



# INSAlgo - Cours 9

...

Sujet du cours : La POO, les classes et comment on peut s'en servir en algo

Cours en Python car la syntaxe est simple et compréhensible pour les débutants ! ٩(●~●)۶

# Les paradigmes de programmation

Impérative



Fonctionnelle

Orientée Objet

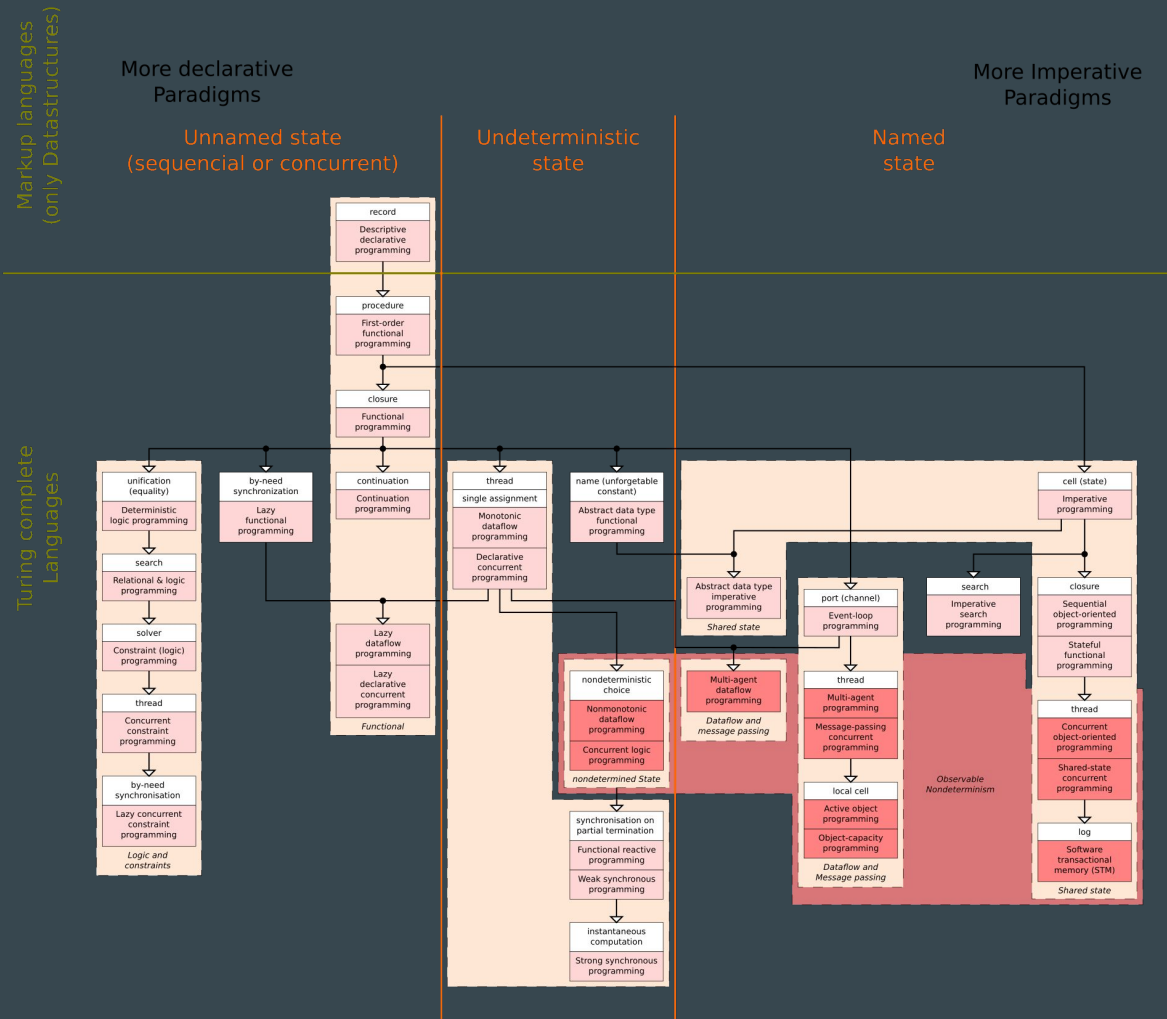


- **Python**: What if everything was a dict?
- **Java**: What if everything was an object?
- **JavaScript**: What if everything was a dict "and" an object?
- **C**: What if everything was a pointer?
- **APL**: What if everything was an array?
- **Tcl**: What if everything was a string?
- **Prolog**: What if everything was a term?
- **LISP**: What if everything was a pair?
- **Scheme**: What if everything was a function?
- **Haskell**: What if everything was a monad?
- **Assembly**: What if everything was a register?
- **Coq**: What if everything was a type/proposition?
- **COBOL**: WHAT IF EVERYTHING WAS UPPERCASE?
- **C#**: What if everything was like Java, but different?
- **Ruby**: What if everything was monkey patched?
- **Pascal**: BEGIN What if everything was structured? END
- **C++**: What if we added everything to the language?
- **C++11**: What if we forgot to stop adding stuff?
- **Rust**: What if garbage collection didn't exist?
- **Go**: What if we tried designing C a second time?
- **Perl**: What if shell, sed, and awk were one language?
- **Perl6**: What if we took the joke too far?
- **PHP**: What if we wanted to make SQL injection easier?
- **VB**: What if we wanted to allow anyone to program?
- **VB.NET**: What if we wanted to stop them again?
- **Forth**: What if everything was a stack?
- **ColorForth**: What if the stack was green?
- **PostScript**: What if everything was printed at 600dpi?
- **XSLT**: What if everything was an XML element?
- **Make**: What if everything was a dependency?
- **m4**: What if everything was incomprehensibly quoted?
- **Scala**: What if Haskell ran on the JVM?
- **Clojure**: What if LISP ran on the JVM?
- **Lua**: What if game developers got tired of C++?
- **Mathematica**: What if Stephen Wolfram invented everything?
