Python - Programmation Orientée Objet

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Python - Programmation Orientée Objet

Qu’est-ce que la Programmation Orientée Objet ?

- La **POO** est est un paradigme de programmation informatique. Elle consiste en la **définition** et l'**interaction** de briques logicielles appelées **objets**. Un **objet** représente un **concept**, une **idée** ou toute **entité** du monde physique (personne, voiture, dinosaure).
- Lorsque que l'on programme avec cette méthode, la première question que l’on se pose est :
  « qu'est-ce que je manipule ? »
- Alors qu'en programmation **Procédurale**, c'est plutôt :
  « qu'est-ce que je fait ? »

A quoi sert-elle ?

- Elle permet de **découper** une grosse **application**, généralement floue, en une multitude d'**objets** interagissant entre eux
- La POO améliore également la **maintenabilité**. Elle facilite les **mises à jour** et l'ajout de **nouvelles fonctionnalités**.
- Elle permet de faire de la **factorisation** et évite ainsi un bon nombre de lignes de code
- La réutilisation du code fut un argument déterminant pour venter les avantages des langages orientés objets.

Les paradigmes de la POO

La POO repose sur plusieurs concepts importants :

- *__Accessibilité__ (ou __Visibilité__)
- __Encapsulation__
- __Polymorphisme__
- __Héritage__
- __Abstraction__
- *__Interfaces__
- __Fonctions Anonymes__
- __Généricité__

Qu’est-ce qu’un objet en programmation?

Commençons par définir les objets dans le mode réel:

- Ils possèdent des **propriétés propres** : Une chaise a 4 pieds, une couleur, un matériaux précis...
- Certains objets peuvent **faire des actions** : la voiture peut rouler, klaxonner...
- Ils peuvent également **interagir entre eux** : l’objet roue tourne et fait avancer la voiture, l’objet cric monte et permet de soulever la voiture...

Le concept d’objet en programmation s’appuie sur ce fonctionnement.

Qu’est-ce qu’un objet en programmation?

Il faut distinguer ce qu’est l’objet et ce qu'est la définition d’un objet

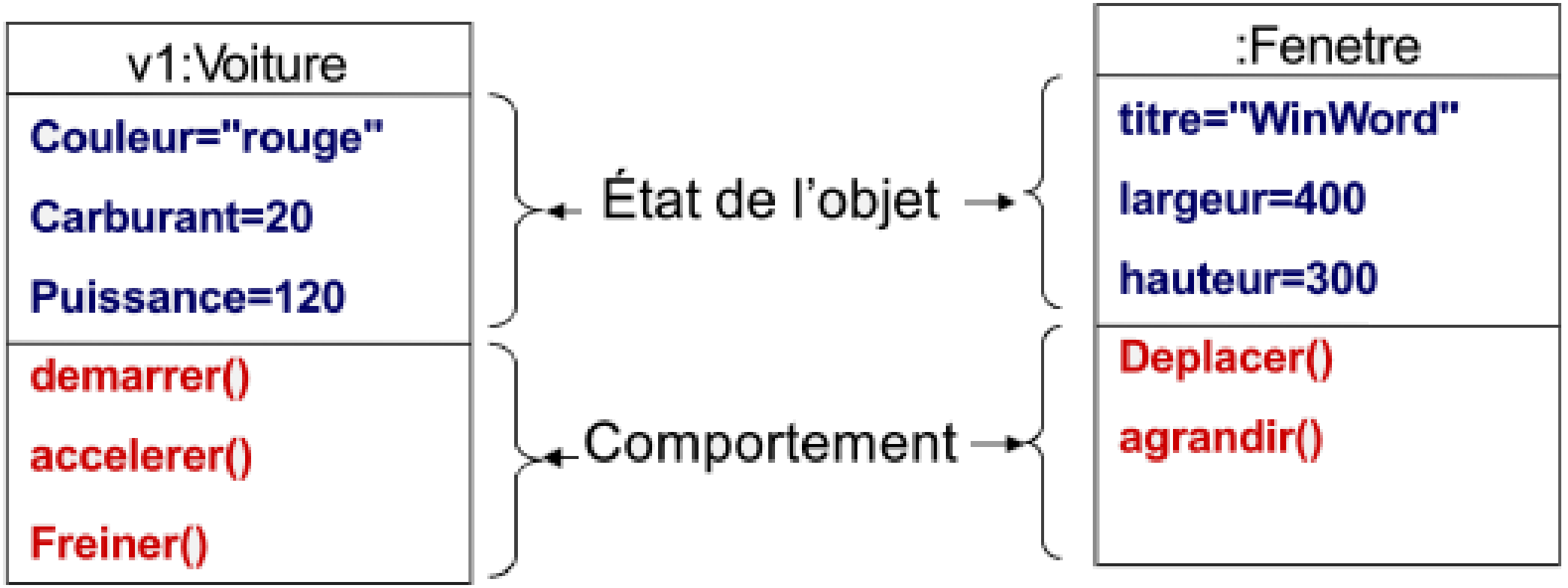
- **Le concept de l’objet** (ou définition/structure)
  - Permet d’indiquer ce qui compose un objet, c'est-à dire quelles sont ses propriétés, ses actions…
- **L’instance d’un objet**
  - C’est la création réelle de l’objet : *Objet Chaise*
  - En fonction de sa définition : *4 pieds, bleu…*
  - Il peut y avoir **plusieurs instances** : *Plusieurs chaises, de couleurs différentes, matériaux différents…*

