Python objet

Christian Nguyen

Département d'informatique
Université de Toulon

Introduction

Python est un langage de programmation de haut niveau, interprété, faiblement typé et offrant typage et liaison [1] dynamique.

Absolument tout y est **objet** : types, variables, structures, fonctions, modules, ...

Toutes les classes héritent de la classe `object`.

Toute les fonctionnalités d'un langage objet : classe, héritage multiple, surcharge d'opérateurs, ...

Associé à une bibliothèque d'interface graphique, il offre un environnement de prototypage efficient [2].

1. binding : association d'une requête à une méthode d'un objet
2. optimisation de la consommation des ressources utilisées dans la production d'un résultat

Définition d'une classe

Syntaxe : mot-clé `class`, nom de la classe et un deux points. Il *faut* l'accompagner d'une docstring.

Convention de nommage : PEP 8 (*Python Enhancement Proposals*), convention dite Camel Case[3].

```
class Point2D:
    """
    definition et manipulation d'un point
    dans le plan euclidien
    """

    def __init__(self):
        """constructeur : point a l'origine par defaut"""
        self.x, self.y = 0.0, 0.0
```

3. classe : première lettre de chaque mot en majuscule,
   méthodes : tout en minuscule séparé par underscore

Constructeur

Il se définit comme une fonction mais il a pour nom invariable `__init__`. Les méthodes entourés de part et d'autre de deux blancs soulignés sont des méthodes spéciales.

Notons que, dans la définition de cette méthode particulière, on passe un paramètre de nom `self` (la référence à l'objet est systématiquement passé en premier paramètre).

Dans ce constructeur, on initialise deux attributs d'instance (préfixés du mot-clé self) de nom `x` et `y`.

Pour créer un objet de cette classe, on écrit : `pt1 = Point2D()`.

Remarque : concernant le destructeur (`__del__`), il vaut mieux éviter de le (re)définir (« It is not guaranteed that __del__() methods are called for objects that still exist when the interpreter exits. »)[4].

4. voir aussi, PEP 442 – Safe object finalization

Accès aux membres d'une classe

Python a une philosophie particulière sur ce point, il propose « l'encapsulation par courtoisie ».

- aucun utilisateur d'une classe n'est censé passer outre les protections offertes par l'interface de la classe.

On peut définir des accesseurs et des mutateurs mais ils n'ont pas vraiment lieu d'être dans la philosophie du langage (qui les considère en outre comme contraignants).

Python leur préfère la notion de *propriété*.

Attributs de classe

La définition d'un attribut de classe se fait directement dans le corps de la classe, après la définition (et la docstring !), avant la définition du constructeur.

Pour y accéder, on préfixe le nom de l'attribut de classe par le nom de la classe.

```
class Point2D:
    """
    definition et manipulation d'un point
    dans le plan euclidien
    """
    numero = 0

    def __init__(self):
        ...
```

Méthodes d'objet, méthodes de classe

Les méthodes d'instance, ou **méthodes d'objet**, comportent dans leur définition le paramètre `self`. Quand on crée un nouvel objet, les attributs de l'objet sont propres à l'objet créé.

En revanche, les méthodes sont contenues dans la classe qui définit un objet.

Une **méthode de classe** prend en premier paramètre `cls` (la classe de l'objet). De plus, on doit utiliser la fonction built-in `classmethod` pour l'identifier comme méthode de classe.

```
class Point2D:
    ...
    def combien(cls):
        print("{} objets crees".format(cls.numero))
    combien = classmethod(combien)  # dans la classe
```

Exemple de méthode de classe

Un constructeur alternatif (Stack Overflow)

```
>> class Y(object):
...     def __init__(self, astring):
...         self.s = astring
...     @classmethod
...     def fromlist(cls, alist):
...         x = cls('')
...         x.s = ','.join(str(s) for s in alist)
...         return x
...     def __repr__(self):
...         return 'Y(%r)' % self.s
...
>>> y1 = Y('xx')
>>> y1
Y('xx')
>>> y2 = Y.fromlist(range(3))
>>> y2
Y('0,1,2')
```

Méthodes statiques

On peut également définir des **méthodes statiques**.

Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent pas le paramètre `cls` (ni `self`). Elles travaillent donc indépendamment de toute donnée (qu'elles ne peuvent modifier).

```
class Test:
  def afficher():
    """methode chargee d'afficher quelque chose"""
    ...
  afficher = staticmethod(afficher)
```

Elles servent principalement à définir un espace de nommage, à associer des fonctions à une classe en particulier.

Introspection

Explorer un objet, connaître ses méthodes ou attributs.

