

Séance 3

Classe et objet



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Tuple
 - Séquence ordonnée non modifiable d'éléments
 - Emballage et déballage, affectation multiple
 - Tuple nommé
- Objet
 - Création d'un objet
 - Accès aux attributs et appels de méthode
 - Programmation orientée objet

Objectifs

- Définition de **classes**
 - Définition, constructeur et initialisation
 - Définition et appel de méthode
- Programmation **orientée objet**
 - Méthodes “spéciales” (égalité et représentation d’objets)
 - Visibilité des attributs, encapsulation
 - Conception orientée objet

Classe



Objet et classe

- Un objet est une **instance** d'une classe

Une classe est un modèle à partir duquel on construit des objets

- La classe définit **deux éléments** constitutifs des objets

Les attributs et les fonctionnalités de l'objet



Marque : Sony-Ericsson
Modèle : S500i
Couleur : Mysterious Green
Batterie : 80%
Luminosité : 60%



Marque : Sony-Ericsson
Modèle : S500i
Couleur : Spring Yellow
Batterie : 35%
Luminosité : 90%

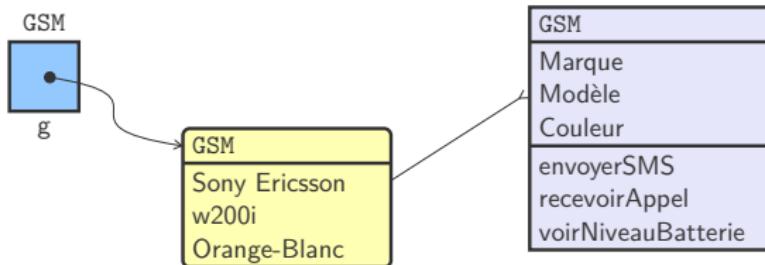
Attribut et fonctionnalité

- Un attribut est une **donnée** stockée dans un objet

Les valeurs des attributs définissent l'état de l'objet

- Une **fonctionnalité** permet d'effectuer une action

Obtenir une information sur l'objet ou donner un ordre



Définir une classe

- Définition d'une classe avec le mot réservé `class`

La classe contient des méthodes

- Initialisation d'un objet par la méthode `__init__`

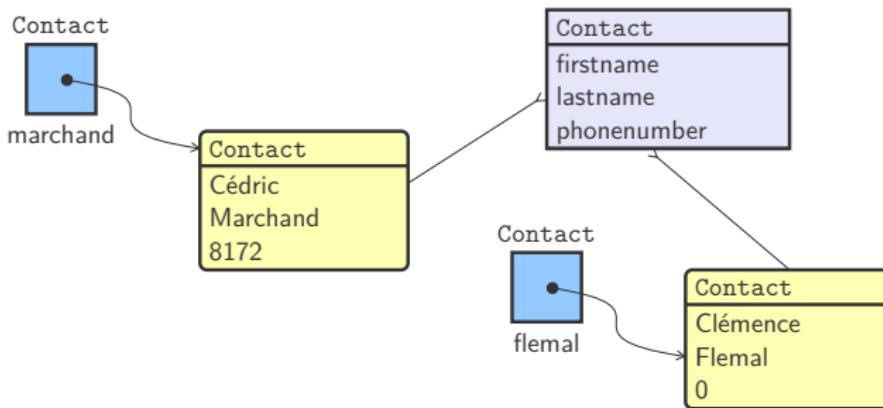
Le paramètre `self` référence l'objet à construire

```
1 class Contact:  
2     def __init__(self, firstname, lastname, phonenumbers):  
3         self.firstname = firstname  
4         self.lastname = lastname  
5         self.phonenumbers = phonenumbers
```

Constructeur

- Méthode appelée au moment de la **création d'un objet**
- **Initialise l'objet**, en donnant une valeur à ses variables

```
1 marchand = Contact("Cédric", "Marchand", 2693)
2 flemal = Contact("Clémence", "Flemal", 0)
```



