

INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET (POO)

CONCEPT DE LA POO

La Programmation Orientée Objet (POO) est un paradigme de programmation. Elle utilise des "objets" pour modéliser des concepts du monde réel. Les objets sont des instances de "classes". Les classes définissent des propriétés (attributs) et des comportements (méthodes) des objets. La POO favorise la réutilisation du code et la modularité. Elle permet de créer des programmes plus faciles à comprendre et à maintenir.

AVANTAGES DE LA POO

- Modularité : facilite la gestion et la maintenance du code.
- Réutilisation : les classes peuvent être réutilisées dans différents programmes.
- Extensibilité : possibilité d'ajouter de nouvelles fonctionnalités sans modifier le code existant.
- Encapsulation : protège les données en limitant leur accès.
- Abstraction : simplifie la complexité en cachant les détails d'implémentation.

DIFFÉRENCES ENTRE POO ET PROGRAMMATION PROCÉDURALE

Caractéristique	POO	Programmation procédurale
Structure	Basée sur des objets	Basée sur des fonctions
Encapsulation	Oui	Non
Réutilisation du code	Forte	Faible
Abstraction	Oui	Non
Extensibilité	Facile	Difficile

TERMINOLOGIE DE LA POO (CLASSE, OBJET, MÉTHODE, ATTRIBUT)

- **Classe** : Plan de construction pour les objets.
- **Objet** : Instance d'une classe.
- **Méthode** : Fonction définie dans une classe.
- **Attribut** : Variable définie dans une classe.

EXEMPLES DE POO DANS LA VIE QUOTIDIENNE

- **Voiture** : Classe avec des attributs (couleur, marque) et des méthodes (démarrer, arrêter).
- **Animal** : Classe avec des attributs (espèce, âge) et des méthodes (manger, dormir).
- **Compte Bancaire** : Classe avec des attributs (solde, numéro) et des méthodes (dépôt, retrait).
- **Livre** : Classe avec des attributs (titre, auteur) et des méthodes (ouvrir, fermer).

CLASSES ET OBJETS

DÉFINITION DE CLASSE

En POO, une classe est un modèle qui définit les propriétés et les comportements d'un objet. Elle sert de plan pour créer des objets. Une classe regroupe des variables (attributs) et des fonctions (méthodes). Les classes permettent de structurer et organiser le code. Elles facilitent la réutilisation et la maintenance du code.

CRÉATION D'UNE CLASSE

Pour créer une classe en Python, on utilise le mot-clé `class` suivi du nom de la classe.

```
class MaClasse:  
    pass
```

Le mot-clé `pass` indique que la classe est vide pour le moment.

INSTANCIATION D'UN OBJET

Pour créer un objet (instance) à partir d'une classe, on appelle la classe comme une fonction.

```
mon_objet = MaClasse()
```

mon_objet est une instance de **MaClasse**.

CONSTRUCTEUR (INIT)

Le constructeur est une méthode spéciale appelée lors de la création d'un objet. En Python, le constructeur est défini par la méthode `__init__`.

```
class MaClasse:  
    def __init__(self, param1, param2):  
        self.param1 = param1  
        self.param2 = param2
```

DIFFÉRENCE ENTRE CLASSE ET OBJET

- **Classe :**

- Modèle ou plan pour créer des objets.
- Définit les attributs et méthodes.

- **Objet :**

- Instance d'une classe.
- Représente un élément unique avec des valeurs spécifiques.

ATTRIBUTS ET MÉTHODES

DÉFINITION DES ATTRIBUTS

Les attributs sont des variables associées à une classe ou à des objets. Ils stockent des données spécifiques aux objets ou à la classe. Les attributs peuvent être d'instance ou de classe. Ils définissent les propriétés des objets créés à partir de la classe.

DÉFINITION DES MÉTHODES

Les méthodes sont des fonctions définies dans une classe. Elles définissent le comportement des objets de la classe. Les méthodes peuvent être d'instance, de classe ou statiques. Elles permettent de manipuler les attributs et d'interagir avec l'objet.