La fonction dir prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

```
>>> dir(pt1)
['__class__', '__delattr__', '__dict__', '__dir__',
...
'combien', 'id', 'n', 'x', 'y']
```

L'attribut spécial __dict__ : par défaut, tous les objets construits possédent cet attribut spécial, un dictionnaire composé des noms des attributs et des valeurs de ces attributs.

```
>>> pt1.__dict__
{'y': 0.0, 'x': 0.0, 'id': 0}
```

Introduction

Concept propre à quelques langages (Python, Ruby, ...). Elle change l'approche objet et le principe d'encapsulation.

Rappel : encapsulation = principe qui consiste à cacher ou à protéger les attributs d'un objet ; la plupart ne doivent pas être accessibles depuis l'extérieur de la classe.

- principes d'accès qui indiquent si un attribut est public ou privé.
- notions d'accesseurs et de mutateurs (implantation contraignante).

En Python, les attributs sont accessibles directement par défaut mais pour certains, on peut créer des propriétés.

- un moyen *transparent* de manipuler des attributs d'objet.

Mise en œuvre

Dans tous les cas, il semble y avoir un accès direct à l'attribut.

Dans la définition de la classe, on précise si un attribut doit être accessible ou modifiable grâce à certaines propriétés. Ces dernières agissent différemment en fonction du contexte dans lequel elles sont appelées.

Par exemple, si on les appelle pour modifier un attribut, elles vont rediriger vers une méthode qui gère ce cas.

Une propriété est une instance de la classe property. Elle attend quatre paramètres, tous optionnels :

- la méthode donnant accès à l'attribut,
- la méthode modifiant l'attribut,
- la méthode appelée quand on souhaite supprimer l'attribut,
- la méthode appelée quand on demande de l'aide sur l'attribut.

Exemple

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit les accesseurs et les mutateurs.

```
class Protegee:
  def __init__(self):
    self.__x = 0.0  # notez le double underscore

  def __get_x(self):
    return self.__x

  def __set_x(self, px):
    self.__x = px

  # l'attribut x pointe vers une propriete
  x = property(__get_x, __set_x)
```

Exemple

Le « Name Mangling » complique l'accès, depuis l'extérieur de la classe, à un attribut commençant par un double underscore.

Dans l'exemple, on déclare que l'attribut `x` est une propriété, pour laquelle on définit la méthode d'accès (l'accesseur) et celle de modification (le mutateur).

Quand on accède à `pt1.x`, la propriété redirige vers la méthode `__get_x` et quand on souhaite modifier la valeur de l'attribut par `pt1.x = 1.0`, la propriété appele la méthode `__set_x` en lui passant en paramètre la nouvelle valeur.

Remarque : si on ne définit qu'un accesseur, l'attribut ne pourra pas être modifié.

Rappel : il est possible de définir une 3ème méthode qui sera appelée quand on fera `del pt1.x` et une 4ème méthode qui sera appelée quand on fera `help(pt1.x)`.

Les méthodes spéciales

Ces méthodes sont utiles lorsque l'on souhaite effectuer un traitement récurrent sur certains attributs (par exemple si l'on souhaite enregistrer un objet dès que l'on modifie l'un de ses attributs).

Ce sont des méthodes d'instance qui contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

La plus utilisée est le **constructeur** (méthode `__init__`) qui prend un nombre variable d'arguments et permet de contrôler la création de nos attributs.

Le **destructeur** (méthode `__del__`) doit être (re)défini avec circonspection car tenter de contrôler la destruction d'un objet est dangereux considérant le ramasse miette (garbage collector) de Python.

Les méthodes spéciales

Représentation de l'objet : deux méthodes spéciales permettent de contrôler comment l'objet est représenté et affiché.

La méthode `__repr__` affecte la façon dont est affiché l'objet quand on tape directement son nom. On la redéfinit quand on souhaite faciliter le debug sur certains objets.

La méthode `__str__` est appelée pour afficher l'objet avec `print` (sinon `__repr__`) ou pour convertir l'objet en chaîne de caractères.

Les méthodes spéciales

Accès aux attributs : comment accéder ou modifier les attributs d'un objet.

La méthode `__getattr__` permet de définir une méthode de gestion des attributs qui ne sont pas trouvés par Python (l'attribut recherché est passé sous la forme d'une chaîne de caractères).

La méthode `__setattr__` définit l'accès à un attribut destiné à être modifié. Cette méthode permet de déclencher une action, par exemple enregistrer l'objet, dès qu'un attribut est modifié.

```
def __setattr__(self, nom_attr, val_attr):
  # pourquoi pas self.nom_attr = val_attr ?
  object.__setattr__(self, nom_attr, val_attr)
  self.enregistrer()
```

Les méthodes spéciales

La méthode `__delattr__` est appelée quand on souhaite supprimer un attribut de l'objet, en faisant `del objet.attribut` par exemple. Elle prend en paramètre, outre `self`, le nom de l'attribut que l'on souhaite supprimer.

