Chapitre 2

Objets dans Python. Structures et fonctionnalités

2.1 Propriétés essentielles. Exemples

En Python pratiquement tout est *objet* au sens technique du terme : une entité qui combine une structure de données (conteneur), avec des fonctionnalités spécifiques à ces données (méthodes, attributs), et un espace de noms, souvent hiérarchique et partagé. Les objets font aussi partie d'un système d'aiguillage, qui permet d'exécuter des opérations diverses en fonction du type des opérandes. Cet aiguillage est *implicite* et *statique* (au sens que le code utilisateur n'inclut pas des commandes de délégation de contrôle), spécifié par la construction des classes et leur héritage, et élimine la présence d'un fleuve de conditionnelles dans le programme :

```
if type(x)==A: ... elif type(x)==B: ... elif ...
```

Les fonctions, le code compilé, et plusieurs environnements (espaces de noms, typiquement : dictionnaires) sont également des objets, disposant d'un état interne, modifiable par le programme utilisateur. Ceci a facilité l'évolution de Python lui-même, du langage, et des mécanismes du compilateur-interprète, ce qui a permis la construction d'optimiseurs comme Numba (voir 7.2).

Ainsi notre premier programme-exemple va être également évolutif, nous allons y ajouter de nouvelles fonctionnalités, discutées dans le texte qui suit. Ceci dit, les objets ne suivent pas toujours les mêmes règles. Les accès et les modifications (ou : l'« interface » des objets si on veut employer cette terminologie, mais elle n'est pas universelle dans le monde des langages) des objets primitifs, immuables, comme les nombres, suivent des protocoles différents des objets appartenant aux classes définies par le programmeur. Cette observation peut paraître triviale et évidente, mais ces différences changent avec chaque nouvelle version du langage.

Notre « Hello World », est une classe qui implémente en Python les nombres complexes, $x+\sqrt{-1}y$, prédéfinis déjà en Python : `x + y*1j`. Ici, ce seront des paires (x, y), implémentées comme instances d'une simple classe : `z = C(x,y)`. Certains éléments de la construction seront utiles dans d'autres exemples de structures numériques/algébriques : les vecteurs ou les spineurs, les nombres duaux de Clifford [31], les quaternions, etc., donc ceci peut

inspirer plus d'un exemple de nature scientifique ou technique, basé sur les mathématiques appliquées. Commençons par les dispositifs d'initialisation et l'affichage d'une instance :

```
class C(object):
    def __init__(self,x,y=0.0):
        self.re=x;  self.im=y
    def __repr__(self):
        return 'C'+str((self.re,self.im))
b=C(1.0,2.0); print(b)
```

Ceci suffit pour créer et afficher quelques variables. Nous répétons qu'un tel programme doit être compréhensible (et même banal) pour le lecteur : le sens du paramètre **self**, la méthode **__init__**, etc., car nous ne pouvons remplacer plusieurs bons livres pédagogiques.

La syntaxe **C(object)** est redondante en Python 3. Ici toutes les classes appartiennent au « nouveau style » et héritent de la classe **object**, même si on les définit en style ancien :

```
class C:
    ...
```

Nous allons – là où applicable – choisir cette notation plus courte.

Pour l'affichage on n'a pas besoin d'afficher les deux composantes séparément, l'affichage standard des tuples est une astuce utile, combinant **x,y** lors de la conversion en chaîne. Mais l'objet **b** est beaucoup plus qu'un tuple à deux composantes. La commande **dir(b)** affiche la liste : **['__class__', ... '__dict__', '__dir__', '__doc__', '__init__', '__repr__', ..., 'im', 're']**, presque 30 items. Les chaînes (constantes) **'im', 're'** sont là, mais les accès aux champs **b.re** etc. passent par la variable **__dict__**, un dictionnaire interne qui constitue l'essence (passive, contrastée avec les items actifs : méthodes) d'une instance d'objet. Ce dictionnaire contient **{'im' : 1.0, 're' : 1.0}**, et l'information à assimiler par les apprentis, est qu'en Python les objets sont des dictionnaires augmentés par quelques références, et les noms de variables d'instance sont des clés de ce dictionnaire. Plus tard nous verrons des exceptions très importantes, mais cette stratégie d'implémentation caractérise d'autres langages dynamiques à objets, comme Smalltalk ou Javascript. On peut se poser quelques questions concernant ce côté dynamique des structures nommées :

