Langage Python

Programmation orientée objet (POO) - partie 1

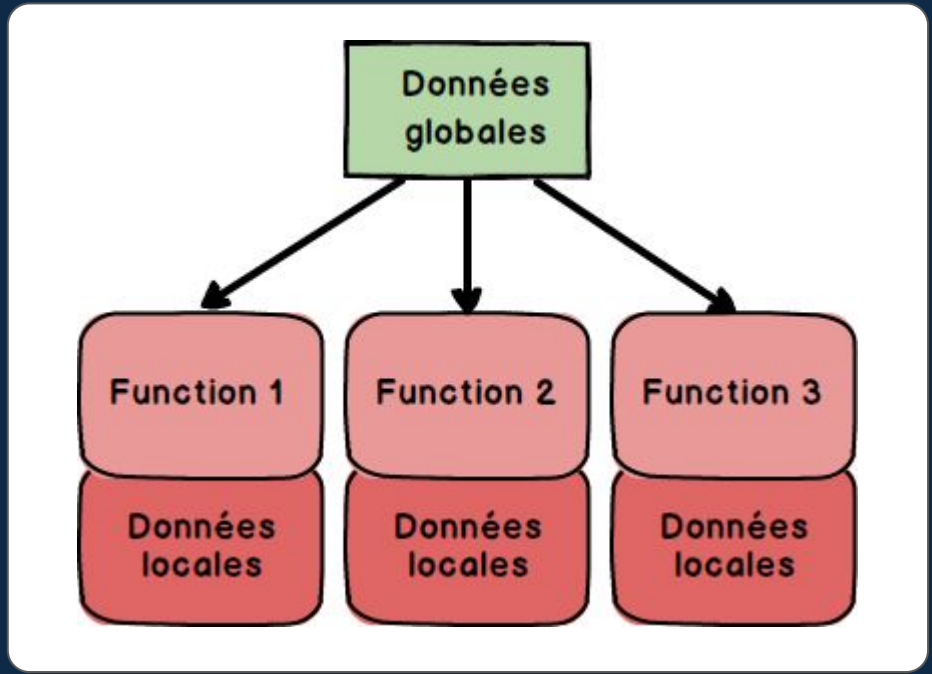
Formation POEC Cybersécurité
Théo Hubert

Concept

- *Programmation Impérative & Programmation procédurale*

Programmation impérative (paradigme) :
- Séquences d'instructions
- Affectations
- Conditions
- Boucles

Programmation procédurale (paradigme) :
- Fonctions
- Modularité
- Meilleur visibilité



Source Image https://waytolearnx.com/2018/09/difference-entre-programmation-procedurale-et-orientee-objet.html

