Richard Gomez

Le petit Python orienté objet

Programmation orientée objet avec Python 3



ellipses

Références sciences

Le petit Python orienté objet

Programmation orientée objet avec Python 3

Richard Gomez

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-064065
©Ellipses Édition Marketing S.A., 2022
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

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 constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

À Pepe Belinchón Cantarero (mi bisabuelo), Martina Martínez Roldán (mi bisabuela), Pilar Belinchón Martínez (mi abuela), Pedro Sáez Martínez (mi abuelo), Brígida Sáez (mi madre), Juliana Sáez (mi tía), Manolo Sáez (mi tío), Pedrito Sáez (mi tío), Federico Gómez Rosell (mi abuelo), Antonio Gómez Minaya (mi padre), David Gomez (mon frère), Didier Calas, Farid Belkalem, Ivan Revel, Stéphane Jammes, Stéphane Baseilhac, Ludovic Jacquet, les Reeds, Alexandre Trijoulet, Ysé Commenges, Yanis Riguet (mon fils) et Estelle Chaumont.

À la mémoire de Rufino Jimenez Rosell, Bautista Villanueva Saez, Cédric Darolles et Nicolas Guilloux.

Prologue

À propos de Python

Python est un langage de programmation permettant de faire de la programmation impérative, fonctionnelle et orientée objet. Il fut inventé en 1989 par Guido Van Rossum et publié en 1991. Il s'agit d'un langage sous licence libre que l'on peut télécharger gratuitement sur le site officiel `www.python.org`, et installer sur n'importe quel ordinateur.



Figure 1. Guido van Rossum, créateur du langage Python

Python est un des langages les plus utilisés au monde et son succès ne cesse d'augmenter. Le classement mondial RedMonk le place en deuxième position derrière Javascript, à égalité avec Java (on consultera [10], par exemple).

Un programme écrit en Python n'est pas aussi rapide qu'un programme écrit en `C++`, mais la simplicité de sa syntaxe permet de concevoir facilement et rapidement des projets de grande envergure. De nombreux développeurs écrivent une première version de leurs logiciels en Python avant de passer à un langage compilé tel que `C++` ou Java.

Le monde du big data et le monde de l'intelligence artificielle ont massivement adopté Python. Aujourd'hui, Python est sans conteste le meilleur langage pour implémenter de l'intelligence artificielle. Python est en outre très utilisé par les géants du web comme Google, Dropbox et Instagram (Youtube est en partie conçu avec Python). Il a également été adopté par le milieu universitaire, les étudiants, la plupart des scientifiques, surtout les mathématiciens, et par de nombreux systèmes éducatifs comme l'enseignement secondaire français, par exemple. Aujourd'hui, les lycéens français écrivent dès la classe de seconde des programmes en Python pour faire des mathématiques et de la physique.

Les qualités de Python sont nombreuses et séduisantes :

- syntaxe élégante, claire et simple ;
- facile à apprendre ;
- en évolution permanente (on en est à la version 3.9.5) ;
- des centaines de milliers de bibliothèques enrichissant le langage dans tous les thèmes imaginables.

Autre qualité importante : le langage Python permet d'entrer de manière simple et naturelle dans le monde de la programmation orientée objet. On entend parfois dire que la POO est complexe et abstraite, Python nous prouve le contraire.

Adopter ce langage est d'autant plus facile que la communauté Python est très impliquée dans l'open source (partage libre et ouvert), l'entraide et la diffusion des idées et savoirs-faire.

À propos de ce livre

Le présent ouvrage est dédié à la programmation orientée objet avec Python. À l'origine, il était prévu pour contenir deux parties : Python objet d'une part, et Tkinter (*graphical user interface*) d'autre part. Malheureusement, son volume nous a contraint à scinder le contenu en deux livres.

Ce livre est la suite logique du *Petit Python* [6] qui décrit toutes les syntaxes du langage (mots-clé, fonctions internes, aliasing, types, etc.) mais sans entrer dans la programmation objet.

Comme avec le précédent ouvrage, nous avons privilégié :

- une présentation claire, simple et directe ;
- une rigueur mathématique ;
- et la volonté de présenter chaque notion à fond.

Chaque point est illustré par de nombreux exemples (des programmes plus ou moins longs). Chaque chapitre s'achève par un travail d'introspection qui peut paraître difficile au premier abord, mais qui s'avère précieux, surtout quand on est confronté à des programmes qui ne tournent pas comme on voudrait. L'introspection permet en effet de comprendre les choses en profondeur et répond à des questions que l'on finit par se poser quand on rencontre des problèmes. Ces approfondissements rendent nos erreurs plus faciles à débusquer et corriger.

On peut lire ce livre dans l'ordre, comme un cours de programmation, mais on peut aussi l'utiliser comme un aide-mémoire permettant de retrouver la syntaxe de n'importe quelle méthode spéciale ou le fonctionnement interne de n'importe quelle structure de données. On peut également l'utiliser pour approfondir des points particuliers en étudiant les chapitres de son choix dans l'ordre que l'on veut : les références

et les renvois sont nombreux et facilitent une telle démarche. La table des matières est écrite dans cet esprit : on y détaille chaque section, chaque sous-section et chaque sous-sous-section. Un index riche de 534 entrées se trouve à la fin du livre.

Le premier chapitre est une présentation générale du paradigme objet. Le lecteur constatera que contrairement à une idée répandue, ce paradigme n'est pas difficile à appréhender et se révèle au contraire naturel et intuitif.

Le chapitre suivant énumère les mots réservés et les fonctions internes dédiés à la POO. Le livre se poursuit avec une classification des types internes fondamentale pour bien se repérer dans le monde Python (*data model*).

Le chapitre 4 est consacré aux notions d'identité et d'égalité en valeur. C'est l'occasion d'approfondir nos connaissances sur les opérateurs de comparaison « `==` » et « `is` ». Tout le monde connaît la différence entre ces deux opérateurs, mais il y a des subtilités que le lecteur ne soupçonne peut-être pas ici (on y réserve de petites surprises).

Un long chapitre est ensuite consacré à la notion de classe chez Python et en présente toutes les facettes.

Un chapitre important est consacré aux méthodes : méthodes liées aux objets, méthodes statiques et méthodes de classes. Nous verrons que ces notions ne sont pas compliquées, contrairement à ce que laissent croire certaines documentations.

Nous consacrons le chapitre 7 au type `object` et le chapitre 8 au type *function*.

Nous étudions en détail les constructeurs, initialiseurs et destructeurs, ainsi que le mécanisme d'instanciation aux chapitres 9 et 10.

Nous décrivons de manière précise au chapitre 11 les règles suivies par Python pour accéder à différents espaces de noms, selon le scope où l'on se trouve.

Nous décrivons méticuleusement au chapitre 12 les méthodes spéciales liées à l'affichage (représentations formelle et informelle, et formatage des chaînes).

Le chapitre 13 donne une présentation approfondie des objets `None`, `Ellipsis` et `NotImplemented`. Ces objets s'utilisent de manière particulière.

Un long chapitre est consacré aux opérations d'arité 2. Il permet de comprendre ce qui se cache derrière un calcul comme « `x + y` », par exemple. On apprend dans ce chapitre à surcharger les opérateurs selon notre bon vouloir.

Une longue série de chapitres étudient les méthodes spéciales ainsi que les méthodes typiques associées à des structures de données fondamentales telles que les objets subscriptables, les séquences, les itérables, les conteneurs, les tables de hachage, les ensembles, les tables de correspondance, les objets appelables, les objets algébriques, etc.

Les chapitres consacrés aux objets appelables (23) et aux objets algébriques (24) sont illustrés par des programmes implémentant des objets mathématiques comme les fonctions réelles d'une ou plusieurs variables réelles, les variables aléatoires à support fini, dénombrable ou compact, les variables aléatoires continues, les quaternions, les vecteurs, les groupes de permutations, etc.

Le chapitre sur les tables de hachage est un petit livre en soi. Ses 65 pages se divisent en 6 sections. On commence par montrer l'intérêt d'une telle structure en comparant ses performances à celles d'une structure naïve comparable. On décrit précisément le système d'adressage et on l'illustre avec des exemples traités à la main, de manière élémentaire. On montre par un calcul de probabilité à quel point les collisions sont impossibles à éviter dans une table, quel que soit le hachage utilisé. On décrit ensuite les tables de hachage de type interne. On étudie les propriétés que devrait avoir une bonne fonction de hachage. On étudie le problème posé par les objets mutables de type interne (Python interdit l'insertion d'un tel objet dans une table de hachage de type interne). On énonce les 5 commandements à suivre quand on crée des types d'objets qu'on insère dans des tables de hachage. Enfin, on étudie la fonction interne `hash` (c'est une fonction très sophistiquée).

Nous consacrons le chapitre 25 aux gestionnaires de contexte et aux gestionnaires décorateurs.

Un chapitre important est consacré à l'héritage : son fonctionnement, son utilité et son évolution au cours de l'histoire du langage Python. Nous décrivons en détail l'algorithme utilisé par Python dans le calcul de l'héritage (algorithme C3). Le chapitre qui suit est consacré à la fonction `super` et à la notion de coopération entre classes.

Le chapitre 28 est consacré à une petite application : un programme simulant un écosystème très basique.

Ensuite, on étudie les méthodes spéciales permettant d'accéder aux attributs en lecture, écriture et effacement.

Les chapitres 30 et 31 sont dédiés aux descripteurs et aux propriétés (les objets de type `property`). La notion de descripteur permet de comprendre, par exemple, comment Python implémente les méthodes.

Le chapitre 32 est consacré aux métaclasses. Une métaclasse est une classe de classes.

Le chapitre 33 étudie les classes abstraites, notamment les modules `abc`, `collections.abc` et `numbers`.

Le chapitre 34 énumère toutes les méthodes spéciales du langage Python (appelées parfois méthodes magiques). Il permet de retrouver rapidement le nom et la syntaxe de chacune de ces méthodes.

Le chapitre 35 est consacré aux fils d'exécution (*threading*), et le 36 aux réseaux. On y montre comment concevoir un serveur capable de faire communiquer plusieurs clients simultanément grâce aux classes `socket` et `Thread` des modules `socket` et `threading`.

Le chapitre 37 est une annexe consacrée aux listes chaînées. C'est l'occasion d'expliquer de manière simple ce que sont une variable et un pointeur. On constatera là aussi que contrairement à ce que l'on croit, les pointeurs sont faciles à comprendre. On montre la différence entre un langage à variables et pointeurs et Python, qui est un langage à espaces de noms, et non un langage à variables. On verra que les listes simplement ou doublement chaînées sont des structures de données assez faciles à comprendre et qu'elles ont apporté une avancée décisive dans la programmation. On profite de cette occasion pour émuler le type `list` sans utiliser la moindre séquence.

Le chapitre 38 est une annexe consacrée à un mode de représentation des entiers relatifs très classique, à savoir le complément à 2. On profite de ce chapitre pour faire quelques rappels sur la base 2.

Le chapitre 39 est une annexe consacrée au module `urllib` (URL *library*). Ce module permet de récupérer des données sur le réseau Internet.

