Programmation Python

Introduction

F ALIN

d'après https ://docs.python.org

François ALIN

SUPPORT DE COURS

FORMATION PYTHON LYCÉE DIADÈME

Janvier 2023

Table des matières

I Les Bases

1 Pourquoi Python 9

2 Mode d'emploi de l'interpréteur Python 11
2.1 Lancement de l'interpréteur 11
2.2 Mode interactif 11
2.3 Encodage du code source 12

3 Introduction informelle à Python 13
3.1 Utilisation de Python comme une calculatrice 13
3.1.1 Les nombres 13
3.1.2 Chaînes de caractères 15
3.1.3 Listes 20
3.2 Premiers pas vers la programmation 22

4 Outils de contrôle de flux 25
4.1 L'instruction if 25
4.2 L'instruction `for` 26
4.3 La fonction `range()` 26
4.4 Les instructions `break`, `continue` et les clauses `else` au sein des boucles 28
4.5 L'instruction `pass` 29
4.6 L'instruction match 29
4.7 Définir des fonctions 32

4.8 Davantage sur la définition des fonctions 34
4.8.1 Valeur par défaut des arguments . . . 34
4.8.2 Les arguments nommés . . . 35
4.8.3 Paramètres spéciaux . . . 37
4.8.4 Listes d'arguments arbitraires . . . 40
4.8.5 Séparation des listes d'arguments . . . 41
4.8.6 Fonctions anonymes . . . 41
4.8.7 Chaînes de documentation . . . 42
4.8.8 Annotations de fonctions . . . 43
4.9 Aparté : le style de codage 43

5 Structures de données . . . 45
5.1 Compléments sur les listes 45
5.1.1 Utilisation des listes comme des piles . . . 46
5.1.2 Utilisation des listes comme des files . . . 47
5.1.3 Listes en compréhension . . . 47
5.1.4 Listes en compréhensions imbriquées . . . 49
5.2 L'instruction del 50
5.3 n-uplets et séquences 51
5.4 Ensembles 52
5.5 Dictionnaires 53
5.6 Techniques de boucles 54
5.7 Plus d'informations sur les conditions 56
5.8 Comparer des séquences avec d'autres types 57

II Python avancé

6 Modules . . . 61
6.1 Les modules en détail 62
6.1.1 Exécuter des modules comme des scripts . . . 64
6.2 Les dossiers de recherche de modules 64
6.2.1 Fichiers Python « compilés » . . . 65
6.3 Modules standards 65
6.4 La fonction `dir()` 66
6.5 Les paquets 67
6.5.1 Importer * depuis un paquet . . . 69
6.5.2 Références internes dans un paquet . . . 70
6.5.3 Paquets dans plusieurs dossiers . . . 70

7 Les entrées/sorties . . . 71
7.1 Formatage de données 71
7.1.1 Les chaines de caractères formatées (f-strings) . . . 72
7.1.2 La méthode de chaine de caractères `format()` . . . 73
7.1.3 Formatage de chaînes à la main . . . 75
7.1.4 Anciennes méthodes de formatage de chaînes . . . 76

7.2 Lecture et écriture de fichiers 76
7.2.1 Méthodes des objets fichiers 77
7.2.2 Sauvegarde de données structurées avec le module json 79

8 Erreurs et exceptions 81
8.1 Les erreurs de syntaxe 81
8.2 Exceptions 81
8.3 Gestion des exceptions 82
8.4 Déclencher des exceptions 85
8.5 Chaînage d'exceptions 86
8.6 Exceptions définies par l'utilisateur 87
8.7 Définition d'actions de nettoyage 87
8.8 Actions de nettoyage prédéfinies 89
8.9 Levée et gestion de multiples exceptions non corrélées 90
8.10 Enrichissement des exceptions avec des notes 91
8.11 Theory : Unit testing 94
8.11.1 What is Unit Testing ? 94
8.11.2 Steps to test a unit 94
8.11.3 Manual Vs Automated 95
8.11.4 Benefits 95
8.11.5 Example : Add two numbers 95
8.11.6 Conclusion 96

Les Bases

1 Pourquoi Python 9

2 Mode d'emploi de l'interpréteur Python 11
2.1 Lancement de l'interpréteur
2.2 Mode interactif
2.3 Encodage du code source

3 Introduction informelle à Python 13
3.1 Utilisation de Python comme une calculatrice
3.2 Premiers pas vers la programmation

