Langages Dynamiques

Cours 5

kn@lri.fr
http://www.lri.fr/~kn

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

Syntaxe des objets

Python est avant tout un langage objet. La syntaxe pour déclarer les classes est la suivante :

```
class Circle:

    def __init__(self, x, y, r):
        self.x = x
        self.y = y
        self.radius = r

    def length(self):
        return 2 * math.pi * self.r

    def area(self):
        return math.pi * self.r ** 2

    def move(self, x, y):
        self.x += x
        self.y += y
```

Qu'y a t'il dans ce programme ?

- Une définition de classe avec le mot-clé `class`
- Des définitions de méthodes, avec le mot clé `def` (comme pour les fonctions) dans le bloc de la classe
- Un *constructeur* appelé `__init__(…)`
- Toutes les méthodes prennent en premier argument l'objet dans lequel on se trouve (pas de `this` implicite, comme en Java)

On peut en suite créer des objets avec :

```
>>> c = Circle(1.5, 1.5, 3)
>>> c.area()
28.274333882308138
>>> c.x
1.5
```

Contrôle d'accès

Il n'y a aucun contrôle d'accès aux éléments de la classe. Les attributs peuvent être librement lus et modifiés, il n'y a pas moyen de les marquer comme privés.

```
>>> c = Circle(1.5, 1.5, 3)
>>> c.area()
28.274333882308138
>>> c.radius = 0
>>> c.area()
0.0
>>> c.radius = 0
>>> c.length = "foo"
```

On peut non seulement modifier des attributs, mais aussi en ajouter et écraser des méthodes !

Méthodes statiques

On peut utiliser un *décorateur* pour déclarer des méthodes comme statiques, i.e. appartenant à la classe et non pas à l'objet.

```
class Circle:
      ...   # tout comme avant

      @classmethod
      def readFromFile(cls, path):
          with open(path) as f:
             line = f.readline().strip()
             x, y, r = line.split(",")
             return cls(x, y, r)          #Équivalent à Circle(...)
```

Héritage

L'héritage s'indique assez naturellement au moment de la déclaration de la classe. Si on imagine une classe `Shape` dont on souhaite hériter :

```
class Circle (Shape):

      def __init__(self, x, y, r):
          super().__init__(self, x, y)
          self.radius = r

      ...
```

La fonction `super()` renvoie la classe parente de celle dans la quelle on se trouve. Donc `super().__init__(self, ...)` appelle le constructeur de la classe parente sur l'objet self que l'on est en train d'initialiser.

Héritage multiple

L'héritage multiple est possible, il suffit d'ajouter plusieurs classes parentes :

```
class Circle (Shape, Colored):

      def __init__(self, x, y, r, c):
          Shape.__init__(self, x, y)
          Colored.__init__(self, c)
          self.radius = r

      ...
```

Attention, si deux classes parentes possèdent les même méthodes, l'ordre de résolution est important. On peut connaître l'ordre de résolution avec la méthode `.mro()` (*method résolution order*) d'une classe.

MRO

```
class A:
      def __init__(self):
          pass
      def f(self):
          print ("Dans A")

class B:
      def __init__(self):
          pass
      def f(self):
          print ("Dans B")

class C(A, B):
      def __init__(self):
          pass
```

```
>>> c = C()
>>> c.f()
Dans A
>>>  C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
            <class 'object'>]
```

Conclusion Objets

- Python est un langage Orienté Objet « classique » (méthodes, attributs, héritage, ...).
- La bibliothèque standard fait une grande utilisation des objets
- Les propriétés usuelles ne sont pas disponibles (private, public, ...) ⇒ Attention, on peut facilement casser son code.

Plan

Souvenons nous ...

$$\{ x \in \mathbb{N} \mid x \bmod 2 = 0 \}$$

Notation mathématique *compacte* qui permet de définir l'ensemble des éléments *d'un autre ensmble* qui *vérifient une propriété*.

$$\{ (x,y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1 \}$$

« L'ensembles des points dont la norme vaut 1 » (i.e. le cercle de rayon 1)

C'est une notation très puissante, compacte et *déclarative* (elle exprime ce qu'on veut, pas comment on le calcule).

Tableaux en compréhension

Python permet de définir des tableaux en compréhension :

```
pairs = [ x for x in range(100) if x % 2 == 0 ]
#pairs vaut [ 0, 2, 4, 6, 8, … , 98 ]
```

La syntaxe (simplifiée) est :

```
[ e_r for x in e_l if c ]
```

où

- e_l doit être une expression renvoyant un tableau (directement, ou une fonction qui renvoie un tableau, ou `range`, ...)
- e_r est une expression résultat qui peut contenir la variable `x`
- `x` est une variable qui va prendre tour à tour les valeurs des éléments de e_l
- `c` est une expression booléenne (qui peut utiliser `x`) si elle est fausse, la valeur de `x` n'est pas utilisée et on passe à la suivante.

Tableaux en compréhension (2)

Python permet de définir des tableaux en compréhension :

Le code :

```
[ er for x in el if c ]
```

est équivalent à :

```
res = []
for x in el:
    if c:
        res = res + [er]
```

Exemple :

```
>>> [ (x-1, x+1) for x in range(10) if x % 3 = 0 ]
[(-1, 1), (2, 4), (5, 7), (8, 10)]
>>>
```

Utilité

Un tableau donné par comprhénsion permet :

- de parcourir un ensemble de valeur
- de le filtrer pour garder des valeurs pertinentes
- de recombiner les données dans une liste résultante

C'est une opération tellement courante que les concepteurs de Python on jugé nécessaire de la mettre dans le langage.

Plan

Fonctions anonymes

Python supporte une syntaxe pour la déclaration de fonctions anonymes :

```
lambda x,y,z: x+y+z
```

```
lambda x : x < 10
```

Ces fonctions anonymes sont des *expressions*. Elle permettent d'éviter la création d'une (petite) fonction et peuvent rendre le code plus lisible. Par exemple:

```
data = ... # tableau de dictionnaires de la forme { "annee": int, "titre": str }

s_date = sorted(data, key=lambda d: d["annee"])
```

Ici la fonction sorted prend en argument supplémentaire une fonction permettant de dire comment extraire la clé de tri à partir d'une entrée du tableau.

Itérateurs

Les fonctions anonymes sont particulièrement utiles avec les itérateurs :`map` (builtins), `filter` (builtins), `reduce` (module `functools`)

```
from functools import reduce

data = .... # un tableau d'entiers

data_even = filter (lambda x: x % 2 == 0, data)
data_square = map (lambda x: x**2, data)
data_prod = reduce(lambda x, y: x * y, data, 1)
```

Fichiers CSV