- **Malbolge**: What if there is no god?

# Beaucoup d'options...

- Imperative
  - procedural
  - **Object-Oriented**
- Declarative
  - Functional
  - Logic
  - Reactive
- Concurrent
- Constrained
- Distributed
- Generic
- Metaprogramming

...



# Pourquoi la POO ?

Réunir données et traitement

Abstraction des traitements

Code plus générique

*Rendre le code plus lisible (...)*

Conteneuriser, chaque bloc de code à un but précis



# Les briques de base : l'objet et la classe (😎)

## La classe

C'est ce qu'on définit, c'est là qu'on décrit à quoi ressemble un objet, quels seront ses **attributs** et ses **méthodes**



## L'objet

C'est ce qu'on manipule dans le programme, on sait ce qu'il contient et comment l'utiliser grâce à la **classe** qui définit son comportement



# Mais alors du coup en code ça donne quoi ?

```
class Point:
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def afficher(self):
        print(self.x, self.y)
```

```
point = Point(1, 2)
print(point.x)    # 1
point.afficher() # 1 2
```

```
class Point {
    int x;
    int y;
    Point(int px, int py) {
        x = px;
        y = py;
        this.x = val;
        // Equivalent
    }
    void afficher() {
        System.out.print(x);
        System.out.println(" "+y);
    }
}
```

```
// Dans le main :
Point point = new Point(1, 2);
System.out.println(point.x)
point.afficher(); // 1 2
```

# Qu'est-ce qui se passe donc ?

```
class Point:
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def afficher(self):
        print(self.x, self.y)

point = Point(1, 2)
point.afficher() # 1 2
```

**x** : attribut

**y** : attribut

**afficher** : méthode

**\_\_init\_\_** : méthode spéciale  
(constructeur)

**self** désigne l'objet depuis lequel la  
méthode est appelée

point.afficher() -> dans afficher,  
self est l'objet mon\_point.

Point est la classe, mon\_point est  
un objet de la classe Point

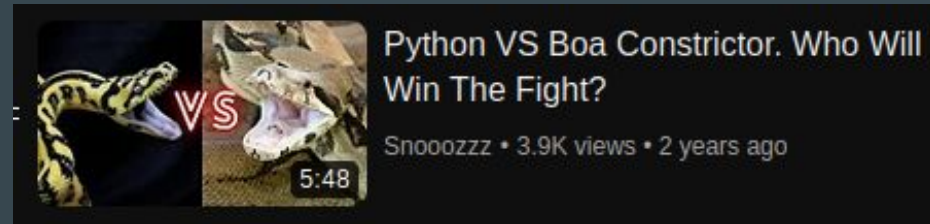
# Le CONSTRUCTEUR (important)

Le constructeur est une méthode spéciale, elle est appelée à la création d'un objet.