4 Outils de contrôle de flux 25
4.1 L'instruction if
4.2 L'instruction `for`
4.3 La fonction `range()`
4.4 Les instructions `break`, `continue` et les clauses `else` au sein des boucles
4.5 L'instruction `pass`
4.6 L'instruction match
4.7 Définir des fonctions
4.8 Davantage sur la définition des fonctions
4.9 Aparté : le style de codage

5 Structures de données 45
5.1 Compléments sur les listes
5.2 L'instruction del
5.3 n-uplets et séquences
5.4 Ensembles
5.5 Dictionnaires
5.6 Techniques de boucles
5.7 Plus d'informations sur les conditions
5.8 Comparer des séquences avec d'autres types

1. Pourquoi Python

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

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

Python est le langage parfait pour vous.

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

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

Python vous permet de découper votre programme en modules qui peuvent être réutilisés dans d'autres programmes Python. Il est fourni avec une grande variété de modules standards que vous

pouvez utiliser comme base de vos programmes, ou comme exemples pour apprendre à programmer. Certains de ces modules donnent accès aux entrées/sorties, aux appels système, aux connecteurs réseaux et même aux outils comme Tk pour créer des interfaces graphiques.

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

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

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

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

2. Mode d'emploi de l'interpréteur Python

2.1 Lancement de l'interpréteur

L'un des atouts de python c'est qu'il ne nécessite pas d'environnement de developpment complexe pour être utilisé. Un simple terminal de commande peut faire l'affaire. Suivant l'environnement sur lequel vous travaillez, ouvrez un terminal de commande. Pour vous assurer que python est correctement installé, et connaitre sa version il vous suffit de taper à l'invite du shell la commande `python -V`

```
(base) francoisalin@MacBook-Pro-de-Francois-2 ~ % python -V
Python 3.9.12
(base) francoisalin@MacBook-Pro-de-Francois-2 ~ %
```

En retour, si Python est correctement installé, vous obtiendrez le numéro de la version installée.

2.2 Mode interactif

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

```
$ python
Python 3.9.12 (main, Apr  5 2022, 01:53:17)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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

2.3 Encodage du code source

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

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

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

où encoding est un des codecs géré par Python.

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

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

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

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

3. Introduction informelle à Python

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

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

Quelques exemples :

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

3.1 Utilisation de Python comme une calculatrice

Essayons quelques commandes Python simples. Démarrez l'interpréteur et attendez l'invite primaire, >>>.

3.1.1 Les nombres

L'interpréteur agit comme une simple calculatrice : vous pouvez lui entrer une expression et il vous affiche la valeur. La syntaxe des expressions est simple : les opérateurs +, -, * et / fonctionnent

comme dans la plupart des langages (par exemple, Pascal ou C) ; les parenthèses peuvent être utilisées pour faire des regroupements. Par exemple :

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

Les nombres entiers (comme 2, 4, 20) sont de type int, alors que les décimaux (comme 5.0, 1.6) sont de type float.

L'opérateur division (/) retourne toujours un float. Pour effectuer une division entière on utilise l'opérateur //. Pour calculer le reste de la division entière on dispose de l'opérateur modulo %.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.2 Chaînes de caractères

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

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

En mode interactif, l'interpréteur affiche les chaînes de caractères entre guillemets. Les guillemets et autres caractères spéciaux sont protégés avec des barres obliques inverses (backslash en anglais). Bien que cela puisse être affiché différemment de ce qui a été entré (les guillemets peuvent changer), les deux formats sont équivalents. La chaîne est affichée entre guillemets si elle contient un guillemet simple et aucun guillemet, sinon elle est affichée entre guillemets simples. La fonction `print()` affiche les chaînes de manière plus lisible, en retirant les guillemets et en affichant les caractères spéciaux qui étaient protégés par une barre oblique inverse :

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

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

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

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

L'exemple suivant :

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

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

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

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

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

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

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

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

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

Pour concatener plusieurs chaines de caractères, il suffit de les placer entre parenthèses. Cela ne fonctionne cependant qu'avec les chaînes littérales, pas avec les variables ni les expressions :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Par exemple :

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

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

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

Utiliser un indice trop grand produit une erreur :

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

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

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

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

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

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

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

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

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

3.1.3 Listes

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

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

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

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

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

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

Les listes gèrent aussi les opérations comme les concaténations :

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

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

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

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

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

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

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

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

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

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

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

3.2 Premiers pas vers la programmation

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

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

Cet exemple introduit plusieurs nouvelles fonctionnalités.

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

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

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

vous venez de saisir la dernière ligne). Notez bien que toutes les lignes à l'intérieur d'un bloc doivent être indentées au même niveau.

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

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

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

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

4. Outils de contrôle de flux

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

4.1 L'instruction if

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

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

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

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

4.2 L'instruction `for`

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

Par exemple :

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

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

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

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

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

4.3 La fonction `range()`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>>
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

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

Lorsqu'elle utilisée dans une boucle, la clause `else` est donc plus proche de celle associée à une instruction `try` que de celle associée à une instruction `if` : la clause else d'une instruction `try` s'exécute lorsqu'aucune exception n'est déclenchée, et celle d'une boucle lorsqu'aucun `break` n'intervient.

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

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

4.5 L'instruction pass

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

Par exemple :

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

On utilise couramment cette instruction pour créer des classes minimales :

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

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

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

4.6 L'instruction match

La commande match prend une expression en paramètre et compare sa valeur à différentes valeurs fournies dans un plusieurs blocs. Son fonctionnement est semblable au `switch case` du C, du java ou du javaScript. Par contre en Python seul le premier cas valide est exécuté.

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

