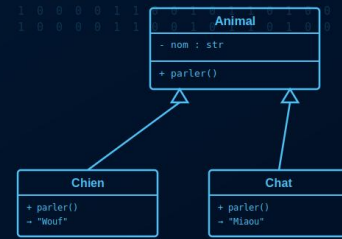


VI. Programmation orientée objet (POO)



```

class Animal:
    def __init__(self, nom):
        self.nom = nom
    def parler(self):
        raise NotImplementedError

class Chien(Animal):
    def parler(self):
        return "Wouf !"
    
```

M. Martin

1

1

Problème

Un projet grandit : le code devient difficile à maintenir si tout est écrit dans des fonctions isolées

Solution

La **programmation orientée objet** organise le code autour de **classes** et **d'objets**.

- Une **classe** définit un modèle qui regroupe données (**attributs**) et comportements (**méthodes**)
- Un **objet** est une instance de classe

2

2

Pourquoi la POO ? (cont.)

Avantages

- Modéliser des concepts du monde réel ou scientifique
- Réutiliser et étendre facilement le code
- Structurer les projets complexes
- Améliorer lisibilité et maintenance

En Python, tout est objet : entiers, chaînes, fonctions, modules.

3

3

Définir une classe et créer un objet

Problème

Comment représenter un point du plan avec ses coordonnées et une opération pour le déplacer ?

Solution

```
class Point :
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def déplacer(self, dx, dy):
        self.x += dx
        self.y += dy

p1 = Point(1, 2) # création d'un objet "p1"
p1.déplacer(3, -1) # appel d'une méthode de Point
print(p1.x, p1.y) # 4 1
```

4

4

Définir une classe et créer un objet (cont.)

- `__init__()` est le **constructeur** appelé lors de l'instanciation
- `self` est une référence à l'objet courant
- Chaque objet possède ses **propres valeurs** (x, y)
- Les méthodes définies dans la classe sont partagées par toutes les instances

On peut créer plusieurs objets indépendants à partir de la même classe :

```
p1 = Point(0, 0)
p2 = Point(5, 5)

p1.deplacer(1, 1)
print(p1.x, p1.y) # 1 1
print(p2.x, p2.y) # 5 5 (indépendant de p1)
```

Remarque : si `__init__()` est absent, Python fournit un constructeur vide par défaut.

5

5

Attributs d'instance et de classe

Problème

Comment distinguer une information propre à chaque objet (ex. position d'un point spécifique) d'une information **commune à toutes les instances** (ex. compteur global) ?

Solution

```
class Compteur :
    instances = 0 # attribut de classe partagé
    def __init__(self):
        Compteur.instances += 1 # incrément global

c1 = Compteur()
c2 = Compteur()
print(Compteur.instances) # 2
```

- **Attributs d'instance** : propres à chaque objet (définis dans `__init__()`)
- **Attributs de classe** : partagés entre toutes les instances

6

6

Encapsulation

Problème

Comment empêcher une modification directe de données sensibles (ex. le solde d'un compte bancaire) ?

Solution

```
class CompteBancaire:
    def __init__(self, solde):
        self.__solde = solde # attribut privé

    def deposter(self, montant):
        self.__solde += montant

    def afficher(self):
        return self.__solde
```

Convention :

- x = public
- _x = protégé (usage interne)
- __x = privé (masqué, non accessible directement)

7

7

Méthodes spéciales

Problème

Comment rendre une classe **plus naturelle à utiliser**, comme les objets Python natifs (+, len(), ==, ...) ?

Solution

```
class Vecteur:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __repr__(self):
        return f"Vecteur({self.x}, {self.y})"
    def __add__(self, other):
        return Vecteur(self.x + other.x, self.y + other.y)

v1 = Vecteur(1, 2)
v2 = Vecteur(3, 4)
print(v1 + v2) # Vecteur(4, 6)
```

- + appelle __add__()
- len(obj) appelle __len__()
- obj[i] appelle __getitem__() qui permet un accès par index

8

8

Problème

Comment éviter de réécrire le même code pour des objets qui ont des caractéristiques proches (ex. une voiture ou un vélo) ?