Objet

- Un objet est une **instance** d'une classe

L'instanciation d'une classe produit un objet

- Stockage d'une **référence** vers l'objet

L'adresse où l'objet se situe en mémoire

```
1 marchand = Contact("Cédric", "Marchand", 2693)
2 print(marchand)
```

```
<__main__.Contact object at 0x10066ae48>
```

Variable d'instance

- **Variables d'instance** attachées à un objet définissent son état
Chaque objet possède ses propres copies de ces variables
- Accès aux variables d'instance avec l'**objet cible**
Ou self à l'intérieur du code de la classe

```
1 print(marchand.firstname)
2 print(flemal.phonenumber)
```

```
Cédric
0
```

Méthode

- Méthode attachée à un objet réalisant une action dessus
La méthode reçoit d'office un paramètre self, l'objet cible
- Appel d'une méthode sur un objet cible avec le point (.)

```
1 class Contact:  
2     def __init__(self, firstname, lastname):  
3         self.firstname = firstname  
4         self.lastname = lastname  
5         self.phonenumber = 0  
6  
7     def setphonenumber(self, number):  
8         self.phonenumber = number  
9  
10 marchand = Contact("Cédric", "Marchand")  
11 marchand.setphonenumber(2693)
```

Plusieurs constructeurs

- Paramètre optionnel pour offrir **plusieurs constructeurs**

Définit une valeur par défaut pour les variables d'instance

```
1 class Contact:  
2     def __init__(self, firstname, lastname, phonenum=0):  
3         self.firstname = firstname  
4         self.lastname = lastname  
5         self.phonenum = phonenum  
6  
7     def setphonenum(self, number):  
8         self.phonenum = number  
9  
10    marchand = Contact("Cédric", "Marchand", 2693)  
11    flemal = Contact("Clémence", "Flemal")
```

Définir un vecteur dans le plan (1)

- Deux variables d'instance pour représenter les **coordonnées**
- Une méthode `norm` pour calculer la **longueur du vecteur**

```
1 from math import sqrt
2
3 class Vector:
4     def __init__(self, x, y):
5         self.x = x
6         self.y = y
7
8     def norm(self):
9         return sqrt(self.x ** 2 + self.y ** 2)
10
11 u = Vector(1, -1)
12 print(u.norm())
```

Définir une musique

- Un titre, une liste d'artistes et une durée
- Méthode hasArtist teste si un artiste a composé la musique

```
1 class Music:  
2     def __init__(self, title, artists, duration):  
3         self.title = title  
4         self.artists = artists  
5         self.duration = duration  
6  
7     def hasAuthor(self, name):  
8         return name in self.artists  
9  
10 m1 = Music('Easy Come Easy Go', ['Alice on the roof'], 213)  
11 print(m1.hasAuthor('Stromae'))
```

False

A photograph of a large stack of red and white striped popcorn boxes, tilted at an angle. The boxes are stacked horizontally, creating a tall, rectangular structure. The word "POPCORN" is printed vertically on the side of the top box. The boxes are resting on a light-colored, polished floor. Scattered around the base of the stack are numerous small, colorful plastic Easter eggs in various colors like yellow, green, red, and orange. In the background, there's some dark furniture and a sign that says "incase".

incase

Programmation orientée objet

Représentation d'un objet

- La méthode `__str__` construit une **représentation de l'objet**

Renvoie une chaîne de caractères lisible de l'objet

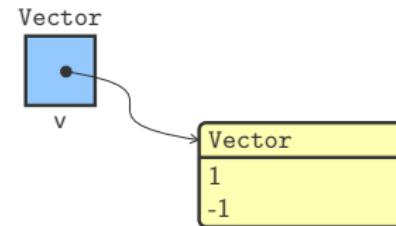
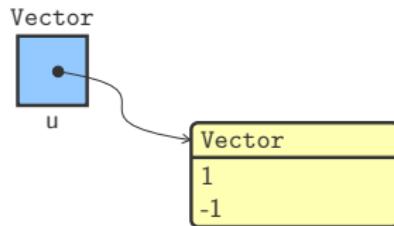
```
1 class Vector:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5  
6     def __str__(self):  
7         return '(' + str(self.x) + ', ' + str(self.y) + ')'  
8  
9 u = Vector(1, -1)  
10 print(u)
```

```
(1, -1)
```