```
1  def http_error(status):
2          match status:
3                  case 400:
4                  return "Bad request"
5          case 404:
6                  return "Not found"
7          case 418:
8                  return "I'm a teapot"
9          case _:
10                 return "Something's wrong with the internet"
```

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

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

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

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

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

Observez bien cet exemple ! Le premier filtre contient simplement deux littéraux. C'est une sorte d'extension des filtres littéraux. Mais les deux filtres suivants mélangent un littéral et un nom de variable. Si un tel filtre réussit, il provoque l'affectation à la variable. Le quatrième filtre est constitué de deux variables, ce qui le fait beaucoup ressembler à l'affectation multiple (x, y) = point.

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

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

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

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

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

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

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

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

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

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

Voici d'autres caractéristiques importantes de cette instruction :

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

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

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

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

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

Ce filtre, lorsqu'il est comparé à une séquence de deux points, réussit et capture le second dans la variable p_2 ;

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

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

```
from enum import Enum

class Color(Enum):
        RED = 'red'
        GREEN = 'green'
        BLUE = 'blue'

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

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

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

4.7 Définir des fonctions

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

```
>>>
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

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

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

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

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

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

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

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

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

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

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

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

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

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

4.8 Davantage sur la définition des fonctions

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

4.8.1 Valeur par défaut des arguments

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

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

Cette fonction peut être appelée de plusieurs façons :

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

— en fournissant une partie des arguments facultatifs : `ask_ok('OK to overwrite the file?', 2);`
— en fournissant tous les arguments : `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or`

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

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

```
i = 5

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

i = 6
f()
```

affiche 5.

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

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

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

affiche :

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

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

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

4.8.2 Les arguments nommés

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

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

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

```
parrot(1000)                                          # 1 positional
    argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword
    arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword
    arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional
    arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1
    keyword
```

mais tous les appels qui suivent sont incorrects :

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

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

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

Quand un dernier paramètre formel est présent sous la forme `**name`, il reçoit un dictionnaire (voir Les types de correspondances — dict) contenant tous les arguments nommés à l'exception de ceux correspondant à un paramètre formel. Ceci peut être combiné à un paramètre formel sous la forme `*name` (décrit dans la section suivante) qui lui reçoit un n-uplet contenant les arguments positionnés au-delà de la liste des paramètres formels (*name doit être présent avant `**name`).

Par exemple, si vous définissez une fonction comme ceci :

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

Elle pourrait être appelée comme ceci :

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

et, bien sûr, elle affiche :

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

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

4.8.3 Paramètres spéciaux

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

Voici à quoi ressemble une définition de fonction :

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

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

Les arguments positionnels-ou-nommés

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

Paramètres positionnels uniquement

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

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

Arguments nommés uniquement

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

Exemples de fonctions

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

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

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

```
>>>
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
La deuxième fonction pos_only_arg restreint le passage aux seuls arguments
 ↪  par position car il y a un / dans la définition de fonction :

>>>
>>> pos_only_arg(1)
1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Récapitulatif

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

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

Quelques conseils :

— utilisez les paramètres positionnels si vous voulez que le nom des paramètres soit masqué à l'utilisateur. Ceci est utile lorsque les noms de paramètres n'ont pas de signification réelle, si vous voulez faire respecter l'ordre des arguments lorsque la fonction est appelée ou si vous avez besoin de prendre certains paramètres positionnels et mots-clés arbitraires ;

— utilisez les paramètres nommés lorsque les noms ont un sens et que la définition de la fonction est plus compréhensible avec des noms explicites ou si vous voulez empêcher les utilisateurs de se fier à la position de l'argument qui est passé ; dans le cas d'une API, utilisez les paramètres seulement positionnels pour éviter de casser l'API si le nom du paramètre est modifié dans l'avenir.