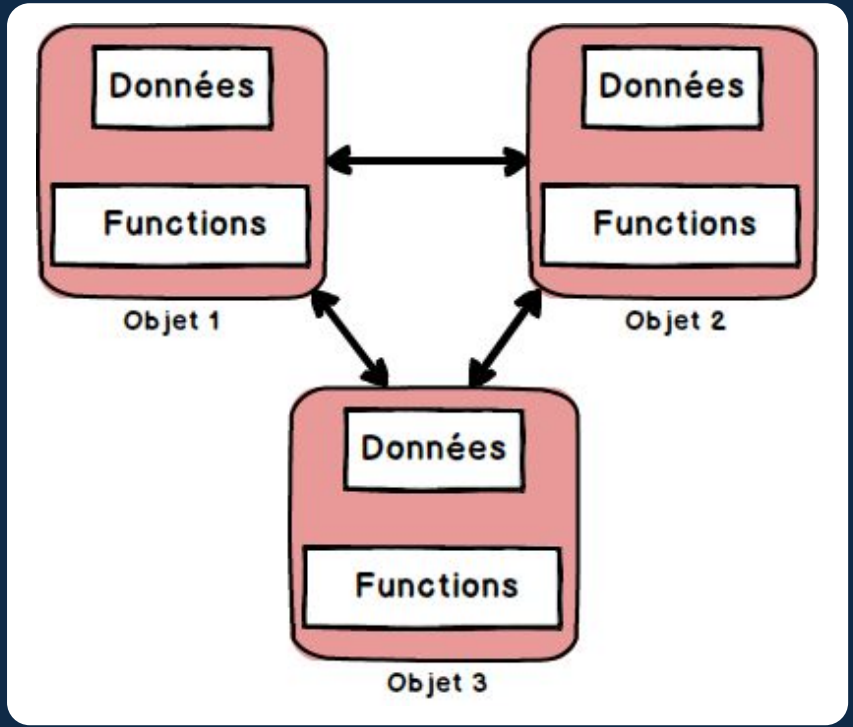
Concept

- *Programmation orientée objet (POO)*

La programmation orientée objet (POO) est **un paradigme** (représentation des choses, un modèle cohérent partagé à travers différents langages qui permettent son usage).

La POO permet de modéliser son code sous forme d'**objets** ayant des **propriétés** et de **méthodes** et qui interagissent entre eux plutôt qu'une séquence d'instructions.

"Les objets désignent le plus souvent **des variables complexes**, elles-mêmes composées de variables ou de fonctions."



Source Image https://waytolearnx.com/2018/09/difference-entre-programmation-procedurale-et-orientee-objet.html

Introduction

- *Histoire de la POO*

1960 Le paradigme de programmation qu'est la POO a été défini par les norvégiens Ole-Johan Dahl et Kristen Nygaard au début de la décennie 1960.

1970 Plus tard, leurs travaux furent repris et amendés dans les années 1970 par l'américain Alan Kay.

1980 À partir des années 1980, les principes de la POO sont appliqués dans de nombreux langages comme Eiffel , C++ ou encore Objective C (une autre extension objet du C utilisé, entre autres, par l'iOS d'Apple).

1990 Les années 1990 ont vu l'avènement des langages orientés objet dans de nombreux secteurs du développement logiciel, et la création du langage Java par la société Sun Microsystems. Le succès de ce langage, plus simple à utiliser que ses prédécesseurs, a conduit Microsoft à riposter en créant au début des années 2000 la plate-forme .NET et le langage C#, cousin de Java.

Introduction

- *La POO aujourd'hui*

De nos jours, de très nombreux langages permettent d'utiliser les principes de la POO dans des domaines variés : **Java** et **C#** bien sûr, mais aussi **PHP** (à partir de la version 5), **VB.NET**, **PowerShell**, **Python**, etc. Une connaissance minimale des principes de la POO est donc indispensable à tout informaticien, qu'il soit développeur ou non.

Introduction

- *Intérêt de la POO*

- Le code est séparé en plusieurs *classes,* ce qui **améliorera la lisibilité** de votre application, chaque classe ayant ses propres propriétés et son propre contexte.
- Une approche **modulaire** est assez simple à mettre en place car vos objets sont cloisonnés et ne peuvent interagir entre eux uniquement si cela a été clairement spécifié.
- La **réutilisation de code** au travers de classes que vous seriez amené à utiliser dans plusieurs projets..

“Si la POO ne permet pas fondamentalement de faire plus de choses que la programmation procédurale, elle permet toutefois de mieux organiser son code. Elle facilite aussi le travail coopératif et la maintenance à long terme”