La méthode `__hasattr__` renvoie `True` si l'attribut existe, `False` sinon.

Python offre la possibilité d'utiliser des chaînes de caractères pour les noms d'attributs :

```
objet = MaClasse()
getattr(objet, "nom")
setattr(objet, "nom", val)
delattr(objet, "nom")
hasattr(objet, "nom")
```

Conteneur

Les objets conteneurs sont principalement les chaînes de caractères, les listes et les dictionnaires.

Tous ont un point commun : ils contiennent d'autres objets, auxquels on peut accéder grâce à l'opérateur [].

Trois méthodes importantes :

- `__getitem__` appelée quand on écrit `objet[index]`,
- `__setitem__` appelée pour `objet[index] = valeur`,
- `__delitem__` appelée pour `del objet[index]`.

Exemple de conteneur

Une classe enveloppe d'un dictionnaire.

```
class MyDict:
  def __init__(self):
    self._dico = {}

  def __getitem__(self, pi):
    return self._dico[pi]

  def __setitem__(self, pi, pv):
    self._dico[pi] = pv
```

Autres méthodes de conteneurs

La méthode `__contains__` est utilisée, via le mot-clé `in`, quand on souhaite savoir si un objet se trouve dans un conteneur.

```
lnb = [1, 2, 3, 4, 5]
8 in lnb  # lnb.__contains__(8)
```

La méthode `__len__` permet de connaître la taille d'un objet conteneur, elle ne prend aucun paramètre et renvoie une taille sous la forme d'un entier.

```
lnb = [1, 2, 3, 4, 5]
len(lnb)  # lnb.__len__()
```

Les méthodes mathématiques

D'autres méthodes spéciales permettent la surcharge d'opérateurs mathématiques.

La méthode `__add__` permet de surcharger l'opérateur +, elle prend en paramètre l'objet que l'on souhaite ajouter (par exemple, ajouter une quantité à un objet d'une classe `Date`).

Pour prendre en compte une opérande gauche qui ne serait pas une instance de la classe, il suffit de préfixer le nom des méthodes spéciales par un **r**. Exemple : `__radd__`.

Pour offrir une forme de polymorphisme i.e garantir différentes actions en fonction du type de l'objet à ajouter, il y a la possibilité de tester le résultat de `isinstance()` qui prend en compte l'héritage [5].

5. ce qui n'est pas le cas de `type()` à usage impératif

Les méthodes mathématiques

Sur le même modèle, il existe les méthodes :

- `__sub__` : surcharge de l'opérateur -
- `__mul__` : surcharge de l'opérateur *
- `__truediv__` : surcharge de l'opérateur /
- `__floordiv__` : surcharge de l'opérateur // (division entière)
- `__mod__` : surcharge de l'opérateur % (modulo)
- `__pow__` : surcharge de l'opérateur ** (puissance)

Il est également possible de surcharger les opérateurs +=, -=, etc. en préfixant les noms de méthode par un **i**.

```
>>> class Nope:
...     def __iadd__(self, pp):
...         print('not implemented')
>>> n1 = Nope()
>>> n1 += 1
not implemented
```

Les méthodes de comparaison

La surcharge des opérateurs de comparaison ==, !=, <, >, <=, >= se fait par des méthodes qui prennent en paramètre l'objet à comparer à `self`, et doivent renvoyer un booléen.

Il s'agit respectivement des méthodes : `__eq__`, `__ne__`, `__lt__`, `__gt__`, `__le__`, `__ge__`.

Si l'interprète ne parvient pas à effectuer une opération de comparaison, il renvoit une exception (`TypeError`).

Un mot sur les tris

Pour trier une séquence de données, Python nous propose deux méthodes :

- la méthode de liste `sort`, qui travaille sur la liste-même et change donc son ordre, si c'est nécessaire,
- la fonction `sorted`, une fonction *built-in*, qui travaille sur n'importe quel type de séquence (tuple, liste ou même dictionnaire) et qui ne modifie pas l'objet d'origine, mais en retourne un nouveau.

La méthode `list.sort` ou la fonction `sorted` ont toutes deux un paramètre optionnel, appelé `key`. Cet argument attend une fonction.

Exemple de tri

Pour trier une liste de **tuples** d'étudiants avec nom, âge et moyenne en fonction de leur moyenne : `sorted(etudiants, key=lambda col: col[2])`.