4.8.4 Listes d'arguments arbitraires

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

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

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

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

4.8.5 Séparation des listes d'arguments

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

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

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

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

4.8.6 Fonctions anonymes

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

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

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

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

4.8.7 Chaînes de documentation

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

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

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

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

Voici un exemple de chaîne de documentation multi-lignes :

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

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

4.8.8 Annotations de fonctions

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

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

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

4.9 Aparté : le style de codage

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

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

— utilisez des indentations de 4 espaces et pas de tabulation.

— 4 espaces constituent un bon compromis entre une indentation courte (qui permet une profondeur d'imbrication plus importante) et une longue (qui rend le code plus facile à lire). Les tabulations introduisent de la confusion et doivent être proscrites autant que possible ;

— faites en sorte que les lignes ne dépassent pas 79 caractères, au besoin en insérant des retours à la ligne.

Vous facilitez ainsi la lecture pour les utilisateurs qui n'ont qu'un petit écran et, pour les autres, cela leur permet de visualiser plusieurs fichiers côte à côte ;

— utilisez des lignes vides pour séparer les fonctions et les classes, ou pour scinder de gros blocs de code à l'intérieur de fonctions ;
— lorsque c'est possible, placez les commentaires sur leurs propres lignes ;
— utilisez les chaînes de documentation ;
— utilisez des espaces autour des opérateurs et après les virgules, mais pas juste à l'intérieur des parenthèses : a = f(1, 2) + g(3, 4) ;
— nommez toujours vos classes et fonctions de la même manière ; la convention est d'utiliser une notation UpperCamelCase pour les classes, et minuscules_avec_trait_bas pour les fonctions et méthodes. Utilisez toujours self comme nom du premier argument des méthodes (voyez Une première approche des classes pour en savoir plus sur les classes et les méthodes) ;
— n'utilisez pas d'encodage exotique dès lors que votre code est censé être utilisé dans des environnements internationaux. Par défaut, Python travaille en UTF-8. Pour couvrir tous les cas, préférez le simple ASCII ;
— de la même manière, n'utilisez que des caractères ASCII pour vos noms de variables s'il est envisageable qu'une personne parlant une autre langue lise ou doive modifier votre code.

5. Structures de données

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

5.1 Compléments sur les listes

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

`list.append(x)`

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

`list.extend(iterable)`

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

`list.insert(i, x)`

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

`list.remove(x)`

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

`list.pop([i])`

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

`list.clear()`

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

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

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

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

`list.count(x)`

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

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

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

`list.reverse()`

Inverse l'ordre des éléments dans la liste.

`list.copy()`

Renvoie une copie superficielle de la liste. Équivalent à a[:]. L'exemple suivant utilise la plupart des méthodes des listes :

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

You might have noticed that methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. 1 This is a design principle for all mutable data structures in Python.

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

5.1.1 Utilisation des listes comme des piles

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

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

5.1.2 Utilisation des listes comme des files

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

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

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

5.1.3 Listes en compréhension

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

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

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

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

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

ou, de manière équivalente :

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

ce qui est plus court et lisible.

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

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

et c'est équivalent à :

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

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

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

```
>>>
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1
[x, x**2 for x in range(6)]
^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

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

5.1.4 Listes en compréhensions imbriquées

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

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

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

Cette compréhension de liste transpose les lignes et les colonnes :

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

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

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

lequel est lui-même équivalent à :

```
>>>
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

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

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

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

5.2 L'instruction del

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

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

del peut aussi être utilisée pour supprimer des variables :

```
>>>
>>> del a
```

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

5.3 n-uplets et séquences

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

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

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

Comme vous pouvez le voir, les n-uplets sont toujours affichés entre parenthèses, de façon à ce que des n-uplets imbriqués soient interprétés correctement ; ils peuvent être saisis avec ou sans parenthèses, même si celles-ci sont souvent nécessaires (notamment lorsqu'un n-uplet fait partie

d'une expression plus longue). Il n'est pas possible d'affecter de valeur à un élément d'un n-uplet ; par contre, il est en revanche possible de créer des n-uplets contenant des objets muables, comme des listes.