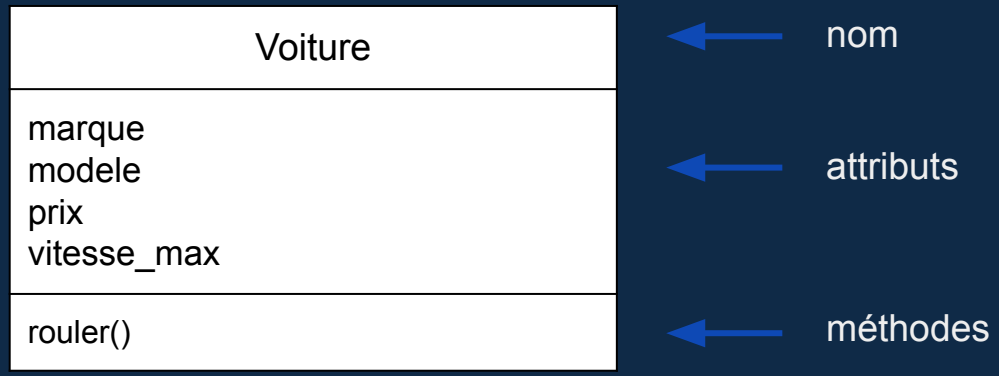
POO

- *Une classe*

Une classe permet de définir la **structure des éléments d'un ensemble d'objets** ayant des caractéristiques communes (**modèle d'objet**). On pourra dire qu'une classe fonctionne comme un moule qui donnerait la forme à des gâteaux.

- Les variables qui décrivent la structure interne de la classe sont appelées **attributs**.

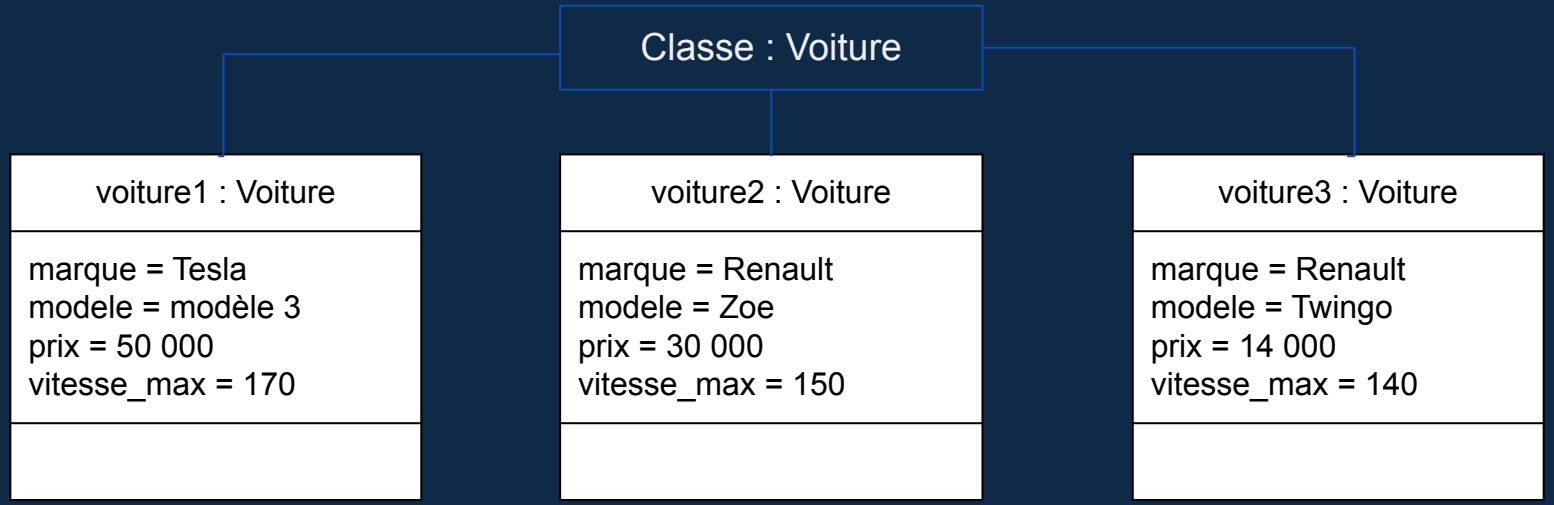
- Les fonctions qui décrivent le comportement de la classe sont appelées **méthodes**.