Remerciements

Ce livre est le fruit d'une longue compilation d'informations récoltées et mûries au fil des années, à l'occasion de différents projets de programmation et de différents programmes d'enseignement. Je tiens à remercier chaleureusement mon ami et collègue Jocelyn Etienne pour son soutien, sa finesse, ses remarques et l'aide apportée dans la correction de cet ouvrage. Je veux aussi remercier mon collègue et ami Stéphane Lévy avec lequel j'ai pu échanger régulièrement sur Python. Lui aussi a soutenu ce projet et a contribué à corriger certains chapitres. Je remercie Joris van der Hoeven et Massimiliano Gubinelli de l'équipe TEXMACS pour les améliorations qu'ils ont apportées à ce document (voir par exemple [8]). Je remercie ma compagne Estelle Chaumont pour son soutien et la patience dont elle a fait preuve pendant la rédaction de ce texte. Sans sa bienveillance et son amour, ce livre n'aurait probablement pas existé. Je remercie mes parents, Antoine et Brigitte, pour leur soutien et leur bienveillance, surtout quand je travaille chez eux à Torreilles (Pyrénées Orientales). Je remercie le petit village de Guitera-Les-Bains (Corse) qui m'a offert la paix et la tranquilité nécessaires pour réfléchir aux notions présentées dans cet ouvrage. Je remercie Paul de Laboulaye, Corinne Baud et les éditions Ellipses pour avoir eu la gentillesse de s'être intéressés à ce document et pour avoir accepté de le publier.

Les programmes exposés dans ce livre ont été testés avec Python 3.8.5.

Ce livre a été tapé avec TEXMACS 1.99.16 (éditeur de textes scientifiques, logiciel libre GNU, `www.texmacs.org`).

À Montpellier,

à Villeneuve-Lès-Maguelone,

à Torreilles,

à Guitera-Les-Bains,

2020-2022



Conventions

1. Chaque programme en langage Python est présenté dans un encart gris (police de caractères à chasse fixe).

2. Le nom d'une méthode spéciale commence et termine toujours par deux underscores, comme par exemple chez `__getattr__`, ce qui peut alourdir le texte. Il arrive parfois que nous écrivions tout simplement *getattr* (en italique) au lieu de `__getattr__`. Bien sûr, on ne confondra pas la méthode *getattr* avec la fonction interne `getattr` qui elle s'écrit sans underscores.

3. Quand on renvoie par exemple à la section 5 du chapitre 3 de ce livre, on écrit 3.5. De même, quand on renvoie à la sous-section 7.2 du chapitre 3, on écrit 3.7.2. Dans toute référence de ce type, le premier nombre indique le numéro du chapitre où se trouve la cible.

4. Certains énoncés sont numérotés (définitions, remarques, etc.) La définition 12.3, par exemple, désigne le 3-ème énoncé numéroté du chapitre 12. Dans la référence d'un tel énoncé, le premier nombre indique le numéro du chapitre où se trouve la cible. Même convention avec les figures et les tableaux.

5. Quand on étudie une classe, il arrive parfois, dans un soucis de clarté, que l'on ne présente pas tout son contenu d'emblée. Par exemple, la classe `A` ci-dessous possède deux méthodes nommées `ceci` et `cela` :

```
class A:
    def ceci(self):
        print("Reeds rocks")
    def cela(self):
        print("SpacePopDave")
```

mais on pourrait présenter `A` en commençant par `ceci` :

```
class A:
    def ceci(self):
        print("Reeds rocks")
```

et en terminant par `cela` :

```
def cela(self):
    print("SpacePopDave")
```

en gardant à l'esprit que le script de `cela` est écrit dans celui de `A` avec, bien entendu, une indentation de 4 cadratins (nous avons omis l'indentation dans `cela` dans un souci de lisibilité).

Table des matières

Prologue 5

1 Paradigme orienté objet 33

1.1 Introduction 33
1.2 Objets et méthodes 34
1.3 Notion de classe 37
1.4 Héritage, surcharge et polymorphisme 39
1.4.1 Héritage 39
1.4.2 Surcharge 40
1.4.3 Surcharge des opérateurs usuels 43
1.4.4 Polymorphisme 44
1.5 Agrégation et composition 44
1.6 Méthodes d'instances, méthodes de classes et méthodes statiques 46
1.7 Langage UML 47

2 Mots réservés et fonctions internes orientés objet 51

2.1 Mots réservés orientés objet 51
2.2 Données internes (module `builtins`) 52
2.3 Données internes orientées objet 54

3 Types offerts par Python 55

3.1 Introduction 55
3.2 Catégories I et III 56
3.3 Catégorie II : les types d'exceptions 57
3.4 Catégories IV et V 57
3.5 Interfaces offertes par Python 58
3.6 Hiérarchie officielle des types internes 59
3.7 Introspection 60

4 Objets 63

4.1 Introduction 63
4.2 Valeur 64
4.2.1 Définition 64
4.2.2 Valeur des objets de type interne 65
4.2.2.1 Nombres 65
4.2.2.2 Chaînes 65
4.2.2.3 Booléens 66

4.2.2.4 Conteneurs . . . 66
4.2.2.5 Objets de type `object` . . . 66
4.2.3 Valeur d'une instance de classe . . . 67
4.2.4 Mécanisme derrière « `==` » . . . 68
4.3 Identité . . . 70
4.3.1 Définition . . . 70
4.3.2 Bizarrerie en coulisse . . . 70
4.3.3 Cas des objets mutables . . . 72

5 Classes . . . 75

5.1 Classes chez Python . . . 75
5.2 Créer une classe . . . 75
5.3 Variables d'une classe (attributs) . . . 76
5.3.1 Définition . . . 76
5.3.2 Attributs-données et attributs-méthodes . . . 78
5.3.3 Identificateurs recommandés . . . 79
5.3.4 Dictionnaire `__dict__` . . . 80
5.4 Instancier une classe . . . 81
5.5 Méthodes . . . 82
5.6 Filiation et héritage . . . 83
5.7 Attributs *name* et *qualname* . . . 86
5.8 Fonction `type` . . . 88
5.9 Docstring . . . 89
5.10 Attribut *slots* . . . 90
5.11 Décorer une classe . . . 93
5.12 Introspection . . . 93
5.12.1 Afficher les parents d'une classe . . . 93
5.12.2 Retour au dictionnaire . . . 94
5.12.3 Une bizarrerie avec les attributs privés . . . 95

6 Méthodes . . . 97

6.1 Méthodes d'instances . . . 97
6.2 Méthodes de classes . . . 98
6.3 Méthodes statiques . . . 99
6.4 Cas d'utilisation . . . 100
6.4.1 Généralités . . . 100
6.4.2 Exemples internes . . . 101
6.4.2.1 `float.fromhex` . . . 101
6.4.2.2 `int.from_bytes` . . . 102
6.4.2.3 `str.maketrans` . . . 102
6.4.2.4 `dict.fromkeys` . . . 103
6.4.2.5 `bytearray.fromhex` . . . 103
6.4.2.6 `bytes.fromhex` . . . 104
6.4.3 Cas des fonctions d'un module . . . 104
6.4.4 Exemples avec des matrices . . . 105
6.4.5 Exemples avec des dates . . . 108

6.5 Introspection . . . 110
6.5.1 Explorer *method* . . . 110
6.5.2 Explorer `classmethod` . . . 111
6.5.3 Explorer `staticmethod` . . . 113
6.6 Bilan . . . 115

7 Type `object` . . . 117

7.1 Introduction . . . 117
7.2 Exploration rapide . . . 118
7.3 Attributs basiques . . . 119
7.3.1 Attribut `__name__` . . . 119
7.3.2 Attribut `__qualname__` . . . 119
7.3.3 Attribut `__bases__` . . . 119
7.3.4 Attribut `__dict__` . . . 120
7.4 Méthodes pour instancier . . . 121
7.4.1 Méthode `__new__` . . . 121
7.4.2 Méthode `__init__` . . . 123
7.5 Méthodes pour afficher . . . 125
7.5.1 Méthode `__repr__` . . . 125
7.5.2 Méthode `__str__` . . . 125
7.5.3 Méthode `__format__` . . . 125
7.6 Méthodes pour comparer . . . 126
7.7 Attributs pour introspection . . . 127
7.7.1 Attribut `__class__` . . . 127
7.7.2 Attribut `__doc__` . . . 128
7.7.3 Méthode `__dir__` . . . 128
7.7.4 Méthode `__sizeof__` . . . 129
7.8 Méthodes accesseur, mutateur et effaceur . . . 129
7.9 Attributs liés à l'héritage . . . 130
7.9.1 Méthode `mro` . . . 130
7.9.2 Attribut `__mro__` . . . 130
7.9.3 Méthode `__subclasses__` . . . 131
7.10 Méthode pour hacher . . . 131
7.11 Curiosités . . . 132
7.11.1 Mise en abyme . . . 132
7.11.2 Bound ou unbound ? . . . 133
7.11.3 Retour à l'espace des noms de `object` . . . 135
7.11.4 Retour à `__subclasses__` . . . 136
7.11.5 Retour à `mro` . . . 137
7.11.6 Dernier coup d'œil . . . 138

8 Type *function* . . . 139

8.1 Paramètres . . . 139
8.1.1 Introduction . . . 139
8.1.2 Paramètres de position sans valeur par défaut . . . 140
8.1.3 Paramètres nommés sans valeur par défaut . . . 140

8.1.4 Paramètres positionnels ou nommés sans valeur par défaut 140
8.1.5 Paramètres avec valeur par défaut 140
8.1.6 Paramètre *var-positional* 141
8.1.7 Paramètre *var-keyword* 141
8.1.8 Signatures 142
8.1.9 Valeur par défaut et effet de bord 145
8.2 Arguments 146
8.3 Exemple : émulation de `max` 148
8.4 Expérience 150
8.5 Passage par valeur, variable ou référence ? 150
8.6 Attributs spécifiques 151
8.6.1 `__name__` 152
8.6.2 `__qualname__` 152
8.6.3 `__doc__` 152
8.6.4 `__annotations__` 153
8.6.5 `__module__` 153
8.6.6 `__globals__` 153
8.6.7 `__defaults__` 153
8.6.8 `__kwdefaults__` 154
8.6.9 `__closure__` 154
8.6.10 `__code__` 154
8.7 Instanciation 154
8.8 Ajouter des attributs à une fonction 156
8.9 Fermeture d'une fonction 158
8.9.1 Notion de variable libre 158
8.9.2 Contenu de la fermeture 159
8.9.3 Calcul de la fermeture 160
8.9.4 À quoi sert la fermeture ? 162
8.9.5 Quelques exemples amusants 163
8.9.6 Retour à une ancienne énigme 165
8.9.7 Émuler une classe avec une fonction 166
8.9.8 Conclusion 168
8.10 Annotations 168
8.10.1 Introduction 168
8.10.2 Module `typing` 169
8.10.3 Annotations contraignantes 171
8.10.3.1 Fonction `checktypes` 171
8.10.3.2 Fonction `ensure_annotations` du module `ensure` 172
8.10.3.3 Fonction `validate` 172

9 Constructeurs et initialiseurs 175

9.1 Préliminaire 175
9.2 Mécanisme d'instanciation 175
9.3 Constructeurs 178
9.4 Pourquoi les constructeurs sont des méthodes statiques ? 180
9.5 Initialiseurs 183

