

B201A Informatique appliquée

Séance 3

Classe et objet

Sébastien Combéfis, Quentin Lurkin

2017–2018



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
- Définition et utilisation de tuples nommés

■ Objet

- Création d'un objet
- Accès aux attributs et appels de méthode
- Introduction à la programmation orientée objet

Objectifs

- Définition de **classes**
 - Définition, constructeur et initialisation
 - Variable d'instance
 - 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



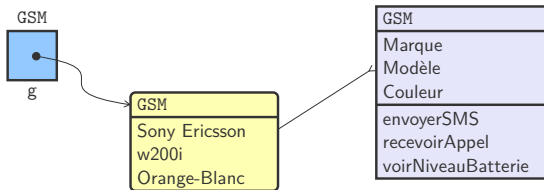
Attribut et fonctionnalité

- 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



Utilisation d'un objet

- Pour pouvoir créer des objets, il faut une **classe**

Une définition unique permet de créer plusieurs objets

- Une fois créée, interaction avec **attribut et fonctionnalité**

Utilisation de l'opérateur d'accès/appel sur l'objet

```
1 # Construction d'objets
2 maxime = Person("Maxime", "Hockey")
3 elise = Person("Elise", "Space")
4
5 # Accès à un attribut
6 print(maxime.firstname)           # Affiche 'Hockey'
7
8 # Appel d'une méthode
9 print(maxime.hasfriend(elise))    # Affiche 'True'
```


Définir une classe

- **Définition** d'une classe avec le mot réservé `class`
 - Corps de la classe est un bloc de code indenté
 - Le corps de la classe peut contenir des définitions de méthodes
- Classe minimale grâce à l'**instruction `pass`**

Aussi appelée instruction vide car ne fait rien

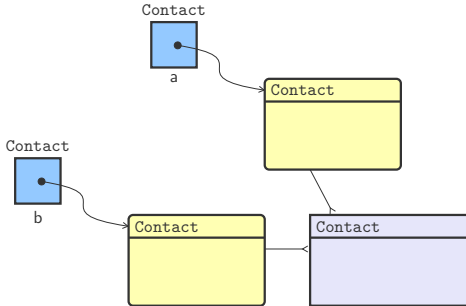
```
1 class Contact:  
2     pass
```

Créer une instance

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

À partir d'une classe, on crée autant d'objets que l'on veut

```
1 a = Contact()  
2 b = Contact()
```



Définir un constructeur

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

Admet au moins un paramètre qui est `self`

- Le paramètre `self` référence l'**objet à construire**

Permet d'accéder aux attributs et fonctionnalités de l'objet

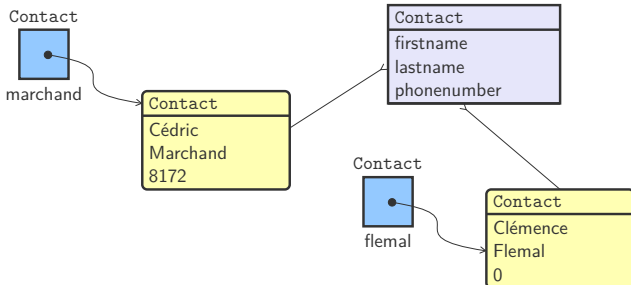
```
1 class Contact:
2     def __init__(self, firstname, lastname, phonenumber):
3         self.firstname = firstname
4         self.lastname = lastname
5         self.phonenumber = phonenumber
```

Appeler un 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 et référence

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

L'instanciation d'une classe produit un objet

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

L'adresse où l'objet se situe en mémoire

```
1 print(marchand)
2 print(flemal)
```

```
<__main__.Contact object at 0x109678748>
<__main__.Contact object at 0x109678780>
```

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
```

Plusieurs constructeurs

- Paramètre optionnel pour offrir **plusieurs constructeurs**
 - Définit une valeur par défaut pour les variables d'instance
 - D'abord les obligatoires, puis les optionnelles

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

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 méthodes

- Comme pour le constructeur pour avoir **plusieurs méthodes**

Utilisation de la valeur spéciale `None` comme valeur par défaut

```
1 class Contact:
2     # [...]
3     def changename(self, firstname=None, lastname=None):
4         if firstname is not None:
5             self.firstname = firstname
6         if lastname is not None:
7             self.lastname = lastname
```

- `lurkin.changename('John', 'Doe')`
- `lurkin.changename('John')`
- `lurkin.changename(lastname='Doe')`

Définir un vecteur dans le plan (1)

- Deux variables d'instance pour représenter les coordonnées

Les deux variables `self.x` et `self.y` représentent (x, y)

- Une méthode `norm` pour calculer la longueur du vecteur

La norme vaut $\sqrt{x^2 + y^2}$

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

1.4142135623730951

Exemple : 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

Exemple : Définir une personne

- Deux éléments particuliers à relever dans la classe Person
 - Variables d'instance pas en paramètre du constructeur
 - Une méthode ne renvoie pas forcément quelque chose

```
1 class Person:
2     def __init__(self, firstname, lastname):
3         self.firstname = firstname
4         self.lastname = lastname
5         self.friends = []
6
7     def addfriend(p):
8         self.friends.append(p)
9
10    def hasfriend(p):
11        return p in self.friends
```

Mot réservé self

- La **variable d'instance** est accessible dans toute la classe

Existe en mémoire pendant toute la durée de vie de l'objet

- Opposée à la **variable locale** qui n'existe que dans la méthode

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