Qu’est-ce qu’un objet en programmation?

- Un objet est une structure informatique définie par un **état** et un **comportement**.
  - L’**état** regroupe les **valeurs instantanées** de tous les **attributs de l’objet**. Il peut changer dans le temps.
  - Le **comportement** décrit les **actions** et les réactions de l’objet. Autrement dit le comportement est défini par **les opérations que l’objet peut effectuer**. Généralement, c’est le comportement qui modifie l’état de l’objet.

Exemple

Objet = état + comportement



Il s'agit ici d'un diagramme d'objet

Identité d'un objet

- En plus de son état, un objet possède une **identité** qui caractérise son existence propre.
- Cette identité s'appelle également **référence** de l'objet.
- En terme informatique de bas niveau, l'identité d'un objet représente son **adresse mémoire**.
- Deux objets **ne peuvent pas avoir la même identité**:
  c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire

Résumé

- La **POO** est un **paradigme de programmation** basé sur la manipulation d'**objets**, représentant des entités ou concepts du monde réel.
- Elle **découpe les applications** complexes en **objets**, améliorant ainsi la maintenabilité et favorisant la réutilisation du code.
- **Concepts clés** : Accessibilité, Encapsulation, Héritage, Polymorphisme, Abstraction, Interfaces, Fonctions Anonymes, Généricité.
- Un objet combine **état** (propriétés actuelles) et **comportement** (actions possibles), avec une **identité unique** (adresse mémoire).

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Définition de Classes

Qu’est-ce qu’une Classe ?

Un **Classe** (`class`) permet de regrouper tous les éléments qui représenteront un Objet : ses **attributs**, ses **propriétés**, ses **méthodes**
On dit qu'une classe représente le concept de l'objet.

Dans les langages fortement typés, **la création d'une classe** aboutira à la création d'un **nouveau Type**.

Instanciation

- Les objets qui sont **définis à partir** d'une classe **appartiennent à celle-ci**.
- Ce processus s'appelle l'**Instanciation**
- On passe du **concept** (classe) à l'objet **réel** (instance/objet)
- La **classe est unique** mais les **objets** qui en **dérivent** peuvent être nombreux.

Les classes et instances

En python, on définit une classe avec le bloc **class**. Tous les blocs **def** à l'intérieur créeront des **méthodes relatives à la classe** et non des fonctions.

```
class Chien:
  """ Représentation d'un chien """

  def __init__(self,nom,age,race):
    self.nom = nom
    self.age = age
    self.race = race

  def aboyer(self):
    print(f"Wouf Wouf {self.nom}")
```

Elements d'une classe

Élément	Caractéristiques	Détails
Attributs : Variables d'instance	- Nom - Valeur initiale (optionnelle)	**État** de l'objet
Méthodes : Fonctions liées à l'instance	**Signature** : - Nom - Paramètres	**Comportement** de l'objet
Constructeur	- Méthode __init__ - Paramètres	Appelés à la **création** de l'objet
Destructeur	Rarement utilisé, varie selon les langages	Méthode particulière appelée pour libérer la mémoire à la **suppression**

Le constructeur __init__

Le **constructeur** est le point d'entrée pour la création d'une **instances/objet** du **type de la classe (instanciation)**.

- Il s'agit d'une **méthodes** dite **Dunder** ou **Magique** (dont le nom commence et fini par deux caractères **underscore**).

```
class Chien:
  def __init__(self,nom,age,race):
    self.nom = nom
    self.age = age
    self.race = race
```

Instancier une classe

Le constructeur est appelé lorsque l'on souhaite instancier une classe, on écrit le **nom de la classe** et non __init__

```
chien_1 = Chien("REX", 12, "Berger Allemand")
```

Une fois la **variable** renvoyant vers **l'instance** de Chien créée, on peut la manipuler et utiliser ses méthodes.

```
chien_1.aboyer() # Wouf Wouf REX
```

Le paramètre de méthode 'self'

Pour référencer **l'objet instancié** lors de la déclaration des méthodes d'instance on utilise un **paramètre supplémentaire obligatoire** en **premier** (la norme est de le nommer **self**).