9.5.1 Introduction . . . 183
9.5.2 Appel de l'initialiseur . . . 184
9.5.3 Utilisation courante . . . 184
9.5.4 Exemple . . . 185
9.5.5 Dérivation et Coopération . . . 185
9.5.6 Confusion fréquente . . . 188
9.6 Manipulations . . . 188
9.7 Instanciation des types internes . . . 193
9.7.1 Cas des objets mutables . . . 193
9.7.2 Cas des objets non mutables . . . 194
9.7.3 Cas des classes . . . 195
9.8 Cas courants où l'on surcharge le constructeur . . . 196
9.8.1 Dérivation d'un type interne d'objets non mutables . . . 196
9.8.2 Dérivation de la classe `type` . . . 198
9.9 Application à des *design patterns* . . . 200
9.9.1 Singleton . . . 200
9.9.2 Borg . . . 203
9.9.3 Compteur d'instances . . . 206

10 Destructeurs et ramasse-miettes . . . 209

10.1 Mot réservé `del` . . . 209
10.2 Destructeur (*finalizer*) . . . 210
10.3 La commande `del` est-elle utile ? . . . 215
10.4 Exemple : retour au design pattern compteur . . . 216

11 Espaces de noms . . . 219

11.1 Introduction . . . 219
11.2 Opérations sur les variables . . . 220
11.3 Espace des noms internes . . . 221
11.4 Espace des noms globaux . . . 221
11.5 Espace de noms locaux . . . 222
11.5.1 Introduction . . . 222
11.5.2 Variable globale et variable locale pointant vers le même objet . . . 223
11.5.3 Paramètres avec valeur par défaut . . . 224
11.6 Résolution d'un nom dans le corps d'une fonction . . . 225
11.7 Espace des noms associé à une classe . . . 228
11.8 Espace des noms associé à une instance . . . 229
11.9 Vie et mort des variables d'une classe et d'une instance . . . 231
11.10 Résolution d'un nom dans le corps d'une classe . . . 233
11.10.1 Énoncé des règles . . . 233
11.10.2 Exemples . . . 235
11.10.3 Une fonction à l'intérieur d'une classe . . . 236
11.10.4 Curiosités . . . 238
11.11 Outils pour l'introspection . . . 240
11.11.1 Fonction `dir` . . . 240
11.11.2 Attribut `__dict__` . . . 241

11.11.3 Fonction `globals` . . . 241
11.11.4 Fonction `locals` . . . 241
11.11.5 Fonction `vars` . . . 242
11.12 Notion de portée (*scope*) . . . 242
11.13 À propos du tiret bas (*underscore*) . . . 243

12 Méthodes spéciales pour l'affichage . . . 245

12.1 Méthode *repr* . . . 245
12.1.1 Introduction . . . 245
12.1.2 Penser à l'héritage . . . 246
12.1.3 Représentation basée sur l'identité . . . 246
12.1.4 Représentation des objets de type interne . . . 247
12.1.4.1 Cas de `object` . . . 247
12.1.4.2 Cas de `classmethod`, `enumerate`, `filter`, `map`, `property`, `reversed`, `staticmethod` et `zip` . . . 248
12.1.4.3 Cas de *function* . . . 248
12.1.4.4 Cas de `memoryview` . . . 249
12.1.4.5 Cas de `super` . . . 249
12.1.4.6 Cas de `type` . . . 250
12.1.4.7 Cas de *method* . . . 251
12.1.5 Représentation par défaut d'une instance de classe . . . 252
12.1.6 Une surcharge pratique pour débugger . . . 253
12.2 Méthode *str* . . . 254
12.2.1 Introduction . . . 254
12.2.2 Lien entre *str* et *repr* . . . 255
12.2.3 Cas des objets conteneurs de type interne . . . 257
12.3 Méthode *format* . . . 259
12.3.1 Introduction . . . 259
12.3.2 Mécanisme derrière la fonction `format` . . . 260
12.3.3 Mécanisme derrière `str.format` . . . 260
12.3.4 Surcharger *format* . . . 261
12.4 Exemple amusant . . . 262

13 `None`, `Ellipsis` et `NotImplemented` . . . 265

13.1 Objet `None` . . . 265
13.1.1 Introduction . . . 265
13.1.2 Affichage . . . 266
13.1.3 Attributs . . . 266
13.1.4 Retour par défaut . . . 267
13.1.5 Utilisé comme valeur par défaut . . . 267
13.1.6 Utilisé pour désactiver une méthode héritée . . . 268
13.2 Objet `Ellipsis` . . . 268
13.2.1 Introduction . . . 268
13.2.2 Affichage . . . 269
13.2.3 Utilisation . . . 269
13.2.4 Exemple . . . 270

13.3 Objet `NotImplemented` . . . 272
13.3.1 Introduction . . . 272
13.3.2 Usage général . . . 272
13.3.3 Usage dans les opérateurs d'arité 2 . . . 273
13.3.4 Ne pas confondre avec `NotImplementedError` . . . 275

14 Opérateurs d'arité 2 . . . 277

14.1 Notion d'opérateur symétrique . . . 277
14.2 Méthodes spéciales d'opérateurs . . . 278
14.3 Symétrique d'une méthode d'opérateur . . . 278
14.4 Mécanisme derrière les opérateurs arithmétiques . . . 279
14.5 Mécanisme derrière les opérateurs de comparaison . . . 280
14.5.1 Égalité et non égalité . . . 281
14.5.2 Ordre . . . 284
14.6 Opérateurs in situ (*in place*) . . . 286
14.7 Priorités . . . 288
14.8 Expériences . . . 288
14.8.1 Expérience 1 . . . 288
14.8.2 Expérience 2 . . . 289
14.8.3 Expérience 3 . . . 289
14.8.4 Expérience 4 . . . 290
14.8.5 Expérience 5 . . . 291
14.8.6 Expérience 6 . . . 291

15 Objets subscriptables . . . 293

15.1 Définition . . . 293
15.2 Méthode *getitem* . . . 294
15.3 Méthode *missing* . . . 295
15.4 Méthode *setitem* . . . 297
15.5 Méthode *delitem* . . . 297
15.6 Opération « `for k in x` » . . . 298
15.7 Opération « `k in a` » . . . 299
15.8 Méthode spéciale *index* . . . 301
15.9 Exemple : matrices . . . 302

16 Slicing . . . 305

16.1 Rappels . . . 305
16.1.1 Rudiments . . . 305
16.1.2 Syntaxes . . . 305
16.1.3 Robustesse du slicing . . . 306
16.2 Mécanisme derrière le slicing . . . 307
16.3 Objets de type `slice` . . . 309
16.4 Reconditionnement . . . 310
16.5 Méthode `slice.indices` . . . 311
16.6 Annexe : *index* . . . 312

17 Séquences . . . 315

17.1 Définition . . . 315
17.2 Les séquences sont itérables . . . 316
17.3 Les séquences supportent l'opération « `x in a` » . . . 316
17.4 Slicing . . . 317
17.4.1 Rappels . . . 317
17.4.2 Exemple consistant . . . 318
17.4.3 Une autre version de cet exemple . . . 322
17.5 Concaténation et répétition . . . 323
17.6 Méthodes spéciales . . . 328
17.6.1 Tour d'horizon . . . 328
17.6.2 Méthode *len* . . . 328
17.6.3 Lien entre *len* et `reversed` . . . 329
17.6.4 Méthode *reversed* . . . 331
17.7 Méthodes typiques . . . 332
17.8 Exemple complet : séquences géométriques . . . 335
17.8.1 Introduction . . . 335
17.8.2 Slicing . . . 336
17.8.3 Changer la longueur et la raison . . . 338
17.8.4 Surcharge de *setitem* . . . 339
17.8.5 Surcharge de *delitem* . . . 340
17.8.6 Surcharge de méthodes typiques . . . 342
17.8.6.1 Surcharge de `append` . . . 342
17.8.6.2 Surcharge de `extend` . . . 343
17.8.6.3 Surcharge de `copy` . . . 344
17.8.6.4 Surcharge de `index` . . . 344
17.8.6.5 Surcharge de `reverse` . . . 345
17.8.6.6 Surcharge de `pop` . . . 345
17.8.7 Surcharge de *add* et *iadd* . . . 346
17.9 Exemple complet par dérivation . . . 347
17.9.1 Introduction . . . 347
17.9.2 Slicing . . . 349
17.9.3 Changer la longueur et la raison . . . 349
17.9.4 Surcharge de *setitem* . . . 350
17.9.5 Surcharge de *delitem* . . . 350
17.9.6 Surcharge de méthodes typiques . . . 352
17.9.6.1 Méthodes à éliminer . . . 353
17.9.6.2 Surcharge de `append` . . . 353
17.9.6.3 Surcharge de `extend` . . . 353
17.9.6.4 Surcharge de `copy` . . . 354
17.9.6.5 Pas besoin de surcharger `index` . . . 354
17.9.6.6 Pas besoin de surcharger `reverse` . . . 355
17.9.6.7 Surcharge de `pop` . . . 355
17.9.7 Surcharge de *add* et *iadd* . . . 356

18 Itérables . . . 359

18.1 Définition . . . 359
18.2 Mécanisme derrière « `for k in x` » . . . 359
18.3 Cas particulier d'un objet muni de *getitem* . . . 360
18.4 Fonction `iter` . . . 363
18.5 Fonction `next` . . . 364
18.6 Surcharger *iter* et *next* : usage courant . . . 365
18.7 Itérateur versus pseudo-itérateur . . . 367
18.8 Exemples complets . . . 369
18.8.1 Suites de Syracuse . . . 369
18.8.2 Itérer sur un entier . . . 370
18.9 Émulation . . . 371
18.10 Décompression . . . 373
18.11 Opérateur splat . . . 373
18.12 Introspection . . . 374
18.13 Compléments sur `iter` et `next` . . . 375
18.13.1 Appel de `iter` avec une fonction et une sentinelle . . . 375
18.13.2 Appel de `next` avec valeur `default` . . . 377
18.14 Expériences . . . 377
18.15 En résumé . . . 378

19 Conteneurs . . . 381

19.1 Définition . . . 381
19.2 Mécanisme derrière « `k in x` » . . . 381
19.3 Conteneurs de type interne . . . 382
19.4 Surcharge de *contains* : exemples simples . . . 383
19.4.1 Conteneur universel . . . 383
19.4.2 Conteneur très exotique . . . 383
19.4.3 Idéaux de $\mathbb{Z}$ . . . 384
19.5 Cas particulier des itérables infinis . . . 385
19.6 Émulation . . . 386
19.7 Exemples . . . 387
19.7.1 Retour à Syracuse . . . 387
19.7.2 Divisibilité chez les entiers . . . 388
19.7.3 Intervalles de $\mathbb{R}$ . . . 390

20 Tables de hachage . . . 393

20.1 Introduction . . . 393
20.1.1 Définition . . . 393
20.1.2 Implémentation naïve des ensembles finis . . . 394
20.1.3 Ensemble naïf versus `set` . . . 395
20.1.4 Implémentation naïve des dictionnaires . . . 398
20.2 Hachage-adressage quand l'univers est fini . . . 399
20.2.1 Fonctionnement . . . 399
20.2.2 Émulation . . . 401
20.3 Hachage-adressage quand l'univers est infini . . . 403
20.3.1 Principe de base . . . 403

