## *1-* Créer une classe en Python

On utilise le mot clé class.
Le nom de la classe doit commencer en majuscule (UML), même si c'est pas obligé en Python (philosophie de la responsabilité).

**Classe avec le mot clé pass**

On peut pour créer des classes rapidement pour le prototypage par exemple, avec le mot clé pass. On ajoutera ensuite des attributs à la demande.

```
# classe User
class User:
    pass

#Instancier et ajout d'attributs
user11 = User()
user1.nom = "Dupont"

print(user1.nom)
```

| User |
| --- |
| nom |

***Classe rapide***

## 1- Créer une classe en Python : suite

### Classe Vecteur, notion de constructeur

```python
#Créer une classe
from math import sqrt

class Vecteur:
    #constructeur avec deux attributs
    def __init__(self, x, y):
        self.x = x
        self.y = y

    #un seul paramètre, self
    def __str__(self):
        """fonction pour afficher les objets,
        ici du vecteur au format mathématique"""
        return "(" + str(self.x) +  "," + str(self.y) + ")"

    @property  #décorateur de propriété = la fonction devient un attribut
    def norme(self):
        "Calcul de la norme d'un vecteur"
        return sqrt(self.x**2 + self.y**2)

    #alias
    longueur = norme
```

Fonction qui construit l'objet de ce type.

Fonction d'affichage propre des objets de cette classe avec print.

| Vecteur |
| --- |
| + x |
| + y |

***Diagramme***

***Une classe Python***

## 2- Créer des objets d'une classe donnée

### Instancier = créer un objet

```
#Instancier
vecteur1 = Vecteur(-4, 0)
print(vecteur1) #une sortie formatée par __str__(.)
#print(vecteur1.norme())

#sorties utilisant les attributs norme et longueur
print(vecteur1.norme)
print(vecteur1.longueur)
```

***Un objet***

| Vecteur |
| --- |
| + x |
| + y |

***Diagramme***

### Le destructeur

```
#En général on ne définit pas un destructeur (garbage collector)
def __del__(self):
    print("L'objet" + str(self) + " a été détruit.")

#Détruire explicitement un objet par l'instruction del (ce n'est pas une fonction)
del vecteur1
print(vecteur1.longueur) #NameError: name 'vecteur1' is not defined
```

## 3- Surcharge d'opérateur

On peut vouloir additionner des objets (avec +), ou utiliser d'autres opérateurs courants comme <, >, etc. On parle de surcharge d'opérateurs.

```
class Vecteur:
    #constructeur avec deux attributs
    def __init__(self, x, y):
        self.x = x
        self.y = y

    #destructeur
    def __del__(self):
        print("L'objet" + str(self) + " a été détruit.")

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

    #Redéfinition d'un opérateur
    def __add__(self, other):
        return Vecteur(x=self.x + other.x, y=self.y + other.y)

    @property  #décorateur de propriété
    def norme(self):
        "Calcul de la norme d'un vecteur"
        return sqrt(self.x**2 + self.y**2)

    longueur = norme#alias
```

***Une classe avec __add__***

other c'est **vecteur2**

```
#Instancier
vecteur1 = Vecteur(-4, 0)
vecteur2 = Vecteur(0, 4)
print(vecteur1 + vecteur2)
```

***Instanciation***

On peut maintenant additionner nos objets comme s'il s'agissait de vulgaires nombres. On fait en fait : **vecteur1**.*__add__*(**vecteur2**).

L'opération a néanmoins un sens (addition de 2 vecteurs en maths) : on ne modifie pas l'opérande +.

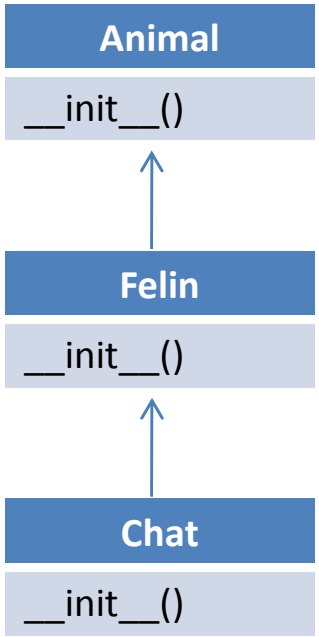
## 4- L'héritage simple

Une classe fille peut hériter d'une classe mère. Elle récupère les données et comportement de la mère.

```
# Héritage: utiliser super()
class Animal:
    def __init__(self):
        print("Je suis un animal")

class Felin(Animal):
    def __init__(self):
        super().__init__()
        print("Je suis un felin")

class Chat(Felin):
    def __init__(self):
        # Appel du constructeur parent
        super().__init__()
        print("Je suis un chat")

chat = Chat()
print(isinstance(chat, Chat))
print(isinstance(chat, Felin))
print(isinstance(chat, Animal))
```

***Classes avec héritage***

Appel explicite du constructeur du parent.



