottante binaire est supposée acquise.

L'erreur de représentation se rapporte au fait que certains (en fait, presque tous) nombres à virgules ne peuvent être représentés exactement en binaire (base deux). C'est la principale raison qui fait que Python (ou Perl, C, C++, Java, Fortran, et nombre d'autres langages) affichent souvent un autre nombre à virgule que celui attendu.

Pourquoi ? 1/10 n'est pas exactement représentable en base deux. presque toutes les machines actuelles (Novembre 2000) utilise l'arithmétique en virgule flottante IEEE-754, et la plupart des plate-formes appliquent aux `float` Python la norme "IEE-754 double précision". Un *double IEE-754* contient 53 bits significatifs, aussi lors de l'entrée de `0.1`, l'ordinateur tente de former la fraction la plus proche avec un numérateur entier de 53 bits et un dénominateur égal à une puissance entière de 2.

L'égalité *cible* :

$$\frac{1}{10} = \frac{J}{2^N}$$

peut être réécrite en :

$$J = \frac{2^N}{10}$$

Nous savons que J doit avoir exactement 53 bits, c'est à dire :

$$2^{52} \leq J < 2^{53}$$

ou encore :

$$2^{52} \leq \frac{2^N}{10} < 2^{53}$$

soit :

$$52 + \log_2(10) \leq N < 53 + \log_2(10)$$

Le calcul de

```
>>> import math
>>> 52 + math.log(10,2)
55.32192809488736
```

nous permet d'obtenir la valeur de N : 56

La meilleure valeur possible pour J est donc l'arrondi de la division par 10 de 2^{56} :

```
>>> q, r = divmod(2 ** 56, 10)
>>> print(q, r)
7205759403792793 6
```

Comme le reste r est supérieur à 5, nous arrondissons q à l'entier supérieur, soit 7205759403792794

La meilleure approximation de 1/10 en IEE-754 "double précision" est donc $7205759403792794/2^{56}$

En simplifiant, la fraction devient $3602879701896397/2^{55}$

Notez que comme nous avons arrondi à la valeur supérieure, cette fraction est légèrement supérieure à 1/10. Si nous avions conservé la valeur de q, elle aurait été légèrement ingérieure à 1/10. Mais en aucun cas, elle ne pouvait être exactement égale à 1/10 !

Ainsi, l'ordinateur ne "voit" jamais 1/10 : ce qu'il voit est la fraction ci-dessus, la meilleure approximation qu'il puisse obtenir :

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Si nous multiplions cette fraction par 10^{55}, nous pouvons voir la valeur des 55 décimales :

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

ce qui signifie que le nombre exact stocké dans l'ordinateur est égal à :

$0,1000000000000000055511151231257827021181583404541015625_{10}$.

Plutôt que d'afficher la totalité des chiffres, beaucoup de langages (y compris les anciennes versions de Python) arrondissent le résultat à 17 chiffres significatifs)

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Les modules **fractions** ou **decimal** simplifient tous 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

Appendice

16.1 Mode interactif

16.1.1 Gestion d'erreur

Lorsqu'une erreur se produit, l'interpréteur affiche un message d'erreur et une pile d'appels. En mode interactif, il revient à l'invite primaire ; lorque l'entrée provinet d'un fichier, il s'arrête en fournissant un code d'erreur non nul après avoir affiché la pile des 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 produisent une sortie de Python avec un code erreur non nul ; c'est le cas pour des incohérences internes ou des cas d'insuffisance de mémoire. Tous les messages d'erreurs sont écrits sur le flux d'erreur standard ; les affichages normaux des commandes exécutées sont écrits sur le flux de sortie standard.

La frappe du caractère d'interruption clavier (Control-C ou DEL) sur l'invite primaire ou secondaire annule l'entrée et revient à l'invite primaire [1]. Une interruption clavier durant l'exécution d'une commande déclenche l'exception `KeyboardInterrupt`, qui peut être gérée par une instruction `try`.

16.1.2 Scripts Python exécutables

Sur les systèmes Unix BSD, les scripts Python peuvent être rendus directement exécutable en écrivant au début du script la ligne

```
#!/usr/bin/env python3.4
```

(à condition que l'interpréteur soit accessible par la variable d'environnement **`PATH`**) et en rendant le fichier exécutable. Les caractères **`#!`** doivent être les deux premiers caractères du fichier. Sur certaines plates-formes, cette première ligne doit se terminer par une fin de ligne Unix (**`'\n'`**) et non une fin de ligne Windows (**`'\r\n'`**). Notez que le croisillon (*hash character, pound character*) **`'#'`** est utilisé pour débuter les commentaires Python.

Le script est rendu exécutable avec la commande **`chmod`** command.

```
$ chmod +x mon_script.py
```

Sur les systèmes Windows, il n'y a pas de notion de "mode exécutable". L'installateur Python associe automatiquement les fichiers d'extension **`.py`** avec **`python.exe`**, ce qui fait qu'un double-clic sur un fichier Python le lancera en tant que script. L'extension peut également être **`.pyw`**, dans ce cas, la fenêtre de console qui apparaît normalement est supprimée.