On peut lui passer des paramètres comme n'importe quelle méthode.

Permet d'initialiser les attributs de notre objet selon les valeurs passées en paramètre et renvoie donc une **instance** de cet objet.

En python : `__init__`





# La surcharge d'opérateurs

Pour utiliser des opérateurs tels que +, \*, [],... sur vos objets.

On définit une méthode spéciale. Exemple pour le + :

```
class Point:
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def afficher(self):
        print(self.x, self.y)

    def __add__(self, point2):
        self.x += point2.x
        self.y += point2.y
```

```
p1 = Point(1, 2)
p2 = Point(2, 3)

p1 + p2
// Equivalent : p1.__add__(p2)

p1.afficher() # 3 5
```

# La surcharge d'opérateurs

Les classes de python surchargent déjà certains opérateurs :

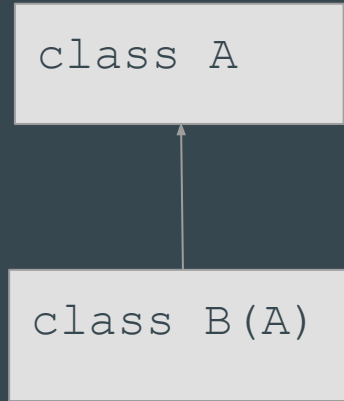


# La surcharge d'opérateurs

| Magic Method                                 | When it gets invoked (example)                                   | Explanation  |
|--|--|--|
| <code>__new__(cls [,...])</code>             | <code>instance = MyClass(arg1, arg2)</code>                      | <code>__new__</code> is called on instance creation  |
| <code>__init__(self [,...])</code>           | <code>instance = MyClass(arg1, arg2)</code>                      | <code>__init__</code> is called on instance creation |
| <code>__cmp__(self, other)</code>            | <code>self == other</code> , <code>self &gt; other</code> , etc. | Called for any comparison                            |
| <code>__pos__(self)</code>                   | <code>+self</code>   | Unary plus sign                                      |
| <code>__neg__(self)</code>                   | <code>-self</code>   | Unary minus sign                                     |
| <code>__invert__(self)</code>                | <code>~self</code>   | Bitwise inversion                                    |
| <code>__index__(self)</code>                 | <code>x[self]</code>   | Conversion when object is used as index              |
| <code>__nonzero__(self)</code>               | <code>bool(self)</code>  | Boolean value of the object                          |
| <code>__getattr__(self, name)</code>         | <code>self.name</code> # name doesn't exist                      | Accessing nonexistent attribute                      |
| <code>__setattr__(self, name, val)</code>    | <code>self.name = val</code>                                     | Assigning to an attribute                            |
| <code>__delattr__(self, name)</code>         | <code>del self.name</code>                                       | Deleting an attribute                                |
| <code>__getattribute__(self, name)</code>    | <code>self.name</code>   | Accessing any attribute                              |
| <code>__getitem__(self, key)</code>          | <code>self[key]</code>   | Accessing an item using an index                     |
| <code>__setitem__(self, key, val)</code>     | <code>self[key] = val</code>                                     | Assigning to an item using an index                  |
| <code>__delitem__(self, key)</code>          | <code>del self[key]</code>                                       | Deleting an item using an index                      |
| <code>__iter__(self)</code>                  | <code>for x in self</code>                                       | Iteration  |
| <code>__contains__(self, value)</code>       | <code>value in self</code> , <code>value not in self</code>      | Membership tests using <code>in</code>               |
| <code>__call__(self [,...])</code>           | <code>self(args)</code>  | "Calling" an instance                                |
| <code>__enter__(self)</code>                 | <code>with self as x:</code>                                     | <code>with</code> statement context managers         |
| <code>__exit__(self, exc, val, trace)</code> | <code>with self as x:</code>                                     | <code>with</code> statement context managers         |
| <code>__getstate__(self)</code>              | <code>pickle.dump(pk1_file, self)</code>                         | Pickling   |
| <code>__setstate__(self)</code>              | <code>data = pickle.load(pk1_file)</code>                        | Pickling   |