- Lors de l'appel de ces méthodes, **on ne devra pas renseigner cet argument**.

```
def __init__(self,nom,age,race):
  self.nom = nom
  self.age = age
  self.race = race

def aboyer(self):
  print(f"Wouf Wouf {self.nom}")
```

```
chien_1 = Chien("REX", 12, "Berger Allemand")
chien_1.aboyer() # Wouf Wouf REX
```

Constructeur par défaut (sans paramètres)

Lorsque l'on crée une **nouvelle classe vide**, on **pourrait penser** qu'il est **impossible de l'instancier si aucun constructeur n'est définit**.

En réalité, **il existe un constructeur vide par défaut** (implicite) dans toute classe qui **n'a pas encore de constructeur**.

```
class Personne:
    pass

p = Personne()  # Aucun __init__ défini, mais ça marche
```

Dès le moment où l'on en ajoute un nous-même, ce constructeur **disparaît**.

Les attributs

Les **attributs** sont un **ensemble de variables** permettant de définir les caractéristiques de notre objet (aussi appelés **variables d'instance**).

- Il est en général **définit** et **affecté** dans le **constructeur**.
- Il peut être **accédé** et **réaffecté** via la notation **objet.attribut**

```
chien_1.age = 6
print(f"Le chien s'appelle {chien_1.nom}, il a {chien_1.age} ans")
print(f"{chien_1} est donc né en {date.today().year-chien_1.age}")
```

Attributs par référence

Un **objet** est une valeur par référence, c'est-à-dire qu'il est **mutable** et qu'on peut ainsi **le passer en paramètre de fonction ou de méthode** et voir **s'opérer des changements** en cas de modifications éventuelles de ses attributs.

```
def change_nom(chien, nouveau_nom):
  chien.nom = nouveau_nom

chien_1.nom = "REX"
print(chien_1.nom) # REX
change_nom(chien_1, "Bill")
print(Chien_1.nom) # Bill
```

Les attributs implicites

Avancé

Ces attributs sont créés **par défaut** lors de la manipulation des classes et s'utilisent via la syntaxe **Dunder** (double underscore).
Pour la classe on a:

- **__name__**: le nom de la classe
- **__doc__**: commentaire associé à la classe
- **__dict__**: le dictionnaire des attributs statiques
- **__bases__**: un tuple des classes dont celle-ci hérite
- **__module__**: contient le nom du module dans lequel la classe a été définie

Pour l'instance on a:

- **__class__**: la classe de l'objet.
- **__dict__**: la liste des attributs d'instance

```
class MaClasse:
  """ une classe """
  test = 0
  def __init__(self):
    self.test1 = 1

cl = MaClasse()

# Classe
print(MaClasse.__name__) # MaClasse
print(MaClasse.__doc__) # une classe
print(MaClasse.__dict__) # {"test": 0, ...}
print(MaClasse.__bases__) # (<class 'object'>,)
print(MaClasse.__module__) # __main__

# Instance
print(cl.__class__) # <class '__main__.MaClasse'>
print(cl.__class__.__name__) # MaClasse
print(cl.__dict__) # {'test1': 1}
print(cl.__doc__) # une classe
```

La notion de visibilité/accessibilité

L'**indicateur de visibilité** est ce qui sert à indiquer **depuis où** on peut **accéder** à l'**élément** qui le suit.

Visibilité	Description	Classe	Sous-classe	Extérieur
public	Accès non restreint	✓	✓	✓
protected	Accès depuis la même classe ou depuis une classe dérivée (cf héritage)	✗	✓	✓
private	Accès uniquement depuis la même classe	✗	✗	✓

- En Python, la visibilité repose sur la confiance du développeur, pas sur une restriction technique comme avec Java ou C#.

- Les attributs peuvent être protégés avec des conventions de nommages, mais aucune interdiction stricte n'est appliquée :
  - Les attributs et méthodes publiques sont nommés « **ainsi** »
  - Les attributs et méthodes protégés sont nommés « **_ainsi** »
  - Les attributs et méthodes privés sont nommés « **__ainsi** »
- Pour les attributs privé, cette convention est nommée **name mangling** empêchant un accès direct via **objet.__attribut** (mais toujours possible avec **objet._nom-classe__nom-attribut**)
- Si un contrôle d'accès réel est nécessaire, on le gère via les propriétés (@property) ou des méthodes dédiées.