POO

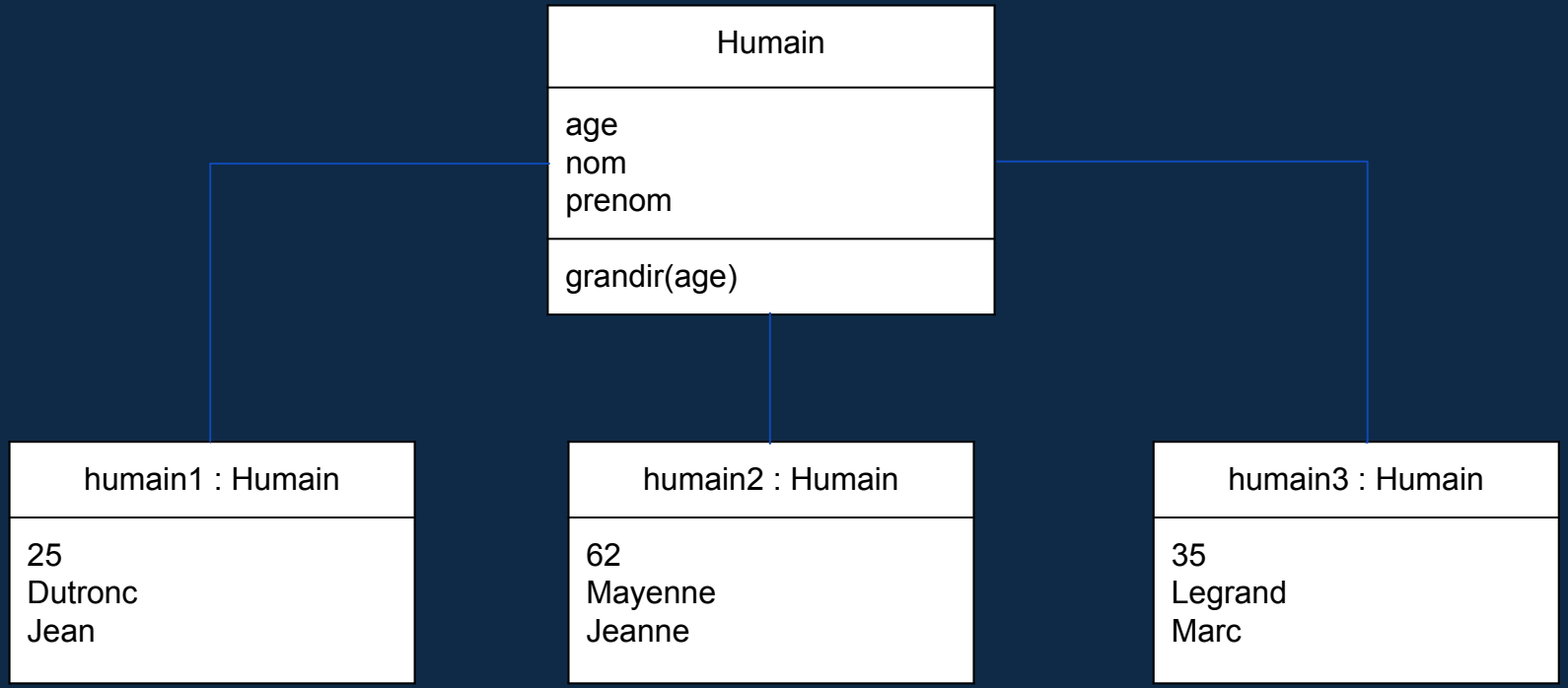
- *Un objet*

“Un objet représente un **concept, une idée ou toute entité du monde physique**, comme une voiture, une personne ou encore une page d'un livre. Il possède **une structure interne et un comportement**, et il sait interagir avec ses pairs”. On peut également voir un **objet** comme une **instance** d'une classe.



