جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

ENSAH

Programmation Python

2023-2024

Prof : Anouar RAGRAGUI

Plan du cours

- Chapitre 1: Introduction
- Chapitre 2: Les bases du langage Python
- **Chapitre 3: Notions avancées du langage Python**

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

ENSAH

Chapitre 3: Notions avancées du langage Python

2023-2024

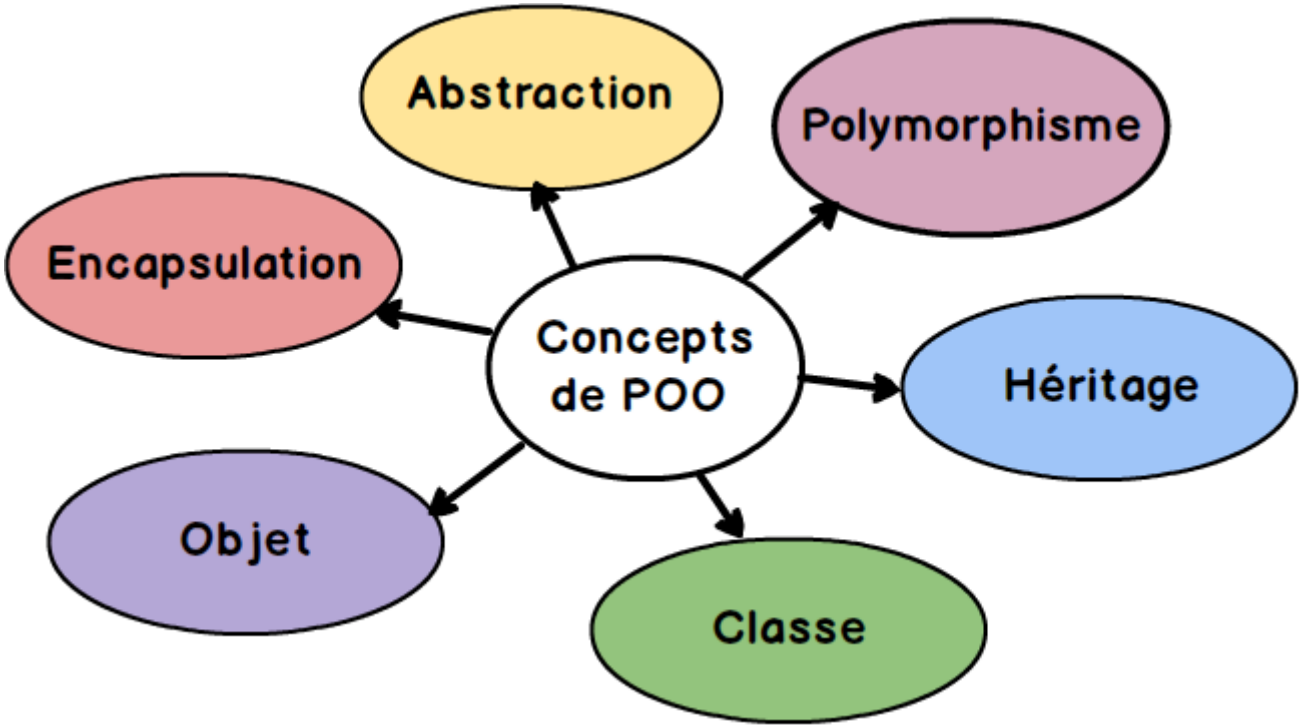
Prof : Anouar RAGRAGUI

Plan

- La programmation orientée objet avec Python
- Manipulation des fichiers
- Les exceptions et la gestion des erreurs
- Accès aux bases de données
- Expressions régulières
- Librairie standard
- Interfaces graphiques et Tkinter

- **La programmation orientée objet (POO)** est un concept de programmation très puissant qui permet de **structurer** ses programmes d'une manière nouvelle.
- En **POO**, on définit un « objet » qui peut contenir des « attributs » ainsi que des « méthodes » qui agissent sur lui-même.
- La **POO** permet de rédiger du code plus compact et mieux ré-utilisable.
- De plus, la **POO** amène de nouveaux concepts tels que le polymorphisme ou bien encore l'héritage.
- En **Python**, on utilise une « **classe** » pour construire un objet.

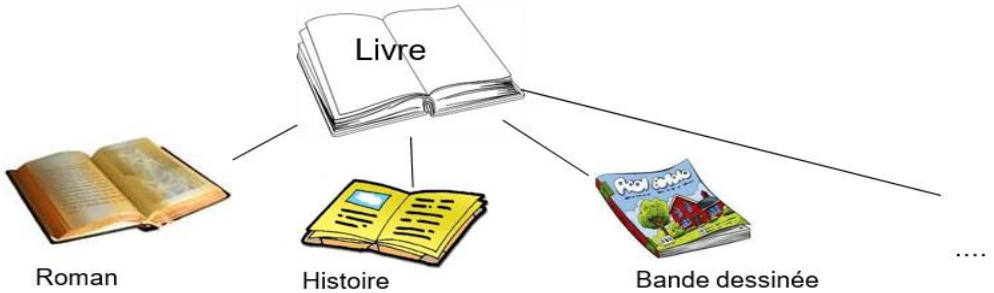
- Les grands principes de la **P.O.O.** sont



- Un **objet** est une entité qui comprend :
  - une partie structurelle qui décrit son état (caractéristiques) et ses liens avec d'autres objets (**attributs ou propriétés**)
  - une partie opérationnelle qui décrit ses comportements (**méthodes**)
- Un **objet** est créé selon un modèle ou un prototype qu'on appelle une **classe.**
- Les **objets** sont des instances d'une classe.
- Exemple : Dans une application de gestion d'une bibliothèque, on peut trouver plusieurs **objets** de **type livre** : ***Roman***, ***Histoire***, ***Bande dessinée***,