Les méthodes

Une **méthode** est l'équivalent d'une **fonction** qui est **associée** à un **objet** ou à une **classe**. Pour faire **appel** à une méthode, il faudra utiliser la notation **Classe.méthode()** ou **objet.méthode()**.

Une méthode d'instance peut accéder aux **attributs** de **l'objet auquel elle est associée** en passant encore une fois par le **paramètre self**, qu'elle doit avoir en tant que **premier paramètre**:

```
def aboyer(self):
  print(f"Wouf Wouf {self.nom}")
```

Une méthode peut réaliser tout ce qu'une fonction faisait de base, mais est en général utilisée pour **éviter d'avoir à passer en argument** des **valeurs** qui sont **déjà dans les attributs de l'objet**:

```
def afficher(self):
  print(f"Mon chien a {self.age} ans, il s'appelle {self.nom} de la race {self.race}")
```

Une méthode participe ainsi activement à la réalisation d'un code plus propre et à la mise en place du **DRY (Don't repeat yourself)** dans le cadre d'un programme.

Les propriétés

Avancé

- Les propriétés sont **trois méthodes magiques** qui sont appelées en cas de **récupération (getattr)**, **d'affectation (setattr)** ou de **suppression (delattr)** d'un attribut.
- Il est ainsi possible de **surcharger/override** ces méthodes magiques pour **en modifier le fonctionnement**.
- Cela évite ainsi d'avoir à répéter des lignes de codes et également la création de méthodes destinées à contrôler et à modifier les affectations ou les récupérations d'attributs d'objets (nommé getters et setters).

```
ma_temperature = Temperature(37.5)
ma_temperature.celcius = 25
print(ma_temperature.fahrenheit) # 99.5
```

```
class Temperature:
  def __init__(self, value):
    self.value = value

  def __getattr__(self,name):
    if name == 'celsius':
      return self.value
    if name == 'fahrenheit':
      return self.value * 1.8 + 32
    raise AttributeError(name)

  def __setattr__(self, name, value):
    if name == 'celcius':
      self.value = value
    if name == 'fahrenheit':
      self.value = (value - 32) / 1.8
    else :
      super().__setattr__(name, value)
```

Les propriétés

Avancé

- Python fournit également un décorateur **@property**. il permet d'appeler une **méthode** comme si on tentait **d'accéder** à un **attribut de l'objet** portant le **même nom**.
- Le décorateur **@.setter** permet d'appeler la méthode **méthode** comme si on tentait **de définir l'attribut de l'objet** portant le **même nom**.

```
@property
def nom(self):
  return self._nom

@nom.setter
def nom(self, nom):
  self._nom = nom

objet.nom = "le nom"
print(objet.nom)
```

```
@property
def age(self):
  today = date.today()
  age = today.year - self.birth_date.year - ((today.month, today.day) < (self.birth_date.month, self.birth_date.day))
  return age
```

Exercice

1. Créer un classe **Gateau**
2. Ajouter les attributs suivants et les initialiser dans le constructeur :
    1. **nom gâteau**:str
    2. **temps cuisson**:int
    3. **liste ingrédients**:list de str
    4. **étapes recettes**: list de str
    5. **nom du créateur**: str
3. Ajouter une méthode qui affiche les ingrédients de la recette
4. Instancier un objet gâteau qui affiche les ingrédients ainsi que les étapes de préparation du gâteau.

Exercice

1. Créer un classe **CompteBancaire** qui représente un compte bancaire, ayant pour attributs :
    1. **numeroCompte**:int
    2. **nom**:str
    3. **solde**:int
2. Créer un constructeur ayant comme paramètres: numero_compte, nom, solde.
3. Créer une méthode Versement() qui gère les versements
4. Créer une méthode Retrait() qui gère les retraits
5. Créer une méthode Agios() permettant d'appliquer les agios à un pourcentage de 5% du solde.
6. Créer une méthode afficher() permettant d'afficher les détails sur le compte.

Les attributs de classe

En plus des **attributs** liés à **un objet/instance**, il est possible de faire appel à ce qu'on appelle des **attributs de classe**.

- Ils sont **partagés** par l'ensemble **des objets de ce type**, ils sont **liés à la classe elle-même**.
- On peut par exemple se servir des attributs de classe pour compter facilement les objets instanciés de cette classe ou pour acceder à des valeurs communes à tous les éléments de ce type.

- Pour accéder à un attribut de classe, on doit se servir de la syntaxe **Classe.attribut**.

```
class Chien:
  instances_chien = 0
  nom_latin = "Canis lupus familiaris"

  def __init__(self, age, nom, race):
    Chien.instances_chien += 1
    self.age = age
    self.nom = nom
    self.race = race
```

```
print(f"Il y a {Chien.instance_chien} instances de chiens dont le nom latin est :{Chien.nom_latin}")
# Il y a 2 instances de chiens dont le nom latin est : Canis lupus familiaris
```

Les méthodes de classe

Une **méthode de classe** est **une méthode** qui, comme pour un attribut de classe, est **liée à la classe** et non à **l'objet**.