Solution

```
class Vehicule:
    def __init__(self, nom):
        self.nom = nom

    def rouler(self, distance):
        print(f"{self.nom} roule sur {distance} km")

class Voiture(Vehicule):
    def rouler(self, distance) :
        print(f"{self.nom} parcourt {distance} km sur route")

class Velo(Vehicule):
    def rouler(self, distance) :
        print(f"{self.nom} parcourt {distance} km à vélo")
```

9

9

Problème

Comment utiliser une **même interface** pour plusieurs objets différents ?

Solution

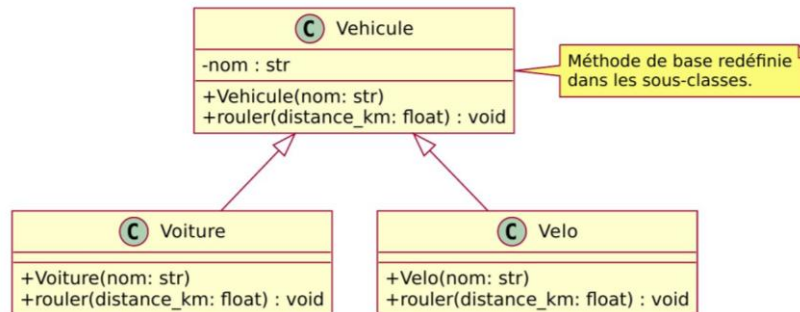
Avec le **polymorphisme** : la méthode rouler() est appelée sans savoir si l'objet est une voiture ou un vélo.

```
flotte = [Voiture("Clio"), Velo("VTT")]
for v in flotte:
    v.rouler(10) # appel identique --> comportement différent
```

10

10

Héritage et polymorphisme : Véhicules



11

11

Problème

Comment obliger les sous-classes à **implémenter certaines méthodes** ?

Solution

- Faire de la classe une classe abstraite
- Le décorateur `@abstractmethod` impose l'implémentation dans les sous-classes

```

from abc import ABC, abstractmethod

class Forme(ABC):
    @abstractmethod
    def aire(self):
        pass

class Cercle(Forme):
    def __init__(self, r):
        self.r = r
    def aire(self):
        return 3.14 * self.r**2
    
```

12

12

Problème

Comment créer des fonctions qui ne dépendent pas d'un objet en particulier ?

Solution

- Le décorateur `@staticmethod` indique que `carre()` ne dépend pas d'un objet en particulier

```
class Outils:
    @staticmethod
    def carre(x):
        return x**2

print(Outils.carre(5)) # 25
```

13

13

Problème

Comment créer des fonctions qui agissent sur la classe entière ?

Solution

- Le décorateur `@classmethod` indique que `incrémenter()` agit sur compteur pour tout objet

```
class Exemple :
    compteur = 0
    @classmethod
    def incrémenter(cls):
        cls.compteur+=1
```

14

14



Problème

Comment représenter des liens entre objets (utilisation, regroupement, dépendance) ?

Solution

Trois relations structurantes, de la plus faible à la plus forte cohésion permettent de représenter ces liens :

1. **Association** : un objet en utilise un autre
2. **Agrégation** : un objet regroupe des objets indépendants
3. **Composition** : un objet contient des sous-objets qui n'existent pas en dehors de lui



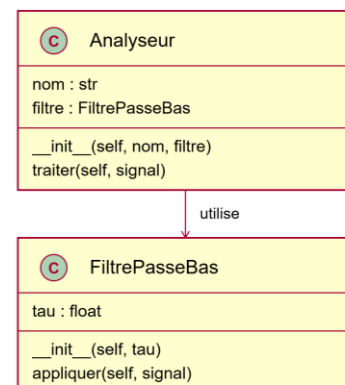
15



Un objet « Analyseur » utilise un objet « FiltrePasseBas » pour traiter un signal, mais ils peuvent exister séparément

```
class FiltrePasseBas:
    def __init__(self, tau):
        self.tau = tau
    def appliquer(self, signal):
        # ..traitement..
        return signal # (placeholder)

class Analyseur:
    def __init__(self, nom, filtre:FiltrePasseBas):
        self.nom = nom
        self.filtre = filtre
    def traiter(self, signal):
        return self.filtre.appliquer(signal)
```