20.3.2 Traitement des collisions . . . 404
20.3.3 Exemple à la main . . . 405
20.3.4 Émulation . . . 406
20.3.5 Défauts . . . 410
20.3.5.1 Nécessité d'une bonne fonction de hachage . . . 410
20.3.5.2 Modélisation . . . 410
20.3.5.3 Paradoxe des anniversaires . . . 411
20.3.5.4 Clusters . . . 415
20.3.6 Variantes . . . 415
20.4 Tables de hachage de type interne . . . 417
20.4.1 Taille dynamique . . . 417
20.4.2 Collections d'objets ou de valeurs ? . . . 418
20.4.3 Objets hachables . . . 420
20.4.3.1 Généralités . . . 420
20.4.3.2 Récursivité . . . 421
20.4.3.3 Introspection . . . 421
20.4.4 Propriétés requises chez *hash* . . . 423
20.4.4.1 Être entier . . . 423
20.4.4.2 Être constant dans le temps . . . 423
20.4.4.3 Être invariant modulo *eq* . . . 424
20.4.4.4 Expérience de pensée : un hachage non invariant modulo *eq* . . . 424
20.4.4.5 Suivre une distribution uniforme . . . 427
20.4.5 Problème avec les mutables de type interne . . . 427
20.4.6 Classe-identités versus classe-valeurs . . . 429
20.4.7 Problème avec les classes-identités . . . 430
20.4.8 Problème avec les classes-valeurs . . . 432
20.4.9 Les 5 commandements . . . 436
20.5 Expériences . . . 436
20.5.1 Expérience 1 . . . 436
20.5.2 Expérience 2 . . . 437
20.5.3 Expérience 3 . . . 437
20.5.4 Expérience 4 . . . 438
20.5.5 Expérience 5 . . . 439
20.5.6 Expérience 6 . . . 439
20.5.7 Expérience 7 . . . 440
20.6 Fonction interne `hash` . . . 441
20.6.1 Évolution de `hash` au fil du temps . . . 441
20.6.2 Python interdit qu'un hachage donne -1 . . . 443
20.6.3 Définition de `hash(n)` pour `n` entier . . . 444
20.6.4 Émulation du *hash* de `int` . . . 445
20.6.5 Résidus et sommes digitales . . . 445
20.6.6 Émulation du *hash* de `int` avec des sommes digitales . . . 446
20.6.7 Compléments sur les opérateurs logiques (base 2) . . . 447
20.6.7.1 Répunit . . . 447
20.6.7.2 Opération AND . . . 447
20.6.7.3 Opération OR . . . 448

20.6.7.4 Opération XOR 448
20.6.7.5 Décalage logique vers la gauche (*left shift*) 448
20.6.7.6 Décalage logique vers la droite (*right shift*) 449
20.6.8 Émulation du *hash* de `int` avec des opérateurs logiques 449
20.6.9 Une remarque sur le nombre M_{61} 450
20.6.10 Émulation de la fonction interne `hash` 451
20.6.11 Émulation du *hash* de `str` 452
20.6.12 Émulation du *hash* de `tuple` 454
20.6.13 Émulation du *hash* de `Fraction` 454
20.6.14 Émulation du *hash* de `float` 455
20.6.15 Émulation du *hash* de `complexe` 456
20.6.16 Bibliothèque de hachage `hashlib` 456
20.6.17 Expérience 457

21 Ensembles 459

21.1 Introduction 459
21.2 Méthodes typiques 459
21.2.1 Méthodes de base 459
21.2.2 Opérations 460
21.2.3 Tests 461
21.2.4 Mises à jour 461
21.2.5 Autre 462
21.3 Méthodes spéciales 463
21.3.1 Méthodes spéciales de base 463
21.3.2 Opérations 464
21.3.3 Opérations in situ 465
21.3.4 Comparaisons 466
21.4 Exemple complet 467
21.4.1 Méthodes de base 467
21.4.2 Méthodes spéciales de base 468
21.4.3 Opérations typiques 469
21.4.4 Tests typiques 470
21.4.5 Mises à jour 470
21.4.6 Méthodes `pop`, `clear` et `copy` 471
21.4.7 Opérateurs arithmétiques 472
21.4.8 Opérateurs in situ 472
21.4.9 Comparaisons 472
21.5 Exemple avec des ensembles infinis 473

22 Tables de correspondance 477

22.1 Introduction 477
22.1.1 Première présentation 477
22.1.2 Deuxième présentation 478
22.1.3 Troisième présentation 479
22.1.4 Quatrième présentation 479

22.2 Méthodes courantes chez une table de correspondance 479
22.2.1 Introspection dans `dict` 480
22.2.2 Méthodes de base 480
22.2.3 Mutations typiques 481
22.2.3.1 Méthode `update` 481
22.2.3.2 Méthode `clear` 481
22.2.3.3 Méthodes `pop` et `popitem` 482
22.2.4 Longueur et itération 482
22.2.5 Méthodes `items`, `keys` et `values` 482
22.2.6 Méthodes `fromkeys` 483
22.2.7 Méthode `copy` 483
22.3 Exemple complet 484

23 Objets appelables 491

23.1 Définition 491
23.2 Mécanisme derrière un appel 491
23.3 Cas typiques 493
23.3.1 Fonctions et méthodes 493
23.3.2 Décorateurs gestionnaires de contexte 493
23.3.3 Classes (émulation de *call* dans `type`) 493
23.4 Émuler les fonctions numériques 494
23.4.1 Fonctions réelles d'une variable réelle 494
23.4.2 Fonctions réelles de plusieurs variables réelles 497
23.5 Émuler les variables aléatoires 499
23.5.1 Support fini 499
23.5.2 Support dénombrable 502
23.5.3 Support compact 503
23.5.4 Support non compact 505
23.5.4.1 Inversion de la fonction de répartition 505
23.5.4.2 Cas de la loi normale 506

24 Objets algébriques 509

24.1 Introduction 509
24.2 Type `bool` 509
24.3 Types `int` (et `bool`) 511
24.3.1 Attributs typiques 511
24.3.2 Méthodes typiques 511
24.3.3 Méthode spéciale *index* 512
24.3.4 Opérateurs typiques d'arité 2 512
24.3.5 Opérateurs typiques d'arité 1 513
24.3.6 Opérateurs bit à bit d'arité 2 513
24.3.7 Opérateur bit à bit d'arité 1 513
24.3.8 Opérateurs de comparaison 513
24.3.9 Méthodes spéciales pour fonctions internes 513
24.3.10 Méthodes spéciales pour fonctions du module `math` 514
24.3.11 Implémentation 514

24.4 Type `float` . . . 514
24.4.1 Attributs typiques . . . 514
24.4.2 Méthodes typiques . . . 514
24.4.3 Opérateurs typiques d'arité 2 . . . 514
24.4.4 Opérateurs typiques d'arité 1 . . . 515
24.4.5 Opérateurs de comparaison . . . 515
24.4.6 Méthodes spéciales pour fonctions internes . . . 515
24.4.7 Méthode spéciale pour la fonction `trunc` du module `math` . . . 515
24.4.8 Instances particulières . . . 515
24.4.9 Implémentation . . . 516
24.5 Type `complex` . . . 516
24.5.1 Instanciation . . . 516
24.5.2 Méthode typique . . . 517
24.5.3 Opérateurs typiques d'arité 2 . . . 517
24.5.4 Opérateurs typiques d'arité 1 . . . 517
24.5.5 Opérateurs de comparaison . . . 517
24.5.6 Méthodes spéciales pour fonctions internes . . . 517
24.6 Type `decimal.Decimal` . . . 517
24.6.1 Intérêt du type `Decimal` . . . 517
24.6.2 Notion de contexte . . . 519
24.6.3 Signaux, `flags` et `traps` . . . 520
24.6.4 Méthodes d'un objet de type `Context` . . . 521
24.6.5 Méthodes d'un objet de type `Decimal` . . . 521
24.6.6 Instances particulières . . . 521
24.7 Type `fractions.Fraction` . . . 522
24.7.1 Introduction . . . 522
24.7.2 Attributs et méthodes . . . 522
24.8 Classe abstraite `Number` . . . 523
24.9 Priorités entre opérateurs . . . 524
24.10 Classe algébrique en général . . . 524
24.11 Dériver un type numérique . . . 526
24.12 Exemple : quaternions . . . 527
24.13 Exemple : permutations . . . 533
24.14 Exemple : vecteurs . . . 537
24.15 Exemple : fonctions et variables aléatoires . . . 541
24.15.1 Retour aux fonctions réelles d'une variable réelle . . . 541
24.15.2 Retour aux variables aléatoires réelles . . . 541

25 Gestionnaires de contexte . . . 545

25.1 Introduction . . . 545
25.2 Signature de *enter* et *exit* . . . 546
25.3 Gestionnaire customisé . . . 548
25.4 Gestionnaire-décorateur . . . 549

26 Héritage . . . 553

26.1 Taxonomie et notion d'héritage . . . 553

26.2 Ordre de résolution d'un attribut (MRO) 556
26.2.1 Linéarisation 556
26.2.2 Position du problème 557
26.2.3 Ordre local 558
26.2.4 Depth first 558
26.2.5 From left to right, depth first : LRDF 559
26.2.6 Monotonie 560
26.2.7 Algorithme LRDF amélioré : KEEP LAST LRDF 561
26.2.8 Exemple fâcheux 562
26.2.9 Solution adoptée par Python 3 563
26.3 Algorithme C3 564
26.4 Superclasse 569
26.4.1 Ne pas confondre les superclasses avec la superclasse 569
26.4.2 Fonction `super` 570
26.5 Un peu d'histoire 571
26.5.1 Python 2.1 571
26.5.2 Python 2.2 : arrivée des classes modernes 572
26.5.3 Python 2.3 : adoption de l'algorithme C3 575

27 Fonction `super` 577

27.1 Notion de superclasse 577
27.2 Fonction `super` 578
27.3 Utilisation élémentaire 580
27.4 Utilisation un peu moins élémentaire 582
27.5 Classes coopératives 583
27.5.1 Introduction 583
27.5.2 Signature robuste 586
27.5.3 Stopper la chaîne des délégations 587
27.6 Adaptateur 588
27.7 Exemple complet 589
27.8 Introspection 594
27.9 Émulation 598

28 Projet : dans la savane 601

28.1 Objectif 601
28.2 Population globale 601
28.3 Classe origine 602
28.4 Êtres vivants, faune et flore 603
28.5 Programme principal 606
28.6 Enrichissements 607