16.1.3 Le fichier de démarrage interactif

Lors des sessions interactives de Python, il est courant d'avoir à entrer quelques commandes fréquemment utilisées chaque fois que l'interpréteur est lancé. On peut automatiser ceci en créant une variable d'environnement nommée `PYTHONSTARTUP` dont la valeur sera le nom d'un fichier contenant les commandes de démarrage. C'est similaire à la fonctionnalité **`.profile`** des shells Unix.

Ce fichier n'est lu que lors des sessions interactives, pas lorsque Python lit des commandes en provenance d'un script, ni quand **`/dev/tty`** est explicitement donné en tant que source de commandes (ce qui fonctionne, à part ça, comme une session interactive). Il est exécuté dans le même espace de noms que celui où les commandes interactives sont exécutées, donc les objets qu'il définit ou importe pourront être utilisés sans

1. Un problème dans le package *GNU Readline* peut produire une comportement différent

qualification dans la session interactive. On peut même modifier les invites de commande **sys.ps1** and **sys.ps2** dans ce fichier.

Si l'on veut lire un fichier de démarrage supplémentaire à partir du répertoire en cours, on peut le programmer dans le fichier de démarrage global avec une commande du genre **if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())**. Si l'on veut utiliser le fichier de démarrage dans un script, on doit l'expliciter dans le script :

```
import os
nom_fic = os.environ.get('PYTHONSTARTUP')
if nom_fic and os.path.isfile(nom_fic):
    with open(nom_fic) as f:
        commandes_demarrage = f.read()
    exec(commandes_demarrage)
```

16.1.4 Les modules de personnalisation

Python fournit deux moyens permettant de le personnaliser : **sitecustomize** et **usercustomize**. Pour voir comment faire, il faut déjà trouver la localisation sur votre site du répertoires **site-packages**. Lancez Python et exécutez ce code :

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.4/site-packages'
```

Vous pouvez maintenant créer un fichier de nom **usercustomize.py** dans ce répertoire et y mettre tout ce que vous y voulez. Il sera lu à chaque invocation de Python, sauf si Python est démarré avec l'option **-s** qui désactive l'importation automatique.

sitecustomize fonctionne de la même façon, mais est d'habitude créé par un administrateur de la machine dans le répertoire global **site-packages**, et il est importé avant **usercustomize**. Voir la documentation du module **site** pour plus de détails.

Annexe A

GLOSSAIRE

>>>

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

...

The default Python prompt of the interactive shell when entering code for an indented code block or within a pair of matching left and right delimiters (parentheses, square brackets or curly braces).

2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as **lib2to3**; a standalone entry point is provided as **Tools/scripts/2to3**. See *2to3 - Automated Python 2 to 3 code translation*.

abstract base class

Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like **hasattr()** would be clumsy or subtly wrong (for example with *magic methods*). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by **isinstance()** and **issubclass()**; see the **abc** module documentation. Python comes with many built-in ABCs for data structures (in the **collections.abc** module), numbers (in the **numbers** module), streams (in the **io** module), import finders and loaders (in the **importlib.abc** module). You can create your own ABCs with the **abc** module.

argument

A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. **name=**) in a function call or passed as a value in a dictionary preceded by ******. For example, **3** and **5** are both keyword arguments in the following calls to **complex()**:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by *****. For example, **3** and **5** are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the *Calls* section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on *the difference between arguments and parameters*, and PEP 362.

attribute
A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file
A *file object* able to read and write *bytes-like objects*.

> **See also:** A *text file* reads and writes `str` objects.

bytes-like object
An object that supports the *Buffer Protocol*, like `bytes`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode
Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.
A list of bytecode instructions can be found in the documentation for *the dis module*.

class
A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

coercion
The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See PEP 343.

CPython
The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator
A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.
The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):
    ...
f = staticmethod(f)
```

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

The same concept exists for classes, but is less commonly used there. See the documentation for *function definitions* and *class definitions* for more about decorators.

descriptor

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see *Implementing Descriptors*.

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

docstring

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

duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `if`. Assignments are also statements, not expressions.

extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

file object

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

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*, . Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object

A synonym for *file object*.

finder

An object that tries to find the *loader* for a module. It must implement either a method named `find_loader()` or a method named `find_module()`. See PEP 302 and PEP 420 for details and `importlib.abc.Finder` for an *abstract base class*.

floor division

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See PEP 238.

function

A series of statements which returns some value to a caller. It can also be passed zero or more *argument* which may be used in the execution of the body. See also *parameter*, *method*, and the *Function definitions* section.

function annotation

An arbitrary metadata value associated with a function parameter or return value. Its syntax is explained in section *Function definitions*. Annotations may be accessed via the `__annotations__` special attribute of a function object.