- Pour en définir une on utilise le décorateur **@classmethod**
- Elle a accès **à l'état de la classe** par le biais d'un paramètre que l'on nomme **cls** par convention, il est en **premier** (à la place de `self`) et **pointe vers la classe** et non l'objet.
- Elle permet donc de **manipuler les attibuts de classes**, d'appeler **d'autres méthodes de classes** ou encore de **créer de nouvelles instances**.

- On accèdera aux attributs de classe avec **cls.attribut**
- Pour faire appel à une méthode de classe on utilise la syntaxe suivante : **Classe.méthode_de_classe**.

```
@classmethod
def afficher_nombre_chiens(cls):
  print(f"Il y a {cls.nombre_chiens} chiens instanciés")
```

```
chien = Chien(5, "Rex", "Berger Allemand")
Chien.afficher_nombre_chiens()
# Il y a 1 chiens instanciés
```

Les méthodes statique

Une **méthode statique** est **une méthode liée à la classe** tout comme une méthode de classe mais **elle ne reçoit aucun premier argument implicite** (`self` ou `cls`).

- Cette méthode ne peut pas accéder ou modifier l'état de la classe (mais elle peut y accéder indirectement via Classe.méthode).
- Elle est destinées à avoir un comportement qui ne change pas, elle est comme une fonction classique en soit.
- Les méthodes statiques sont donc souvent utilisé dans un but utilitaire pour afficher du texte ou utilisé d'autres méthodes.

- Pour faire une méthode statique, il faut donc utiliser le décorateur **@staticmethod** et on l'appellera dans le coeur de notre programme (comme pour les méthodes de classe) la syntaxe **Classe.méthode()**:

```
class Chien:
    nombre_chiens = 2

    @staticmethod
    def seuil_chien(max):
        print(f"Il y a {max - Chien.nombre_chiens} places disponibles dans le refuge")

# Appel via la classe
Chien.seuil_chien(10)  # Il y a 8 places disponibles dans le refuge
```

Différence entre méthode de classe et méthode statique

Méthode de classe	Méthode statique
Une méthode de classe prend comme **premier paramètre cls** (la classe)	Une méthode statique n'a **pas d'arguments par défaut**
Une méthode de classe peut **accéder et modifier l'état d'une classe via le paramètre cls**	Une méthode statique ne peut pas **accéder ou modifier l'état d'une classe sans utiliser la syntaxe avec le nom de la classe**.
La méthode class prend **la classe comme paramètre pour connaître l'état de cette classe (cls)**	Les méthodes statiques **ne connaissent pas l'état de la classe**. Ces méthodes sont utilisées pour effectuer certaines tâches utilitaires en prenant certains paramètres, comme des fonctions.
Utilisation du décorateur **@classmethod**	Utilisation du décorateur **@staticmethod**

Exercice

1. Créer un classe **WaterTank** qui possédera **les attributs d'instance** suivants :
    1. **Poids** de la citerne à **vide**: float
    2. **Capacité maximale**: float
    3. **Niveau de remplissage**:float
2. Créer les **méthodes** suivantes propre à chaque instance de classe:
    1. Méthode indiquant le **poids total**
    2. Méthode pour **remplir la citerne** avec un **nombre de litre d'eau**
    3. Méthode pour **vider la citerne** d'eau d'un **nombre de litre d'eau**
3. Créer un **atribut de classe** qui contiendra **la totalité des volumes d'eau** des citernes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Héritage

Le concept de l'héritage

L'héritage est un mécanisme fortement utilisé dans la programmation orienté objet.

- Une classe peut **hériter** d'une **autre classe**, dans ce cas elle en possédera **les membres** (méthodes / attributs), on dit aussi qu'elle **dérive** de l'autre classe.
- On parle alors de **classe fille/enfant** (spécialisé) et de **classe mère/parent** (général)
- Pour **réaliser un héritage** en Python il suffit **d'ajouter des parenthèses** après le nom de la classe que l'on créé et d'y **ajouter la classe dont l'on souhaite hériter**.

Exemples

```
class Chien(Mammifere):
  pass
```

- `Chien` est un enfant de la classe `Mammifere`
- La classe `Mammifere` est une sorte de la classe `Animal`
- La classe `Animal` est une sorte de la classe `ÊtreVivant`

Chaque **parent** est un plus **général** que son **enfant**. Et inversement, chaque **enfant** est plus **spécialisé** que son **parent**.
L'**enfant** aura donc **les caractéristiques du parent** auxquelles s'ajoute ses **spécificités**.