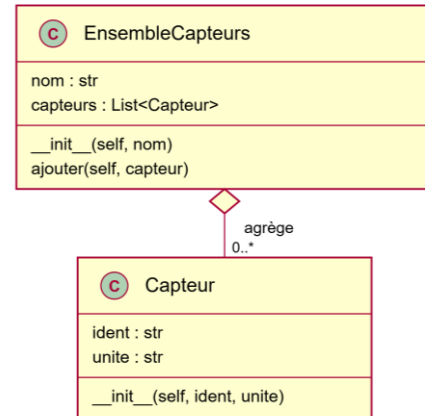
16

Agrégation - exemple

- Un objet « EnsembleCapteurs » regroupe des objets « Capteur »
- Les capteurs peuvent exister sans l'ensemble

```
class Capteur:
    def __init__(self, ident, unite):
        self.ident = ident
        self.unite = unite

class EnsembleCapteurs:
    def __init__(self, nom):
        self.nom = nom
        self.capteurs = []
    def ajouter(self, capteur: Capteur):
        self.capteurs.append(capteur)
```



17

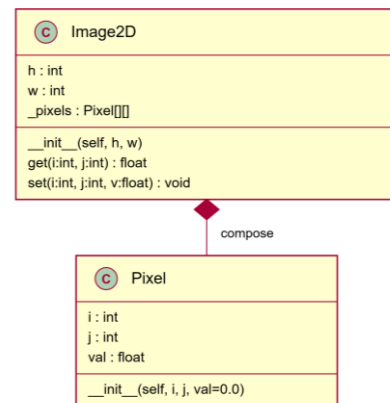
17

Composition - exemple

- Un objet « Image2D » est composée d'objets « Pixel »
- Les objets « Pixel » n'existent pas hors de l'objet « Image2D »

```
class Pixel:
    def __init__(self, i, j, val=0.0):
        self.i = i
        self.j = j
        self.val = val

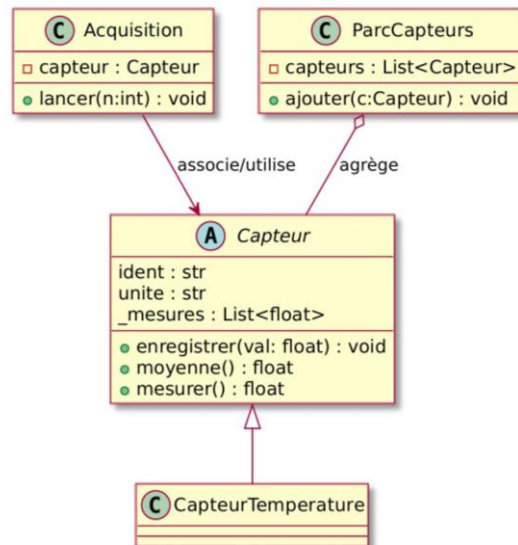
class Image2D:
    def __init__(self, h, w):
        self.h = h
        self.w = w
        self._pixels = [
            [Pixel(i, j) for j in range(self.w)]
            for i in range(self.h)]
    def get(self, i, j):
        return self._pixels[i][j].val
    def set(self, i, j, v):
        self._pixels[i][j].val = v
```



18

18

Exemple complet



19

19

Relations + héritage

```

from abc import ABC, abstractmethod

class Capteur(ABC):
    def __init__(self, ident, unite):
        self.ident = ident
        self.unite = unite
        self._mesures = []
    def enregistrer(self, valeur):
        self._mesures.append(valeur)
    def moyenne(self):
        return sum(self._mesures) / len(self._mesures) if self._mesures else float("nan")
    @abstractmethod
    def mesurer(self):
        pass

class CapteurTemperature(Capteur):
    def __init__(self, ident):
        super().__init__(ident, "°C")
    def mesurer(self):
        # ..acquisition matérielle simulée..
        return 20.0
    
```