- ▷ Est-il possible de changer le nom de la classe (par exemple un doublon aurait été créé), et l'affichage reflèterait automatiquement ce changement ?
- ▷ Est-ce que les variables globales dans le programme, ou variables internes dans une fonction ont quelque chose à voir avec les dictionnaires ? Dans un langage compilé en code machine, les variables locales résident sur une pile, et les globales peuvent avoir des adresses statiques, ce qui est considérablement plus rapide et l'allocation de mémoire est plus économique, qu'un tableau de hachage, dont l'accès aux éléments prend un certain temps.

La première question est double. Pour l'affichage, l'instance doit connaître le nom de sa classe, on peut demander la valeur de **b.__class__** dans l'exemple ci-dessus. Mais si – comme dans la section 2.2 – une méthode à l'intérieur de la classe créé une nouvelle instance, la connaissance du nom ne suffit pas, l'instance doit appeler le constructeur des instances, qui en Python est la classe elle même : un objet dont la classe est **type**, et qui est appelable comme une fonction. La multi-fonctionnalité des entités en Python est pervasive.

L'attribut **self.__class__** référence cet objet, et pour son nom, il suffit d'accéder à l'attribut **__name__** de la classe : **self.__class__.__name__**. Ces entités remplaceront **C** et **'C'**.

Python est un langage, dont la couche d'*introspection* – la possibilité d'analyser le programme par lui même – est très puissante et complète, et ceci était considéré par Guido Van Rossum comme une de plus importantes propriétés du langage, plus importante que les optimisations « statiques » du code utilisateur. On gagne en flexibilité, donc en adaptativité et en tolérance, en débogage, et en facilité de fusionner les programmes avec les librairies écrites dans d'autres langages. Malgré la pression des communautés qui se battent pour la vitesse de programmes, qui n'ont pas (ou rarement) besoin de dispositifs d'introspection dynamique, cet aspect du langage reste fondamental.

Terminons donc par un exemple d'introspection. Imaginons que notre programme, qui opère avec la classe C, opère aussi avec une autre classe V, représentant des vecteurs avec trois champs **x,y,z**, et aussi avec une classe de quaternions Q à quatre champs, et encore avec une dizaine d'autres structures similaires. Comment éviter la nécessité d'écrire une dizaine de méthodes **__repr__** presqu'identiques ?

La réponse simpliste, non-universelle, consiste à faire hériter les classes **C**, **V** et les autres d'une classe nommée **Repr** :

```
class Repr:
    def __repr__(self):  # La seule méthode
        nom=self.__class__.__name__  # 'C', 'V', etc.
        champs=tuple(self.__dict__.values())
        return nom+str(champs)
class C(Repr): ... # etc.
```

L'affichage de **b** aura la même forme que ci-dessus. Python donne à ses structures de données la possibilité de les analyser en profondeur, et éventuellement d'optimiser leur comportement. Cette vision du système Python comme ouvert, permettant de mettre les doigts presque partout, va nous accompagner jusqu'à la fin de ce livre.

2.1.1 Variables et dictionnaires

Dans un langage compilé en code de bas niveau (« niveau assembleur »), une affectation typique, disons : **x = 7.1**, donne lieu à une réservation par le compilateur d'une adresse dans la mémoire typiquement relative à la zone prévue pour l'activation du module qui définit cette variable [1]. Pendant l'exécution, le code engendré n'aura plus besoin de son nom. On doit cependant savoir quelle est la taille de l'espace réservé. Si le langage est statiquement typé : les types de données affectés aux variables sont connus durant la compilation, le compilateur sait combien d'octets y seront nécessaires. Ceci accélère de manière considérable la gestion des *tableaux indexés*, très importante pour les calculs numériques.