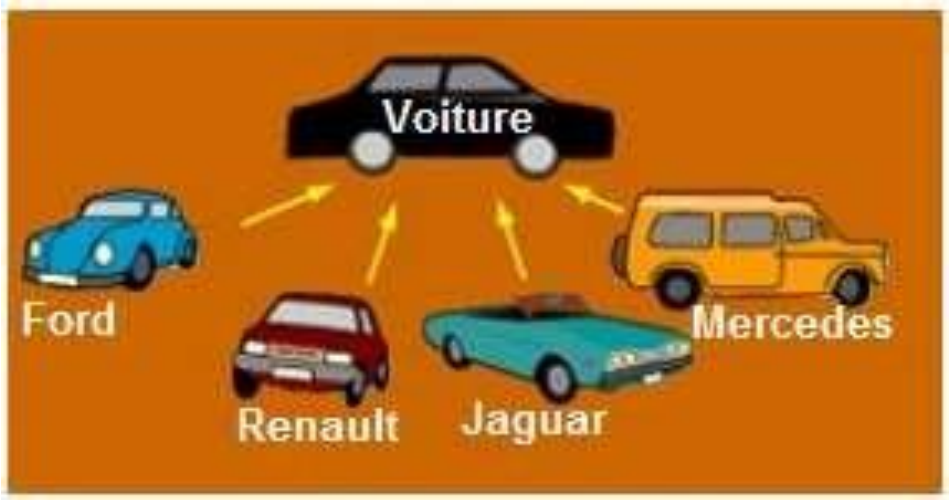
On peut dégager des **types d'objets.**

La notion de **classe** correspond à cette notion de **type d'objet.**

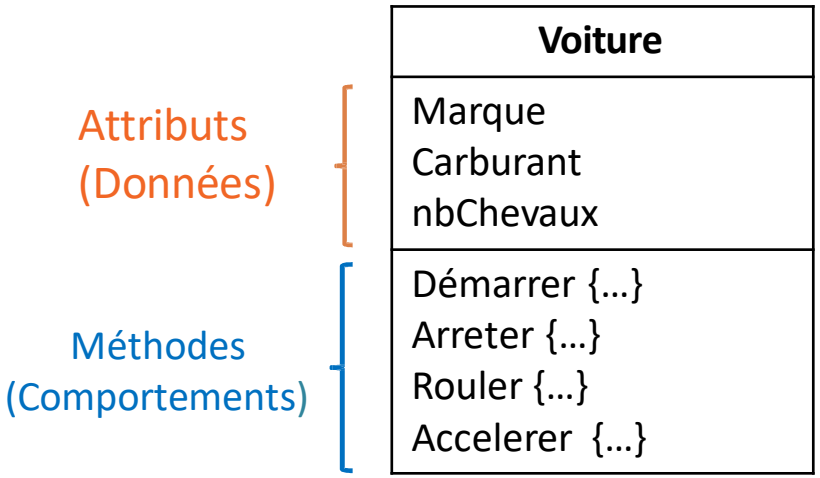


Notion de classe:

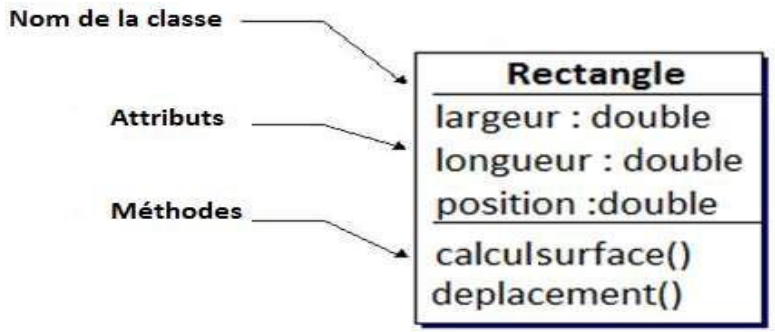
- Un **objet** est une "variable" (presque) qu'il faut déclarer avec son **type**.
- Le **type** d'un **objet** est un type complexe (par opposition aux types primitifs : entier, caractère, …) qu'on appelle une **classe**.
- Un **objet** est une **variable** dont le type est celui de la classe dont il est issu.
- Exemple :



- Une **classe** est une entité:
  - Représentant un ensemble d'objets ayant des données et les mêmes comportements (méthodes)
  - Capable de générer des instances.
- La classe contient :
  - des **attributs (données).**
  - des **méthodes** qui définissent les comportements de classe.
- Une classe permet de créer les objets : Classe = Modèle d'objets.
- Exemple:



- Exemple : Considérons la classe Rectangle définie pour créer les objets représentant des rectangles.



- Un objet de type **Rectangle** est caractérisé par :
  - Ses **attributs** (données) : largeur , longueur, position, …
  - Ses **méthodes** (comportements) : opérations qu'on peut effectuer sur un objet de type rectangle : calcul de la surface, déplacement dans le plan, …
- **Classe :** Regroupement de données (attributs) et de méthodes (fonctions membres)
- **Objet : instance de classe**
  - Attributs et méthodes communs à tous les objets d'une classe
  - Valeurs des attributs propres à chaque objet
- **Encapsulation** En programmation orientée-objet pure : l'encapsulation des données : accès unique des données à travers les méthodes

- En **Python**, le mot-clé `class` permet de créer sa propre classe, suivi du nom de cette classe.
- Comme d'habitude, cette ligne attend un bloc d'instructions indenté définissant le corps de la classe :

```
class Nom_Classe:
    #corps de la classe
    #...
```

- En **Python**, il n'y a pas de véritable distinction entre les modificateurs d'accès **public**, **privé** et **protégé** comme dans certains autres langages de programmation tels que Java.
- Cependant, Python utilise des conventions pour indiquer l'intention de l'accès aux attributs et aux méthodes.

- **Les objets** suivent les règles habituelles concernant leur **initialisation** par défaut
- Il est donc nécessaire de faire **appel** à **une fonction membre** pour attribuer des valeurs aux données d'un objet.
- **Un objet** doit effectuer un certain nombre d'opérations nécessaires à son bon fonctionnement
- L'absence de procédure d'initialisation peut alors devenir **catastrophique**.
- **La fonction membre spéciale est le :`__init__` :**
  - initialise les objets de la classe
  - La méthode init est similaire aux **constructeurs** en C++ et Java.
  - Le paramètre **self** est une référence à l'instance actuelle de la classe et est utilisé pour accéder aux variables **appartenant** à la **classe**.
  - Cette **méthode** est automatiquement appelée lorsqu'une nouvelle instance de la classe est créée.