ATTRIBUTS D'INSTANCE

Les attributs d'instance sont spécifiques à chaque objet. Ils sont définis dans le constructeur avec `self`. Chaque objet a ses propres valeurs d'attributs d'instance. Exemple :

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

ATTRIBUTS DE CLASSE

Les attributs de classe sont partagés par toutes les instances de la classe. Ils sont définis directement dans la classe, en dehors des méthodes. Exemple :

```
class Person:  
    species = "Homo sapiens"
```

Tous les objets de la classe **Person** partagent l'attribut **species**.

MÉTHODES D'INSTANCE

Les méthodes d'instance opèrent sur les attributs d'instance. Elles utilisent `self` pour accéder aux attributs et aux autres méthodes. Exemple :

```
class Person:  
    def greet(self):  
        return f"Hello, my name is {self.name}"
```

MÉTHODES DE CLASSE

Les méthodes de classe opèrent sur les attributs de classe. Elles utilisent `cls` comme premier paramètre. Elles sont définies avec le décorateur `@classmethod`. Exemple :

```
class Person:  
    species = "Homo sapiens"  
  
    @classmethod  
    def get_species(cls):  
        return cls.species
```

MÉTHODES STATIQUES

Les méthodes statiques ne dépendent ni de l'instance ni de la classe. Elles n'utilisent ni `self` ni `cls`. Elles sont définies avec le décorateur `@staticmethod`. Exemple :

```
class Math:  
    @staticmethod  
    def add(a, b):  
        return a + b
```

ACCÈS AUX ATTRIBUTS

Pour accéder aux attributs d'un objet, utilisez la syntaxe `objet.attribut`. Exemple :

```
person = Person("Alice")
print(person.name) # Accède à l'attribut d'instance name
```

MODIFICATION DES ATTRIBUTS

Pour modifier un attribut d'un objet, utilisez la syntaxe `objet.attribut = valeur`. Exemple :

```
person = Person("Alice")
person.name = "Bob" # Modifie l'attribut d'instance name
```

APPEL DES MÉTHODES

Pour appeler une méthode d'un objet, utilisez la syntaxe `objet.méthode()`. Exemple :

```
person = Person("Alice")
print(person.greet()) # Appelle la méthode d'instance greet
```

CONSTRUCTEURS

DÉFINITION DES CONSTRUCTEURS

Les constructeurs sont des méthodes spéciales utilisées pour initialiser les objets. En Python, le constructeur est défini par la méthode `__init__`. Il est appelé automatiquement lors de la création d'un objet. Le constructeur permet de définir et d'initialiser les attributs de l'objet.

CONSTRUCTEUR PAR DÉFAUT

Un constructeur par défaut est un constructeur sans paramètres. Si aucun constructeur n'est défini, Python fournit un constructeur par défaut. Ce constructeur ne fait rien d'autre que créer l'objet.

CONSTRUCTEUR PERSONNALISÉ

Un constructeur personnalisé est un constructeur avec des paramètres. Il permet d'initialiser les attributs de l'objet avec des valeurs spécifiques. Il est défini en ajoutant des paramètres à la méthode `__init__`.

UTILISATION DE `__init__`

La méthode `__init__` est utilisée pour définir un constructeur en Python. Elle prend `self` comme premier paramètre, suivi des autres paramètres. Exemple :

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

INITIALISATION DES ATTRIBUTS

Les attributs de l'objet sont initialisés dans la méthode `__init__`. Les valeurs des paramètres sont assignées aux attributs de l'objet. Exemple :

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

APPEL DU CONSTRUCTEUR

Le constructeur est appelé automatiquement lors de la création d'un objet. Exemple :

```
p = Personne("Alice", 30)
print(p.nom) # Affiche "Alice"
print(p.age) # Affiche 30
```

CONSTRUCTEURS ET HÉRITAGE

Les constructeurs peuvent être utilisés avec l'héritage. Le constructeur de la classe parente peut être appelé avec `super()`. Exemple :

```
class Parent:  
    def __init__(self, nom):  
        self.nom = nom  
  
class Enfant(Parent):  
    def __init__(self, nom, age):  
        super().__init__(nom)  
        self.age = age
```

ENCAPSULATION

DÉFINITION DE L'ENCAPSULATION

L'encapsulation est un principe fondamental de la POO. Elle consiste à regrouper des données et des méthodes qui les manipulent. Elle permet de protéger les données internes d'un objet. Les données ne sont accessibles que via des méthodes spécifiques. Cela améliore la sécurité et l'intégrité des données.