Exemples complet

```
class Mammifere:
  nb_mammifere = 0
  def dormir(self):
    print("Zzzzz")

class Chien(Mammifere):
  def __init__(self, nom, age):
    self.nom = nom
    self.age = age
    Mammifere.nb_mammifere += 1

chien = Chien("Idéfix", "White terrier")
chien.dormir() # Zzzzz
print(Chien.nb_mammifere) # 1
```

L'utilisation de la méthode super()

- Lors d'un **héritage**, il est possible **d'accéder aux attributs et aux méthodes de la classe mère** à partir de la classe enfant.
- Si l'on souhaite avoir accès à la méthode **calc_age(annee)** de la classe **Mammifère** pour se servir du résultat dans la classe enfant, on doit utiliser le mot-clé **super()** pour **accéder à la classe parent**, puis la syntaxe **super().nom_méthode()** pour en **appeler la méthode**.

- Le mot clé **super()** est également utilisé dans le cadre d'un **constructeur** pour faire appel au **constructeur de la classe parent** qui pourrait avoir besoin de paramètres, comme ci-dessous

```
class Mammifere:
    nombre_mammifere = 0
    def __init__(self):
        Mammifere.nombre_mammifere += 1

    def calculer_age(self, annee_naissance: int) -> int:
        return date.today().year - annee_naissance


class Chien(Mammifere):
    def __init__(self, nom: str, annee_naissance: int, race: str):
        super().__init__()
        self.nom = nom
        self.annee_naissance = annee_naissance
        self.race = race

    def age_chien(self) -> int:
        return super().calculer_age(self.annee_naissance)
```

La classe object

Chaque classe du Python va **automatiquement hériter** d'une classe qui se nomme "**object**". Cette classe comporte **une série de méthodes et d'attributs** qui seront ainsi automatiquement hérités par les classes enfants (**__str__**, **__getattr__** et/ou **__setattr__**).

```
class Personne:
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age

personne_1 = Personne("Dupont", "Jean", 40)
print(personne_1)
# Appel __str__ d'object : <__main__.Personne object at 0x00000209C4636F90>
```

Exercice

1. Écrire une classe **Rectangle** en langage Python, permettant de construire un rectangle doté **d'attributs longueur et largeur**.
2. Créer une méthode **perimetre()** permettant de calculer le périmètre du rectangle et une méthode **surface()** permettant de calculer la surface du rectangle
3. Créer une classe fille **Pave héritant de la classe Rectangle** et dotée en plus d'un **attribut hauteur** et d'une autre méthode **volume()** permettant de calculer le volume du Pavé.
4. Surcharger les méthodes **périmètre()** et **surface()** du **Pavé** pour avoir les bon résultats.

Les classes abstraites

Dans notre exemple précédent, nous pourrions avoir `Mammifere` en classe abstraite car par la présence de leurs spécialisations, leur **instanciation devient incohérente**, ***abstraite***.

De la même façon, une **méthode abstraite** est une méthode qui ne contient **pas d'implémentation**.

- Elle n'a **pas de corps** (pas de block de code)
- Une méthode abstraite sera toujours dans une classe abstraite.

Le module ABC (Abstract Base Class)

En Python, le module `abc` permet de définir des **classes abstraites**, c'est-à-dire des classes **non instanciables** servant de **modèles** pour d'autres classes.
Ce module fournit notamment :

- la classe `ABC`, qui permet de **transformer une classe Python ordinaire en classe abstraite**.
- le décorateur `@abstractmethod`, qui permet de **déclarer une méthode abstraite**, c'est-à-dire une méthode que **toutes les classes dérivées doivent implémenter**.

Exemple

```
from abc import ABC, abstractmethod

class Animal(ABC):
  @abstractmethod
  def crier(self):
    pass

class Chien(Animal):
  def crier(self): # definition de crier imposé
    print("Whouaf whouaf!")

# a = Animal() -> Impossible ! car abstraite
c = Chien()
c.crier()
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Polymorphisme

Le concept de Polymorphisme

Le mot **polymorphisme** suggère qu'un élément **définit par son nom** (**identificateur/symbol**) possède **plusieurs formes**.

- Il aura ainsi la capacité de faire **une même action** avec **différents types d'intervenants**.
- Le polymorphisme consiste en l'utilisation d'une **version différente d'une méthode**.

Les types de Polymorphisme

Il y a plusieurs types possibles de **polymorphisme** en **POO**:

- Les polymorphisme **avec signatures différentes**
  - par **Surcharge / Overload** (aussi nommé « **ad hoc** »)
  - **Paramétrique**
- Les polymorphisme de l'**Héritage**
  - par **Masquage / Shadowing**
  - par **Substitution / Override**

En python, seuls les polymorphisme paramétriques (généricité) et par substitution sont possibles.

Exemple