Si les étudiants sont des **instances** :

```
class Etudiant:
  def __init__(self, nom, age, moyenne):
    self.nom = nom
    self.age = age
    self.moyenne = moyenne
```

Deux façon de définir le tri des étudiants :

- définir la méthode spéciale `__lt__` si le paramètre est un nombre,
- utiliser l'argument key : `sorted(etudiants, key=lambda et: et.moyenne)` (dans l'ordre inverse : `reverse=True`).

Optimisation et extension

Les méthodes de tri qui reposent sur des fonctions lambdas [6] ne sont pas le meilleur choix au niveau rapidité.

Le module **operator** propose plusieurs fonctions qui vont s'avérer utiles dans ce contexte, en particulier les fonctions `itemgetter` et `attrgetter`.

```
from operator import itemgetter
sorted(etudiants, key=itemgetter(2))
```

Pour trier une liste d'objets :

```
from operator import attrgetter
sorted(etudiants, key=attrgetter("moyenne"))
```

6. Semantically, lambdas are just syntactic sugar for a normal function definition.

Tris : optimisation et extension

Trier selon plusieurs critères, par exemple tri des étudiants par âge et note moyenne (tri par âge, mais si deux étudiants ont le même âge, tri sur leur moyenne).

```
sorted(etudiants, key=attrgetter("age", "moyenne"))
```

Attention : si deux éléments d'une séquence à comparer sont identiques, leur ordre est conservé. Cette propriété est appelée « stabilité » et permet de chaîner les tris.

Pickle

Deux méthodes du module **pickle** sont utilisées pour redéfinir la façon dont les objets sont enregistrés dans des fichiers (voir aussi PEP 307).

La méthode `__getstate__` est appelée au moment de sérialiser l'objet, juste avant l'enregistrement.
Par défaut, pickle enregistre le dictionnaire des attributs de l'objet à enregistrer (contenu dans l'attribut ___dict___), sinon, il enregistre la valeur renvoyée par `__getstate__`.

La méthode `__setstate__` est appelée au moment de désérialiser l'objet, après la récupération du dictionnaire des attributs (ou un autre type d'objet, auquel cas la définition de `__setstate__` est indispensable).

Héritage simple

```
class A:
    ...
class B(A):
    ...
```

Les objets de type `B` reprennent les méthodes de la classe `A` en même temps que celles de la classe `B` et ce sont celles de la classe `B` qui sont appelées d'abord.

Si une méthode est définie dans différentes classes, celle définie directement dans la classe dont est issu l'objet est choisie, si elle existe, sinon il y a parcours de la hiérarchie de l'héritage.

Héritage simple

On appelle délégation l'appel à une méthode d'une classe parent surchargée depuis la méthode de la classe enfant.

En Python, la délégation n'est pas implicite, il faut y procéder explicitement. On se sert pour cela de la notation `classe.methode(objet)` pour appeler précisément une méthode d'une classe.

Cela est particulièrement utile au niveau des constructeurs.

```
class B(A):
  def __init__(self, px, py):
    A.__init__(self, px)
    self.y = py
```

Héritage simple

Python définit deux fonctions qui peuvent se révéler utiles dans bien des cas :

- `issubclass` vérifie si une classe est une sous-classe d'une autre classe,
- `isinstance` permet de savoir si un objet est issu d'une classe ou de ses classes filles (prise en compte de l'héritage, `type` lui ne considère que le type immédiat de l'objet).

```
>>> a = A() ; b = B()
>>> type(a) is A ; type(b) is A
True
False
>>> isinstance(a, A) ; isinstance(b, A)
True
True
```

Héritage multiple

Syntaxe : `class Fille(Parent1, Parent2):`

Les algorithmes chargés d'effectuer la recherche de la signature d'une méthode parmi les classes mères appartiennent à la famille des Method Resolution Order (ou MRO).

Initialement, l'algorithme était une recherche *depth-first* et *left-to-right* : l'ordre de définition des classes mères conditionnait la recherche d'une méthode.

Actuellement, la recherche des méthodes se fait suivant l'algorithme C3 (dans un souci de rétrocompatibilité, les deux algorithmes sont présents dans la branche 2, distingo par `object`).

Il est recommandé désormais, en cas d'héritage multiple, de systématiquement stipuler `object`, afin que le code puisse être exécuté indifféremment en branche 2 ou 3.