29 Méthodes spéciales derrière les accesseurs 609

29.1 Introduction 609
29.2 Fonctions `getattr`, `setattr`, `delattr` et `hasattr` 610
29.3 Méthode *getattribute* 611

29.3.1 Introduction . . . 611
29.3.2 Exemple de surcharge . . . 612
29.3.3 Attention aux appels récursifs . . . 613
29.3.4 Accesseur de `object` . . . 613
29.3.5 Émulation de l'accesseur de `object` . . . 617
29.4 Méthode *getattr* . . . 618
29.5 Méthode *setattr* . . . 619
29.5.1 Fonctionnement . . . 619
29.5.2 Émulation du mutateur de `object` . . . 621
29.6 Méthode *delattr* . . . 623
29.6.1 Fonctionnement . . . 623
29.6.2 Émulation de l'effaceur de `object` . . . 624
29.7 Surcharger *dir* . . . 625

30 Descripteurs . . . 627

30.1 Introduction . . . 627
30.1.1 Définition . . . 627
30.1.2 Méthode *get* . . . 628
30.1.3 Méthode *set* . . . 630
30.1.4 Méthode *delete* . . . 631
30.1.5 Méthode *set name* . . . 631
30.1.6 Protocole descripteur . . . 632
30.2 Cas des fonctions et méthodes . . . 632
30.2.1 Fonctions . . . 632
30.2.2 Méthodes d'instances . . . 634
30.2.3 Méthodes statiques . . . 636
30.2.4 Méthodes de classes . . . 637
30.3 Cas d'un *member descriptor* (*slots*) . . . 639
30.4 Cas d'un *getset descriptor* . . . 640
30.5 Utilisations élémentaires . . . 643
30.5.1 Première expérience . . . 643
30.5.2 Deuxième expérience . . . 644
30.5.3 Troisième expérience . . . 645
30.6 Utilisation de *set name* . . . 646
30.7 Validateurs . . . 647
30.8 Propriétés . . . 648
30.9 Émulation totale . . . 649
30.10 Retour au problème de *bound versus unbound* . . . 658

31 Type `property` . . . 661

31.1 Introduction . . . 661
31.2 Application simpliste . . . 663
31.3 Méthodes `getter`, `setter` et `deleter` . . . 663
31.4 Émulation . . . 665

32 Métaclasses . . . 667

32.1 Définitions . . . 667
32.2 Dériver `type` . . . 668
32.2.1 Usage courant : surcharger *new* . . . 668
32.2.2 Alternative : surcharger *init* . . . 669
32.3 Instancier une métaclasse . . . 669
32.3.1 Appel direct . . . 670
32.3.2 Mot-clé `class` avec paramètre `metaclass` . . . 670
32.4 À propos du paramètre `metaclass` . . . 671
32.4.1 Avec quoi alimenter ce paramètre ? . . . 671
32.4.2 Avec un objet possédant une méthode *call* . . . 671
32.4.3 Avec une fonction . . . 672
32.5 Exemple . . . 673
32.6 Attributs `mro` et *mro* . . . 674
32.7 Émulation du *call* de `type` . . . 674
32.8 Méthode *init subclass* . . . 675
32.8.1 Introduction . . . 675
32.8.2 Signature . . . 676
32.8.3 Exemple amusant . . . 677
32.8.4 Nature . . . 677
32.8.5 Émulation . . . 678
32.9 Méthodes *instancecheck* et *subclasscheck* . . . 679
32.9.1 Méthode spéciale derrière `isinstance` . . . 679
32.9.2 Méthode spéciale derrière `issubclass` . . . 680
32.9.3 Cas typique . . . 681
32.9.4 Introspection . . . 682

33 Classes abstraites (`abc`) . . . 685

33.1 Définition . . . 685
33.2 Métaclasse `ABCMeta` du module `abc` . . . 685
33.3 Classe `ABC` . . . 687
33.4 Décorateur `abstractmethod` . . . 688
33.5 Décorateurs obsolètes . . . 689
33.6 Méthode `register` . . . 690
33.7 Surcharge de *subclasscheck* dans `ABCMeta` . . . 692
33.8 Surcharge de *instancecheck* dans `ABCMeta` . . . 693
33.9 Méthode *subclasshook* . . . 693
33.10 Cas courants . . . 695
33.11 Exemple : classe abstraite des validateurs . . . 699
33.12 Module `collections.abc` . . . 701
33.13 Module `numbers` . . . 703

34 Aide-mémoire méthodes spéciales . . . 705

34.1 Instanciation et destruction . . . 705
34.2 Dérivation . . . 705
34.3 Espaces de noms . . . 705
34.4 Affichage . . . 705

34.5 Comparaisons . . . 706
34.6 Opérateurs arithmétiques . . . 706
34.6.1 Arité 1 . . . 706
34.6.2 Arité 2 . . . 706
34.7 Valeur booléenne . . . 707
34.8 Subscriptables . . . 707
34.9 Conteneurs et séquences . . . 707
34.10 Itérables . . . 707
34.11 Hachables . . . 707
34.12 Ensembles . . . 707
34.13 Tables de correspondance . . . 708
34.14 Appelables . . . 708
34.15 Objets algébriques . . . 708
34.16 Gestionnaires de contexte . . . 708
34.17 Accesseurs . . . 709
34.18 Descripteurs . . . 709
34.19 Introspection . . . 709
34.20 Autres méthodes . . . 709
34.20.1 Méthode *bytes* . . . 709
34.20.2 Méthode *length hint* . . . 710
34.20.3 Méthode pour objets asynchrones . . . 710
34.20.4 Divers . . . 710
34.21 Créer ses propres méthodes spéciales . . . 711

35 Threading . . . 715

35.1 Notion de fil d'exécution . . . 715
35.2 Fils d'exécution simultanés . . . 717
35.3 Dériver la classe `Thread` . . . 718
35.4 Attribut de type `Thread` . . . 721
35.5 Synchroniser des fils avec des instances de `Lock` . . . 721
35.6 Méthode `join` . . . 724
35.7 Fonctions utiles . . . 726
35.8 Introspection . . . 727
35.9 Parallélisme . . . 728

36 Réseaux . . . 731

36.1 Protocoles et ports . . . 731
36.2 Notion de socket . . . 732
36.2.1 Connecteur . . . 732
36.2.2 Famille et genre . . . 733
36.2.3 Signature de `socket` . . . 733
36.2.4 Adresses . . . 734
36.3 Principes de base . . . 734
36.4 Émettre et recevoir . . . 736
36.5 Mise en pratique . . . 737
36.5.1 Manipulations élémentaires . . . 737

36.5.2 Conseils pratiques . . . 741
36.5.3 Connecter deux ordinateurs à travers le réseau Internet . . . 742
36.6 Envoyer une requête à un serveur HTTP . . . 743
36.7 Expérience avec le navigateur . . . 746
36.8 Recommandations officielles . . . 748
36.8.1 Aléas du réseau . . . 748
36.8.2 Messages de longueurs fixes . . . 748
36.8.3 Messages indiquant leurs longueurs . . . 749
36.8.4 Méthode `sendall` . . . 751
36.8.5 Conclusion . . . 751
36.9 Chat rudimentaire (ping-pong) . . . 751
36.9.1 Client . . . 752
36.9.2 Serveur . . . 752
36.10 Les sockets sont des gestionnaires de contexte . . . 753
36.11 Fonction `select` . . . 755
36.12 Messagerie instantanée . . . 757
36.12.1 Position du problème . . . 757
36.12.2 Chat avec `select` et `setblocking` . . . 758
36.12.3 Chat avec `Thread` . . . 761
36.12.4 Chat avec `Thread` 100 % orienté objet . . . 764
36.12.5 Conclusion . . . 768
36.13 Fonctions pour retrouver un IP ou une adresse . . . 768
36.14 Compléments . . . 770
36.14.1 Attributs d'un socket . . . 770
36.14.2 Méthode `fileno` . . . 771
36.14.3 Méthodes `getsockname` et `getpeername` . . . 771
36.14.4 Méthode `sendfile` . . . 772
36.14.5 Méthodes `gettimeout` et `settimeout` . . . 773
36.14.6 Méthode `getblocking` . . . 773
36.14.7 Méthode `shutdown` . . . 773
36.14.8 Erreurs de connexion . . . 775

37 Annexe : listes chaînées . . . 777

37.1 Pointeurs . . . 777
37.2 Notion de liste simplement chaînée . . . 781
37.2.1 Définition . . . 781
37.2.2 Méthodes courantes . . . 782
37.2.3 Implémentation classique (Pascal) . . . 783
37.3 Émulation des listes simplement chaînées . . . 784
37.4 Émulation du type `list` . . . 789
37.5 Notion de liste doublement chaînée . . . 792

38 Annexe : complément à 2 . . . 795

38.1 Préliminaire . . . 795
38.2 Complément à 2 dans le cas de 4 bits . . . 795
38.3 Trouver la valeur représentée . . . 796

38.4 Généralisation à n bits . . . 796
38.5 Arithmétique . . . 797
38.6 Émulation . . . 797
38.7 Involution . . . 799
38.8 Petite astuce . . . 800
38.9 Retour à Python . . . 800
38.10 Opérateur ~ d'inversion . . . 801
38.11 Réciproques de `bin`, `oct` et `hex` . . . 801

39 Annexe : module `urllib` . . . 805

39.1 Présentation du paquet . . . 805
39.2 Notion de URL . . . 805
39.3 Récupérer une ressource sur le web . . . 806
39.4 Requête HTTP GET . . . 806
39.5 Requête HTTP POST . . . 808
39.5.1 Introduction . . . 808
39.5.2 Requête écrite à la main . . . 809
39.5.3 Fonction `Request` . . . 810
39.6 Exemple amusant . . . 811
39.7 Entête . . . 813
39.8 Attributs d'un objet requête . . . 814
39.9 Attributs d'un objet réponse . . . 816
39.10 Gestion des erreurs . . . 817
39.11 Autres modules . . . 819

Bibliographie . . . 821

Index . . . 823

Chapitre 1
Paradigme orienté objet

Nous présentons ici les concepts fondamentaux de la programmation orientée objet. Le reste du livre ne fait, finalement, que montrer comment ces concepts sont intégrés au langage Python.

1.1 Introduction

La programmation orientée objet (POO) est un paradigme de programmation au même titre que la programmation impérative, séquentielle, procédurale, fonctionnelle, récursive, itérative, logique, etc. Ce paradigme pointe le bout son nez en 1962 avec le langage Simula, créé par Ole-Johan Dahl et Kristen Nygaard, mais la POO naît véritablement dans les années 70 avec les langages Flex, puis Smalltalk (1972), créés par Alan Kay (entre autres), lui-même très influencé par Simula.

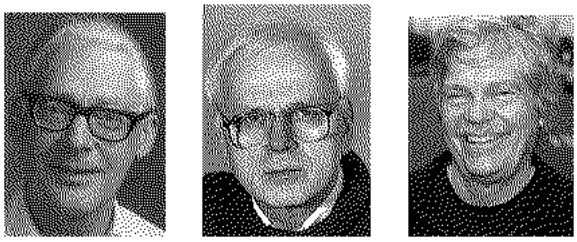


Figure 1.1. O.-J Dahl, K. Nygaard et A. Kay.

Chaque paradigme introduit un ensemble de concepts fondamentaux. Les concepts introduits par la POO sont l'*objet*, la *classe*, l'*encapsulation*, l'*héritage*, le *polymorphisme*, la *surcharge*, le *message*, etc. Il est difficile de donner une description générale, précise et complète de ces concepts, d'une part parce qu'il s'agit de notions complexes qu'il est préférable d'aborder sur des exemples, au cas par cas, et d'autre part, parce qu'il n'existe pas à l'heure actuelle de terminologie universelle propre à la POO.