```
Traceback (most recent call last):
  File "program.py", line 38, in <module>
    print(u.norm())
  File "program.py", line 35, in norm
    return sqrt(x ** 2 + y ** 2)
NameError: name 'x' is not defined
```

Fonction vs méthode (1)

- Une **méthode** est une fonction associée à un objet
 - La méthode peut agir sur les variables d'instance de l'objet
 - La méthode est appelée sur un objet cible
- Fonction agissant sur un **tuple nommé**

Tuple passé en paramètre de manière explicite

```
1 Vector = namedtuple('Vector', ['x', 'y'])
2
3 def norm(v):
4     return sqrt(v.x ** 2 + v.y ** 2)
5
6 u = Vector(1, -1)
7 print(norm(u))
```

Fonction vs méthode (2)

- Une **méthode** est une fonction associée à un objet
 - La méthode peut agir sur les variables d'instance de l'objet
 - La méthode est appelée sur un objet cible
- Méthode agissant sur un **objet cible**

Objet cible passé de manière implicite

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

Résumé : définition d'une classe

- **Deux éléments** à définir dans le corps d'une classe
 - Le constructeur initialise les variables d'instance
 - Les méthodes interrogent ou agissent sur l'objet
- **Plusieurs** constructeurs et versions d'une méthode

À l'aide des paramètres optionnels

```
1 class NomDeLaClasse:
2     def __init__(self, pc1, pc2, ...):
3         self.varinst1 = pc1
4         self.varinst2 = pc2
5         self.autrevarinst = valeur
6
7     def methode(self, pm1, pm2, ...):
8         # ... agir sur l'objet ...
```


Résumé : utilisation d'un objet

- **Création d'un objet** à partir du nom de la classe
 - Appel implicite du constructeur (`__init__`)
 - Même nombre de valeurs (vc_i) que de paramètres (pc_i)
- Stockage d'une **référence vers l'objet** dans une variable

Utilisation de cette variable pour agir sur l'objet

```
1 var = NomDeLaClasse(vc1, vc2, ...)  
2  
3 var.methode(vm1, vm2, ...)
```



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)
```

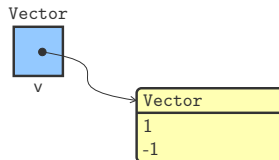
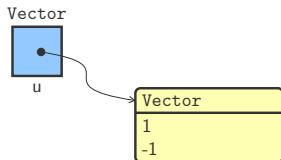
(1, -1)

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
```

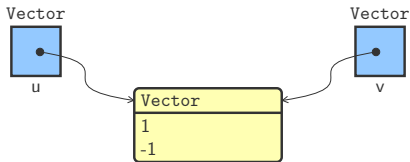


Alias (1)

- 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
```



Alias (2)

■ Qu'affiche le code suivant après exécution ?

- magic1 appelle une méthode sur data
- magic2 modifie la variable locale data

```
1 def magic1(data):  
2     data.append("Coucou")  
3  
4 def magic2(data):  
5     data = "Coucou"  
6  
7 a = [0, 1, 2, 3]  
8 print(a)  
9  
10 magic1(a)  
11 print(a)  
12  
13 magic2(a)  
14 print(a)
```

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
12 u = Vector(1, -1)
13 v = Vector(2, 1)
14 print(u < v)
```

True

Définir un vecteur dans le plan (2)

- Une seule variable d'instance pour les **coordonnées**

Stockée dans un tuple par exemple

- **Choix d'implémentation** complètement libres

Il n'y a pas qu'une seule solution unique

```
1 class Vector:
2     def __init__(self, x, y):
3         self.coords = (x, y)
4
5     def norm(self):
6         return sqrt(self.coords[0] ** 2 + self.coords[1] ** 2)
7
8 u = Vector(1, -1)
9 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 class Vector:
2     # ...
3
4     def __eq__(self, other):
5         return self.__coords == other.__coords
6
7 u = Vector(1, -1)
8 v = Vector(1, -1)
9 print(u == v)           # True
10 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
6 # ou
7 # s = Vector(u.coords[0] + v.coords[0], u.coords[1] + v.coords[1])
```

Variable privée

- Variable privée en préfixant son nom avec `__`

Ne pourra pas être accédée en dehors de la classe

```
1 class Vector:
2     def __init__(self, x, y):
3         self.__x = x
4         self.__y = y
5
6 u = Vector(1, -1)
7 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`

L'appel se fait comme si c'était une variable d'instance publique

```
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`

Où `nom` est celui de la variable à modifier

```
1 class Vector:
2     # ...
3
4     @x.setter
5     def x(self, value):
6         self.__x = value
7
8 u = Vector(1, -1)
9 u.x = 12
10 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 class Vector:
2     def __init__(self, x, y):
3         self.__coords = (x, y)
4
5     @property
6     def x(self):
7         return self.__coords[0]
8
9     @x.setter
10    def x(self, value):
11        self.__coords = (value, self.__coords[1])
12
13 u = Vector(1, -1)
14 u.x = 12
15 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** définit 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
10 r = Rectangle(Vector(1, -1), 100, 50)
11 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 passées en paramètre

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
9 r = Rectangle(Vector(1, -1), 100, 50)
10 print(r)
```

Crédits

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