Héritage multiple

```
class A(object):
    pass

class B(object):
    pass

class C(A, B):
    pass

class D(B,A):
    pass

class E(D,A):
    pass

try:
    class F(A,D):
        pass
except Exception as e:
    print(e)
```

- Cannot create a consistent method resolution order (MRO) for bases A, D

Héritage multiple - Linéarisation

$$L(0) := [O]$$

$$L(A) := [A] + \text{merge}(L(O), [O]) = [A] + \text{merge}([O], [O]) = [A, O]$$

$$L(B) := [B] + \text{merge}(L(O), [O]) = [B] + \text{merge}([O], [O]) = [B, O]$$

$$L(C) := [C] + \text{merge}(L(A), L(B), [A,B]) =$$
$$[C] + \text{merge}([A,O], [B,O], [A,B]) = [C, A, B, O]$$

$$L(D) := [D, B, A, 0]$$

$$L(E) := [E] + \text{merge}(L(D), L(A), [D,A]) =$$
$$[E] + \text{merge}([D,B,A,O], [A,O], [D,A]) = [E, D, B, A, O]$$

$$L(F) := [F] + \text{merge}(L(A), L(D), [A,D]) =$$
$$[F] + \text{merge}([A,O], [D,B,A,O], [A,D]) = ???$$

Héritage multiple - Configuration diamant

La délégation pose problème quand plusieurs classes parents d'une classe fille héritent d'une même classe.

Pour éviter les multiples appels à la même méthode surchargée, il est possible d'utiliser la fonction *built-in* `super()`.

`super()` prend deux arguments, une classe et une instance, qui peuvent être omis dans le contexte d'une classe (sucre syntaxique ou *syntactic sugar*).

`super()` crée un *wrapper*[7] qui permet de court-circuiter l'appel à `__getattribute__()` de l'objet afin d'introduire une nouvelle manière de résoudre l'appel à une méthode de cet objet.

7. voir la section « Décorateurs »

Configuration diamant - Délégation explicite

```
class A():
    def __init__(self):
        pass

class B(A):
    def __init__(self):
        A.__init__(self)

class C(A):
    def __init__(self):
        A.__init__(self)

class D(B,C):
    def __init__(self):
        B.__init__(self)
        C.__init__(self)

d = D()  # B appelle A.__init__() et C appelle A.__init__()
```

Configuration diamant - Délégation implicite

```
class A():
    def __init__(self):
        pass

class B(A):
    def __init__(self):
        super.__init__()

class C(A):
    def __init__(self):
        super.__init__()

class D(B,C):
    def __init__(self):
        super.__init__()

d = D()   # B.__init__(), C.__init__(), A.__init__()
```

Héritage multiple - Exemple pratique

Les exceptions sont un bon exemple, car ce sont des classes hiérarchisées selon une relation d'héritage précise.

Cette relation d'héritage est importante car le type de l'exception qui suit `except` est intercepté mais aussi toutes les classes qui héritent de ce type.

La plupart des exceptions sont levées pour signaler une erreur mais pas toutes (exemple : `KeyboardInterupt`). Pour intercepter toutes les *erreurs* potentielles, on écrira donc `except Exception:`, toutes les exceptions « d'erreurs » étant dérivées de `Exception`.

Création d'exceptions personnalisées : on peut créer sa propre classe exception (qui doit contenir au moins deux choses : un constructeur et la méthode `__str__`), la lever avec `raise`, l'intercepter avec `except`.

Création d'exceptions personnalisées

création

```
class MonException(Exception):
    """ exception levee dans un contexte a definir """
    def __init__(self, message):
        """ stockage du message d'erreur """
        self.message = message
    def __str__(self):
        """ renvoi du message """
        return self.message
```

utilisation

```
...
raise MonException("il y a un probleme")

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonException: il y a un probleme
```

Itérateurs

Un itérateur est un objet qui va être chargé de parcourir un objet conteneur (par exemple une liste).

L'itérateur est créé dans la méthode spéciale `__iter__` de l'objet.

À chaque itération, Python appelle la méthode spéciale `__next__` de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception `StopIteration` si le parcours touche à sa fin.

Python utilise deux fonctions pour appeler et manipuler les itérateurs : `iter` (qui permet d'appeler la méthode spéciale ___iter___ de l'objet passé en paramètre) et `next` (qui appelle la méthode spéciale ___next___ de l'itérateur passé en paramètre).

Itérateurs

```
>>> ch = 'diy'
>>> it = iter(ch)
>>> it
<str_iterator object at 0x7f19326179b0>
>>> next(it)
'd'
>>> next(it)
'i'
>>> next(it)
'y'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Itérateurs

Création d'un itérateur : c'est une classe, avec méthodes `__init__` et `__next__`.

Utilisée par une autre classe par l'intermédiaire de sa méthode `__iter__` ou mise en œuvre directement par la méthode `__next__` dans l'objet conteneur (dans ce cas, la méthode `__iter__` pourra renvoyer `self`). ▶ transparent suivant

Beaucoup de répétitions dans le code produit, surtout si l'on doit créer plusieurs itérateurs pour un même objet (d'où l'utilisation d'itérateurs existants, par exemple celui des listes).
▶ il existe un autre mécanisme, plus simple et plus intuitif : les générateurs.

Itérateurs

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
>>> for char in rev:
...     print(char)
```

Générateurs

Un moyen plus pratique de créer et manipuler des itérateurs, leur puissance tient surtout à leur simplicité et à leur petite taille.