Liste pas forcément à jour et non exhaustives des “méthodes magiques” de Python

# L'héritage : étendre les fonctionnalités d'une classe

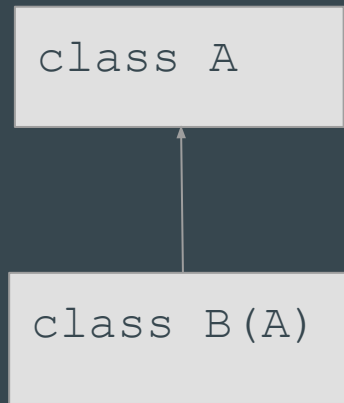


Si la classe A possède des attributs et des méthodes, la classe B les possédera aussi car elle les **héríte** de A.

Ce mécanisme permet d'étendre les fonctionnalités d'une classe, ou de fournir une spécialisation d'une classe plus générale.

Permet de construire votre propre graphe d'héritage avec vos classes pour réduire le code dupliqué.  
Ou permet de modifier le comportement d'une classe préexistante dans le langage.

# L'héritage : la surcharge de méthodes



Si la classe A possède des attributs et des méthodes, la classe B les possédera aussi car elle les **hérite** de A.

La classe **A** peut implémenter une méthode **afficher**. Si cette méthode n'est pas redéfinie dans **B**, appeler **afficher** sur un objet de type **B** appellera la méthode **afficher** de **A**.

Si on redéfinit **afficher** dans **B**, appeler **afficher** sur un objet de type **B** va utiliser la méthode la plus “spécialisée”, c'est-à-dire celle de **B**.

# Exemple de modification d'une classe :

On veut créer une liste qui affiche tout ce qu'on lui ajoute

```
class ListeAffichage(list):  
    def __setitem__(self, key, val):  
        print(f"On insère la valeur {val} à l'emplacement {key}")  
        return super().__setitem__(key, val)  
  
liste = ListeAffichage([1, 2, 3, 4, 5])  
liste[2] = 0; # On insère la valeur 0 à l'emplacement 2
```

Grâce à l'héritage, et à la surcharge, notre classe se comporte exactement comme une liste sauf à l'insertion

# Mais alors en algo ça sert quand ?

Quand on veut manipuler des structures de données plus complexes que juste des nombres, et surtout quand on cherche à regrouper et gérer de données reliés.