ATTRIBUTS PRIVÉS ET PUBLICS

En Python, les attributs publics sont accessibles de l'extérieur. Les attributs privés sont précédés de deux underscores __. Exemple d'attribut public :

```
self.nom = "NomPublic"
```

Exemple d'attribut privé :

```
self.__nom = "NomPrivé"
```

MÉTHODES D'ACCÈS (GETTERS ET SETTERS)

Les getters permettent de récupérer la valeur d'un attribut privé. Les setters permettent de modifier la valeur d'un attribut privé. Exemple de getter :

```
def get_nom(self):  
    return self.__nom
```

Exemple de setter :

```
def set_nom(self, nom):  
    self.__nom = nom
```

PROPRIÉTÉS EN PYTHON

Les propriétés permettent de gérer l'accès aux attributs. Elles utilisent les décorateurs `@property`, `@nom.setter`. Exemple de propriété :

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

@nom.setter
def nom(self, nom):
    self.__nom = nom
```

AVANTAGES DE L'ENCAPSULATION

- Protection des données internes d'un objet.
- Contrôle de l'accès et de la modification des données.
- Facilite la maintenance et l'évolution du code.
- Améliore la sécurité des applications.
- Permet de respecter le principe de responsabilité unique.

HÉRITAGE

DÉFINITION DE L'HÉRITAGE

L'héritage est un concept clé de la programmation orientée objet. Il permet à une classe (dite "enfant") de hériter les attributs et méthodes d'une autre classe (dite "parent"). Cela favorise la réutilisation du code et l'organisation hiérarchique des classes.

CLASSES PARENT ET ENFANT

- Classe parent : La classe dont les attributs et méthodes sont hérités.
- Classe enfant : La classe qui hérite des attributs et méthodes de la classe parent.
- Exemple : **Animal** (parent) et **Chien** (enfant).

SYNTAXE DE L'HÉRITAGE EN PYTHON

La syntaxe pour définir une classe enfant qui hérite d'une classe parent est :

```
class Parent:  
    pass  
  
class Enfant(Parent):  
    pass
```

HÉRITAGE SIMPLE VS HÉRITAGE MULTIPLE

- Héritage simple : Une classe enfant hérite d'une seule classe parent.
- Héritage multiple : Une classe enfant hérite de plusieurs classes parents.
- Exemple d'héritage multiple : `class Enfant(Parent1, Parent2) :`

UTILISATION DU MOT-CLÉ SUPER()

Le mot-clé `super()` est utilisé pour appeler des méthodes de la classe parent. Cela permet de réutiliser le code de la classe parent dans la classe enfant.

```
class Parent:  
    def __init__(self):  
        print("Parent")  
  
class Enfant(Parent):  
    def __init__(self):  
        super().__init__()  
        print("Enfant")
```

MÉTHODES ET ATTRIBUTS HÉRITÉS

- Les méthodes et attributs définis dans la classe parent sont automatiquement disponibles dans la classe enfant.
- Cela inclut les méthodes d'instance, les méthodes de classe et les attributs.

REDÉFINITION DE MÉTHODES (SURCHARGE)

La redéfinition de méthodes permet à une classe enfant de fournir une implémentation spécifique d'une méthode héritée de la classe parent.