- **Exemple: Réaliser une classe Personne** avec deux attributs nom et age et une fonction membre initialise:

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


personne1 = Personne("Ahmed", 30)

print(personne1.nom)
print(personne1.age)
```

```
Ahmed
30
```

- **Remarque : La fonction** `__str__()` **:** contrôle ce qui doit être renvoyé lorsque l'objet de classe est représenté sous forme de chaîne.

- **Exemple:**

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


personne1 = Personne("Ahmed", 30)
print(personne1)
```

```
<__main__.Personne object at 0x000002BE0B97D010>
```

La représentation sous forme de chaîne d'un objet **SANS** la fonction `__str()__`:

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __str__(self) :
        return f"Le nom est {self.nom} et l'age est {self.age}"

personne1 = Personne("Ahmed", 30)
print(personne1)
```

```
Le nom est Ahmed et l'age est 30
```

La représentation sous forme de chaîne d'un objet **AVEC** la fonction `__str()__`:

- Le concept **d'encapsulation** est un concept très utile de la **POO**. Il permet en particulier d'éviter une **modification** par erreur des données d'un objet.
- En effet, il n'est alors pas possible **d'agir directement** sur les données d'un objet
- Il est nécessaire de passer par ses **méthodes** qui jouent le rôle d'interface obligatoire
- Pour mettre en œuvre **l'encapsulation** en **Python,** on utilise souvent des conventions de nommage et des **méthodes spéciales** (appelées méthodes d'accès ou getters/setters).

- **Public** : En Python, les **attributs** et les **méthodes** qui ne sont pas préfixés par un **underscore** _ sont généralement considérés comme **publics** et peuvent être **accédés de l'extérieur de la classe**.
- **Privé** : Les **attributs** et les **méthodes** qui sont préfixés par un double **underscore** __ sont considérés comme **privés** et ne doivent généralement pas **être accédés directement de l'extérieur de la classe**. Cependant, ils ne sont pas complètement privés en Python, car ils peuvent toujours être accédés en utilisant le nom de la classe avec l'attribut ou la méthode.
- **Protégé** : Les **attributs** et les **méthodes** qui sont préfixés par un seul **underscore** _ sont généralement considérés comme **protégés**. Cela signifie qu'ils sont destinés à être utilisés à l'intérieur de la classe et de ses sous-classes, mais pas à **l'extérieur de la classe**. Cependant, il n'y a pas de mécanisme strict pour empêcher l'accès à ces attributs ou méthodes depuis l'extérieur de la classe.

- Exemple 1:

```
class MaClasse:
    def __init__(self):
        self.public_attribut = "Public"
        self._protege_attribut = "Protégé"
        self.__prive_attribut = "Privé"
```

- Exemple 2:

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

    def get_solde(self):
        return self.__solde

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

    def retrait(self, montant):
        if montant <= self.__solde:
            self.__solde -= montant
        else:
            print("Solde insuffisant")


mon_compte = CompteBancaire(1000)
print("Solde actuel:", mon_compte.get_solde())
mon_compte.depot(500)
mon_compte.retrait(200)
print("Nouveau solde:", mon_compte.get_solde())
print("Solde ", mon_compte._CompteBancaire__solde)
```

```
Solde actuel: 1000
Nouveau solde: 1300
Solde  1300
```

- **Méthodes de classe :**
  - Les méthodes de classe sont des méthodes qui agissent sur la **classe elle-même** plutôt que sur des instances de la classe.
  - Elles sont décorées avec `@classmethod` et prennent habituellement **cls** comme premier argument (conventionnellement appelé la classe elle-même).
- **Méthodes d'instance :**
  - Les méthodes d'instance agissent sur **des instances spécifiques** de la classe.
  - Elles prennent **self** comme premier argument, qui est une référence à l'instance elle-même.

- **Exemple** : Supposons que nous ayons une classe **Voiture** avec deux méthodes : une méthode de classe appelée **compteur_voitures()** qui compte le nombre total de voitures créées, et une méthode d'instance appelée **afficher_details()** qui affiche les détails de la voiture instanciée.

```
class Voiture:
    nombre_voitures = 0

    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
        Voiture.nombre_voitures += 1
    @classmethod
    def compteur_voitures(cls):
        return cls.nombre_voitures

    def afficher_details(self):
        print(f"Marque : {self.marque}, Modèle : {self.modele}")
voiture1 = Voiture("Toyota", "Corolla")
voiture2 = Voiture("Honda", "Civic")

print("Nombre total de voitures :", Voiture.compteur_voitures())
voiture1.afficher_details()
voiture2.afficher_details()
```

- Les **attributs de classe** et les **attributs d'instance** sont deux concepts clés en programmation orientée objet.
- **Attribut de classe :**
  - Un attribut de classe est partagé par toutes les instances d'une **classe**.
  - Il est défini à l'intérieur de la classe mais à l'extérieur de toute méthode.
  - Ils sont souvent utilisés pour **stocker des valeurs** ou des propriétés qui sont communes à toutes les **instances** de la classe.
  - Ils peuvent être accédés en utilisant le **nom de la classe** ou une **instance** de la classe.
- **Attribut d'instance :**
  - Un **attribut d'instance** est spécifique à **chaque** instance de la classe.
  - **Chaque instance** a sa propre **copie** de cet attribut.
  - Ils sont généralement définis à l'intérieur du constructeur (`__init__`) en utilisant le mot-clé **self**.
  - Les **attributs d'instance** stockent des valeurs qui sont **uniques** à chaque instance.