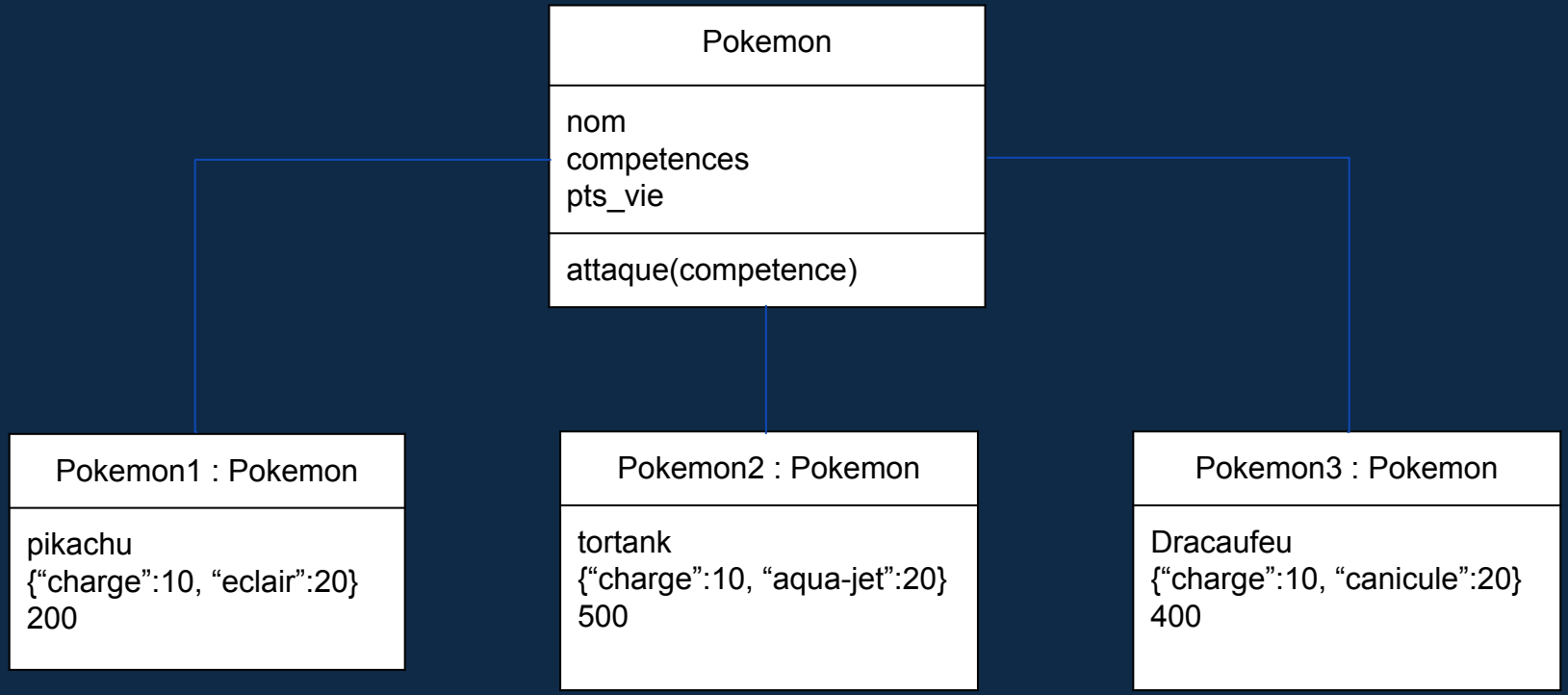
POO

- *Exemple*


Humain
age
nom
prenom
grandir(age)
humain1 : Humain
25
Dutronc
Jean
humain2 : Humain
62
Mayenne
Jeanne
humain3 : Humain
35
Legrand
Marc

POO

- *Exemple*


Pokemon
nom
competences
pts_vie
attaque(competence)
Pokemon1 : Pokemon
pikachu
{“charge”:10, “eclair”:20}
200
Pokemon2 : Pokemon
tortank
{“charge”:10, “aqua-jet”:20}
500
Pokemon3 : Pokemon
Dracaufeu
{“charge”:10, “canicule”:20}
400

POO

- *A retenir*

  - *Les classes sont des ensembles de code qui contiennent des variables et des fonctions et qui vont nous servir à créer des objets*
  - *Les objets créés à partir d'une classe disposent automatiquement des variables et des fonctions définies dans la classe.*

Syntaxe

- *Classe & Constructeur*

Pour créer une **nouvelle classe Python** on utilise le mot clé **class** suivi du nom de notre classe. Par convention le nom d'une classe commence toujours par une **majuscule**.

```
class Voiture :
  pass
```

Le **constructeur** est appelé lors de la création d'un nouvel objet. Il s'agit en fait d'une **méthode particulière** dont le code est exécuté lorsqu'une classe est instanciée.