```
class Parent:  
    def afficher(self):  
        print("Parent")  
  
class Enfant(Parent):  
    def afficher(self):  
        print("Enfant")
```

AVANTAGES DE L'HÉRITAGE

- Réutilisation du code : Évite la duplication de code.
- Organisation : Structure hiérarchique des classes.
- Extensibilité : Facilite l'ajout de nouvelles fonctionnalités.

LIMITES ET DANGERS DE L'HÉRITAGE

- Complexité : Peut rendre le code difficile à comprendre.
- Couplage fort : Les classes enfants sont fortement dépendantes des classes parents.
- Héritage multiple : Peut introduire des conflits et ambiguïtés.

POLYMORPHISME

NOTION DE POLYMORPHISME

Le polymorphisme permet à des objets de différents types de répondre à la même interface. Il permet d'utiliser une seule interface pour représenter différentes implémentations. En Python, le polymorphisme se manifeste souvent par l'utilisation de méthodes communes.

POLYMORPHISME PAR HÉRITAGE

Le polymorphisme par héritage permet à une classe dérivée de remplacer ou d'étendre le comportement d'une classe de base. Les méthodes de la classe de base peuvent être redéfinies dans la classe dérivée. Cela permet d'utiliser des objets de classe dérivée comme s'ils étaient de la classe de base.

POLYMORPHISME PAR COMPOSITION

Le polymorphisme par composition consiste à utiliser des objets d'autres classes pour obtenir un comportement polymorphique. Une classe peut contenir des objets d'autres classes et utiliser leurs méthodes. Cela permet de créer des structures plus flexibles et modulaires.

MÉTHODES POLYMORPHIQUES

Les méthodes polymorphiques sont des méthodes qui peuvent être redéfinies dans des classes dérivées. Elles permettent d'adapter le comportement d'une classe en fonction du type de l'objet. Exemple en Python :

```
class Animal:  
    def parler(self):  
        pass  
  
class Chien(Animal):  
    def parler(self):  
        return "Woof!"  
  
class Chat(Animal):  
    def parler(self):  
        return "Meow!"
```

SURCHARGE DE MÉTHODES

La surcharge de méthodes permet de définir plusieurs méthodes avec le même nom mais des signatures différentes. En Python, cela se fait souvent par l'utilisation de paramètres par défaut. Exemple :

```
class Exemple:  
    def methode(self, a, b=0):  
        return a + b
```

UTILISATION DU POLYMORPHISME EN PYTHON

Le polymorphisme permet de traiter des objets de différentes classes de manière uniforme. Il est couramment utilisé dans les structures de données et les algorithmes génériques. Exemple :

```
animaux = [Chien(), Chat()]
for animal in animaux:
    print(animal.parler())
```

MÉTHODES SPÉCIALES

MÉTHODES MAGIQUES

Les méthodes magiques sont des méthodes spéciales en Python. Elles sont entourées de doubles underscores, par exemple `__init__`. Elles permettent de définir des comportements spécifiques pour les objets. Elles sont appelées automatiquement par certaines opérations. Exemples : `__str__`, `__repr__`, `__len__`, `__getitem__`, etc. Elles facilitent la personnalisation des classes.

MÉTHODE INIT

La méthode `__init__` est le constructeur d'une classe. Elle est appelée automatiquement lors de la création d'une instance. Elle initialise les attributs de l'objet.

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

MÉTHODE STR

La méthode `__str__` définit la représentation en chaîne de caractères d'un objet. Elle est appelée par la fonction `print()` et `str()`.

```
class Person:  
    def __str__(self):  
        return f"{self.name}, {self.age} years old"
```

MÉTHODE REPR

La méthode `__repr__` fournit une représentation officielle de l'objet. Elle est utilisée pour le débogage et le développement.

```
class Person:  
    def __repr__(self):  
        return f"Person(name={self.name}, age={self.age})"
```

MÉTHODE LEN

La méthode `__len__` retourne la longueur d'un objet. Elle est appelée par la fonction `len()`.

```
class MyList:  
    def __len__(self):  
        return len(self.items)
```

MÉTHODE GETITEM

La méthode `__getitem__` permet d'accéder aux éléments d'un objet à l'aide de l'opérateur `[]`.

```
class MyList:  
    def __getitem__(self, index):  
        return self.items[index]
```

MÉTHODE SETITEM

La méthode `__setitem__` permet de modifier les éléments d'un objet à l'aide de l'opérateur `[]`.

```
class MyList:  
    def __setitem__(self, index, value):  
        self.items[index] = value
```

MÉTHODE DELITEM

La méthode `__delitem__` permet de supprimer les éléments d'un objet à l'aide de l'opérateur `del`.