Pour les langages dynamiques, où les types des *variables* n'ont pas de sens pour le compilateur, la stratégie consiste à considérer la variable non pas comme l'adresse de la zone de stockage, mais comme l'endroit où l'on stocke l'adresse de cette zone (le « pointeur »).

1. Pour les langages statiques, la réservation a pu se faire lors de le *déclaration* de la variable, manifeste ou automatique.

Puisque les adresses sont typiquement de même taille, l'indexation peut être rapide, mais le dé-référencement de la donnée est plus lente, à cause de l'indirection.

En Python l'accès aux variables peut être encore plus lent : aussi dans l'environnement global, en dehors des classes, une variable est **une association** entre son nom et sa valeur, cette paire existe lors de l'exécution, et constitue un fragment de l'environnement (*name-space*) de la classe, ou du module en question. Ceci implique la possibilité de décoder les noms des variables pendant l'exécution. On peut demander à l'interprète de trouver la valeur de la variable, dont le *nom* est donné : `eval('b.re')` retourne 1.0. Cela implique que l'accès global sera plus lent que le décodage des variables locales et des arguments des fonctions, le contraire de ce que caractérise les langages compilés en code machine.

L'affectation `x = 7.1` est donc la création d'une association : une entrée dans le dictionnaire représentant l'environnement en vigueur. L'environnement global est un dictionnaire accessible grâce à la fonction `globals`. Ceci permet l'accès interactif aux *variables* – non pas seulement à leurs valeurs. L'affectation ci-dessus peut être codée comme

```
gldict=globals()
gldict['x']=7.1
```

Voici un exemple de traçage sélectif des variables globales, déclenché par l'interruption clavier. L'usager peut alors demander la valeur d'une variable quelconque de l'environnement global.

```
globvar = 0
def runtest():
    global globvar
    for i in range(1000):
        try:
            leVraiCalcul() #  P. ex. print(i*i);  etc.
            globvar +=1    #  Par exemple
        except KeyboardInterrupt:
            x=input("Pause. Quelle variable?: ")
            v=globals()[x]; print(x,"->",v)
        # La boucle reprend...
```

L'appui sur Ctrl-C pendant l'exécution de `runtest()` affiche l'invite, et attend une chaîne de caractères ; si on tape `globvar` (ou autre identifiant *existant*, sans guillements), l'information souhaitée est transmise. Une variable inexistante déclenche une autre exception, qui n'ayant pas été interceptée, arrête le programme.

On peut, grâce à ce décodage dynamique des noms, modifier des variables ou exécuter des fonctions interactivement, pendant que le programme tourne [2].

Il est possible également d'accéder aux variables locales d'une fonction, l'appel `locals()` à l'intérieur de la fonction retourne une copie de l'environnement local en tant que dictionnaire. Cependant, contrairement à `globals()`, le dictionnaire local est une « proxy » de l'environnement, permet de le lire, mais la modification de ses entrées laisse l'environnement original intact. Le stockage des valeurs locales des fonctions est optimisé. Cela n'empêche pas l'usage de `locals()` pour l'analyse interactive d'une fonction, de manière similaire au programme ci-dessus.

2. Plutôt : quand son exécution a été interrompue, afin de passer le contrôle à un agent de contrôle.

2.1.2 Classe comme environnement

L'instance ne stocke pas individuellement les références vers les fonctions-méthodes, car celles-ci sont partagées. La classe possède son propre `__dict__`, et les méthodes d'instances y sont associées avec leurs noms. Ce sont des fonctions *normales* qui peuvent être appelées de cette manière : `C.__repr__(b)`, le « self » prend sa place normale, comme le premier argument.