Ainsi,haque livre consacré à ce thème utilise une terminologie et un jargon qui lui sont propres et qui dépend essentiellement du langage considéré.

Python, comme d'autres langages, supporte plusieurs paradigmes : la programmation impérative structurée, la programmation fonctionnelle et celle qui nous intéresse ici, la programmation orientée objet. Chaque langage objet implémente la POO à sa manière, avec des spécificités propres. Python implémente une POO inspirée de C++ et Modula-3.

1.2 Objets et méthodes

Dans le jargon POO, un *objet* est une structure de données possédant des champs et des méthodes. On peut, dans un premier temps, se représenter un objet comme une fiche en papier bristol rangée dans un fichier. Nous donnons ci-dessous un exemple. Il s'agit d'un objet nommé `rg` représentant un client donné (une personne physique) chez une banque commerciale donnée :

rg	
prénom	"Richard"
nom	"Gomez"
naissance	18 avril 1971
nationalité	France, Espagne
adresse	"19, avenue Georges Guynemer, 81200 Mazamet"
crédit	3200

Tableau 1.1. L'objet `rg` possède 6 champs.

Cet exemple possède 6 champs. Chaque champ possède un nom (un identificateur) : `prénom`, `nom`, `naissance`, `nationalité`, `adresse` et `crédit`. Quand on implémente cet objet avec Python par exemple, l'ordinateur écrit quelque part dans la mémoire l'identificateur `rg` et fait pointer ce dernier vers un autre lieu de la mémoire dans lequel se trouve la structure représentée ci-dessus (tableau 1). On voit que l'indentificateur `prénom` pointe vers la chaîne de caractères `"Richard"`. De même, `nom` pointe vers la chaîne `"Gomez"`, etc. Pour accéder à l'un des champs, on utilise la syntaxe de la *qualification* : si on désire le `nom` de `rg`, on demande `rg.nom` :

```
>>> rg.nom
'Gomez'
>>> rg.nationalité
```

```
['France', 'Espagne']
```

Mais un objet n'est pas qu'une simple fiche. En général, un objet possède des *méthodes*. Une méthode est une fonction (ou une procédure). De manière générale, une méthode peut retourner un résultat, produire un effet, ou modifier des champs. En POO, on applique le principe suivant : l'état d'un objet ne peut être modifié que par une de ses méthodes (même si avec Python on peut déroger à cette règle). Montrons un exemple. On munit `rg` d'une méthode `payer` permettant de mettre à jour son état chaque fois qu'il effectue un paiement. Une fois implémentée cette méthode, `rg` ressemble à ceci :

rg	
prénom	"Richard"
nom	"Gomez"
naissance	18 avril 1971
nationalité	France, Espagne
adresse	"19, avenue Georges Guynemer, 81200 Mazamet"
crédit	3200
payer	*une méthode*

Tableau 1.2. L'objet `rg` possède 6 champs et 1 méthode.

Pour accéder à une méthode, on fait comme avec un champ : si on veut la fonction `payer` de `rg`, on demande `rg.payer`. Pour appeler cette fonction, on fait comme d'habitude : on ajoute des parenthèses. Son utilisation pourrait ressembler à ceci :

```
>>> rg.crédit
3200
>>> rg.payer(200)
>>> rg.crédit
3000
```

Richard Gomez a payé 200 € et son `crédit` est passé de `3200` à `3000`. L'implémentation de cette méthode ressemblerait à ceci :

```
# Pseudo-code
méthode payer(self,somme):
    self.crédit = self.crédit - somme
```

Dans ce code, `rg` est représenté par `self` et on remarque que la signature `(self,somme)` possède 2 paramètres alors qu'en appel on ne fait passer qu'un argument (ce point sera expliqué plus tard). Montrons un autre exemple : une méthode `âge` pour cal-

culer l'âge de la personne représentée par `rg`. Une fois implémentée cette méthode, notre objet ressemble à ceci :

rg	
prénom	"Richard"
nom	"Gomez"
naissance	18 avril 1971
nationalité	France, Espagne
adresse	"19, avenue Georges Guynemer, 81200 Mazamet"
crédit	3000
payer	*une méthode*
âge	*une méthode*

Tableau 1.3. L'objet `rg` possède 6 champs et 2 méthodes.

On utiliserait la fonction `rg.âge` comme ceci :

```
>>> rg.âge()
47
```

L'implémentation ressemblerait à ceci :

```
#Pseudo-code
méthode âge(self):
    année = (on extrait l'année de self.naissance)
    date = (on extrait l'année en cours)
    résultat = date - année
    retourner résultat
```

Remarque 1.1. Les champs traduisent l'état dans lequel se trouve l'objet, tandis que les méthodes décrivent comment évolue l'état. Un objet est à la fois un état et des règles de comportement. C'est ce qu'on appelle l'*encapsulation* des données.

Convention 1.2. Dans ce livre, nous utilisons le mot *attribut* pour désigner les champs et les méthodes d'un objet. Nous utilisons le mot *attribut-donnée* pour désigner les champs. Un objet possède donc des attributs-données, et des attributs-méthodes.

On peut résumer la vie d'un objet comme ceci : l'objet est créé dans un état initial. Il change d'état au cours de l'exécution (plusieurs fois éventuellement). Il interagit avec d'autres objets, ce qui peut provoquer des changements d'état. À un moment donné, l'objet est détruit.

Remarque 1.3. Le mot *objet* du jargon Python n'est pas l'équivalent exact du mot *objet* du jargon POO. Chez Python, tout est objet : les données ordinaires comme les nombres ou les chaînes de caractère sont des objets ; les fonctions sont des objets ; les modules sont des objets ; et même les classes sont des objets.

1.3 Notion de classe

Les objets ne sont pas créés ex nihilo. Chaque objet appartient à une *classe* d'objets. On peut par exemple créer la classe des *nombres complexes*. Ce n'est qu'une fois cette classe implémentée, qu'on peut instancier le nombre $3 + 4$ i, par exemple. Dans le jargon de la POO, on dit que $3 + 4$ i est un objet de la classe des *nombres complexes*. On dit aussi que $3 + 4$ i est une *instance* de la classe des *nombres complexes*. De manière générale, chaque objet est une instance d'une classe donnée. L'usage veut que ce soit dans l'implémentation de la classe que soit définie (déclarée) la structure des futures instances. Plus précisément, c'est dans l'implémentation de la classe qu'on déclare les attributs-données et qu'on implémente les méthodes (chez Python, nous verrons que la déclaration des attributs-données se fait en surchargeant l'initialiseur). Certains auteurs parlent de la classe comme d'un *contrat logiciel*, voir par exemple [1].

Reprenons l'exemple de l'objet `rg` donné à la section précédente. Si on suit le paradigme de la programmation orientée objet, avant de créer `rg` on définit d'abord la classe à laquelle `rg` appartiendra. Notre objet `rg` représente le client d'une banque, nous créons donc une classe nommée `Client`, tout simplement (ensuite, nous pourrons dire que `rg` *est* un `Client`). Cette classe pourrait ressembler à ceci :

`Classe`	`Client`
`prénom`	Chaîne de caractères
`nom`	Chaîne de caractères
`naissance`	Date
`nationalité`	Liste de un ou plusieurs pays
`adresse`	Chaîne de caractères
`crédit`	Nombre
`payer`	*une méthode*
`âge`	*une méthode*

Tableau 1.4. La classe `Client` prévoit 6 champs et implémente 2 méthodes.

Nous pouvons maintenant créer `rg` en instanciant `Client` :

```
# Pseudo-code
rg = (une instance de Client)
```

À ce stade, `rg` ressemble à ceci :

rg	
prénom	
nom	
naissance	
nationalité	
adresse	
crédit	
payer	*méthode* `payer` *de* `Client`
âge	*méthode* `âge` *de* `Client`

Tableau 1.5. Les champs de `rg` n'ont pas été initialisés.

Il est d'usage dans la POO d'implémenter des méthodes *accesseurs* et *mutateurs* pour certains champs (en anglais : *getters* and *setters*). Plus précisément, chaque champ que l'on souhaite visible depuis l'extérieur de la classe (depuis tout lieu du programme situé à l'extérieur du corps de la classe) est muni d'un « `get` ». De même, chaque champ modifiable depuis l'extérieur est muni d'un « `set` ». Pour le champ `prénom`, par exemple, on pourrait implémenter une méthode `prénom_get` qui retourne le contenu du champ, ainsi qu'une méthode `prénom_set` pour remplir ce champ :

```
#Pseudo-code
méthode prénom_get(self):
    retourner self.prénom
méthode prénom_set(self,ch):
    self.prénom = ch
```

Nous n'insisterons pas sur ce point car on peut faire du très bon Python objet sans accesseurs ni mutateurs (nous estimons même que les méthodes `prénom_get` et `prénom_set` ci-dessus sont redondantes).

Nous verrons plus loin que Python permet d'implémenter un *initialiseur* dans chaque classe. Il s'agit d'une méthode qui initialise les champs lors de l'instanciation. Il s'agit alors de faire passer le contenu de chaque champ au moment où l'on crée l'objet. Grâce à un initialiseur rédigé correctement, on pourra créer et « remplir » notre objet `rg` en écrivant ceci :

```
>>> rg = Client("Richard","Gomez",Date(18,4,1971),[France,Espagne],
                "19, avenue Georges Guynemer 81200 Mazamet",3200)
```

Note. Celui qui implémente une classe n'est pas forcément celui qui l'utilise : il arrive souvent qu'on écrive un programme qui utilise des classes qu'on n'a pas créées soi-même. On dit alors qu'on est *client* de la classe utilisée. En principe, le client n'a pas besoin de savoir comment les méthodes sont implémentées. En revanche, il doit

connaître le nom et la signature des méthodes (c'est-à-dire l'*interface* des objets). Si le concepteur de la classe modifie l'implémentation mais pas l'interface, il n'y aura pas d'incidence négative sur le travail du client. C'est un des points forts de la POO : l'encapsulation déjà évoquée. Le paradigme objet permet de travailler localement.

Remarque 1.4. La notion de *classe* chez Python n'est pas tout à fait équivalente à la notion de classe en POO. La différence repose sur un détail. Du point de vue de la POO, `str`, par exemple, est une classe : en effet, on peut instancier des objets de classe `str`, et ces derniers possèdent des méthodes (`count`, `upper`, `split`, etc.) Néanmoins, dans le jargon Python, on préfère dire que `str` est un type (on dit que l'on crée des objets de type `str`) et réserver le mot classe à celles que l'on implémente soi-même. Ceci étant dit, les choses ne sont pas aussi rigides : dans de nombreux documents on parle tantôt de la classe `str`, tantôt du type `str`.