```
class MyList:  
    def __delitem__(self, index):  
        del self.items[index]
```

MÉTHODE ITER

La méthode `__iter__` retourne un itérateur pour l'objet. Elle est appelée par la fonction `iter()`.

```
class MyList:  
    def __iter__(self):  
        return iter(self.items)
```

MÉTHODE NEXT

La méthode `__next__` retourne l'élément suivant de l'itérateur. Elle est appelée par la fonction `next()`.

```
class MyIterator:  
    def __next__(self):  
        if self.index < len(self.items):  
            item = self.items[self.index]  
            self.index += 1  
            return item  
        else:  
            raise StopIteration
```

PROPRIÉTÉS

DÉFINITION DES PROPRIÉTÉS

Les propriétés en Python permettent de contrôler l'accès aux attributs d'une classe. Elles permettent de définir des méthodes pour obtenir, modifier ou supprimer un attribut. Les propriétés sont utilisées pour encapsuler les attributs et ajouter de la logique. Elles augmentent la sécurité et l'intégrité des données. Les propriétés sont définies à l'aide des décorateurs `@property`, `@<attribut>.setter`, et `@<attribut>.deleter`.

ACCESSEURS (GETTERS)

Les accesseurs, ou getters, sont des méthodes qui permettent de lire la valeur d'un attribut. Ils sont souvent utilisés pour ajouter des vérifications ou transformations lors de la lecture. Syntaxe d'un getter avec le décorateur `@property` :

```
class MaClasse:  
    @property  
    def mon_attribut(self):  
        return self._mon_attribut
```

MUTATEURS (SETTERS)

Les mutateurs, ou setters, sont des méthodes qui permettent de modifier la valeur d'un attribut. Ils sont utilisés pour ajouter des vérifications ou transformations lors de l'écriture. Syntaxe d'un setter avec le décorateur `@<attribut>.setter` :

```
class MaClasse:  
    @mon_attribut.setter  
    def mon_attribut(self, valeur):  
        self._mon_attribut = valeur
```

PROPRIÉTÉS AVEC @PROPERTY

Le décorateur `@property` transforme une méthode en une propriété. Il permet d'accéder à une méthode comme s'il s'agissait d'un attribut. Cela simplifie l'interface de la classe et améliore la lisibilité du code.

Exemple d'utilisation :

```
class MaClasse:  
    def __init__(self, valeur):  
        self._mon_attribut = valeur  
  
    @property  
    def mon_attribut(self):  
        return self._mon_attribut
```

PROPRIÉTÉS CALCULÉES

Les propriétés calculées permettent de définir des attributs dont la valeur est calculée dynamiquement. Elles sont utiles pour encapsuler des calculs complexes ou des transformations. Exemple d'utilisation :

```
class Rectangle:  
    def __init__(self, largeur, hauteur):  
        self.largeur = largeur  
        self.hauteur = hauteur  
  
    @property  
    def surface(self):  
        return self.largeur * self.hauteur
```

MÉTHODES DE CLASSE ET MÉTHODES STATISTIQUES

DÉFINITION DES MÉTHODES DE CLASSE

Les méthodes de classe sont des méthodes qui sont liées à la classe elle-même et non à une instance spécifique. Elles peuvent accéder et modifier l'état de la classe qui est partagé par toutes les instances. Elles reçoivent un paramètre implicite `cls` qui fait référence à la classe.

UTILISATION DU DÉCORATEUR @CLASSMETHOD

Le décorateur `@classmethod` est utilisé pour définir une méthode de classe. Syntaxe :

```
class MaClasse:  
    @classmethod  
    def ma_methode_de_classe(cls):  
        pass
```

DÉFINITION DES MÉTHODES STATIQUES

Les méthodes statiques sont des méthodes qui ne dépendent ni de l'instance ni de la classe. Elles ne peuvent pas accéder ou modifier l'état de l'instance ou de la classe. Elles sont définies à l'intérieur de la classe mais sont appelées sans référence à une instance ou à `cls`.

UTILISATION DU DÉCORATEUR @STATICMETHOD

Le décorateur `@staticmethod` est utilisé pour définir une méthode statique. Syntaxe :