Égalité (1)

- L'opérateur d'égalité **compare les références** des variables
Le contenu des objets n'est pas comparé

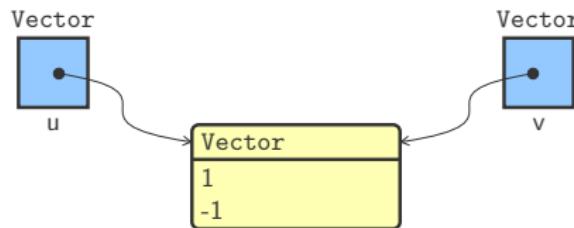
```
1 u = Vector(1, -1)
2 v = Vector(1, -1)
3 print(u == v)                      # False
```



Alias

- Un **alias** est une copie de la référence vers un objet
Il n'y a qu'une seule copie de l'objet en mémoire

```
1 u = Vector(1, -1)
2 v = u
3 print(u == v)                      # True
```



Surcharge d'opérateur (1)

- On peut **redéfinir** les opérateurs arithmétiques

`--add-- pour +, --sub-- pour -, --mul-- pour *...`

```
1 class Vector:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __add__(self, other):
7         return Vector(self.x + other.x, self.y + other.y)
8
9     # ...
10
11 u = Vector(1, -1)
12 v = Vector(2, 1)
13 print(u + v)
```

(3, 0)

Surcharge d'opérateur (2)

- On peut **redéfinir** les opérateurs de comparaison

`--lt-- pour <, --le-- pour <=, --eq-- pour ==...`

```
1 class Vector:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __lt__(self, other):
7         return self.x < other.x or (self.x == other.x and self.y <
8             other.y)
9
# ...
10
11 u = Vector(1, -1)
12 v = Vector(2, 1)
13 print(u < v)
```

False

Définir un vecteur dans le plan (2)

- Une seule variable d'instance pour les coordonnées

Stockée dans un tuple par exemple

```
1 from math import sqrt
2
3 class Vector:
4     def __init__(self, x, y):
5         self.coords = (x, y)
6
7     def norm(self):
8         return sqrt(self.coords[0] ** 2 + self.coords[1] ** 2)
9
10 u = Vector(1, -1)
11 print(u.norm())
```

Égalité (2)

- Surcharge de l'opérateur d'égalité pour **comparer les objets**

Le contenu des objets sera comparé, et non plus les références

- **Comparaison des identités** avec l'opérateur **is**

Comparaison des références des objets

```
1 from math import sqrt
2
3 class Vector:
4     # ...
5
6     def __eq__(self, other):
7         return self.__coords == other.__coords
8
9 u = Vector(1, -1)
10 v = Vector(1, -1)
11 print(u == v)                      # True
12 print(u is v)                      # False
```

Encapsulation (1)

- Les données de l'objet sont **encapsulées** dans l'objet

Ne pas dévoiler les détails d'implémentation en dehors de l'objet

- **Pas d'accès direct** aux variables d'instance

Pas recommandé d'accéder directement aux variables d'instance

```
1 u = Vector(1, -1)
2 v = Vector(2, 1)
3
4 s = Vector(u.x + v.x, u.y + v.y)
5 # ou
6 # s = Vector(u.coords[0] + v.coords[0], u.coords[0] + v.coords[0])
```

Variable privée

- Variable privée en préfixant son nom avec __

Ne pourra pas être accédée en dehors de la classe

```
1 from math import sqrt
2
3 class Vector:
4     def __init__(self, x, y):
5         self.__x = x
6         self.__y = y
7
8 u = Vector(1, -1)
9 print(u.__x)
```

```
Traceback (most recent call last):
  File "program.py", line 9, in <module>
    print(u.__x)
AttributeError: 'Vector' object has no attribute '__x'
```