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

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

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

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

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

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

5.4 Ensembles

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

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

Voici une brève démonstration :

```
>>>
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been
 ↪  removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

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

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

5.5 Dictionnaires

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

Le plus simple est de voir un dictionnaire comme un ensemble de paires clé-valeur au sein duquel les clés doivent être uniques. Une paire d'accolades crée un dictionnaire vide : . Placer une liste de paires clé-valeur séparées par des virgules à l'intérieur des accolades ajoute les valeurs correspondantes au dictionnaire ; c'est également de cette façon que les dictionnaires sont affichés.

Les opérations classiques sur un dictionnaire consistent à stocker une valeur pour une clé et à extraire la valeur correspondant à une clé. Il est également possible de supprimer une paire clé-valeur avec del. Si vous stockez une valeur pour une clé qui est déjà utilisée, l'ancienne valeur

associée à cette clé est perdue. Si vous tentez d'extraire une valeur associée à une clé qui n'existe pas, une exception est levée.

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

Voici un petit exemple utilisant un dictionnaire :

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

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

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

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

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

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

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

5.6 Techniques de boucles

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.7 Plus d'informations sur les conditions

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

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

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

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

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

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

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

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

5.8 Comparer des séquences avec d'autres types

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

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

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

II Python avancé

6 Modules 61
6.1 Les modules en détail
6.2 Les dossiers de recherche de modules
6.3 Modules standards
6.4 La fonction `dir()`
6.5 Les paquets

7 Les entrées/sorties 71
7.1 Formatage de données
7.2 Lecture et écriture de fichiers

8 Erreurs et exceptions 81
8.1 Les erreurs de syntaxe
8.2 Exceptions
8.3 Gestion des exceptions
8.4 Déclencher des exceptions
8.5 Chaînage d'exceptions
8.6 Exceptions définies par l'utilisateur
8.7 Définition d'actions de nettoyage
8.8 Actions de nettoyage prédéfinies
8.9 Levée et gestion de multiples exceptions non corrélées
8.10 Enrichissement des exceptions avec des notes
8.11 Theory : Unit testing

6. Modules

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

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

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

```
# Fibonacci numbers module

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

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

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

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

This does not add the names of the functions defined in fibo directly to the current namespace (see Portées et espaces de nommage en Python for more details) ; it only adds the module name fibo there. Using the module name you can access the functions :

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

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

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

6.1 Les modules en détail

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

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand,

if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, modname.itemname.

Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the import statement that imports names from a module directly into the importing module's namespace. For example :

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

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, fibo is not defined).

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

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

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

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

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

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

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

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

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

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

6.1.1 Exécuter des modules comme des scripts

Lorsque vous exécutez un module Python avec :

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

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

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

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

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

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

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

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

6.2 Les dossiers de recherche de modules

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

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

Vous trouverez plus de détails dans The initialization of the sys.path module search path.

> **NOTE:**
> Note sur les systèmes qui gèrent les liens symboliques, le dossier contenant le script courant est résolu après avoir suivi le lien symbolique du script. Autrement dit, le dossier contenant le lien symbolique n'est pas ajouté aux dossiers de recherche de modules. Après leur initialisation, les programmes Python peuvent modifier leur sys.path. Le dossier contenant le script courant est placé au début de la liste des dossiers à rechercher, avant les dossiers de bibliothèques. Cela signifie qu'un module dans ce dossier, ayant le même nom qu'un module, sera chargé à sa place. C'est une erreur typique, à moins que ce ne soit voulu. Voir Modules standards pour plus d'informations.

6.2.1 Fichiers Python « compilés »

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

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

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

Astuces pour les experts :

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

6.3 Modules standards

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

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

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

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

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

6.4 La fonction `dir()`

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

```
>>>
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__',
↪  '__spec__',
'__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
'_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
'_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
'callstats', 'copyright', 'displayhook', 'dont_write_bytecode',
↪  'exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
'float_repr_style', 'get_asyncgen_hooks',
↪  'get_coroutine_origin_tracking_depth',
'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
↪  'pycache_prefix',
'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth',
↪  'setdlopenflags',
'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace',
↪  'stderr',
'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version',
↪  'version_info',
'warnoptions']
```