- **Exemple :**

```
class Voiture:
    roues = 4  # Attribut de classe
    def __init__(self, couleur):
        self.couleur = couleur  # Attribut d'instance

voiture1 = Voiture("rouge")
print(voiture1.roues)  # Sortie: 4
print(voiture1.couleur)# Sortie: rouge
voiture2 = Voiture("bleu")
print(voiture2.roues)  # Sortie: 4
print(voiture2.couleur)# Sortie: blue
```

- **Remarque**: Les attributs de classe ne peuvent pas être modifiés ni à l'extérieur d'une classe, ni à l'intérieur d'une classe. Puisqu'ils sont destinés à être identiques pour toutes les instances, cela est logique de ne pas pouvoir les modifier via une instance.

- En **Python,** la **surcharge** des opérateurs est souvent appelée méthodes magiques ou méthodes spéciales.
- Ces **méthodes spéciales** commencent et se terminent par un double soulignement (__).
- Voici quelques exemples :
  - Pour surcharger l'opérateur d'addition (+), vous définissez la méthode `__add__`.
  - Pour surcharger l'opérateur de soustraction (-), vous définissez la méthode `__sub__`.
  - Pour surcharger l'opérateur de multiplication (*), vous définissez la méthode `__mul__`.

+	__add__
-	__sub__
*	__mul__
/	__truediv__
//	__floordiv__
<	__lt__ (less than)
>	__gt__ (greater than)
<=	__le__ (less or equal)
>=	__ge__ (greater or equal)
==	__eq__
	...

- Exemple:

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

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)


point1 = Point(1, 2)
point2 = Point(3, 4)

result = point1 + point2
print("Somme des points :", result.x, result.y)
```

```
Somme des points : 4 6
```

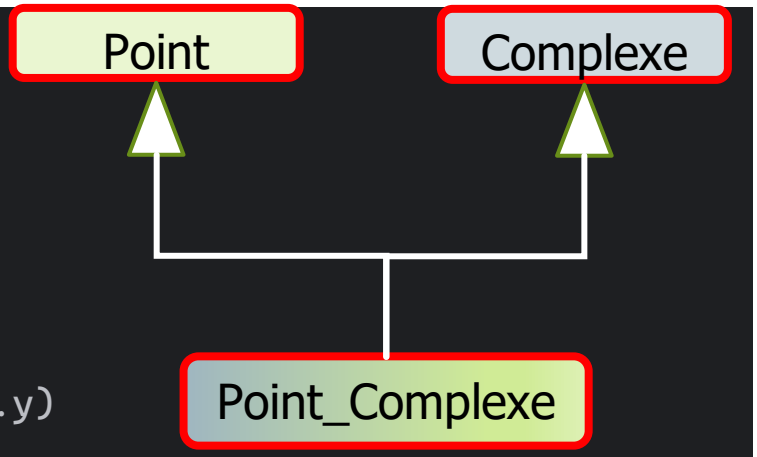
- En **programmation, l'héritage** est la capacité d'une **classe** d'hériter des propriétés d'une classe **préexistante.**
- On parle de classe mère et de classe fille.
- En Python, l'héritage peut être **multiple** lorsqu'une classe fille hérite de plusieurs classes mères.
- Il permet de réutiliser une classe existante:
  - **Relation de parenté** : Lorsqu'une **classe** hérite d'une autre **classe,** elle établit une **relation de parenté.** La classe enfant hérite des **propriétés** et des **comportements** de la classe parent.
  - **Réutilisation du code** : L'héritage permet la **réutilisation** du code, car les **attributs** et les **méthodes** définis dans la **classe parent** peuvent être utilisés par la **classe enfant** sans avoir à les redéfinir.
  - **Extension** : La **classe enfant** peut **étendre** ou **modifier** le comportement de la classe parent **en ajoutant** de nouveaux **attributs** ou **méthodes,** ou en **redéfinissant** les méthodes existantes.

- En Python, lorsqu'on veut créer une **classe héritant** d'une autre classe, on ajoutera après le nom de **la classe fille** le **nom** de la **classe(s) mère**(s) entre parenthèses :
- Syntaxe:

```
class Mere1:
    # contenu de la classe mère 1

class Mere2:
    # contenu de la classe mère 2

class Fille1(Mere1):
    # contenu de la classe fille 1

class Fille2(Mere1, Mere2):
    # contenu de la classe fille 2
```

- Exemple:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def deplace(self, dx, dy):
        self.x += dx
        self.y += dy
    def affiche_point(self):
        print("Position :", self.x, ",", self.y)

class Complexe:
    def __init__(self, reel, img):
        self.reel = reel
        self.img = img
    def affiche_complexe(self):
        print("Partie réelle :", self.reel, ", Partie imaginaire :", self.img)
```



- Exemple:

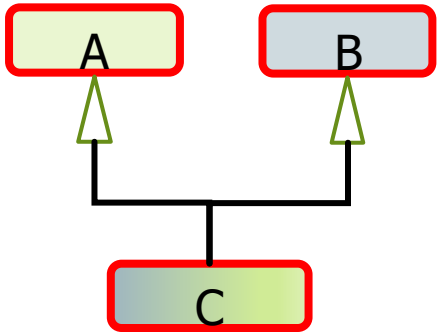
```
class PointComplexe(Point, Complexe):
    def __init__(self, x, y, reel, img):
        # Appel des constructeurs des classes parentes
        Point.__init__(self, x, y)
        Complexe.__init__(self, reel, img)

    def affiche_pc(self):
        self.affiche_point()
        self.affiche_complexe()

pc = PointComplexe(1, 2, 3.0, 4.0)
pc.affiche_pc()
```

- Sortie:

```
Position : 1 , 2
Partie réelle : 3.0 , Partie imaginaire : 4.0
```

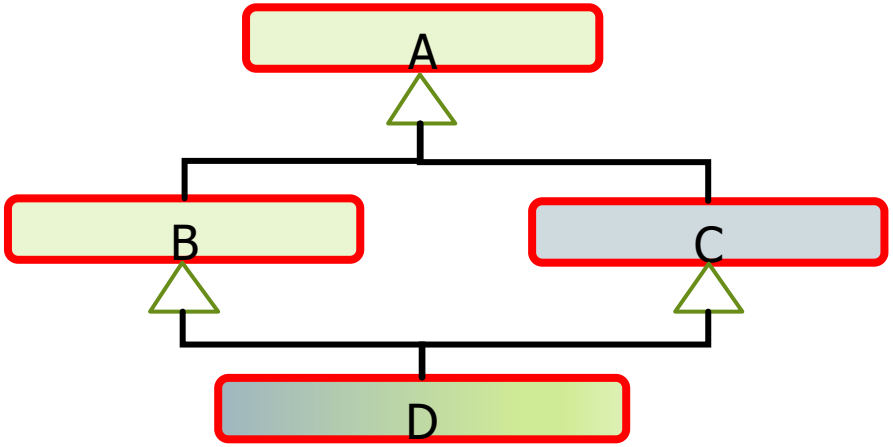
Conflit 1 :