```
class MaClasse:  
    @staticmethod  
    def ma_methode_statique():  
        pass
```

DIFFÉRENCES ENTRE MÉTHODES DE CLASSE, MÉTHODES D'INSTANCE ET MÉTHODES STATIQUES

Type de méthode	Paramètre implicite	Accès à l'instance	Accès à la classe
Méthode d'instance	<code>self</code>	Oui	Non
Méthode de classe	<code>cls</code>	Non	Oui
Méthode statique	Aucun	Non	Non

CAS D'UTILISATION DES MÉTHODES DE CLASSE

Les méthodes de classe sont utiles pour :

- Créer des méthodes de fabrique.
- Accéder et modifier des variables de classe.
- Implémenter des comportements qui concernent la classe dans son ensemble.

CAS D'UTILISATION DES MÉTHODES STATIQUES

Les méthodes statiques sont utiles pour :

- Grouper des fonctions utilitaires liées à la classe.
- Implémenter des comportements qui ne dépendent ni de l'instance ni de la classe.
- Améliorer la lisibilité du code en regroupant les fonctions dans la classe pertinente.

COMPOSITION

DÉFINITION DE LA COMPOSITION

La composition est un concept de la POO où une classe est composée d'une ou plusieurs instances d'autres classes. Elle permet de créer des objets complexes en combinant des objets plus simples. Contrairement à l'héritage, la composition favorise la réutilisation de code sans créer de hiérarchies de classes. Elle est souvent décrite par la relation "a un" (has-a). Par exemple, une voiture "a un" moteur, un volant, des roues, etc.

CRÉATION DE CLASSES COMPOSÉES

Pour créer une classe composée, définissez des attributs qui sont des instances d'autres classes.

```
class Moteur:  
    pass  
  
class Voiture:  
    def __init__(self):  
        self.moteur = Moteur()
```

AVANTAGES DE LA COMPOSITION

- Favorise la réutilisation de code.
- Réduit la complexité des hiérarchies de classes.
- Facilite les modifications et les extensions.
- Permet de créer des objets plus flexibles et modulaires.
- Encourage une conception axée sur les responsabilités.

COMPARAISON AVEC L'HÉRITAGE

Héritage

Relation "est un" (is-a)

Crée des hiérarchies de classes

Moins flexible pour les changements

Partage d'implémentation

Composition

Relation "a un" (has-a)

Crée des objets modulaires

Plus flexible et modulaire

Réutilisation de composants

IMPLÉMENTATION DE LA COMPOSITION EN PYTHON

```
class Moteur:  
    def demarrer(self):  
        print("Moteur démarré")  
  
class Voiture:  
    def __init__(self):  
        self.moteur = Moteur()  
  
    def demarrer_voiture(self):  
        self.moteur.demarrer()
```

UTILISATION DE LA COMPOSITION DANS DES PROJETS RÉELS

- Dans les jeux vidéo, pour créer des entités comme des personnages avec des composants (santé, armure, armes).
- Dans les applications web, pour structurer des modules réutilisables (authentification, gestion des utilisateurs).
- Dans les systèmes embarqués, pour modéliser des composants matériels (capteurs, actionneurs).
- Dans les frameworks, pour créer des bibliothèques modulaires et extensibles.

INTERFACES ET PROTOCOLES

NOTION D'INTERFACE

Une interface définit un contrat que les classes doivent suivre. Elle spécifie les méthodes qu'une classe doit implémenter. En Python, les interfaces ne sont pas explicitement définies. On utilise des classes abstraites pour simuler les interfaces. Les interfaces favorisent la flexibilité et la réutilisabilité du code. Elles permettent la séparation des préoccupations.

IMPLÉMENTATION D'UNE INTERFACE

En Python, on utilise le module `abc` pour créer des interfaces. Voici un exemple de classe abstraite :

```
from abc import ABC, abstractmethod

class MonInterface(ABC):
    @abstractmethod
    def ma_methode(self):
        pass
```

PROTOCOLES EN PYTHON

Les protocoles définissent un ensemble de méthodes et propriétés. Ils ne sont pas explicitement déclarés comme les interfaces. Un objet est considéré conforme à un protocole s'il implémente les méthodes requises. Les protocoles sont utilisés pour le typage statique avec `typing`. Ils permettent une vérification plus flexible des types.

UTILISATION DES PROTOCOLES

Les protocoles sont définis avec le module `typing`. Exemple d'utilisation d'un protocole :