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

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

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

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

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

6.5 Les paquets

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

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

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

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

Les fichiers `__init__.py` sont nécessaires pour que Python considère un dossier contenant ce fichier comme un paquet. Cela évite que des dossiers ayant des noms courants comme string ne masquent des modules qui auraient été trouvés plus tard dans la recherche des dossiers. Dans le plus simple des cas, `__init__.py` peut être vide, mais il peut aussi exécuter du code d'initialisation pour son paquet ou configurer la variable `__all__` (documentée plus loin).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5.1 Importer * depuis un paquet

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

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

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

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

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

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

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

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

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

6.5.2 Références internes dans un paquet

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

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

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

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

6.5.3 Paquets dans plusieurs dossiers

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

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

7. Les entrées/sorties

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

7.1 Formatage de données

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

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

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

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

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

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

Enfin, vous pouvez construire des concaténations de tranches de chaînes vous-même, et ainsi créer n'importe quel agencement. Le type des chaînes a des méthodes utiles pour aligner des chaînes dans une largeur de taille fixe. Lorsque qu'un affichage basique suffit, pour afficher simplement une variable pour en inspecter le contenu, vous pouvez convertir n'importe quelle valeur en chaîne de caractères en utilisant la fonction `repr()` ou la fonction `str()`.

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

Quelques exemples :

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

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

7.1.1 Les chaines de caractères formatées (f-strings)

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

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

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

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

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

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

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

The = specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression :

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

See self-documenting expressions for more information on the = specifier. For a reference on these format specifications, see the reference guide for the Mini-langage de spécification de format.

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

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

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

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

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

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

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

Les arguments positionnés et nommés peuvent être combinés arbitrairement :

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

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

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

This could also be done by passing the table dictionary as keyword arguments with the ** notation.

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

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

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes :

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

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

7.1.3 Formatage de chaînes à la main

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

```
>>>
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
1   1    1
2   4    8
3   9   27
4  16   64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
10 100 1000
```

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

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

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

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

7.1.4 Anciennes méthodes de formatage de chaînes

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

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

Vous trouvez plus d'informations dans la section Formatage de chaines à la printf.

7.2 Lecture et écriture de fichiers

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

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

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

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

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

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

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

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

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

Avertissement Appeler `f.write()` sans utiliser le mot clé with ni appeler `f.close()` pourrait mener à une situation où les arguments de `f.write()` ne seraient pas complètement écrits sur le disque, même si le programme se termine avec succès. Après la fermeture du fichier, que ce soit via une instruction with ou en appelant `f.close()`, toute tentative d'utilisation de l'objet fichier échoue systématiquement.

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

7.2.1 Méthodes des objets fichiers

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

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

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

`f.readline()` lit une seule ligne du fichier ; un caractère de fin de ligne (\n) est laissé à la fin de la chaîne. Il n'est omis que sur la dernière ligne du fichier si celui-ci ne se termine pas un caractère de fin de ligne. Ceci permet de rendre la valeur de retour non ambigüe : si f.readline() renvoie une chaîne vide, c'est que la fin du fichier a été atteinte, alors qu'une ligne vide est représentée par '\n' (une chaîne de caractères ne contenant qu'une fin de ligne).

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

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

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

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

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

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

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

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

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

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

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

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

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

7.2.2 Sauvegarde de données structurées avec le module json

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

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called json can take Python data hierarchies, and convert them to string representations ; this process is called serializing. Reconstructing the data from the string representation is called deserializing. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

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

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

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

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

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

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

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

NOTE:
Note Les fichiers JSON doivent être encodés en UTF-8. Utilisez encoding="utf-8" lorsque vous ouvrez un fichier JSON en tant que fichier texte, que ce soit en lecture ou en écriture. Cette méthode de sérialisation peut sérialiser des listes et des dictionnaires. Mais sérialiser d'autres types de données requiert un peu plus de travail. La documentation du module json explique comment faire.

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

8. Erreurs et exceptions

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

8.1 Les erreurs de syntaxe

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

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

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

8.2 Exceptions

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

```
>>>
>>> 10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

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

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

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

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

8.3 Gestion des exceptions

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

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

L'instruction `try` fonctionne comme ceci :

— premièrement, la clause `try` (instruction(s) placée(s) entre les mots-clés `try` et `except`) est exécutée.

— si aucune exception n'intervient, la clause except est sautée et l'exécution de l'instruction `try` est terminée.

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

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

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

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

```
class B(Exception):
        pass

class C(B):
        pass

class D(C):
        pass

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

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

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

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

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

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

BaseException est la classe mère de toutes les exceptions. Une de ses sous-classes, Exception, est la classe mère de toutes les exceptions non fatales. Les exceptions qui ne sont pas des sous-classes de Exception ne sont normalement pas gérées, car elles sont utilisées pour indiquer que le programme doit se terminer. C'est le cas de SystemExit qui est levée par `sys.exit()` et `KeyboardInterrupt` qui est levée quand l'utilisateur souhaite interrompre le programme.