Création de générateurs simples : par le mot-clé `yield`, qui ne peut s'utiliser que dans le corps d'une fonction (on définit une fonction par type de parcours) et qui est suivi d'une valeur à renvoyer.

1. la fonction commence son exécution sur demande du premier élément (grâce à la fonction `next()`),
2. à l'instruction `yield`, elle renvoie la valeur qui suit et se met en pause,
3. sur demande de l'élément suivant (`next`), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au `yield` suivant.

À la fin de l'exécution de la fonction, l'exception `StopIteration` est automatiquement levée.

Générateurs

```
def intervalle(binf, bsup):
    """generateur des entiers entre binf et bsup
       ANT : binf <= bsup"""

    while binf <= bsup:
        yield binf
        binf += 1
```

Si la méthode spéciale `__iter__` contient un appel à `yield`, alors ce sera ce générateur qui sera appelé pour parcourir la boucle.

Les générateurs utilisent (implicitement) des itérateurs, ce qui est plus confortable pour le codeur (inutile de créer une classe par itérateur, de coder une méthode `__next__`, ni même de lever l'exception `StopIteration`).

Les générateurs comme co-routines

Les générateurs produisent des valeurs, les co-routines consomment des valeurs.

Les co-routines sont un moyen d'altérer le parcours . . .pendant le parcours. Par exemple, dans notre générateur intervalle, on pourrait vouloir passer directement de 5 à 10.

Interrompre la boucle : la méthode `close` permet d'interrompre prématurément la boucle, comme le mot-clé `break` en somme.

```
generateur = intervalle(5, 20)
for nombre in generateur:
    if nombre > 15:
        generateur.close()  # interruption de la boucle
```

Les générateurs comme co-routines

Envoyer des données au générateur : le point d'échange de données se fait au mot-clé `yield`.

`yield valeur` « renvoie » `valeur` qui deviendra donc la valeur courante du parcours. La fonction se met ensuite en pause.

```
def intervalle(binf, bsup):
    while binf <= bsup:
        valeur_recue = yield binf
        binf += 1
        # le generateur a recu quelque chose
        if valeur_recue:
            binf = valeur_recue
            yield binf
```

On peut, à cet instant, altérer le fonctionnement du générateur en lui envoyant une valeur par la méthode `send` : `generateur.send(10)`.

Les générateurs comme co-routines

```
def grep(pattern):
    print("Looking for %s" % pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)

g = grep("python")
g.send(None)  # initialisation du generateur
g.send("to be or not to be")
g.send("python generators rock!")
g.close()
```

Introduction

Les décorateurs sont un moyen simple de modifier le comportement « par défaut » de fonctions ou de classes.

C'est un exemple de métaprogrammation, c'est-à-dire des programmes manipulant d'autres programmes.

Les décorateurs sont des *wrappers*, c'est à dire qu'ils permettent d'exécuter du code avant et après la fonction qu'ils décorent, sans modifier la fonction elle-même.

Une fonction associée à un décorateur ne s'exécutera pas directement mais appellera le décorateur. C'est au décorateur de décider s'il exécute la fonction et dans quelles conditions.

Décorateur artisanal

Tiré du site « Sam et Max ».

```
def decorateur(fonction_a_decorer):

    def wrapper_de_la_fonction():
        print("Avant que la fonction ne s'execute")
        fonction_a_decorer()
        print("Apres que la fonction se soit executee")

    # a ce stade, fonction_a_decorer n'a jamais ete executee
    return wrapper_de_la_fonction

def fonction_intouchable():
    print("Je suis une fonction non modifiable")

fonction_intouchable()

fonction_intouchable = decorateur(fonction_intouchable)
fonction_intouchable()
```

Format le plus simple

Les décorateurs sont des fonctions dans leur définition mais ils doivent prendre en paramètre une fonction et renvoyer une fonction.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateurs grâce à une (ou plusieurs) lignes au-dessus de la définition de fonction.

```
@decorateur
def fonction(...)
```

Le décorateur s'exécute au moment de la *définition* de fonction. Il prend en paramètre une fonction (celle qu'il modifie) et renvoie une fonction (qui peut être la même).

