UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Licence informatique & vidéoludisme
Semestre 1

Méthodologie de la programmation

Chapitre 2

Programmation orientée objet avec python

1. Les objets en python

Objets

Python supporte nativement certains types :

- entiers
- flottants
- chaînes de caractères
- listes
- dictionnaires

chaque instance de ces types est un objet :

0.25, {'Name' : 'Alice', 'Age' : '31'}, ['A','B','C'], "chaîne", etc.

Objets

Chaque objet d'un type donné dispose :

- d'une implémentation qui donne sa **structure interne**
- d'un ensemble de méthodes permettant d'intéragir avec lui, qui régissent son **comportement** : l'interface

Programmation orientée objet (POO)

- un objet représente un concept, une idée, ou toute entité du monde physique ;
- en python, tout est objet ;
- un langage qui permet la POO permet de :
  - créer de nouveaux objets ;
  - les manipuler ;
  - les détruire.

Programmation orientée objet (POO)

- Tous les langages ne sont pas orientés objet même si on peut faire de la programmation objet dans n'importe lequel.
- Certains langages offrent un support natif de la POO : C++, Swift, Java, PHP, Go, Rust, Javascript, Python, ...
- Mais pas tous de la même façon :
  - Avec des classes qu'on instancie (C++, Java, Python)
  - Avec des prototypes qu'on clone (Self, JavaScript)
  - D'autres choses moins clairement nommées (Go)
- Avec différentes façons de typer :
  - Typage fort ou faible
  - Typage statique ou dynamique

Exemple d'objet : la liste

Implémentation

```
1  // listobject.h
2  #include <stdio.h>
3
4  typedef struct {
5      PyObject_HEAD
6      Py_ssize_t ob_size;
7      /* Vector of pointers to list elements. list[0] is ob_item[0], etc.
           */
8      PyObject **ob_item;
9      /* ob_item contains space for 'allocated' elements. The number
10     * currently in use is ob_size.
11     * 0 <= ob_size <= allocated
12     * len(list) == ob_size
13     * ob_item == NULL implies ob_size == allocated == 0
14     */
15     Py_ssize_t allocated;
16 } PyListObject;
```

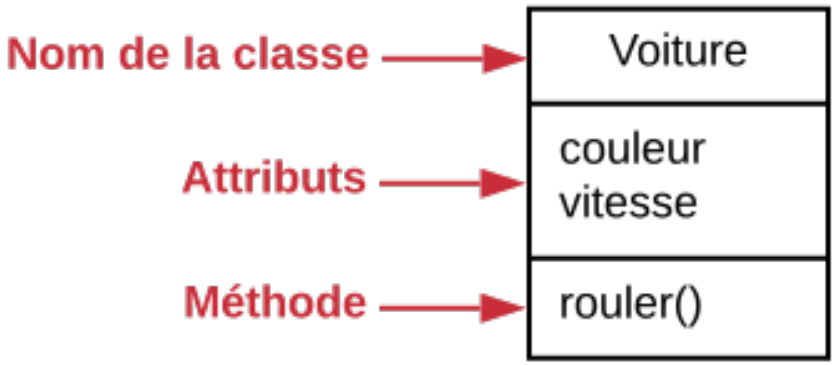
Exemple d'objet : la liste

Méthodes prédéfinies

- L[i], L[i : j], +
- len(), min(), max(), del(L[i])
- L.append(), L.extend(), L.count(), L.index(), L.insert(), L.pop() ,L.remove(), L.reverse(), L.sort(), etc.

Classe en python

Nom et variables



Classe en python : implémentation

- mot clé **class**
- nom : Point
- argument : **superclasse** dont la classe courante hérite

```
1 class Point(object):
2     """
3     attributs et méthodes
4     """
```

les attributs sont partagés par toutes les instances de votre classe

Classe en python : implémentation

```
1 class Point(object):
2     x = 0
3     y = 42
```

les attributs sont partagés par toutes les instances de votre classe

Objets

Chaque objet d'un type donné dispose :

- d'une implémentation qui donne sa **structure interne**
- d'un ensemble de méthodes permettant d'intéragir avec lui, qui régissent son **comportement** : l'interface

Encapsulation

Définition et raison d'être

implémentation et **interface** sont dissociées

- obligation de passer par des **méthodes** pour modifier les **attributs** d'un objet
- protection des représentations internes (privées)
- création de briques logicielles facilement réutilisables (c'est le cas de nombreux modules)
- **interface stable** du point de vue de l'utilisateur, même si l'implémentation change

  … un code peut « casser » si l'implémentation d'une classe change

Encapsulation

Mise en œvre

3 niveaux de restriction de la « visibilité d'une variable »

- publique : visible par tous
- privée : visible par les méthodes définies au sein de la classe
  - convention : __variable-privee__
- protégée (lié à la notion d'héritage et de hiérarchie des classes)
  - convention : _variable-protégée_

Classe *vs* Objet

créer une classe $\neq$ instancier une classe

- Créer une classe : définir une nouvelle classe, écrire ses attributs et méthodes.
- Instancier une classe : créer **un exemplaire d'un objet**, par exemple `L=[1, 2]`.

