Python
Les classes

Nicolas Delestre

INSA
ROUEN NORMANDIE

Classe

Déclaration

- Syntaxe :

```
class NomClasse[(SuperClasse,SuperClasse2,...)]:
""" Documentation """
    def __init__(self[, param1, param2, ...]):
        """ Documentation """
        self.attr1 = ...
        self.attr2 = ...

    def m1(self[, param1, param2, ...]):
        ...
```

- On définit une classe dans un module (un module peut contenir plusieurs classes). On importe une classe, comme on importe une fonction
- L'identifiant d'une classe est en *CamelCase*
- La classe au sommet de l'héritage est la classe `object` qui n'est pas précisée si c'est la superclasse de la classe à définir
- Il n'y a pas de classe abstraite

Attributs, méthodes

- C'est le faite d'affecter une valeur à un attribut (affectation), que l'attribut est ajouté à l'instance
- La bonne pratique veut que l'*initialiseur*[a] (`__init__`) initialise tous les attributs d'instance
- Il n'y a pas de mot clé pour la visibilité des méthodes ou des attributs, mais si l'identifiant commence par un :
  - `_` alors il doit être considéré comme privé (il n'apparaît pas dans la documentation, `help`)
  - `__` alors il est privé et non accessible (sans faire d'introspection)
- Comme les paramètres formels ne sont pas typés, il n'y a pas de surcharge (polymorphisme)

a. très souvent dénommé, à tord, *constructeur*. Le véritable constructeur est la méthode `__new__` que l'on ne redéfinit que si on fait de la méta-programmation

Classe

Classe Point2D (sans la documentation) du module point.py

```
 5 class Point2D:
 6     def __init__(self, x: float, y: float, identifiant: str=None):
 7         self._x = x
 8         self._y = y
 9         self._id = identifiant
10
11     def get_x(self) -> float:
12         return self._x
13
14     def set_x(self, x: float):
15         self._x = x
16
17     def get_y(self) -> float:
18         return self._y
19
20     def set_y(self, y: float):
21         self._y = y
22
23     def get_id(self) -> str:
24         return self._id
```

Objet

Création d'un objet

- On crée un objet en utilisant l'identifiant de la classe suivi, entre parenthèses, des paramètres effectifs à donner à l'initialiseur (pour les paramètres formels qui suivent le `self`)

```
>>> from point import Point2D
>>> pt1 = Point2D(1,2)
```

Invocation d'une méthode

- On invoque habituellement une méthode `m` sur un objet `o` en utilisant la notation pointée classique : `o.m(...)`
- On peut aussi invoquer une méthode comme on appelle une fonction, en préfixant l'identifiant de la méthode par l'identifiant de la classe et en mettant en premier paramètre effectif l'objet

```
>>> pt1.get_x()
1
>>> Point2D.get_x(pt1)
1
```

Héritage

Principe

- Python accepte l'héritage multiple
- On peut redéfinir le comportement des méthodes (polymorphisme par héritage) : on accède à la méthode de la super classe en l'invoquant à l'aide de la deuxième notation
- La bonne pratique veut que l'on désigne la super classe à l'aide de la fonction `super`
  Cf. https://stackoverflow.com/questions/42413670/whats-the-difference-between-super-and-parent-class-name

Classe `Point3D` (sans la documentation) du module `point.py`

```
26 class Point3D(Point2D):
27     def __init__(self, x: float, y: float, z: float, identifiant: str=None):
28         super().__init__(x, y, identifiant)
29         self._z = z
30 
31     def get_z(self) -> float:
32         return self._z
33 
34     def set_z(self, z: float):
35         self._z = z
```

Liaison dynamique de méthode

Principe

- Python réalise la liaison dynamique de méthode : la méthode sélectionnée suite à l'invocation d'une méthode est toujours celle qui est la plus proche (au sens de l'héritage) de la classe de l'objet

```
3  class A:
4      def m1(self):
5          print("m1 de A")
6          self.m2()
7
8      def m2(self):
9          print("m2 de A")
10
11 class B(A):
12     def m2(self):
13         print("m2 de B")
```

```
>>> b=B()
>>> b.m1()
m1 de A
m2 de B
>>> a=A()
>>> a.m1()
m1 de A
m2 de A
```

Algorithme de sélection de méthode dans le cas d'un héritage multiple

- L'algorithme utilise un tri topologique tel que les classes sont étudiées dans l'ordre de déclaration de l'héritage multiple
- L'attribut `__mro__` (*method resolution order*) permet de connaître la liste issue de ce tri

```
class A1:
    def m(self):
        print("m de A1")


class A2:
    def m(self):
        print("m de A2")

class B(A1, A2):
    pass
```

```
>>> B().m()
m de A1
```

```
class A1:
    def m(self):
        print("m de A1")


class A2:
    def m(self):
        print("m de A2")

class B(A2, A1):
    pass
```

```
>>> B().m()
m de A2
```

```
class A:
    def m(self):
        print("m de A")

class B1(A):
    pass

class B2(A):
    def m(self):
        print("m de B2")

class C(B1, B2):
    pass
```

```
>>> C().m()
m de B2
```

```
>>> C.__mro__
(<class 'heritage_multiple_3.C'>, <class 'heritage_multiple_3.B1'>,
<class 'heritage_multiple_3.B2'>, <class 'heritage_multiple_3.A'>, <class 'object'>)
```

Retour sur `__init__`

- Le comportement de la méthode `__init__` est classique

```
class A:
    def __init__(self, a):
        self._a = a

class B(A):
    pass

class C(A):
    def __init__(self, c):
        self._c = c

class D(A):
    def __init__(self, a, d):
        super().__init__(a)
        self._d = d
```

```
>>> a=A(1)
>>> a._a
1
>>> b=B(1)
>>> b._a
1
>>> c=C(1)
>>> c._a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '_a'
>>> d=D(1,2)
>>> d._a
1
>>> d._d
2
```

Conclusion

- Comment on définit une classe
- Python réalise de la liaison dynamique de méthode
- Python accepte l'héritage multiple avec une recherche des attributs et méthodes à l'aide d'un tri topologique
- L'initialiseur `__init__` est une méthode comme les autres

INSA ROUEN NORMANDIE