Graphes

Flux de  
données

Vecteurs

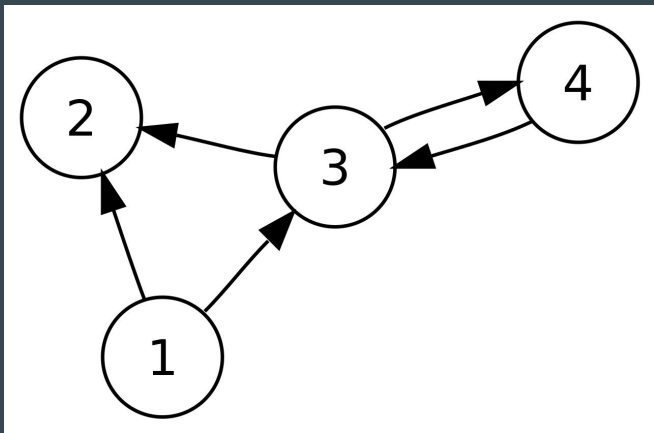
Simulation

Points

Des singes qui font  
des maths !?  
#AoC11

# Les classes dans un graphe

On définit une classe Noeud, avec un attribut nom et un attribut voisins, qui est une liste de Noeud



```
class Noeud:
    def __init__(self, nom, voisins = list()):
        self.nom = nom
        self.voisins = set(voisins)

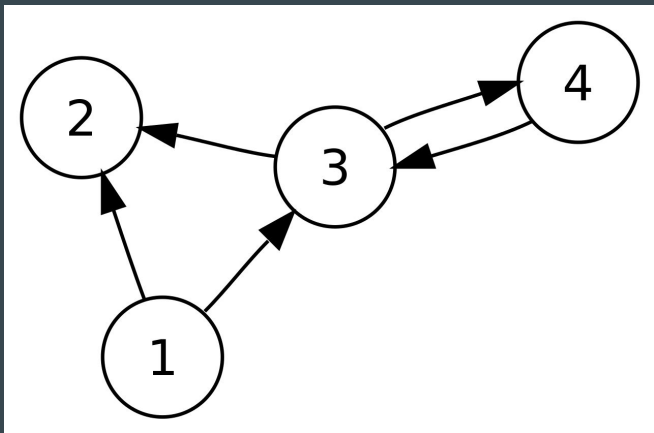
    def ajouter_voisin(self, noeud):
        self.voisins.add(noeud)

    @property
    def nb_voisins(self):
        return len(self.voisins)

    @staticmethod
    def chemin_entre(noeud1, noeud2):
        chemin = ...
        # algo de Dijkstra
        return chemin
```



# Les classes dans un graphe



```
n1 = Noeud(1)
n2 = Noeud(2)
n3 = Noeud(3)
n4 = Noeud(4, [n3])
n1.ajouter_voisin(n2)
n1.ajouter_voisin(n3)
n3.ajouter_voisin(n2)
n3.ajouter_voisin(n4)
a = n3.nb_voisins # 2
Noeud.chemin_entre(n1, n4)
# retourne [n1, n3, n4]
```

# Les limites de la POO

## 1. Inflexibilité de l'héritage

- L'utilisation excessive de l'héritage peut conduire à des hiérarchies profondément imbriquées et à un **couplage** étroit.
- Les sous-classes peuvent se retrouver avec des relations forcées ou maladroites avec la classe mère.

## 2. Le problème de la classe de base fragile

- Les modifications apportées à une classe de base peuvent **briser involontairement des sous-classes**.
- Cela oblige à une coordination minutieuse et peut limiter l'évolution indépendante des classes.

## 3. Violation des principes SOLID

En particulier le **principe de substitution de Liskov (LSP)**, qui stipule que les objets d'une superclasse doivent pouvoir être remplacés par des objets de ses sous-classes sans casser le système.

## 4. Une trop grande importance accordée aux objets

Peut conduire à une tendance à « objectiver » chaque concept, même lorsque des approches procédurales ou fonctionnelles pourraient être plus simples ou plus appropriées.

## 5. Difficulté avec la concomitance

Les abstractions de la POO ne s'alignent pas toujours bien sur les modèles de programmation parallèle et concurrente.

# Le problème du Rectangle → Carré

```
class Rectangle :
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def set_width(self, width):
        self.width = width

    def set_height(self, height):
        self.height = height

    def get_area(self):
        return self.width * self.height

    def __str__(self):
        return f"Rectangle(width= {self.width}, height= {self.height})"
```

# Le problème du Rectangle → Carré

```
class Square(Rectangle):
    def __init__(self, side_length):
        super().__init__(side_length, side_length)

    def set_side(self, side_length):
        self.width = side_length
        self.height = side_length

    def set_width(self, width):
        self.set_side(width)

    def set_height(self, height):
        self.set_side(height)

    def __str__(self):
        return f"Square(side_length= {self.width})"
```

# Le problème du Rectangle → Carré

```
def  
increase_rectangle_area_by_10_percent (rectangle ):  
    rectangle .set_width( rectangle .width * 1.1)
```

# Le problème du Rectangle → Carré

```
# Example usage
rect = Rectangle(3, 4)
print(rect) # Rectangle(width=3, height=4)
print("Area:", rect.get_area()) # Area: 12

square = Square(5)
print(square) # Square(side_length=5)
print("Area:", square.get_area()) # Area: 25

shapes = [rect, square]
for shape in shapes:
    original_area = shape.width * shape.height
    increase_rectangle_area_by_10_percent(shape)
    print(shape, "Area:", shape.get_area(), end=" → ")
    if shape.get_area() == original_area * 1.1:
        print("Correct")
    else:
        print("Wrong")
```