Le constructeur se définit dans une classe comme une fonction avec deux particularités :

- le nom de la fonction doit être **__init__** ;
- la fonction doit **accepter au moins un paramètre**, dont le **nom doit être self**, et qui doit être le premier paramètre.

Le paramètre self représente en fait l'objet cible, c'est-à-dire que c'est une variable qui contient une référence vers l'objet qui est en cours de création. Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

Syntaxe

- *Classe & Constructeur*

Constructeur

Création d'un objet voiture1 et voiture2

```
class Voiture:
    def __init__(self, modele, marque, prix):
        self.modele = modele
        self.marque = marque
        self.prix = prix

voiture1 = Voiture('Tesla', 'Model 3', 45000)
voiture2 = Voiture('Tesla', 'Model S', 65000)

print(voiture1,'\n')
print(voiture1.modele)
print(voiture1.marque)
print(voiture1.prix)
```

```
<__main__.Voiture object at 0x7f3a5aa99190>

Tesla
Model 3
45000
```

Syntaxe

- *Classe & Constructeur*

```
class Voiture:
    def __init__(self, modele, marque, prix=None):
        self.modele = modele
        self.marque = marque
        self.prix = prix

voiture1 = Voiture('Tesla', 'Model 3', 45000)
voiture2 = Voiture('Tesla', 'Model S')

print(voiture2.modele)
print(voiture2.marque)
print(voiture2.prix)
```

```
Tesla
Model S
None
```

Avec Python il n'est **pas possible d'avoir plusieurs constructeurs** mais on peut attribuer des valeurs par défaut aux arguments du constructeur.

Syntaxe

- *Méthodes*

Les méthodes sont des **fonctions définies dans une classe**.

```
class Voiture:
    def __init__(self, modele, marque, prix=None):
        self.modele = modele
        self.marque = marque
        self.prix = prix

    def setPrix(self, prix):
      self.prix = prix

    def printPrix(self):
      print(self.modele)
      print(self.marque)
      print(self.prix)

voiture1 = Voiture('Tesla', 'Model 3', 45000)
voiture2 = Voiture('Tesla', 'Model S')

voiture2.setPrix(65000)

voiture2.printPrix()
```

```
Tesla
Model S
65000
```

Syntaxe

Travaux Pratique

Syntaxe

- *Surcharge*

La surcharge d'opérateurs nous permet d'**utiliser des opérateurs standards** tels que +, -, * etc sur nos **classes personnalisées**. Il est également **possible de surcharger certaines fonctions** tel que print().

Constructeur →

Surcharge de l'opérateur + →

Surcharge de la fonction print() →

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, p2):
        return Point2D(self.x + p2.x, self.y + p2.y)

    def __str__(self):
        return str((self.x, self.y))

p1 = Point2D(2,3);
p2 = Point2D(4,5);

result = p1 + p2

print(result)
```

Syntaxe

- *Surcharge*

Exemple de surcharge n°2 :

Par exemple pour définir ce que signifie l'**égalité de deux objets,** il faut redéfinir l'opérateur == en définissant une méthode **__eq__**.

Constructeur →

Surcharge de l'opérateur == →

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

p1 = Point2D(2,2);
p2 = Point2D(2,2);

if p1 == p2 :
  print("Les 2 vecteurs sont égaux")
```

```
Les 2 vecteurs sont égaux
```

Syntaxe

- *Surcharge d'opérateurs*

Opérateur	Notation	Méthode à définir
Signe positif	+	__pos__
Signe négatif	-	__neg__
Addition	+	__add__
Soustraction	-	__sub__
Multiplication	*	__mul__
Division	/	__truediv__
Égal	==	__eq__
Différent	!=	__ne__

Pour voir la totalité : http://lptms.u-psud.fr/wiki-cours/index.php/Python:_Surcharge

Syntaxe

Travaux Pratique

Fin

La suite :
Programmation Orienté Objet
Partie 2