```python
class Personne:
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age

    def jouer(self):
        print("L'adulte n'a plus le temps de jouer")

class Enfant(Personne):
    def __init__(self, nom, prenom, age, jouet):
        super().__init__(nom, prenom, age)
        self.jouet = jouet

    def jouer(self): # Réécrit la définition de la méthode
        print(f"L'enfant joue avec {self.jouet}")

liste_personnes = [
    Personne("Jean", "Dupont", 30),
    Enfant("Titou", "Enfant", 5, "Légo")
]
for personne in liste_personnes:
    personne.jouer()
```

Duck typing

Le **Duck Typing** est un concept de Python qui permet le polymorphisme par **comportement**. Il ne vérifie **pas le type d'un objet**, mais la **présence des méthodes** ou **attributs nécessaires** à son utilisation.

- **"If it walks like a duck, and it quacks like a duck, then it must be a duck"**.
- Il permet de regrouper différentes classes selon les méthodes qu'elles ont en commun (comparable aux interfaces).

Exemple

```python
class Avion:
    def __init__(self, nom, nb_moteur):
        self.nom = nom
        self.nb_moteur = nb_moteur

    def decoller(self):
        print("L'avion décolle !")
```

```python
class Canard:
    def __init__(self, nom, couleur):
        self.nom = nom
        self.couleur = couleur

    def decoller(self):
        print("Le canard décolle !")
```

```python
liste_volant = [
    Avion("Boeing",8),
    Canard("Daphy", "brun")
]
for volant in liste_volant:
    volant.decoller()
```

Exercice

1. Créer une classe **Personne**, contenant le nom de la personne, son **prénom**, son **numéro de téléphone** et son **email**. Une méthode **__str__** pour afficher les données de la personne.
2. Créer une classe **Travailleur** , qui hérite de la classe Personne et étend avec les attributs **nom d'entreprise, adresse entreprise et téléphone professionnel**. Une méthode **__str__** pour afficher les données et qui **réutilise celle de Personne**.
3. Créer une classe **Scientifique** qui hérite de la classe Travailleur et étend avec les attributs de type list **disciplines** (physique, chimie, mathématique, ...) et **types du scientifique** (théorique, expérimental, informatique...) Une méthode __str__ pour afficher les données et **qui réutilise celle de Travailleur**.

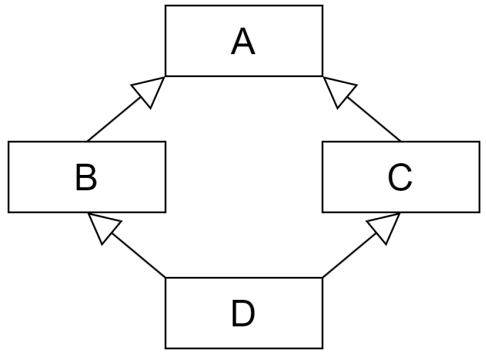
UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Multi-héritage

Le Multi-héritage

Dans la majorité des langages de programmation, l'**héritage multiple** n'est pas supporter pour éviter le problème de l'**héritage en diamant**.

- Si `B` et `C` **héritent** de `A` et `D` **hérite** de `B` et `C`, **quelle version** de `A` **doit être utilisée** par `D` ?



- En Python, c'est possible car l'interpréteur résoud le problème en usant de ce que l'on appelle le « **MRO** » (Method Resolution Order).

Le MRO (Method Resolution Order)

Il s'agit d'une liste contenant **l'ordre d'apparition des classes** servant pour l'héritage d'une classe.

- On peut y accéder avec la méthode: **NomDeClasse.mro()**

Ainsi lors d'un **héritage multiple**, le MRO déterminera l'ordre d'utilisation des méthodes/constructeur et empêcher les conflits.

- Il faudra bien faire attention à se servir du constructeur de la super-classe via l'utilisation du mot-clé **super()**, qui va en réalité chercher dans la MRO le constructeur dont on a besoin pour éviter les conflits.

Exemple

```python
class EtreVivant:
    def se_nourrir(self):
        print("Un être vivant se nourrit")

class Animal(EtreVivant):
    def dormir(self):
        print("L'animal dort")

    def se_nourrir(self):
        print("L'animal mange")

class Carnivore(EtreVivant):
    def chasser(self):
        print("Le carnivore chasse")

    def se_nourrir(self):
        print("Le carnivore mange")
```

```python
class Toutou(Animal, Carnivore):
    """Un chien est à la fois un animal et un carnivore"""

toutou = Toutou()
toutou.se_nourrir() # L'animal mange
toutou.chasser() # Le carnivore mange