La classe peut contenir d'autres entités, on peut définir une variable partagée : `pi = 3.1415926536`. Cette constante sera placée dans le dictionnaire, et accessible par toutes les instances. On trouve `'pi'` en exécutant `dir(b)`, ou une autre instance de `C`, et il n'est pas possible d'en déduire tout de suite si cette entité appartient à l'instance ou à la classe.

On peut modifier de l'extérieur le contenu d'une classe, par exemple `C.pi=1`, mais si on passe par `__dict__`, en exécutant `C.__dict__['pi'] = 0`, on découvre que ceci n'est pas un dictionnaire normal, mais un objet de la classe `mappingproxy({ ...})`. Cet objet est un « emballage », qui rend le résultat en lecture seule, et l'affectation échoue.

Ceci avait pour but d'accélérer l'accès aux attributs, en facilitant l'usage possible du cache, mais le statut actuel de ce dispositif n'est pas clair. Certains suggèrent qu'il s'agissait d'augmenter la protection du contenu de la classe, mais les programmes Python n'ont pratiquement aucune défense contre l'auto-destruction, et l'affectation `Classe.nomMéthode = 0` abîme la Classe, sauf si celle-ci étant intégrée est déjà protégée. Ceci dit, les manipulations du contenu d'une classe peuvent ainsi être optimisées, les clés de ce dictionnaire sont obligatoirement des chaînes, et la liaison entre l'accès aux méthodes, et la couche magique de la classe, est plus directe. Ceci est considéré comme un détail de l'implémentation CPython, voir [26].

Nous terminons cette sous-section par l'observation que la classe de classes est l'objet `type`, qui est sa propre classe. Pour un utilisateur typique de Python ceci n'a pas beaucoup d'importance, mais le développement avancé parfois exige la *création dynamique de classes*, pendant l'exécution du programme. Par exemple, un programme peut lire un fichier JSON avec une description de structures de données hiérarchiques, ou une description d'une base de données relationnelle, et construire une collection d'objets correspondant à sa structure. L'appel de `type(...)` ne sert pas qu'à la vérification de la classe de son argument, mais peut en créer une autre. Ce sujet est repris dans la section 5.5 consacrée aux métaclasses.

2.2 Surcharge des opérateurs

2.2.1 Arithmétique et héritage

Retournons aux nombres complexes, et ajoutons quelques méthodes arithmétiques à notre classe, par exemple l'addition et la multiplication. Les définitions ci-dessous profitent des opérations standard, mais ceci n'est nullement dû à l'héritage, la classe `C` n'héritant d'aucune classe (sauf `object`). Ce que nous appelons souvent la surcharge, est une redéfinition (*overriding*).

```
def __add__(self,ob):
    return C(self.re+ob.re,self.im+ob.im)
```

```
    def __mul__(self,ob):
        x=self.re*ob.re-self.im*ob.im
        y=self.re*ob.im+self.im*ob.re
        return C(x,y)
d = C(1,2)+C(0,1)*C(3,1) #-> C(0,5)
```

Nous avons défini les méthodes **__add__** et **__mul__**, qui sont les « vrais » noms des fonctions correspondantes. Contrairement à C++, les opérateurs unaires et binaires en Python n'ont pas d'identités en tant que fonctions (compilables), ce ne sont que des abréviations syntaxiques. Leur place est dans la grammaire de Python, employée pendant le processus de compilation. La forme **a+b** est considérée équivalente à **a.__add__(b)**. La collection de fonctions implémentant les opérateurs est assez complète : **__sub__**, **__pow__** (puissance) **__gt__**, **__or__**, **__neg__** (moins unaire), **__mod__**, etc.