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

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

```
1  import sys
2
3  try:
4          f = open('myfile.txt')
5          s = f.readline()
6          i = int(s.strip())
7  except OSError as err:
8          print("OS error:", err)
9  except ValueError:
10         print("Could not convert data to an integer.")
11 except Exception as err:
12         print(f"Unexpected {err=}, {type(err)=}")
13         raise
```

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

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

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

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

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

8.4 Déclencher des exceptions

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

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

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

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

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

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

8.5 Chaînage d'exceptions

If an unhandled exception occurs inside an except section, it will have the exception being handled attached to it and included in the error message :

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

During handling of the above exception, another exception occurred:

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

To indicate that an exception is a direct consequence of another, the raise statement allows an optional `from` clause :

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

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

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

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

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

It also allows disabling automatic exception chaining using the from None idiom :

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

Pour plus d'informations sur les mécanismes de chaînage, voir Exceptions natives.

8.6 Exceptions définies par l'utilisateur

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

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

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

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

8.7 Définition d'actions de nettoyage

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

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

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

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

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

Un exemple plus compliqué :

```
>>>
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

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

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

8.8 Actions de nettoyage prédéfinies

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

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

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

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

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

8.9 Levée et gestion de multiples exceptions non corrélées

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

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

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

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

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

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

```
>>>
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10 Enrichissement des exceptions avec des notes

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

après l'exception.

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

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

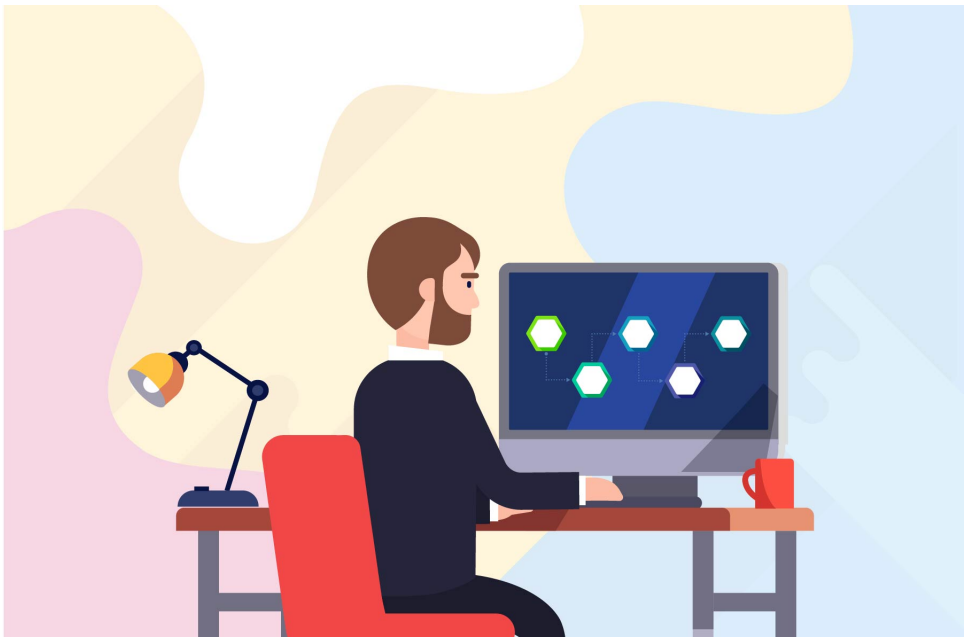
```
>>>
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
+ Exception Group Traceback (most recent call last):
|   File "<stdin>", line 1, in <module>
| ExceptionGroup: We have some problems (3 sub-exceptions)
+-+---------------- 1 ----------------
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|   File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 1
+---------------- 2 ----------------
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|   File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 2
+---------------- 3 ----------------
| Traceback (most recent call last):
|   File "<stdin>", line 3, in <module>
|   File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 3
+------------------------------------
>>>
```

8.11 Theory : Unit testing

.