Premier exemple

```
def mon_decorateur(fonction):
    """premier exemple de decorateur"""
    print("Decorateur avec parametre {0}".format(fonction))
    return fonction

@mon_decorateur
def salut():
    """fonction modifiee par le decorateur"""
    print("Hello")

print(salut())
Decorateur avec parametre <function salut at 0x7f2c4d4fec80>
Hello
```

Modifier le comportement d'une fonction

Définition, dans le corps d'un décorateur, d'une fonction chargée de modifier le comportement d'une autre fonction.

```
def mon_decorateur(fonction):
    """affichage d'un message avant appel d'une
    fonction definie"""

    def fonction_modifiee():
        """fonction renvoyee
        avertissement avant execution de fonction"""

        print("Attention : appel de {0}".format(fonction))
        return fonction()  # /!\

    return fonction_modifiee

@mon_decorateur
def salut():
    print("Hello")
```

Décorateur avec paramètres

Exemple : décorateur chargé d'exécuter une fonction en contrôlant le temps qu'elle met à s'exécuter. Si elle met un temps supérieur à la durée passée en paramètre du décorateur, on affiche une alerte.

L'appel du décorateur, au-dessus de la définition de la fonction, sera de la forme :

```
@controler_temps(2.5) # 2,5 secondes maximum pour la fonction
```

Les parenthèses sont très importantes : la fonction de décorateur prendra en paramètres non pas une fonction, mais les paramètres du décorateur (ici, le temps maximum autorisé pour la fonction). Elle ne renverra pas une fonction de substitution, mais un décorateur.

Deuxième exemple

```
import time

def controler_temps(nb_secs):

    def decorateur(fonction_a_executer):
        """appele lors de la DEFINITION de la fonction"""

        def fonction_modifiee():
            """fonction renvoyee par le decorateur.
            calcul du temps mis par la fonction a s'executer"""

            tps_avant = time.time() # avant execution
            valeur_renvoyee = fonction_a_executer() # execution
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant
            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour s'executer".format( \
                        fonction_a_executer, tps_execution))
            return valeur_renvoyee

        return fonction_modifiee

    return decorateur

@controler_temps(4)
def attendre():
   input("Appuyez sur Entree...")
```

Prise en compte des paramètres

Un décorateur ne doit pas se soucier des paramètres fournis à la fonction.

- utilisation du nombre variable d'arguments.

```
def fonction_modifiee(*pnot_named, **pnamed):
  """fonction renvoyee par le decorateur"""

  tps_avant = time.time() # avant d'executer la fonction
  ret = fonction_a_executer(*pnot_named, **pnamed)
  tps_apres = time.time()
  tps_execution = tps_apres - tps_avant
  if tps_execution >= nb_secs:
    print("La fonction {0} a mis {1} pour s'executer".format( \
          fonction_a_executer, tps_execution))
  return ret
```

Décorateurs appliqués aux définitions de classes

Au lieu de recevoir en paramètre une fonction, le décorateur reçoit une classe.

```
def decorateur(classe):
    print("Definition de la classe {0}".format(classe))
    return classe

@decorateur
class Test:
    pass
```

Chaîner les décorateurs

Modification d'une fonction ou d'une définition de classe par le biais de plusieurs décorateurs.

```
@decorateur1
@decorateur2
def fonction():
```

Exemple d'application (PEP 318)

La classe singleton, une classe qui ne peut être instanciée qu'une fois.

```
def singleton(classe_definie):
    instances = {} # dictionnaire d'instances singletons
    def get_instance():
        if classe_definie not in instances:
            # creation du 1er objet de classe_definie
            instances[classe_definie] = classe_definie()
        return instances[classe_definie]
    return get_instance

@singleton
class Test:
  pass
```

Exemple d'application (PEP 318)

Contrôler les types passés à une fonction.

```
def controler_types(*a_args, **a_kwargs):
    def decorateur(fonction_a_executer):
        def fonction_modifiee(*args, **kwargs):
            """controle les types passes en parametres"""

            # longueur param attendus == param recues ?
            if len(a_args) != len(args):
                raise TypeError("nombre d'arguments attendu != recu")

            # parcours liste arg recus et non nommes
            for i, arg in enumerate(args):
                if a_args[i] is not type(args[i]):
                    raise TypeError("l'argument {0} n'est pas du type " \
                            "{1}".format(i, a_args[i]))

            # parcours liste param recus et nommes
            for cle in kwargs:
                if cle not in a_kwargs:
                    raise TypeError("l'argument {0} n'a aucun type " \
                            "precise".format(repr(cle)))
                if a_kwargs[cle] is not type(kwargs[cle]):
                    raise TypeError("l'argument {0} n'est pas de type" \
                            "{1}".format(repr(cle), a_kwargs[cle]))
            return fonction_a_executer(*args, **kwargs)
        return fonction_modifiee
    return decorateur
```

Introduction

Une classe sert à créer un objet, une métaclasse sert à créer une classe.