Méthode

le constructeur de la classe est une méthode spéciale chargée de l'**initialisation des attributs**

- Parfois c'est la méthode qui a le même nom que la classe ;
- En Python, c'est **la méthode qui s'appelle** `__init__` ;
- La méthode qui est appelée à chaque fois qu'on **instancie** la classe.

Lorsqu'une méthode est appelée sur une instance, elle reçoit cette instance en argument (par exemple `L.append(4)`)

- En Python, c'est le premier argument de la méthode et on le nomme généralement self, par convention.

Classe Point

Exemple de constructeur

1. création d'une classe :

```
1 class Point(object):
2     def __init__(self, x, y):
3     """ crée un objet point """
4         self.x = x
5         self.y = y
```

Classe Point

Exemple de constructeur

1. création d'une classe :

```
1 class Point(object):
2     def __init__(self, x, y):
3     """ crée un objet point """
4         self.x = x
5         self.y = y
```

2. instanciation :

```
1 P = Point(2,1)
2 print(P.x, P.y) # 2 1
```

Rq : ne peut pas modifier directement l'attribut mais on peut y accéder grâce à l'opérateur « . »

Ajout de méthodes

Méthodes spéciales

- On a vu qu'on peut utiliser la même fonction `print` sur différents objets... pourquoi ?
- Il existe des méthodes spéciales qui homogénéisent le fonctionnement des objets en python
  - __init__()
  - __str__()
  - __sizeof__()
  - __len__()
  - __pow__()
  - __floor__()
  - __iter__()
  - __repr__()

Ajout de méthodes

Pretty print, distance

```
1 class Point(object):
2     # Constructeur
3     def __init__(self, x, y):
4         self.x = x
5         self.y = y
6 
7     # Pretty-print
8     def __str__(self):
9         return "<" + str(self.x) + "," + str(self.y) + ">"
```

Utilisation des nouvelles classes

Ici, dans une fonction extérieure à la classe /!\

```
1  # Renvoie le milieu du segment [P1, P2]
2  def point_milieu(P1, P2):
3      x = (P1.x + P2.x) // 2
4      y = (P1.y + P2.y) // 2
5      return Point(x, y) # la fonction renvoie un objet Point
6
7  P = Point(2,1)
8  Q = Point(6,4)
9
10 M = point_milieu(P,Q)
```

Ajout de méthodes

Méthode qui calcule la distance

```
1 P = Point(2,1)
2 Q = Point(6,4)
3 distance = P.dist(Q)
4 print(distance) # 5
```

Ajout de méthodes

Méthode `dist`

```
1  class Point(object):
2      # Constructeur
3      def __init__(self, x, y):
4          self.x = x
5          self.y = y
6
7      # Pretty print
8      def __str__(self):
9          return "<" + str(self.x) + "," + str(self.y) + ">"
10
11     # Nouvelle méthode
12     def dist(self):
13          pass
```

- self est toujours le premier argument des méthodes de la classe

Ajout de méthodes

Méthode `dist`

- que calcule-t-on ?
- de quel(s) module(s) a-t-on besoin ?
- quel(s) paramètre(s) ?
- que renvoie-t-on ?

```
1 def dist(self, ??):
2     return ??
```

```
1 distance = P.dist(Q)
```

Ajout de méthodes

Méthode `dist`

- que calcule-t-on ? Rappel : $PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$
- de quel(s) module(s) a-t-on besoin ?
- quel(s) paramètre(s) ?
- que renvoie-t-on ?

```
1 def dist(self, ??):
2     return ??
```

```
1 distance = P.dist(Q)
```

Ajout de méthodes

Méthode `dist`

- que calcule-t-on ? Rappel : $PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$
- de quel(s) module(s) a-t-on besoin ? fonction `sqrt` module `math`
- quel(s) paramètre(s) ?
- que renvoie-t-on ?

```
1 def dist(self, ??):
2     return ??
```

```
1 distance = P.dist(Q)
```

Ajout de méthodes

Méthode `dist`

- que calcule-t-on ? Rappel : $PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$
- de quel(s) module(s) a-t-on besoin ? fonction `sqrt` module `math`
- quel(s) paramètre(s) ? `self`, second point
- que renvoie-t-on ?

```
1 def dist(self, ??):
2     return ??
```

```
1 distance = P.dist(Q)
```

Ajout de méthodes

Méthode `dist`

- que calcule-t-on ? Rappel : $PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$
- de quel(s) module(s) a-t-on besoin ? fonction `sqrt` module `math`
- quel(s) paramètre(s) ? `self`, second point
- que renvoie-t-on ? valeur flottante

```
1 def dist(self, ??):
2     return ??
```

```
1 distance = P.dist(Q)
```

Ajout de méthodes

Méthode `dist`

```python
from math import sqrt

class Point(object):
    # Constructeur
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Pretty print
    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"

    def dist(self, P2):
        return sqrt((P2.x - self.x)**2 + (P2.y - self.y)**2))
```

Utilisation des classes

Pour utiliser une classe dans un script, il faut importer la classe *via* le **module** dans lequel elle a été créée :

```
1 from point import Point
```

Sources

- Cours de Pablo Rauzy (lien)
- Cours de Jean-Pascal Palus (lien)
- courspython.com