- *Problème* : Les classes A et B possèdent des attributs ou des fonctions portant le même nom



Conflit 2 :

- Problème : Duplication des membres fonctions ou attributs dans tous les objets de D



- **Python** résout le problème de **l'héritage multiple :**
- **Ordre de résolution des méthodes (MRO)** : Python utilise l'algorithme C3 pour calculer l'ordre de résolution des méthodes.
- **La méthode `__mro__`** : Chaque classe Python a un attribut spécial **`__mro__`** qui contient l'ordre de résolution des méthodes.
- **La fonction super()** : La fonction super() est utilisée pour appeler les méthodes de la classe parente dans la hiérarchie d'héritage.
- **Diamond Problem (problème du diamant)** : Lorsque vous avez une structure d'héritage en forme de diamant. Python résout ce problème en garantissant que les méthodes des ancêtres ne sont appelées qu'une seule fois, dans l'ordre approprié de résolution des méthodes.

- Exemple:

```
class Vehicule:
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele
    def afficher_details(self):
        print(f"Marque: {self.marque}, Modèle: {self.modele}")

class Voiture(Vehicule):
    def __init__(self, marque, modele, couleur):
        super().__init__(marque, modele)
        self.couleur = couleur
    def afficher_details(self):
        super().afficher_details()
        print(f"Couleur: {self.couleur}")

class Moto(Vehicule):
    def __init__(self, marque, modele, puissance):
        super().__init__(marque, modele)
        self.puissance = puissance
    def afficher_details(self):
        super().afficher_details()
        print(f"Puissance: {self.puissance} chevaux")
```

- Exemple:

```
voiture = Voiture("Toyota", "Corolla", "Rouge")
moto = Moto("Honda", "CBR500R", 47)

voiture.afficher_details()
moto.afficher_details()
```

- Affichage:

```
Marque: Toyota, Modèle: Corolla
Couleur: Rouge
Marque: Honda, Modèle: CBR500R
Puissance: 47 chevaux
```

- Ce terme désigne la capacité qu'une méthode de s'adapter automatiquement à l'objet manipulé.
- « Poly» signifie « plusieurs », et « morphe » signifie « forme »
- N'a de sens que dans un contexte « ***Héritage*** ».
- En programmation, le **polymorphisme** est la capacité d'une fonction (ou méthode) à se comporter différemment en fonction de l'objet qui lui est passé.
- Une fonction donnée peut donc avoir plusieurs définitions.
- Le polymorphisme en Python se réfère à la capacité d'objets de différentes classes à répondre de manière unique à la même méthode ou fonction.

- Le polymorphisme permet de traiter différents types d'objets de manière uniforme, ce qui est une caractéristique clé de la programmation orientée objet.
- Il existe principalement deux formes de polymorphisme en Python :
  - **Polymorphisme par surcharge de méthode**
  - **Polymorphisme par surcharge d'opérateur**
- **Polymorphisme par surcharge de méthode :**
  - Cela se produit lorsque des classes dérivées (ou sous-classes) fournissent leur propre implémentation d'une méthode déjà définie dans la classe de base (ou super-classe).
  - Le comportement de la méthode varie en fonction de la classe à partir de laquelle elle est appelée.
- **Polymorphisme par surcharge d'opérateur :**
  - Cela permet à des opérateurs tels que +, -, *, etc., d'agir différemment en fonction des types d'objets sur lesquels ils sont appliqués.
  - Python fournit des méthodes spéciales (appelées méthodes du type double souligné) qui peuvent être surchargées pour permettre cette fonctionnalité.

- Exemple: **Polymorphisme par surcharge de méthode**

```
class Animal:
    def parler(self):
        print("Les animaux font du bruit !")
class Chien(Animal):
    def parler(self):
        print("Le chien aboie")
class Chat(Animal):
    def parler(self):
        print("Le chat miaule")

def faire_parler(animal):
    animal.parler()

animal1 = Animal()
faire_parler(animal1)
animal1 = Chien()
faire_parler(animal1)
animal1 = Chat()
faire_parler(animal1)
```

```
Les animaux font du bruit !
Le chien aboie
Le chat miaule
```

- Exemple : **Polymorphisme par surcharge d'opérateur**

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

    def __add__(self, other):
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        elif isinstance(other, (int, float)):
            return Point(self.x + other, self.y)
        else:
            raise TypeError("L'opération d'addition n'est pas supportée.")

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)
p4 = p1 + 1
print(p4.x, p4.y)
```

```
4 6
2 2
```

- En Python, **la gestion des fichiers** est un aspect crucial **pour manipuler** des données stockées de manière persistante sur le disque.
- Une grande partie de l'information est stockée sous forme de **texte** dans des **fichiers**.
- Pour traiter cette information, vous devez le plus souvent **lire** ou **écrire** dans un ou plusieurs fichiers.
- **Python** permet très facilement de manipuler des **fichiers** tout en garantissant une très bonne **portabilité** du code quel que soit le système d'exploitation sur lequel s'exécute le programme.
- Les opérations de manipulation de fichier : **écriture, lecture, création, suppression** sont toutes susceptibles d'échouer. Dans ce cas, les fonctions ou les méthodes produiront une erreur de type `OSError` ou d'un type héritant de cette **exception**.

- La fonction `open()` permet de récupérer un descripteur de **fichier** en passant en paramètre le **chemin de ce fichier.**
- Le descripteur de fichier va nous permettre de réaliser les opérations de **lecture** et/ou **d'écriture** dans le fichier.
- Lorsque les opérations sur le fichier sont terminées, il faut fermer le descripteur de fichier à l'aide de la méthode `close()`
- Syntaxe :