Accesseur

- Accès à une variable privée à l'aide d'un **accesseur**

Méthode qui renvoie une variable d'instance

- Un accesseur se définit avec la **décoration** `property`

```
1 from math import sqrt
2
3 class Vector:
4     # ...
5
6     @property
7     def x(self):
8         return self.__x
9
10 u = Vector(1, -1)
11 print(u.x)                      # 1
```

Mutateur

- Modification d'une variable privée à l'aide d'un **mutateur**

Méthode qui change la valeur d'une variable d'instance

- Un mutateur se définit avec la **décoration** `nom.setter`

```
1 from math import sqrt
2
3 class Vector:
4     # ...
5
6     @x.setter
7     def x(self, value):
8         self.__x = value
9
10 u = Vector(1, -1)
11 u.x = 12
12 print(u.x)                      # 12
```

Encapsulation (2)

- Accès à un objet **uniquement via les méthodes publiques**

Utilisateur indépendants de la représentation interne de l'objet

```
1 from math import sqrt
2
3 class Vector:
4     def __init__(self, x, y):
5         self.__coords = (x, y)
6
7     @property
8     def x(self):
9         return self.__coords[0]
10
11    @x.setter
12    def x(self, value):
13        self.__coords = (value, self.__coords[1])
14
15 u = Vector(1, -1)
16 u.x = 12
17 print(u.x)                      # 12
```

Interface

- L'**interface publique** d'un objet expose ses fonctionnalités

Définit ce que l'utilisateur peut faire avec l'objet

- Interface publique de la classe Vector

- Une variable d'instance coords privée
- Un accesseur et un mutateur pour la coordonnée x
- Une méthode coords publique

Vector
-coords
+x
+norm()

Composition d'objets

- On peut **composer** plusieurs objets ensemble

En utilisant des variables d'instance de type objet

```
1 class Rectangle:
2     def __init__(self, lowerleft, width, height, angle=0):
3         self.__lowerleft = lowerleft
4         self.__width = width
5         self.__height = height
6         self.__angle = angle
7
8     @property
9     def lowerleft(self):
10        return self.__lowerleft
11
12 p = Vector(1, -1)
13 r = Rectangle(p, 100, 50)
14 print(r.lowerleft)          # (1, -1)
```

Réutilisation de code

- On peut réutiliser le code défini pour les objets composés

Il suffit d'appeler les méthodes des variables objet

```
1 class Rectangle:  
2     # ...  
3  
4     def __str__(self):  
5         return "Rectangle en " + str(self.__lowerleft) + " de  
6             longueur " + str(self.__width) + " et de hauteur " + str(  
7                 self.__height) + " incliné de " + str(self.__angle) + "  
8                 degrés"  
9  
r = Rectangle(Vector(1, -1), 100, 50)  
10 print(r)
```

```
Rectangle en (1, -1) de longueur 100 et de hauteur 50 incliné de  
0 degrés
```

Chaine de caractères formatée

- Chaine de caractères **formatée** à partir de valeurs
 - Incrustation de valeurs définie avec des balises {}
 - Valeurs à incruster passées en paramètres de format
- **Même nombre** de balises que de valeurs passée en paramètres

Sinon, l'interpréteur Python produit une erreur d'exécution

```
1 class Rectangle:  
2     # ...  
3  
4     def __str__(self):  
5         return "Rectangle en {} de longueur {} et de hauteur {}  
6         incliné de {} degrés".format(self.__lowerleft, self.__width  
7             , self.__height, self.__angle)  
8  
r = Rectangle(Vector(1, -1), 100, 50)  
print(r)
```

Crédits

- <https://www.flickr.com/photos/booleansplit/3510951967>
- <https://www.flickr.com/photos/goincase/492906260>