En Python 3 la division **__div__** n'existe plus, on dispose de la division entière (euclidienne) **__floordiv__**, qui tronque le résultat, toujours entier, et la division réelle : **__truediv__**. Le module standard **operator** [35] exporte et facilite l'usage des opérateurs comme des fonctions normales, et épargne l'écriture des tirets-bas ; on peut écrire **mod(a,b)** au lieu de **a.__mod__(b)**. La collection contient plus de 100 éléments, plusieurs opérateurs ont uniquement une forme textuelle, mais certains correspondent à des constructions syntaxiques qui dans d'autres langages ne sont pas assimilées aux opérateurs. Par exemple – important pour le développement – l'accès indexé aux éléments d'une séquence (liste, tableau, etc.) : **v = p[n]** est une abréviation de **v = p.__getitem__(n)**.

L'affectation **p[n] = v** signifie **p.__setitem__(n,v)**, et il est possible d'indexer des tranches, par **p.__getitem__(slice(2,9))**, équivalent à **p[2:9]**. Les opérateurs de modification *in situ*, comme **x+=1**, ont également des formes fonctionnelles : **x.__iadd__(1)**.

On ne peut surcharger les opérateurs agissant sur des types prédéfinis, numériques, ou textes (redéfinir **__add__** pour eux), mais traditionnellement l'arithmétique mixte est acceptée par la plupart des langages, on peut écrire **(2-3j)+7**, ou **1.5*(2+1j)**, donc il est souhaitable de prévoir une telle extension à notre classe **C**. Ici il faut ajouter deux choses :

- ▷ Vérifier si l'argument droit de l'opérateur appartient à la classe dont il est question, et sinon, effectuer la conversion appropriée (ou déclencher une exception).
- ▷ Profiter de la convention, que pour les opérateurs binaires, si l'argument gauche est d'un type prédéfini, mais l'argument droit appartient à une classe utilisateur, Python appelle *un autre* opérateur « jumeau » avec les arguments échangés. Ainsi **x.__add__(y)** cause l'exécution de **y.__radd__(x)**, **__mul__** lance **__rmul__** etc.

Voici la partie arithmétique de la classe **C** complétée :

```
    def conjg(self): return C(self.re,-self.im)
    def abs2(self):  return self.re**2 + self.im**2
    def inv(self):   return self.conjg()/self.abs2() # 1/self
    def __neg__(self): return C(-self.re,-self.im)
    def __add__(self,ob):
        if isinstance(ob,C):
            return C(self.re+ob.re,self.im+ob.im)
        return C(self.re+ob,self.im)
    def __radd__(self,ob):  return C(self.re+ob,self.im)
```

```
    def __sub__(self,ob):
        if isinstance(ob,C):
            return C(self.re-ob.re,self.im-ob.im)
        return C(self.re-ob,self.im)
    def __rsub__(self,ob):  return C(ob-self.re,-self.im)
    def __mul__(self,ob):
        if isinstance(ob,C):
            x=self.re*ob.re-self.im*ob.im
            y=self.re*ob.im+self.im*ob.re
            return C(x,y)
        return C(ob*self.re,ob*self.im)
    def __rmul__(self,ob):  return C(ob*self.re,ob*self.im)
    def __truediv__(self,ob):
        if isinstance(ob,C):  return self*ob.inv()
        return C(self.re/ob,self.im/ob)
    def __rtruediv__(self,ob): return self.inv()*ob
```

Il faut aussi prévoir la possibilité d'appliquer les fonctions élémentaires habituelles, exp, log, sin, etc. Ceci peut être un exercice en sélection du code dans une fonction :

```
import math as M
def exp(x):
    if isinstance(x,C): # Aiguillage manuel
        rho=exp(x.re)
        return C(rho*M.cos(x.im),rho*sin(x.im))
    else:
        return M.exp(x)
```