```
f = open("monfichier.txt)
# faire des opérations sur le contenu du fichier
f.close()
```

- Python fournit une syntaxe spéciale : le '`with`'.
- Syntaxe:

```
with open("monfichier.txt") as f:
    # faire des opérations sur le contenu du fichier
```

- La syntaxe `with` appelle automatiquement la méthode `close()` du descripteur de fichier à la sortie du bloc

- Pour obtenir le contenu du **fichier** dans une **chaîne de caractères** :

```
with open("texte.txt") as f:
    contenu = f.read()
print(contenu)
```

- Pour obtenir le contenu du **fichier** sous la forme d'un **liste de chaînes de caractères** (un élément par ligne) :

```
with open("texte.txt") as f:
    lignes = f.readlines()
print(lignes)
```

ou

```
with open("texte.txt") as f:
    for ligne in f:
        print(ligne)
```

- Remarque :
  - Pour **supprimer** le **caractère de retour à la ligne** , on peut utiliser la méthode `str.rstrip()`
  - Il est possible de préciser la famille d'encodage d'un **fichier** grâce au paramètre `encoding` :

```
with open("texte.txt",encoding="utf-8") as f:
    for ligne in f:
        print(ligne.rstrip('\n'))
```

- Lorsqu'on **ouvre un fichier,** il faut préciser le **mode d'ouverture** qui dépend du type du **fichier** et des opérations que l'on souhaite réaliser.
- Le **mode** est représenté par une chaîne de caractères qui est passée à la fonction `open()` avec le paramètre mode.

Mode	Description
r	ouverture en lecture (mode par défaut)
w	ouverture en écriture (efface le contenu précédent)
x	ouverture uniquement pour création (l'ouverture échoue si le fichier existe déjà)
+	ouverture en lecture et écriture
a	ouverture en écriture pour ajout en fin de fichier
b	fichier binaire
t	fichier texte (mode par défaut)

- Exemple:

```
# ouverture en écriture
with open("dialogues.txt", mode="w") as f:
```

- Pour écrire d'un bloc dans un fichier, on peut utiliser la méthode `write()`.
- Pour écrire une liste de lignes, il faut utiliser la méthode `writelines()`.
- Exemple:

```
lignes = ["-ligne1 \n",
          "- ligne2.\n"]

with open("texte.txt", mode="a") as f:
    f.writelines(lignes)
```

- Remarque:
  - Attention pour les fichiers textes, ces méthodes n'ajoutent pas de caractères de fin de ligne, il faut donc les écrire explicitement.
  - Il faut se rappeler que le mode d'ouverture en écriture (w) remplace intégralement le contenu du fichier. Pour ajouter à la fin du fichier, il faut ouvrir le fichier **en mode ajout** (a).

- Les **fichiers** sont organisés selon une structure arborescente dans laquelle un **nœud** peut être soit un **fichier** soit un **répertoire**.
- Même si tous les **systèmes d'exploitation** suivent le même principe, il existe des différences majeures d'organisation.
- Exemple:
  - Le système MS-Windows utilise un système de fichiers multi-têtes (c:, d:, e:…) tandis que les systèmes *nix et MacOS utilisent un système mono-tête dont la racine est /.
  - Dans la représentation des chemins de fichiers, MS-Windows utilise le caractère \ pour séparer les composants d'un chemin C:\\Users\desktop\ Tandis que les systèmes *nix et MacOs utilisent le caractère / comme /home/document
- Toutes ces nuances peuvent rendre difficiles l'écriture d'un programme portable d'un système à l'autre. Heureusement, la bibliothèque standard Python fournit plusieurs solutions pour aider les développeurs :
  - **Le module os.path** fournit des fonctions élémentaires pour gérer les chemins de fichiers.
  - **Le module pathlib** est un module de haut-niveau qui permet à la fois de manipuler un chemin mais également d'interagir avec le fichier ou le répertoire désigné par ce chemin. (L'élément central du module est la classe **Path**)

Le module os.path

- `join()` : Cette fonction permet de créer un chemin en utilisant le séparateur approprié pour le système.

```
import os.path as path

chemin = path.join("fichiers", "monfichier.txt")
print(chemin)
# Sous Windows affiche fichiers\monfichier.txt
# Sous *nix ou MacOS, affiche fichiers/monfichier.tx
```

Le module pathlib:

```
from pathlib import Path

# Créer le chemin en utilisant pathlib
chemin = Path("fichiers", "monfichier.txt")
print(chemin)
```

Le module pathlib:

- La classe Path possède la méthode **open()** qui accepte les mêmes paramètres que la fonction **open()** sauf le chemin qui est déjà représenté par l'objet lui-même :

```
from pathlib import Path

chemin = Path("texte.txt")

with chemin.open() as f:
    for ligne in f:
        print(ligne)
```

ou

```
from pathlib import Path

chemin = Path("texte.txt")
contenu = chemin.read_text()
print(contenu)
```

- Il est possible de réaliser des **opérations élémentaires** sur les **fichiers** et les répertoires soit avec **le module os** soit avec **le module pathlib.**
- Les **modules os** et **shutil** sont historiquement les premiers modules qui ont été introduits en Python. Ils proposent surtout des fonctions alors que le module **pathlib** est orienté objet avec notamment la classe Path.
- **Connaître le répertoire de travail:**

```
import os
print(os.getcwd())
```

ou

```
import pathlib
print(pathlib.Path.cwd())
```

- **Supprimer un fichier:**

```
import os
os.remove("monfichier.txt")
```

ou

```
import pathlib
p = pathlib.Path("texte.txt")
p.unlink()
```

- **Créer un répertoire**

```
import os
os.mkdir("monrepertoire")
```

ou

```
import pathlib
p = pathlib.Path("monrepertoire")
p.mkdir()
```

- **Supprimer un répertoire**