20

Relations + héritage (cont.)

```
class Acquisition:
    """Association : utilise un Capteur pour acquérir."""
    def __init__(self, capteur: Capteur):
        self.capteur = capteur
    def run(self, n):
        for _ in range(n):
            self.capteur.enregistrer(self.capteur.mesurer())

class ParcCapteurs:
    """Agrégation : contient des Capteur (vie indépendante)."""
    def __init__(self, nom):
        self.nom = nom
        self.capteurs = []
    def ajouter(self, c: Capteur):
        self.capteurs.append(c)
```

21

Bonnes pratiques dans le choix des relations

- Préférer une association simple si le cycle de vie des objets n'est pas lié
- Choisir la composition si les objets internes ont une durée de vie qui dépend de l'objet composite
- Choisir l'agrégation pour des collections d'objets autonomes
- Favoriser la composition plutôt que l'héritage quand une relation « à un » suffit

22

22

Problème

Comment contrôler l'accès à un attribut (ex. interdire qu'une température soit inférieure au zéro absolu) ?

```
class Celsius:
    def __init__(self, temp=0):
        self.temp = temp

c = Celsius(25)
c.temp = -300 # !!! Température impensable
```

- Ici, rien n'empêche d'attribuer une valeur **physiquement impossible**
- Dans certains langages, on utiliserait des **getters/setters** (getTemp(), setTemp()) pour contrôler
- En Python, on préfère une solution plus simple et élégante : le décorateur **@property**

23

23

Problème

- **@property** permet d'utiliser une méthode comme un attribut
- Sert à ajouter des contrôles sans changer la syntaxe

```
class Celsius:
    def __init__(self, temp=0):
        self.temp = temp

    @property
    def temp(self):
        return self._temp

    @temp.setter
    def temp(self, valeur):
        if valeur < -273.15:
            raise ValueError("Température impossible")
        self._temp = valeur
```

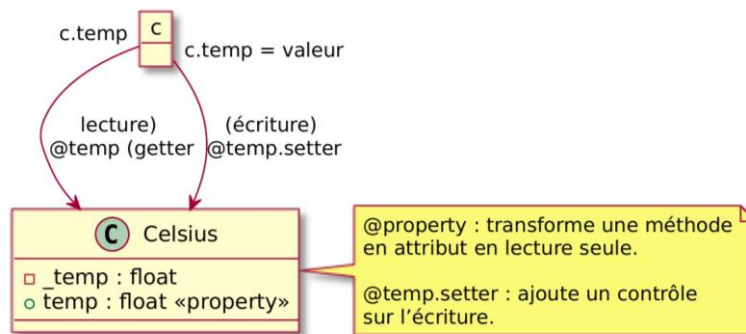
24

24

Décorateur @property (cont.)

```
c = Celsius(20)
print(c.temp) # lit en utilisant @property -> 20
c.temp = 100 # écrit en utilisant @temp.setter
print(c.temp) # 100
c.temp = -300 # déclenche ValueError
```

Accès à une propriété Python



25

25

Itérateurs

- **Itérable** : objet utilisable dans `for ... in` (liste, chaîne, fichier, etc.)
- **Itérateur** : tout objet implémentant `__iter__()` et `__next__()`

Problème

Comment faire d'une classe un itérateur ?

Solution

```
class Compteur : # Classe rendue itérable
    def __init__(self, n) :
        self.n = n
        self.i = 0
    def __iter__(self) :
        return self
    def __next__(self) :
        if self.i < self.n :
            v = self.i
            self.i += 1
            return v
        raise StopIteration

for x in Compteur(3):
    print(x) # Affiche 0, 1, 2
```

26



La POO en Python permet de :

- Créer des classes et objets
- Utiliser l'héritage et le polymorphisme
- Protéger les données en utilisant l'encapsulation
- Redéfinir le comportement avec les méthodes spéciales
- Contrôler l'accès avec @property
- Écrire des objets itérateurs et générateurs

C'est un paradigme incontournable pour structurer des projets complexes.