et similaire pour d'autres fonctions, qui choisiront la variante selon l'argument. On peut penser que cette solution est naturelle, voire unique. Ceci est inexact. Elle est différente de l'implémentation du *moins unaire*, la fonction **__neg__**. Ici la délégation de l'opération selon l'argument se fait de manière interne : c'est le système d'aiguillage des méthodes qui s'en charge. Pas de conditionnelles dans le code Python, on écrit : **(-expr)**. La philosophie « tout est objet » n'englobe pas tout de la même façon, les fonctions dans le module **math** (ou **cmath** pour les nombres complexes) ne sont pas des méthodes, et n'appellent pas les méthodes de leurs arguments. Dans plusieurs autres librairies, par exemple pour les itérateurs, c'est différent. Le programmeur peut choisir l'interface procédurale, et appeler **next(g)** pour obtenir l'élément suivant engendré par l'itérateur **g**, de classe quelconque disposant de la méthode **__next__**. En la définissant, ou en construisant un générateur, la fonction **next** délègue son action à cette méthode.

Mais la fonction **exp** dans **math** ne peut effectuer une telle opération. Si elle le pouvait, son code aurait pu avoir (conclusion un peu simplifiée) la forme :

```
def exp(x): return x.__exp__()

# ... et dans le programme :
import math as M
```

```
class C:
    def __exp__(self):
        rho=exp(self.re)
        return C(rho*M.cos(self.im),rho*sin(self.im))
```

Il faudrait, bien sûr, prévoir la méthode **__exp__** dans la classe des nombres réels, ce qui conceptuellement et techniquement serait facile, mais n'a pas été fait. La séparation de fonctions mathématiques dans un module extérieur à charger était un choix humain, probablement dicté par l'observation selon laquelle les procédures mathématiques sont peu utiles pour le plus grand nombre, et il est redondant de les avoir dans l'espace de noms en permanence.

2.2.2 Héritage des classes numériques. Surcharge de *__new__*

La surcharge de **__new__** est rarement employée, mais intéressante, car elle touche au sens de l'*abstraction* en Python, et favorise une réutilisation cohérente des structures du langage pour des domaines différents. Nous allons montrer comment cet héritage peut être appliqué pour la simplification du code. Ceci n'est pas une simplification au sens d'une élimination globale du code, c'est seulement une simplification du code utilisateur (pas d'aiguillage explicite), qui a besoin de construire et d'appliquer des fonctions polymorphes, adaptables à des types variés, et leur coercition en méthodes pose des difficultés. Ces questions ont contribué à l'introduction des « classes abstraites » générales (ABC) et numériques [1, 32].

La création typique d'une nouvelle classe inclut l'exécution de la méthode standard, surchargeable **__init__(self,...)**, qui initialise le contenu de l'instance crée, et c'est presque tout, ce qui est enseigné aux non-experts. Au moment de l'initialisation, l'instance a déjà été crée, la mémoire de base (les attributs standard) est allouée, les variables de base comme **__class__**, **__dict__**, etc., sont instanciées (mais **__dict__** est normalement vide). Cependant, pour certaines classes – par exemple pour les nombres – ceci n'a pas lieu, car les instances concernées ***sont immuables***, elles n'ont pas de dictionnaire d'attributs. Si on crée un entier : **x = 3.14**, on peut changer **x**, mais la constante 3.14 est figée jusqu'à la fin de son existence (la coupure de toutes les références vers elle). La méthode **__init__** existe pour tous objets, les nombres et la classe **object** comprise, mais elle ne fait rien. Tout se déroule au moment de la *création* de l'instance, de l'allocation de mémoire.

Ceci est le rôle de la fonction **__new__**, une *méthode statique*, agissant dans le domaine de classes. Construisons une sous-classe de nombres réels. Ici **__new__** ne fait qu'appeler le constructeur de la superclasse, ce qui engendre un nombre réel normal.

```
import math as M
class R(float):
    def __new__(cls, value):  # cls : nouvelle classe, "R"
        return  super().__new__(cls, value)
        # super(R, cls) est : float
```

Voici l'instanciation de la classe **R** :

```
x=R(6);  y=R(8.1)
z=x/y
w=M.exp(x)
```