```
import os
os.rmdir("monrepertoire")
```

```
import pathlib
p = pathlib.Path("monrepertoire")
p.rmdir()
```

- **Vérifier qu'un fichier existe :**

```
import os.path
os.path.exists("monfichier.txt")
```

```
import pathlib
p = pathlib.Path("monfichier.txt")
p.exists()
```

- **Lister le contenu d'un répertoire :**

```
import os
liste_fichiers = os.listdir("monrepertoire")
```

```
import pathlib
p = pathlib.Path("monrepertoire")
liste_fichiers = list(p.iterdir())
```

- **Rechercher des fichiers**

```
import pathlib
liste_fichiers = list(pathlib.Path.cwd().glob("**/*.py"))
```

- Toutes **les erreurs** qui se produisent lors de l'exécution d'un programme Python sont représentées par une **exception**.
- Une **exception** est un objet qui contient des **informations** sur le **contexte** de l'erreur. Lorsqu'une exception survient et qu'elle n'est pas traitée alors elle produit une interruption du programme et elle affiche sur la sortie standard un message ainsi que la pile des appels (stacktrace).
- La pile des appels présente dans l'ordre la **liste des fonctions** et des **méthodes** qui étaient en cours d'appel au moment où **exception** est survenue.
- Exemple:

```
def do_something_wrong():
    1 / 0

def do_something():
    do_something_wrong()

do_something()
```

```
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 7, in <module>
    do_something()
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 5, in do_something
    do_something_wrong()
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 2, in do_something_wrong
    1 / 0
    ~~^~~
ZeroDivisionError: division by zero
```

- Un **programme** est capable de traiter le problème à l'origine de **l'exception.**
- Exemple: Si le programme demande à l'utilisateur de saisir un nombre et que l'utilisateur saisit une valeur erronée, le programme peut simplement demander à l'utilisateur de saisir une autre valeur. Plutôt que de faire échouer le programme, il est possible d'essayer de réaliser un traitement et, s'il échoue de proposer un traitement adapté :
- Exemple:

```
nombre = input("Entrez un nombre : ")
try:
    nombre = int(nombre)
except ValueError:
    print("Désolé la valeur saisie n'est pas un nombre.")
```

- Si la fonction `int` ne peut pas créer un entier à partir du paramètre, elle produit une exception de **type ValueError**. Donc après le bloc `try`, on ajoute une instruction `except` pour le type **ValueError**
- L'intérêt de la structure du `try except` est qu'elle permet de dissocier dans le code **la partie du traitement** normal de la **partie du traitement des erreurs.**

- Lorsqu'une **exception** survient dans un bloc `try`, elle interrompt immédiatement l'exécution du bloc et **l'interpréteur** recherche un bloc `except` pouvant traiter l'exception.
- Il est possible d'ajouter plusieurs blocs `except` à la suite d'un bloc `try`.
- Chacun d'entre eux permet de coder un traitement particulier pour chaque type d'erreur :
- Exemple:

```
try:
    numerateur = int(input("Entrez un numérateur : "))
    denominateur = int(input("Entrez un dénominateur : "))
    resultat = numerateur / denominateur
    print("Le resultat de la division est", resultat)
except ValueError:
    print("Désolé, les valeurs saisies ne sont pas des nombres.")
except ZeroDivisionError:
    print("Désolé, la division par zéro n'est pas permise.")
```

- Remarque : Si le même traitement est applicable pour des **exceptions** de types différents, il est possible de fournir un seul bloc **except** avec le **tuple** des exceptions concernées :
- Exemple:

```
try:
    numerateur = int(input("Entrez un numérateur : "))
    denominateur = int(input("Entrez un dénominateur : "))
    resultat = numerateur / denominateur
    print("Le resultat de la division est", resultat)
except (ValueError, ZeroDivisionError):
    print("Désolé, quelque chose ne s'est pas bien passé.")
```

- Les **exceptions** possèdent une représentation sous forme de chaîne de caractères pour fournir un message.
- Pour avoir accès à **l'exception**, on utilise la **syntaxe suivante :**

```
nombre = input("Entrez nombre : ")
try:
    nombre = int(nombre)
except ValueError as e:
    print(e)
```

- Dans l'exemple ci-dessus, on précise que **l'exception** de type **ValueError** est accessible dans le bloc **except** sous le nom e ce qui permet d'afficher **l'exception** (c'est-à-dire le message d'erreur).

- Il est possible d'ajouter une `clause else` après les blocs `try except`.
- Le bloc **else** est exécuté uniquement si le bloc **try** se termine normalement, c'est-à-dire sans qu'une exception ne survienne.
- Exemple:

```
try:
    numerateur = int(input("Entrez un numérateur : "))
    denominateur = int(input("Entrez un dénominateur : "))
    resultat = numerateur / denominateur
except (ValueError, ZeroDivisionError):
    print("Désolé, quelque chose ne s'est pas bien passé.")
else:
    print("Le resultat de la division est", resultat)
```

- Le **bloc else** permet de distinguer la partie du code qui est susceptible de produire une **exception** de celle qui fait partie du **comportement** nominal du code mais qui ne produit pas **d'exception**.

- Dans certain cas, on souhaite réaliser un traitement après le **bloc try** que ce dernier **se termine correctement** ou bien **qu'une exception soit survenue.**
- Dans cas, on place le code dans un `bloc finally`.
- Exemple:

```
try:
    numerateur = int(input("Entrez un numérateur : "))
    denominateur = int(input("Entrez un dénominateur : "))
    resultat = numerateur / denominateur
    print("Le resultat de la division est", resultat)
except (ValueError, ZeroDivisionError):
    print("Désolé, quelque chose ne s'est pas bien passé.")
finally:
    print("afficher ceci quel que soit le résultat")
```

- Un bloc **finally** est systématique appelé même si le bloc **try** est interrompu par une instruction.

- Il est possible de signaler une exception grâce au mot-clé `raise`.
- Exemple:

```
if x < 0:
    raise ValueError
```

- Pour la plupart des **exceptions**, il est possible de passer en paramètre un **message** pour décrire le cas **exceptionnel** :

```
if x < 0:
    raise ValueError("La valeur ne doit pas être négative")
```

- Plusieurs exceptions décrivent des erreurs très courantes. Il est intéressant de bien les connaître pour écrire des programmes capables de traiter les erreurs éventuelles :
  - `ValueError`: Signale que la valeur n'est pas correcte.