# Le problème du Rectangle → Carré

《onyr ★ nixos》《09 - POO》python main.py

```
Rectangle(width=3, height=4)
```

Area: 12

```
Square(side_length=5)
```

Area: 25

```
Rectangle(width=3.3000000000000003, height=4) Area:  
13.200000000000001 -> Correct
```

```
Square(side_length=5.5) Area: 30.25 -> Wrong
```

# Le problème du Rectangle → Carré

Une correction ?

```
def
corrected_increase_rectangle_area_by_10_percent (rectangle):
    if isinstance (rectangle , Square):
        rectangle.set_side (rectangle.width * sqrt(1.1))
    else:
        rectangle.set_width( rectangle.width * 1.1)
```



# Le problème du Rectangle → Carré

Une correction ?

```
def
corrected_increase_rectangle_area_by_10_percent (rectangle):
    if isinstance (rectangle , Square):
        rectangle.set_side (rectangle.width * sqrt(1.1))
    else:
        rectangle.set_width( rectangle.width * 1.1)
```

# Le problème du Rectangle → Carré

```
⟨onyr ★ nixos⟩ ⟨09 - POO⟩ python main.py
```

```
Rectangle(width=3, height=4)
```

```
Area: 12
```

```
Square(side_length=5)
```

```
Area: 25
```

```
Rectangle(width=3.3000000000000003, height=4) Area: 13.200000000000001 -> Correct
```

```
Square(side_length=5.244044240850759) Area: 27.500000000000001 -> Wrong
```

```
⟨onyr ★ nixos⟩ ⟨09 - POO⟩ python
```

```
Python 3.12.4 (main, Jun 6 2024, 18:26:44) [GCC 13.3.0] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> 25*1.1
```

```
27.500000000000004
```

```
>>> exit()
```

Maintenant on doit gérer des erreurs d'arrondis...

# Le problème du Rectangle → Carré

```
class Rectangle :  
    def __init__(self, width, height):  
        self.width = width  
        self.height = height  
  
    def set_width(self, width):  
        self.width = width  
  
    def set_height(self, height):  
        self.height = height  
  
    def get_area(self):  
        return round(self.width * self.height, 6)  
  
    def __str__(self):  
        return f"Rectangle(width= {self.width}, height= {self.height})"
```

Maintenant on doit gérer des erreurs d'arrondis...

# Le problème du Rectangle → Carré

```
# Example usage
rect = Rectangle(3, 4)
print(rect)    # Rectangle(width=3, height=4)
print("Area:", rect.get_area())    # Area: 12

square = Square(5)
print(square)   # Square(side_length=5)
print("Area:", square.get_area())   # Area: 25

shapes = [rect, square]
for shape in shapes:
    original_area = shape.width * shape.height
    corrected_increase_rectangle_area_by_10_percent(shape)
    print(shape, "Area:", shape.get_area(), end=" -> ")
    if shape.get_area() == round(original_area * 1.1, 6):
        print("Correct")
    else:
        print("Wrong")
```

Maintenant on doit gérer des erreurs d'arrondis...

# Le problème du Rectangle → Carré

《onyr ★ nixos》《09 - POO》 python main.py

Rectangle(width=3, height=4)

Area: 12

Square(side\_length=5)

Area: 25

Rectangle(width=3.3000000000000003, height=4) Area: 13.2 -> Correct

Square(side\_length=5.244044240850759) Area: 27.5 -> Correct

# Comment s'en sortir

- Utiliser un autre paradigme ;)
  - Ne pas mettre de d'héritage entre Square et Rectangle
- Considerer ces objects comme immutables

## Critiques supplémentaires

- Overengineering
- Les arbres d'héritage complexes peuvent devenir lourds
- Fuites d'encapsulation
- Les modèles Getter/Setter conduisent parfois à des « données publiques dans des membres privés ».
- Mauvaise utilisation et complexité
- La POO peut nécessiter une conception initiale importante et est dangereuse à étendre

# Conclusion



# Fin du cours

Place à quelques exos. Comme d'habitude  
rdv sur le discord pour les voir.

Autheurs:

- Anciens d'INSAlgo
- Onyr