Python itself does not assign any particular meaning to function annotations. They are intended to be interpreted by third-party libraries or tools. See PEP 3107, which describes some of their potential uses.

__future__

A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it becomes the default:

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

garbage collection

The process of freeing memory when it is not used anymore. Pyhon performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

generator

A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks-up where it left-off (in contrast to functions which start fresh on every invocation).

generator expression

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

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

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the `functools.singledispatch()` decorator, and PEP 443.

GIL

See *global interpreter lock*.

global interpreter lock

The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a “free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hashable

An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE

An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path

A list of locations (or *path entries* that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importer

An object that both finds and loads a module; both a *finder* and *loader* object.

importing

The process by which Python code in one module is made available to Python code in another module.

interactive

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

interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file object*, and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in *Iterator Types*.

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the *Sorting HOW TO* for examples of how to create and use key functions.

keyword argument

See *argument*.

lambda

An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [arguments]: expression`

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list

A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements are O(1).

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = [':#04x'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See PEP 302 for details and `importlib.abc.Loader` for an *abstract base class*.

mapping

A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` *abstract base classes*. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder

A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from *path entry finder*.

metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in *Customizing class creation*.

method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order.

module

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*

See also *package*.

module spec

A namespace containing the import-related information used to load a module.

MRO

See *method resolution order*.

mutable

Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple

Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time()`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package

A PEP 420 *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

See also *module*.

nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class

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

object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

package

A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also *regular package* and *namespace package*

parameter

A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. **abs()**).
- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare * in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

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

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *****, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with ******, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on *the difference between arguments and parameters*, the **inspect.Parameter()** class, the *Function definitions* section, and PEP 362.

path entry
: A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder
: A *finder* returned by a callable on **sys.path_hooks** (i.e. a *path entry hook* which knows how to locate modules given a *path entry*.

path entry hook
: A callable on the **sys.path_hook** list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
: One of the default *meta path finder* which searches an *import path* for modules.

portion
: A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in PEP 420.

positional argument
: See *argument*.

provisional API
: A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

  Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

  This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See PEP 411 for more details.

provisional package
: See *provisional API*.

Python 3000
: Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic
: An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a **for** statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name

A dotted name showing the "path" from a module's global scope to a class, function or method defined in that module, as defined in PEP 3155. For top-level functions and classes, the qualified name is the same as the object's name:

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

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. **email.mime.text**:

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

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The **sys** module defines a **getrefcount()** function that programmers can call to return the reference count for a particular object.

regular package

A traditional *package* such as a directory containing an **__init__.py** file.

See also *namespace package*

__slots__

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

An *package* which supports efficient element access using integer indices via the **__getitem__()** special method and defines a **__len__()** method that returns the length of the sequence. Some built-in sequence types are **list**, **str**, **tuple**, and **bytes**. Note that **dict** also supports **__getitem__()** and **__len__()**, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

The **collections.abc.Sequence** abstract base class defines a much richer interface that goes beyond just **__getitem__()** and **__len__()**, adding **count()**, **index()**, **__contains__()**, and **__reversed__()**. Types that implement this expanded interface can be registered explicitly using **register()**.

single dispatch

A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice

An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, **[]** with colons between numbers when several are given, such as in **variable_name[1:3:5]**. The bracket (subscript) notation uses **slice** objects internally.

special method

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in *Special method names*.

statement

A statement is part of a suite (a "block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

struct sequence

A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can either be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

text encoding

A codec which encodes Unicode strings to bytes.

text file

A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically.

> **See also:** A *binary file* reads and writes `bytes` objects.

triple-quoted string

A string which is bound by three instances of either a quotation mark (`'`) or an apostrophe (`"`). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type

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

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`.See PEP 278 and PEP 3116, as well as `bytes.splitlines()`) for an additional use.

view

The objects returned from `dict.keys()`, `dict.values()` and `dict.items()` are called dictionary views. They are lazy sequences that will see changes in the underlying dictionary. To force the dictionary view to become a full list use `list(dictview)`. See *Dictionary view objects*.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also *pyvenv - Creating virtual environments*

virtual machine

A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "`import this`" at the interactive prompt.

Annexe B

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation. Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the *Reporting Bugs* page for information on how to do so. New volunteers are always welcome ! Many thanks go to :

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content ;
- the Docutils project for creating reStructuredText and the Docutils suite ;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors. It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You !

Annexe C

HISTORY AND LICENSE

C.1 History of the software

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

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

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/)). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

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

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

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

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 3.4.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.4.3 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 3.4.3 alone or in any derivative

version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2015 Python Software Foundation ; All Rights Reserved" are retained in Python 3.4.3 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 3.4.3 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 3.4.3.
4. PSF is making Python 3.4.3 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 3.4.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.4.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.4.3, 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 3.4.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http ://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives ; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes) : "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle) : 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL : http ://hdl.handle.net/1895.22/1013."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 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.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 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.

Annexe D

COPYRIGHT