```
int('a')
```

```
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 1, in <module>
    int('a')
ValueError: invalid literal for int() with base 10: 'a'
```

  - `TypeError`: Signale que le type de la donnée n'est pas correct pour l'instruction à exécuter.

```
1 + "a"
```

```
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 1, in <module>
    1 + "a"
    ~~^~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- `IndexError`: Signale que l'on veut accéder à un élément d'une liste, d'un n-uplet ou d'une chaîne de caractères avec un index invalide.

```
"hello"[100]
```

```
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 1, in <module>
    "hello"[100]
    ~~~~~~~^^^^^
IndexError: string index out of range
```

- `KeyError`: Signale que la clé n'existe pas dans un dictionnaire.

```
d = {}
d['une cle']
```

```
Traceback (most recent call last):
  File "C:\Users\DELL\Desktop\teste\tt\pythonProject\main.py", line 2, in <module>
    d['une cle']
    ~^^^^^^^^^^^
KeyError: 'une cle'
```

- **Python** permet facilement d'interagir avec des **systèmes de bases de données** relationnelles (**SGBDR**).
- Cette facilité est en partie due à la définition d'une **API standard** d'accès aux bases de données décrite par Python Database API Specification v2.0.
- Pour accéder à une **base de données** vous devrez installer avec le **module pip** le package pour vous connecter au SGBDR de votre choix.
- La procédure de connexion peut avoir des spécificités selon le **SGBDR** mais l'interaction avec la base **reste standard**.
- Dans notre chapitre nous prendrons comme exemple **MySQL**. Nous partirons du principe qu'il existe un serveur **MySQL** actif sur la machine hôte

- Pour vous connecter à une **base de données MySQL** vous devez:
  - **Installer MySQL Server**: Assurez-vous que MySQL Server est installé sur votre système. Vous pouvez télécharger et installer MySQL depuis le site officiel de MySQL (https://dev.mysql.com/downloads/).
  - **Installer le connecteur MySQL:** qui est un paquet nommé **mysql-connector-python** et qui est disponible sur **PyPI.** Ou utiliser l'instruction dans le terminal ***pip install mysql-connector-python***
  - **Configurer et utiliser le connecteur** : Vous devez **importer** le connecteur MySQL dans votre code et configurer la connexion à la base de données en utilisant les paramètres appropriés (**nom d'utilisateur, mot de passe, hôte, port,** etc.).
- Remarque : N'oubliez pas de **tester** la connexion à la base de données après avoir configuré le connecteur pour vérifier que tout fonctionne correctement. Utilisez un **bloc try** pour cela.

- Le module **mysql.connector** fournit la méthode **connect** qui permet de retourner un objet qui représente la **connexion** vers la **base de données.**
- Vous devez fournir les paramètres **host, user** et **password** pour donner l'adresse du **SGBDR**, le **login** et le **mot de passe** de connexion.
- Vous pouvez également fournir le paramètre **database** pour indiquer quelle base de données vous souhaitez utiliser.
- Exemple:

```
import mysql.connector

db = mysql.connector.connect(
    host="localhost",
    user="login",
    password="mot_de_passe",
    database="demo_python"
)

# faire quelque chose d'utile avec la connexion

db.close()
```

- La **connexion** à la base de données se comporte comme un gestionnaire de ressource.
- Cela signifie que nous pouvons employer la syntaxe `with` pour nous assurer de fermer la connexion correctement.
- De plus, il est utile d'employer l'opérateur ** pour faire un **unpack** de **dictionnaire**. Il permet de passer les paramètres de connexion du dictionnaire en tant qu'arguments nommés à la fonction **mysql.connector.connect()**.
- Ainsi nous pouvons représenter les paramètres de connexion comme un simple dictionnaire :

```
import mysql.connector

connection_params = {
    'host': "localhost",
    'user': "login",
    'password': "mot_de_passe",
    'database': "demo_python",
}

with mysql.connector.connect(**connection_params) as db :
    # faire quelque chose d'utile avec la connexion
    pass
```

- **L'interaction** avec la base de données se fera à travers un **curseur**.
- Cet objet permet à la fois **d'envoyer** des requêtes et de **consulter** les résultats quand il s'agit d'une requête de consultation de données.
- Pour créer un **curseur**, on appelle la méthode `cursor()` sur la connexion. Il est recommandé de fermer un curseur lorsqu'il n'est plus utilisé.
- Comme un **curseur** est également un **gestionnaire de ressource**, nous pouvons également employer la syntaxe `with` :

```
import mysql.connector

connection_params = {
    'host': "localhost",
    'user': "TDIA",
    'password': "123456",
    'database': "test",
}

with mysql.connector.connect(**connection_params) as db :
    with db.cursor() as c:
        # faire quelque chose d'utile avec le curseur
        pass
```

- **L'insertion** de **données** se fait avec une **requête SQL** de type **insert**.
- Pour **exécuter** une requête, nous utilisons la méthode `execute` du **curseur**.

```
with mysql.connector.connect(**connection_params) as db:
    with db.cursor() as c:
        c.execute("INSERT INTO utilisateur (nom, email, PW, age) "
                  "VALUES ('TDIA', 'TDIA@email.com', '1234', 20)")
        db.commit()
    print("Insertion réussie.")
```

- Remarque:
  - L'appel à la méthode `commit()` de la connexion qui permet de **valider** les **modifications**. Si vous n'appelez pas cette méthode, la transaction avec la base de données ne sera pas terminée et aucune ligne ne sera insérée en base de données.
  - **Le connecteur MySQL** fournit la propriété non standard **autocommit** sur la connexion. Si cette propriété vaut **True** alors un commit est automatiquement fait après chaque **exécution** de requête.
  - On peut contrôler **le flux d'exécution** du programme lorsqu'une **erreur** survient, en fournissant une méthode pour gérer ces **erreurs.**