1.4 Héritage, surcharge et polymorphisme

1.4.1 Héritage

Les langages orientés objet permettent d'établir une relation de filiation entre différentes classes, et comme chacun sait, la filiation permet *l'héritage*. Prenons un exemple : nous souhaitons écrire un programme de géométrie dans lequel on utiliserait des classes `Polygone`, `Triangle`, `TriangleRectangle`, `Quadrilatère`, `Parallélogramme`, `Rectangle`, `Carré`, etc. Il est possible alors de définir la classe `Triangle` comme une classe fille de la classe `Polygone` :

```
# Pseudo-code
classe Polygone :
    # code implémentant cette classe
    (...)
classe Triangle fille de Polygone :
    # code implémentant cette classe
    (...)
```

Cette filiation possède la conséquence suivante : la classe `Triangle` hérite automatiquement de tous les attributs de la classe `Polygone`. L'héritage permet ainsi de factoriser du code. On interprète cette relation en disant que la classe `Triangle` est une *spécialisation* de la classe `Polygone`.

Note. On dit que `Triangle` est *fille* de `Polygone` ; `Polygone` est *mère* de `Triangle` ; `Triangle` *dérive* de `Polygone` ; `Polygone` est la *super-classe* de `Triangle`.

Reprenons notre exemple `Polygone` en le munissant d'une méthode `tracer` permettant de représenter graphiquement ses instances à l'écran. On peut implémenter cette méthode de la manière suivante. On suppose qu'un objet de classe `Polygone` possède un attribut `sommets` contenant la liste des coordonnées `(x,y)` de chaque sommet. On écrit alors :

```
# Pseudo-code
classe Polygone :
    méthode tracer(self) :
        n = nombre d'items dans self.sommets
        pour k de 0 à n-1 :
            a = self.sommets[k]
            b = self.sommets[(k+1)%n]
            tracer le segment joignant a et b
    (...)
```

Imaginons maintenant que nous implémentons `Triangle` et `Quadrilatère` comme filles de `Polygone`, `TriangleRectangle` comme fille de `Triangle`, `Parallélogramme` comme fille de `Quadrilatère`, `Rectangle` comme fille de `Parallélogramme`, et `Carré` comme fille de `Rectangle` (on laisse au lecteur le soin de dessiner l'arbre généalogique correspondant à cette *taxonomie*). Comme cela a été dit, il n'y a pas besoin d'implémenter une méthode `tracer` pour chacune de ces classes, puisqu'elles héritent toutes de la classe `Polygone` (on prend soin, bien sûr, de munir les instances d'un attribut `sommets`). On pourrait maintenant faire l'expérience suivante :

```
# Pseudo-code
x = (une instance de Triangle initialisée)
x.tracer()
```

On verrait alors apparaître un beau triangle à l'écran.

1.4.2 Surcharge

Reprenons l'exemple de `Polygone` et ses classes dérivées. Supposons que nous souhaitons implémenter une méthode `aire` retournant l'aire d'un objet. Supposons que nous ne savons pas calculer l'aire d'un polygone en général, mais que nous souhaitons malgré tout pouvoir appeler `aire` sur n'importe quel objet dérivant de `Polygone`. Alors, dans l'implémentation de la classe `Polygone`, nous écrivons ceci :

```
# Pseudo-code
classe Polygone
    méthode aire(self) :
       raise NotImplementedError
```

Ce qui provoquera bien sûr une erreur chaque fois qu'on appellera `aire` sur un `Polygone` :

```
>>> x = (une instance de Polygone initialisée)
>>> afficher x.aire()
NotImplementedError
```

Supposons par ailleurs que nous implémentons des méthodes `longueur` et `largeur` dans les classes `TriangleRectangle` et `Rectangle` qui retournent les dimensions de l'objet. Dans ce cas, nous pouvons munir `TriangleRectangle` et `Rectangle` d'une méthode `aire` satisfaisante :

```
# Pseudo-code
classe TriangleRectangle :
    méthode longueur(self) : (...)
    méthode largeur(self) : (...)
    méthode aire(self) :
        retourner self.longueur() * self.largeur() / 2
```

Pour la classe `Rectangle`, on écrira ceci :

```
# Pseudo-code
classe Rectangle :
    méthode longueur(self) : (...)
    méthode largeur(self) : (...)
    méthode aire(self) :
        retourner self.longueur() * self.largeur()
```

Nous venons de *surcharger* la méthode `aire` chez `TriangleRectangle` et `Rectangle`. Désormais, nous pouvons faire l'expérience ci-dessous :

```
# Pseudo-code
u = (une instance de Rectangle initialisée)
afficher u.aire()
v = (une instance de TriangleRectangle initialisée)
afficher v.aire()
```

sans provoquer d'erreur. Nous pouvons même utiliser `aire` avec une instance de `Carré` : ce dernier dérive de `Rectangle`. En revanche, chaque fois que nous appellons `aire` sur une figure géométrique qui n'est ni un rectangle, ni un triangle rectangle, on obtient un `NotImplementedError`.

On peut se demander comment le langage a géré les conflits surgissant lors de la résolution du nom `u.aire`. En effet, on pourrait croire que l'instance `u` possède le `aire` hérité de `Polygone` et le `aire` implémenté dans `Rectangle`. Il n'en est rien : le fait de définir `aire` dans `Rectangle` a pour effet de faire oublier la fonction `aire` héritée de la super-classe, il n'y a donc pas de conflit. C'est le principe de la *surcharge*.

Remarque 1.5. On rappelle au lecteur souhaitant écrire une adaptation en Python de ce que nous venons de faire qu'il existe de nombreuses formules d'aire pour le triangle. Citons par exemple la formule du déterminant :

$$ABC = \frac{1}{2}\det(\overrightarrow{AB}, \overrightarrow{AC})$$

ou la formule du produit vectoriel :

$$ABC = \frac{1}{2}\|\overrightarrow{AB} \wedge \overrightarrow{AC}\|$$

ou encore la formule de Héron (Héron d'Alexandrie, 1er siècle après J.-C) :

$$ABC = \sqrt{p\,(p-a)\,(p-b)\,(p-c)}$$

où a, b et c sont les longueurs des côtés et p le demi-périmètre.

On peut, comme nous venons de le voir, surcharger une méthode afin de la remplacer, mais on peut également surcharger une méthode afin de l'étendre (la spécialiser, par exemple) :

```
# Pseudo-code
classe Animal :
    méthode parle(self) :
        raise NotImplementedError
classe Chat fille de Animal :
    méthode parle(self) :
        afficher "miaou"
classe Chien fille de Animal :
    méthode parle(self) :
        afficher "waouh"
classe Siamois fille de Chat:
    méthode parle(self) :
        superclasse.parle()
        afficher "fffff"
```

Une fois ces classes implémentées, nous faisons l'expérience ci-dessous :

```
>>> natacha = (une instance de Siamois)
>>> natacha.parle()
miaou
fffff
```

La méthode `parle` a affiché `"miaou"` puis `"fffff"`. En effet, la méthode `parle` de la siamoise `natacha` appelle la méthode `parle` de la super-classe avant de lâcher son `"fffff"`.

Note. Notre classe `Animal` n'a pas vocation à être instanciée. Cette classe sert à centraliser tout ce que les chats, chiens, singes, etc. ont en commun. Il n'y a donc rien d'étonnant à ce qu'une éventuelle instance d'`Animal` ne possède pas de méthode

`parle`. On interprète cette situation en disant qu'une instance de `Animal` n'est pas assez spécialisée pour savoir parler. En effet, comment parle un animal en général ? Dans le jargon de la POO, on dit que `Animal` est une *classe abstraite*. Le module `abc` permet d'implémenter les classes abstraites en nous fournissant la classe `ABC` (*abstract base class*), la métaclasse `ABCMeta` (*abstract class metaclass*) et le décorateur `abstractmethod`, voir chapitre 33.

Note. Dans la plupart des langages orientés objets, on distingue deux notions : la surcharge et la redéfinition (ou spécialisation). En anglais, on parle de *overloading* et *overriding*. Surcharger au sens de « overloading » une méthode `action` dans une classe `A` signifie écrire une deuxième méthode `action` dans `A`, mais avec une signature différente. Chez Python, ceci aurait pour effet d'écraser (remplacer) la première fonction `action`. Ainsi, chez Python, la notion d'overloading n'existe pas à proprement parler. Si on souhaite que la méthode `action` possède plusieurs signatures, il suffit d'utiliser des paramètres avec valeurs par défaut (chapitre 8). On peut donc dire que les paramètres avec valeur par défaut remplacent la notion d'overloading. De manière générale, surcharger au sens de « overriding » une méthode `action` chez une classe `A` signifie qu'on a déjà écrit une méthode `action` dans une classe ancêtre de `A` et qu'on écrit une autre `action` dans `A` elle-même. Cela n'a pas pour effet d'écraser la première `action`. Cela a pour effet de redéfinir (ou spécialiser) la méthode `action` chez les instances de `A`. Nous prenons le parti dans ce livre (comme dans la plupart des textes en français sur Python) de traduire *overriding* par surcharge (et non pas par redéfinition).

1.4.3 Surcharge des opérateurs usuels

Python permet la surcharge des opérateurs classiques, comme par exemple le `+` de l'addition. Voici un exemple. On suppose qu'une instance de `Complexe` représente un nombre complexe et possède deux attributs `réel` et `imag` correspondant à la partie réelle et à la partie imaginaire, respectivement. On suppose avoir surchargé la méthode spéciale permettant l'affichage d'une instance :

```
>>> z = Complexe(5,3)
>>> z
5 + 3 i
```

On surcharge alors l'opérateur `+` comme ceci :

```
# Pseudo-code
classe Complexe :
    redéfinition de +(self,other) :
        re = self.réel + other.réel
        im = self.imag + other.imag
        retourner Complexe(re,im)
```

Le résultat espéré est celui-ci :

```
>>> u = Complexe(5,3)
>>> v = Complexe(2,8)
>>> u + v
10 + 11 i
```

Note. Cet exemple est purement pédagogique. Il n'y a pas besoin d'implémenter une classe pour faire du calcul dans $\mathbb{C}$ puisque Python possède déjà le type `complex`.

Nous retiendrons ceci : la surcharge permet de définir `x + y` pour des objets `x` et `y` appartenant à nos propres classes. Le lecteur savait déjà que les classes `str`, `tuple` et `list` surchargent l'opérateur `+`.

1.4.4 Polymorphisme

La possibilité de pouvoir utiliser le même nom pour des méthodes différentes, comme nous l'avons vu avec `aire` chez `TriangleRectangle` et `aire` chez `Rectangle` (exemple plus haut) est appelée *polymorphisme*. L'utilisateur de ces classes n'a pas besoin de connaître le mode de calcul de l'aire, ni son implémentation. Il a juste besoin de savoir que pour obtenir l'aire de `x`, on appelle `x.aire`. Sur ce même exemple (polygones et dérivés), nous pouvons également citer la méthode `tracer` : lors d'un appel `x.tracer()`, on ne se préoccupe pas de savoir à quelle classe appartient `x` (à condition que celle-ci dérive de `polygone` ou d'une autre classe dans laquelle on aurait pris soin d'implanter là aussi une méthode du même nom). C'est un peu comme avec le bouton ON-OFF des appareils électroniques : nous ne savons pas ce qui se cache derrière, mais nous savons l'utiliser car l'interface est toujours la même : on appuie sur le bouton pour mettre l'appareil en marche, et on rappuie pour l'éteindre. Même chose chez Python quand on écrit `x + y` sans se préoccuper de savoir si `x` et `y` sont des nombres ou des séquences.

