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.

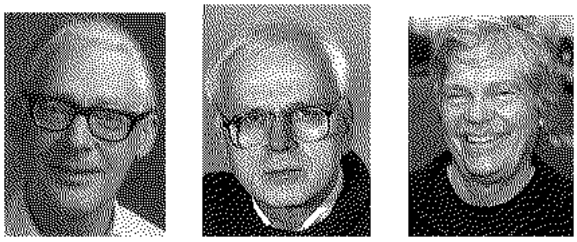

Figure 1.1. O.-J Dahl, K. Nygaard et A. Kay.

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
```