La méthode `__init__` est là pour initialiser un objet non pour le créer. La méthode qui s'en charge, c'est `__new__`.

Quand on tente de construire un objet :

1. On demande à créer un objet, en écrivant par exemple Personne("Doe", "John").
2. La méthode `__new__` de la classe (ici Personne) est appelée et se charge de construire un nouvel objet.
3. Si `__new__` renvoie une instance de la classe, on appelle le constructeur `__init__` en lui passant en paramètres cette nouvelle instance ainsi que les arguments passés lors de la création de l'objet.

La méthode new

C'est une méthode statique, ce qui signifie qu'elle ne prend pas `self` en paramètre car son but est de créer une nouvelle instance de classe, l'instance n'existe pas encore.

Cependant, elle prend la classe manipulée elle-même. Les autres paramètres passés à la méthode `__new__` seront transmis au constructeur.

Exemple :

```
def __new__(cls, nom, prenom):
    print("appel de __new__ de la classe {}".format(cls))
    # on laisse le travail a object
    return object.__new__(cls, nom, prenom)
```

Créer une classe dynamiquement

Les classes sont également des objets : par défaut, toutes les classes sont modelées sur la classe `type`. Cela signifie que :

1. Quand on crée une nouvelle classe (class Personne : par exemple), Python appelle la méthode `__new__` de la classe `type`.
2. Une fois la classe créée, on appelle le constructeur `__init__` de la classe `type`.

La classe type prend trois arguments pour se construire :

- le nom de la classe à créer,
- un tuple contenant les classes dont la nouvelle classe va hériter,
- un dictionnaire contenant les attributs et méthodes de la classe.

Exemple

```
def creer_personne(personne, nom, prenom):
    """fonction qui joue le role de constructeur pour
       la classe Personne"""
    personne.nom = nom
    ...

# dictionnaire des methodes
methodes = {
    "__init__": creer_personne,
    "presenter": presenter_personne,
}

# creation dynamique de la classe
Personne = type("Personne", (), methodes)
```

Définition d'une métaclasse

Une classe peut posséder une autre métaclasse que `type`.

Construire une métaclasse se fait de la même façon que construire une classe. Les métaclasses héritent de `type`.

Deux méthodes sont nécessaires :

- la méthode `__new__`, appelée pour créer une classe et qui prend quatre paramètres :
  - la métaclasse servant de base à la création de la nouvelle classe,
  - le nom de la nouvelle classe,
  - un tuple contenant les classes dont héritent la classe à créer,
  - le dictionnaire des attributs et méthodes de la classe à créer.

Définition d'une métaclasse

- la méthode `__init__`, appelée pour construire la classe et qui prend les mêmes paramètres que `__new__`, sauf le premier, qui n'est plus la métaclasse servant de modèle mais la classe que l'on vient de créer.

On précise dans la ligne de la définition de la classe qu'une classe prend comme métaclasse autre chose que `type` :

```
class MaClasse(metaclass=MaMetaClasse):
    pass
```

Rôles des métaclasses

Les métaclasses sont généralement utilisées pour des besoins assez complexes. L'exemple le plus répandu est une métaclasse chargée de tracer l'appel de ses méthodes.

Exemple plus simple, garder les classes créées dans un dictionnaire prenant comme clé le nom de la classe et comme valeur la classe elle-même.

Ainsi, en plus des widgets dont on dispose dans une bibliothèque destinée à construire des interfaces graphiques, on peut créer ses propres classes héritant des classes de la bibliothèque.

La classe mère de tous les widgets s'appellera `Widget`. De cette classe hériteront les classes `Bouton`, `CaseACocher`, `Menu`, `Cadre`, etc. L'utilisateur de la bibliothèque pourra par ailleurs en dériver ses propres classes.

Exemple

```
trace_classes = {}

class MetaWidget(type):

    """metaclasse pour les widgets
    herite de type car c'est une metaclasse.
    ecrit dans le dictionnaire trace_classes a chaque fois
    qu'une classe sera creee"""

    def __init__(cls, nom, bases, dico):
        """constructeur appele quand on cree une classe"""
        type.__init__(cls, nom, bases, dico)
        trace_classes[nom] = cls

class Widget(metaclass=MetaWidget):
    """classe mere de tous les widgets"""
    pass

class Bouton(Widget):
    """une classe definissant le widget bouton"""
    pass
```