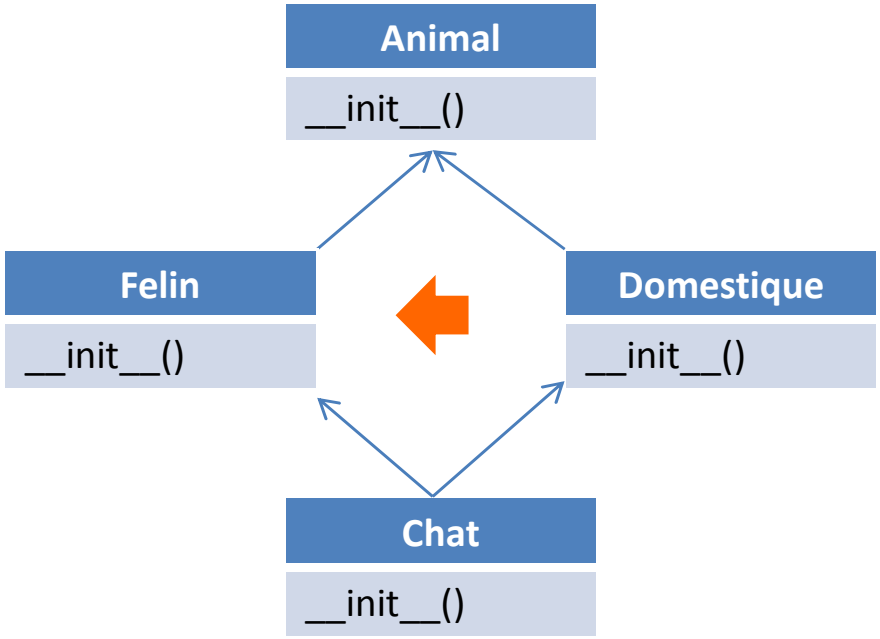

***Hiérarchie***

## 4- L'héritage multiple

Une classe fille peut hériter de plus d'une classe mère : héritage multiple (en losange). Utile parfois, très utilisé dans Django. Dans d'autres langages on utilise les interfaces.

```
# Héritage multiple (en losange)
class Animal:
    def __init__(self):
        print("Je suis un animal")

class Felin(Animal):
    def __init__(self):
        super().__init__()
        print("Je suis un felin")

class Domestique(Animal):
    def __init__(self):
        super().__init__()
        print("Je suis un animal domestique")

class Chat(Felin, Domestique):
    def __init__(self):
        super().__init__()
        print("Je suis un chat")

chat = Chat()
print(isinstance(chat, Domestique))
```

***Classes avec héritage multiple***




***Diagramme***

Le chat est animal qui est un félin domestique

## 5- Encapsulation, getters, setters et property

On définit les getters = méthodes publiques pour accéder aux données des objets, et les setters pour modifier ces données. Une méthode peut être « transformée » en attribut : c'est la notion de property.

```
# Getters, setters, property
class Vecteur:
    #constructeur avec deux attributs
    def __init__(self, x, y):
        self.__x = x # privé
        self.y = y

    # Méthodes d'accès/modif et property
    def getX(self):
        return self.__x

    def setX(self, val):
        self.__x = val
    x = property(getX, setX)

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

    #Redéfinition d'un opérateur
    def __add__(self, other):
        return Vecteur(x=self.x + other.x, y=self.y + other.y)

    @property
    def norme(self):
        "Calcul de la norme d'un vecteur"
        return sqrt(self.x**2 + self.y**2)

    longueur = norme#alias
```

Cet attribut est private

***Classes avec héritage multiple***

| Vecteur |
| --- |
| - x |
| + y |
| + getX() |
| + setX() |

***Diagramme***

```
#Instancier
vecteur1 = Vecteur(-4, 0)
vecteur1.x = 5
print(vecteur1)
```

***Instanciation***

La méthode est vue comme un attribut
@property est un décorateur

## 6- Attributs et méthodes statiques, constantes

On n'a pas besoin d'instancier un objet pour accéder aux attributs statiques (attributs de classe) ou pour utiliser une méthode statique (ou méthode de classe).

```
# Attributs et méthodes statiques
class Maclasse:
    # Attribu statique
    attribut = "Coucou"

    def __init__(self):
        print("Pas grd chose")

    # méthode statique
    @staticmethod   # décorateur
    def methodeStatique(x, y):
        return x + y

    def autre(self):
        return self.__class_.methodeStatique(1, 2)

# Récup de l'attribut
print(Maclasse.attribut1)

# Utiliser la méthode
print(Maclasse.methodeStatique(5, 6))
```

Ce décorateur indique une méthode statique

Appel de la méthode statique

***Aspect statique***

## 7- Une classe abstraite

On ne peut créer un objet avec une telle classe. On peut l'utiliser comme classe technique ou pour résoudre des problèmes de conception.  Il n' y a pas de mot clé en python pour la créer.

Utiliser pass dans les méthodes qui doivent être redéfinies dans la classe fille.

Rendre la classe abstraite

```
>>> class MaClasse:
...     def __init__(self):
...         if self.__class__ == MaClasse
...             raise Exception("Construction interdite")
...         else:
...             print("Constructeur")
...     def methodeSansContenu(self):
...         pass
...
>>> objet = MaClasse()
Exception: Construction interdite
```

*Classe abstraite*

```
>>> class ClasseFille(MaClasse):
...     def __init__(self):
...         print("Appel depuis la classe fille")
...         super().__init__()
...
...     def methodeSansContenu(self):
...         print("Appel de la méthode redéfinie")
...
>>> objet = ClasseFille()
Appel depuis la classe fille
Constructeur
```

*La classe fille*