Note. De manière générale, dans le monde de la POO, Les notions de overloading et overriding ont beaucoup à voir avec le polymorphisme.

1.5 Agrégation et composition

Un champ d'un objet `x` ne contient pas forcément une donnée simple telle qu'un nombre ou une chaîne, il peut aussi contenir un autre objet (ou plus exactement pointer vers un autre objet). Si un champ de `x` pointe vers un objet `y`, on dit que l'objet `x` contient l'objet `y`.

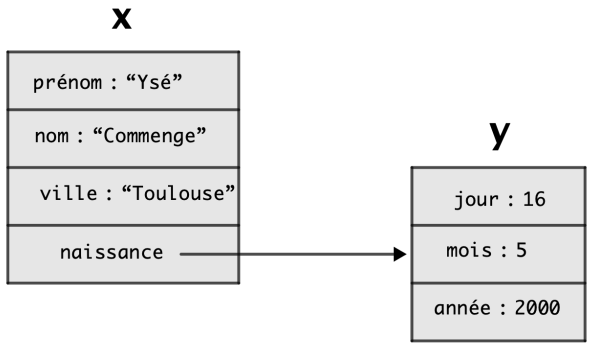


Figure 1.2. L'objet `x` ci-dessus représente une personne. Son champ `naissance` pointe vers `y` qui représente une date.

Si par ailleurs notre programme est écrit de telle sorte que l'objet `y` disparaisse au moment où `x` est effacé, on dit qu'il y a *composition* entre `x` et `y` (le contenu ne survit pas au contenant). Si au contraire, l'objet `y` survit après que `x` soit effacé, on dit qu'il y a *agrégation*. Certains auteurs préfèrent dire *agrégation forte* et *agrégation faible*, au lieu de composition et agrégation. Voici un exemple basé sur l'idée que toute voiture possède un moteur. On suppose que toute instance de `Moteur` possède un numéro de série :

classe Moteur	
série	Un entier

Tableau 1.6. L'attribue `série` est un nombre entier.

Et que toute instance de `Voiture` possède un moteur :

classe Voiture	
moteur	Un Moteur

Tableau 1.7. L'attribut `moteur` est un `Moteur`.

Imaginons que nous avons implanté ces deux classes en langage Python. Si maintenant on crée un `Moteur` `m`, puis une `Voiture` `v` en faisant pointer `v.moteur` vers `m`, nous obtenons une agrégation faible. En effet, si l'objet `v` est détruit, l'objet `m` demeure accessible et Python ne le détruit pas. En revanche, si la création d'une voiture `w` commande la création d'un moteur sans lui donner de nom (autre que le nom qualifié `w.moteur`) on aura une agrégation forte. En effet, la destruction de `w` rendrait `w.moteur` inaccessible et le ramasse-miettes ne tarderait pas à le détruire (c'est ainsi que fonctionne Python : tout objet inaccessible est assez vite détruit, voir chapitre 10). Notons qu'on pourrait implémenter une agrégation symétrique basée sur l'idée que toute voiture possède un moteur et que tout moteur est installé dans une voiture :

classe Moteur	
série	Un entier
voiture	Une Voiture

classe Voiture	
moteur	Un Moteur

Tableau 1.8. Une voiture possède un moteur et un moteur possède une voiture

Tout ceci est un peu théorique pour l'instant. On retiendra néanmoins qu'un attribut peut pointer vers un objet aussi complexe que désiré, et que ce dernier peut avoir un attribut pointant lui-même sur cet objet.

1.6 Méthodes d'instances, méthodes de classes et méthodes statiques

Jusqu'à présent nous n'avons parlé que de *méthodes d'instances*. Chez Python, les méthodes sont implémentées à l'aide de fonctions écrites dans le corps d'une classe :

```
# Pseudo-code
class Polygone:
    def tracer(self,c):
        (...trace self avec la couleur c...)
```

Pour Python, `Polygone.tracer` est une fonction. Instancions `x` de type `Polygone` :

```
>>> x = Polygone(...)
```

Alors `x.tracer` est une méthode de l'instance `x` (la fonction a donné une méthode d'instance). Il faut bien comprendre que `Polygone.tracer` et `x.tracer` sont deux objets distincts. Certes, tous deux sont appelables, mais leurs signatures sont distinctes. La signature du premier est `(self,c)` tandis que celle du second est `(c)`. Si on appelle `x.tracer` en faisant passer deux arguments,

```
>>> x.tracer(1,2)
Error (...)
```

on déclenche un message d'erreur signalant qu'il y a un argument de trop.

Voici ce qui s'est passé : l'appel `x.tracer(1,2)` a déclenché l'appel `Polygone.tracer(x,1,2)`. La règle suivie par Python est simple : l'appel d'une méthode d'instance déclenche l'appel de la fonction correspondante en faisant passer l'instance en premier argument (ici on voit que `x` est envoyé comme premier argument). Par convention, le premier paramètre d'une fonction écrite dans le corps d'une classe est nommé `self` pour bien se rappeler qu'il s'agit de l'instance d'appel.

On peut dire qu'une méthode puise des informations dans l'instance qui appelle et bien sûr, dans les arguments passés en appel.

Mais en programmation objet, il n'y a pas que les méthodes d'instances. Il y a aussi les *méthodes de classes* et les *méthodes statiques*. chez Python,

- une méthodes de classe est une méthode qui est liée à la classe et non à une instance ; ainsi, elle puise les informations dans la classe à laquelle elle est liée et dans les arguments passés en appel ;
- une méthode statique est une méthode qui n'est liée ni à la classe, ni à une instance ; ainsi, elle ne puise les informations que dans les arguments passés en appel.

Chaque langage définit (éventuellement) ces notions de façon plus ou moins différente. Les langages C++ et Java, par exemple, définissent les méthodes statiques comme chez Python. En revanche, ces langages ne connaissent pas les méthodes de classes. Le langage Smalltalk définit les méthodes de classes de manière différente.

Chez Python, les méthode sont implémentées en écrivant des fonctions dans le corps de la classe. Pour obtenir une méthode de classe, on décore cette fonction avec `@classmethod`. Pour obtenir une méthode statique, on décore avec `@staticmethod`. On retiendra ceci :

- une méthode de classe fait passer le type de l'instance en premier argument ;
- une méthode statique ne fait passer ni l'instance, ni son type en argument.

1.7 Langage UML

UML (Unified Modeling Language) est un langage à base de pictogrammes et diagrammes utilisé pour la conception et le développement de systèmes, en particulier de programmes orientés objet. Il s'agit d'un outil incontournable pour les projets de grande envergure impliquant des informaticiens et leurs clients. Dans un projet de programme orienté objet, le langage UML permet de spécifier les classes, leurs liens, les acteurs, les activités du programme, etc. Le but de cette section est de décrire brièvement quelques éléments de UML, afin de se faire une première idée.

Note. Il n'est pas nécessaire de connaître UML pour faire du Python objet (mais il n'est pas mauvais de savoir qu'un tel langage existe).

Dans le langage UML, une classe est représentée par un tableau divisé en 3 cases : le nom, les champs et les méthodes :

le nom
les champs
les méthodes

Tableau 1.9. Représentation UML d'une classe.

Chaque champ est inscrit avec son nom, mais aussi sa visibilité, multiplicité, type et éventuelle valeur initiale, selon la syntaxe ci-dessous :

```
visibilité nom [multiplicité] : type = valeur initiale
```

La visibilité ne concerne pas vraiment Python, néanmoins, il faut savoir que dans la plupart des langages orientés objet, chaque champ possède un degré de visibilité parmi les degrés suivants :

- `+` public (*public*)
- `#` protégé (*protected*)
- `~` package (*package*)
- `-` privé (*private*)

Un champ privé, par exemple, est un champ qui n'est accessible que depuis l'intérieur de la classe. Dans l'idéal, l'encapsulation chère à la POO voudrait que tout champ soit privé, ce qui obligerait le programme à utiliser des accesseurs et des mutateurs.

Pour comprendre la multiplicité, nous donnons un exemple. Supposons que chaque client est représenté par une instance de `Client` et possède un ou deux numéros de téléphone (un fixe et un mobile). Au lieu d'écrire :

```
Tél1 : int
Tél2 : int
```

on écrit

```
Tél[2] : int
```

De manière analogue, chaque méthode est inscrite avec son nom, sa visibilité, sa signature et le type de ce qui est éventuellement retourné. Dans la signature, chaque paramètre est inscrit avec sa direction, son nom et son type :

```
visibilité nom(dir x : type...) : type
```

La direction est *in*, *out* ou *inout*. Voici un exemple :

```
évolue(in x:int, in y:float, in z:Date) : void
```

Il s'agit d'une méthode nommée `évolue` possédant 3 paramètres `x` (entier), `y` (décimal) et `z` (date). Cette méthode ne retourne rien (*void*).

Le diagramme correspondant à la classe `Client` définie à la section 1.3 ressemblerait à ceci :

Client
+ prénom : str + nom : str + naissance : Date + nationalité [2] : Pays + adresse : str + crédit : float
+ payer(in somme:float, out crédit:float) : void + âge() : int

Tableau 1.10. Représentation UML de la classe `Client`.

Tous les attributs de notre exemple sont publics car l'objectif de ce livre est la programmation avec Python, et comme cela a été dit, Python ne s'embête pas avec la notion de visibilité des attributs.

Nous avons parlé de relations entre les classes comme l'héritage, l'agrégation, la composition, mais il en existe d'autres comme l'*association*, la *dépendance*, etc. Chacune de ces relations possède sa propre représentation UML :

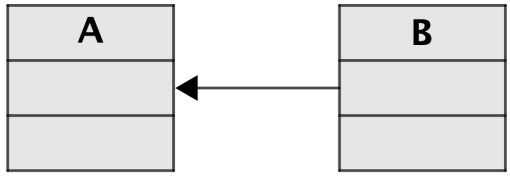


Figure 1.3. Héritage : `B` dérive de `A`

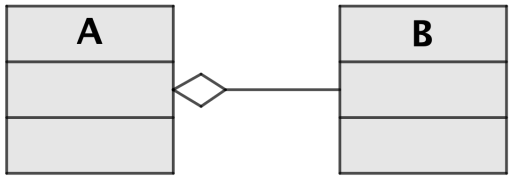


Figure 1.4. Agrégation : un `A` possède un `B`

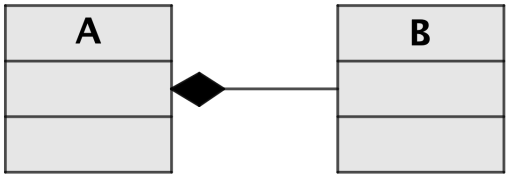


Figure 1.5. Composition : un `A` possède un `B`

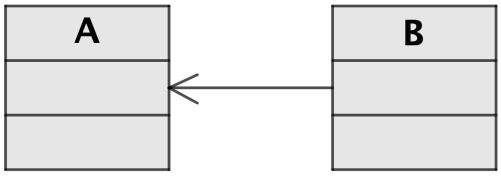


Figure 1.6. Un type d'association (il en existe d'autres)