```
from typing import Protocol

class MonProtocole(Protocol):
    def ma_methode(self) -> str:
        pass
```

DIFFÉRENCES ENTRE INTERFACES ET PROTOCOLES

Aspect	Interface	Protocole
Déclaration	Classe abstraite avec abc	Protocol du module <code>typing</code>
Utilisation	Héritage explicite	Conformité implicite
Vérification	À l'exécution	À la compilation (typage statique)
Flexibilité	Moins flexible	Plus flexible

EXEMPLE D'INTERFACE EN PYTHON

```
from abc import ABC, abstractmethod

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

class Chien(Animal):
    def parler(self):
        return "Woof"

class Chat(Animal):
    def parler(self):
        return "Meow"
```

GESTION DES EXCEPTIONS EN POO

CONCEPT D'EXCEPTION

Une exception est une erreur détectée pendant l'exécution d'un programme. Les exceptions interrompent le flux normal du programme. Elles sont gérées pour éviter la terminaison brutale du programme. Exemples courants : division par zéro, fichier non trouvé. En POO, les exceptions sont des objets. Elles héritent de la classe **BaseException**. Les exceptions peuvent être levées et attrapées. Python fournit plusieurs types d'exceptions intégrées.

LEVÉE D'EXCEPTION

Pour lever une exception, utilisez le mot-clé `raise`. La syntaxe est : `raise ExceptionType("message")`. Exemple :

```
raise ValueError("Invalid value")
```

Vous pouvez lever des exceptions personnalisées. Le message fournit des détails sur l'erreur. Le programme s'arrête si l'exception n'est pas gérée.

GESTION DES EXCEPTIONS AVEC TRY/EXCEPT

Utilisez `try` et `except` pour gérer les exceptions. Syntaxe :

```
try:  
    # Code pouvant provoquer une exception  
except ExceptionType:  
    # Code de gestion de l'exception
```

Exemple :

```
try:  
    result = 10 / 0  
except ZeroDivisionError:  
    print("Division par zéro interdite")
```

UTILISATION DE FINALLY

Le bloc `finally` s'exécute toujours, qu'il y ait exception ou non. Syntaxe :

```
try:  
    # Code pouvant provoquer une exception  
except ExceptionType:  
    # Code de gestion de l'exception  
finally:  
    # Code qui s'exécute en tout cas
```

Exemple :

```
try:  
    file = open('data.txt', 'r')  
except FileNotFoundError:  
    print("Fichier non trouvé")  
finally:  
    print("Exécution du bloc finally")
```

CRÉATION D'EXCEPTIONS PERSONNALISÉES

Créez des exceptions personnalisées en héritant de `Exception`. Syntaxe :

```
class MyException(Exception):
    pass
```

Exemple :

```
class InvalidAgeError(Exception):
    def __init__(self, age):
        self.age = age
        super().__init__(f"Invalid age: {age}")
```

Levez l'exception avec `raise InvalidAgeError(age)`.

LES EXCEPTIONS INTÉGRÉES EN PYTHON

Python fournit plusieurs exceptions intégrées :

- **ValueError** : Valeur incorrecte
- **TypeError** : Type incorrect
- **IndexError** : Index hors limites
- **KeyError** : Clé non trouvée
- **FileNotFoundException** : Fichier non trouvé
- **ZeroDivisionError** : Division par zéro

Consultez la documentation pour la liste complète.

BONNES PRATIQUES DE GESTION DES EXCEPTIONS

- Gérer les exceptions spécifiques plutôt que générales.
- Utiliser des messages d'erreur clairs et informatifs.
- Éviter de masquer les erreurs.
- Utiliser `finally` pour le nettoyage des ressources.
- Documenter les exceptions levées par les fonctions.
- Créer des exceptions personnalisées pour des cas spécifiques.
- Tester le code pour les scénarios d'exception.