8.11.1 What is Unit Testing ?

There are different types of testing in software engineering, all serving different purposes. We will consider one that is performed by developers : unit testing. This is probably the most popular methodology : it implies testing the behavior of a unit of an application. A unit is a portion of code that does exactly one task, so it's the smallest testable part of an application.



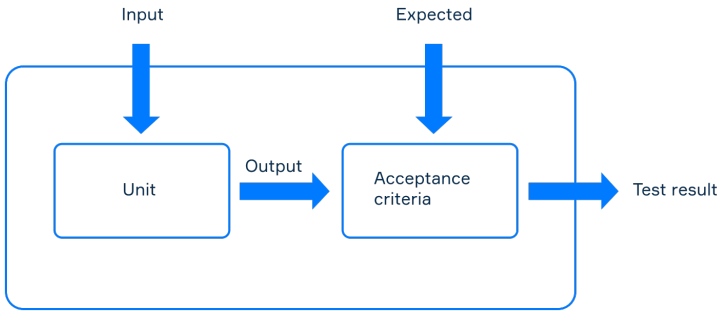
Unit testing is usually done by the author of the tested unit to confirm that the unit does its work well. In object-oriented programming, a unit can be a class or a function. In procedural programming, a unit is a function or a procedure.

8.11.2 Steps to test a unit

The main idea of unit testing is to isolate a specific unit and check that it works properly.

Generally, each unit behaves in a similar way : it consumes input data, performs some action on it and produces output data. We can use this behavior to verify the code. The test goes through the following stages :

— We create two datasets : the input and the expected output ;
— We define acceptance criteria : some conditions determining if the unit works as expected (usually, the acceptance criterion is a comparison of the actual output with the expected one) ;
— We pass the created input dataset to the tested unit ;
— The input invokes code of the tested unit ;
— The code produces an output ;
— The produced output is checked by the acceptance criteria, which compare the actual output with the expected one ;
— Acceptance criteria return the result : pass or fail....



Note : there is no step called "we thoroughly examine the code".

> **NOTE:**
> In unit testing, we know nothing about what the unit actually does with input arguments, but we know exactly what result is expected for each set of input arguments.

8.11.3 Manual Vs Automated

All these steps can be carried out manually or be automated. It is in fact very uncommon to opt for manual unit testing, and there are a couple of sound reasons for that. First, testing is a really repetitive process : every time you make a small change in your code, you have to retest it. Doing this manually would be daunting. Second, unit testing can be easily automated, so there is simply no reason to do it manually.

Automated test cases are reusable. Suppose you add a new feature to your program ; you can execute existing test cases to check whether the new feature has affected the application.

8.11.4 Benefits

First, unit testing allows developers to find bugs at the early stages of development – as you hopefully remember, unit testing is done during the development process, unlike many other testing types. The sooner you find a bug, the less effort you spend on fixing it. It means we can save some time and money !

Secondly, unit testing protects your code from further incorrect changes. Like a butterfly may cause a tornado, even the smallest changes can drastically affect the behavior of your application. In that case, unit testing is a part of regression testing, where we re-run tests to ensure that code still works as expected after it has been changed.

Finally, unit testing makes the integration process easier. The correctness of the whole program depends on each unit and the interaction between them. Since correctness of individual units is approved by unit tests, developers can focus on building interaction between them.

8.11.5 Example : Add two numbers

Suppose we're developing a calculator that has several basic functions : add, subtract, multiply and divide. The application can be divided into 4 units according to these functions. Let's try to apply unit testing to the add function which takes two parameters, x and y.

```
x = 2
y = 2
expected = 4

output = add(x, y)

if expected == output:
print("Passed")
else:
print("Failed")...
```

The pseudocode consists of three parts. First of all, we initialize the test data. Then, we invoke our test subject : the add function. In the end, we compare the actual result with the expected result. If the add function returns 4, your test case passed ; otherwise, it failed. Easy as that !

As you can see, unit testing does not depend on the implementation of the unit. There is no need to look under the hood unless our unit tests fail.

8.11.6 Conclusion

We have covered quite a major topic – unit testing. It is a kind of testing that checks the behavior of the smallest testable pieces of code. Unit testing considers a unit as a black box ; it only checks the result of a unit without internal details. It can be automated easily, so it is often used to confirm that the unit still works properly after code has been changed.

Unit testing is an essential part of developer's daily work. Even though it is quite time-consuming, it protects your code from bugs and saves you a lot of energy at the end of the day.