# [<class '__main__.Toutou'>,
# <class '__main__.Animal'>, <class '__main__.Carnivore'>,
# <class '__main__.EtreVivant'>, <class 'object'>]
print(Toutou.mro())
```

Exercice

```python
class Address:
  def __init__(self, street, city):
      self.street = str(street)
      self.city = str(city)

  def show(self):
      print(self.street)
      print(self.city)
```

```python
class Person:
    def __init__(self, name, email):
        self.name = name
        self.email= email

    def show(self):
        print(self.name + ' - ' + self.email)
```

1. Créer la classe **Contact** qui **hérite à la fois** de **Address** et **Person**, cette classe doit implémenter la méthode **show()**
2. Créer une classe **Notebook** qui contient un **dictionnaire** qui associe **les noms des personnes** à un **objet Contact**. (Pas besoin d'héritage)
   - Cette classe devra avoir une méthode **show()**
   - Cette classe doit avoir une méthode **add(self, name, email, street, city)**
3. Tester le code suivant :

```python
notes = Notebook()
notes.add('Alice', '<alice@example.com>', 'Lv 24', 'Sthlm')
notes.show()
```

```
=== Alice ===
Alice - <alice@exemple.fr>
lv 24
sthlm
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Gestion des exceptions

Qu'est-ce qu'une exception ?

Une **exception** est un **événement anormal** qui se produit **pendant l'exécution d'un programme** et qui **interrompt son déroulement**. On parle d'**exception** car il s'agit d'un cas particulier que le programme n'a pas su gérer automatiquement (ex : division par zéro).

- Pour **réaliser un programme fonctionnel**, il faut **anticiper les erreurs possibles** et les traiter pour qu'**elles soient non bloquantes**.
- Nous pouvons également **lever** des exceptions même si le programme ne trouve aucun problèmes (ex: mauvaise saisie).

Attraper une exception

Pour attraper une exception, il faut faire appel à un bloc de type **try…except…else…finally**. Ce bloc est constitué de quatre parties :

- Le bloc **try** sert à contenir l'ensemble du **code que l'on souhaite exécuter** et qui **pourrait poser problème** lors de l'exécution.
- Le bloc **except** sert à **récupérer l'exception** dans le but de **la traiter** de façon à ce **qu'elle ne bloque pas le programme**.
- Le bloc **else** sert à **exécuter du code** dans le cas où **aucune exception n'a été récoltée**.
- Le bloc **finally** sert quant à lui à **exécuter du code à la fin de l'ensemble du bloc** peut importe il y a eu une exception ou non.

Exemple

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def set_age(self,age):
        if age < 0:
            raise ValueError
        if age > 100:
            raise Exception
        self.age = age
```

```
person = Person("John", 18)

try:
    person.set_age(20)
except ValueError:
    print("Saisie invalide !")
except Exception:
    print("Une autre exception a été levée")
else:
    print("Saisie valide !")
finally:
    print("après le try, avec ou sans exeception levées")
```

Exceptions personnalisées

Nous pouvons également créer une exception nous-même, il nous suffit de créer une classe qui héritera d'**Exception** ou de **BaseException**.

```
class AgeInvalideException(Exception):
    pass

def input_age():
    try:
        age = int(input("Saisir votre Age : "))
        if age<= 0 or age >=120:
            raise AgeInvalideException("Age invalide")
    except AgeInvalideException as aie:
        print(aie)
        return -1
    else:
        print("Age valide !")
        return age
```

Exercice

Via la gestion des exceptions et la levée d'exceptions personnalisées, vous devrez réaliser un programme en console qui **demandera à l'utilisateur un login** ne devant **comporter que des lettres** et un **mot de passe ne comportant que des chiffres**. Dans le cas contraire, vous devrez **lever une exception** qui ne **devra pas stopper** le fonctionnement du programme mais **s'afficher afin d'informer à l'utilisateur que ses informations sont incorrectes**

```
Veuillez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aaa
Veuillez entrer un mot de passe SVP (celui-ci ne doit posséder que des chiffres) : dd
Le mot de passe ne dois posséder que des nombres !

Veuillez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : Aa
Il ne dois y avoir que des minuscules dans le login !
Veuillez entrer un mot de passe SVP (celui-ci ne doit posséder que des chiffres) : 47

Veuillez entrer un login SVP (celui-ci ne doit posséder que des lettres minuscules) : aa
Veuillez entrer un mot de passe SVP (celui-ci ne doit posséder que des chiffres) : 47
```