POO

C’est quoi la POO en Python

La POO est une approche différente pour concevoir et organiser le code, basée sur des objets qui possèdent des variables et des fonctions spécifiques.

Les principaux objectifs de la POO sont de créer des scripts clairs, bien structurés, modulables et faciles à maintenir et à déboguer. Nous explorerons ces avantages de manière pratique dans la suite.

Python est un langage fortement orienté objet, ce qui implique que la notion d'objets est au cœur même du langage.

En réalité, presque tout en Python est un objet, même si nous ne nous en sommes pas rendu compte jusqu'à présent. Les types de données tels que les chaînes de caractères (str), les entiers (int), les listes (list), etc., sont des objets. Les fonctions sont également des objets

Definition d’un objet en programmation

En informatique, le concept d'objets s'inspire de la vie réelle où un objet possède des caractéristiques et permet d'effectuer des actions. Par exemple, un crayon a des caractéristiques telles que sa taille, sa couleur, sa forme, etc., et permet d'écrire ou de dessiner.

En informatique, un objet est un bloc de code cohérent qui possède ses propres variables (équivalentes aux caractéristiques des objets réels) et fonctions (actions). Les objets informatiques peuvent être simples ou complexes, tout comme les objets du quotidien.

Dans Python, les variables et fonctions d'un objet sont appelées attributs, plus précisément attributs de données pour les variables et méthodes pour les fonctions spécifiques à l'objet.

Dans la plupart des langages informatiques, on utilise plutôt le terme "membres" pour désigner les variables et fonctions d'un objet, "propriété" pour les variables, et "méthodes" pour les fonctions.

Definition d’une class

Une classe est une structure contenant généralement des variables et des fonctions. Elle sert de plan pour créer des objets qui partagent le même ensemble d'attributs de données et de méthodes de base.

Les classes sont considérées comme les principaux outils de la POO, car elles permettent de mettre en œuvre des concepts fondamentaux tels que l'héritage, l'encapsulation et le polymorphisme. Nous explorerons ces concepts en détail plus tard.

Pour l'instant, retenez simplement que créer une classe équivaut à définir un nouveau type d'objets ou un nouveau type de données. Une classe agit comme un modèle pour la création d'objets d'un certain type.

Création d'une class

Pour créer une nouvelle classe Python on utilise le mot clef class suivi du nom de notre classe. Ici, on va créer une classe Utilisateur qui va être très simple pour le moment.

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

    def dire_bonjour(self):
        print(f"Bonjour, je m'appelle {self.nom} et j'ai {self.age} ans.")

personne1 = Personne("Alice", 25)
personne1.dire_bonjour()  # Affiche "Bonjour, je m'appelle Alice et j'ai 25 ans."
```

Une classe est un modèle pour créer des objets. Elle définit les attributs et les méthodes que les objets de cette classe posséderont.
Les attributs sont des variables qui stockent les données associées à un objet.
Les méthodes sont des fonctions qui effectuent des opérations sur les attributs d'un objet.

Encapsulation et visibilité des attributs :

L'encapsulation est le principe de regrouper les données et les méthodes associées dans une même entité, la classe.
Les attributs peuvent être publics, privés ou protégés pour contrôler leur accès depuis l'extérieur de la classe.
Les méthodes d'accès, comme les getters et les setters, permettent d'interagir avec les attributs de manière contrôlée.

- Les attributs publics sont accessibles depuis n'importe où. Ils sont nommés sans préfixe particulier (par exemple, nom).
- Les attributs protégés sont conventionnellement indiqués par un seul underscore en préfixe (par exemple, _age).
- Les attributs privés sont conventionnellement indiqués par deux underscores en préfixe (par exemple, __salaire).

Encapsulation et visibilité des attributs :

l'attribut nom est un attribut public. Il peut être accédé et modifié directement en utilisant la syntaxe objet.attribut.

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

personne = Personne("Alice")
print(personne.nom)  # Accès à l'attribut public "nom"
personne.nom = "Bob"  # Modification de l'attribut public "nom"
```

Encapsulation et visibilité des attributs :

Attributs protégés : Les attributs protégés sont conventionnellement indiqués en ajoutant un seul underscore _ en préfixe à leur nom. Bien que les attributs protégés puissent être accédés et modifiés depuis l'extérieur de la classe, il est généralement recommandé de les considérer comme étant réservés à un usage interne de la classe ou de ses sous-classes. Cela signifie que leur accès direct depuis l'extérieur de la classe n'est pas encouragé.

```python
class Personne:
    def __init__(self, nom):
        self._age = 25  # Attribut protégé

personne = Personne("Alice")
print(personne._age)  # Accès à l'attribut protégé "_age"
personne._age = 30    # Modification de l'attribut protégé "_age"
```

Encapsulation et visibilité des attributs :

Attributs privés : Les attributs privés sont conventionnellement indiqués en ajoutant deux underscores __ en préfixe à leur nom. Ils sont destinés à être utilisés uniquement à l'intérieur de la classe et ne doivent pas être accessibles depuis l'extérieur.

```python
class Personne:
    def __init__(self, nom):
        self.__salaire = 5000  # Attribut privé

personne = Personne("Alice")
print(personne.__salaire) #Erreur d'accès
print(personne._Personne__salaire)  # Accès à l'attribut privé "__salaire" de manière non recommandée
personne.__salaire = 6000          # Tentative de modification de l'attribut privé "__salaire"
```

Héritage et polymorphisme

L'héritage est un mécanisme qui permet à une classe d'hériter des attributs et des méthodes d'une autre classe appelée classe parente ou classe de base. La classe qui hérite est appelée classe enfant ou classe dérivée. L'héritage permet la réutilisation du code et la création d'une hiérarchie de classes.

```
class ClasseEnfant(ClasseParent):
    # Définition de la classe enfant
    ...
```

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

    def dire_bonjour(self):
        print(f"Bonjour, je m'appelle {self.nom}.")

class Employe(Personne):
    def __init__(self, nom, salaire):
        super().__init__(nom)
        self.salaire = salaire

    def travailler(self):
        print(f"{self.nom} travaille dur.")

class Manager(Employe):
    def __init__(self, nom, salaire, equipe):
        super().__init__(nom, salaire)
        self.equipe = equipe

    def organiser_reunion(self):
        print(f"{self.nom} organise une réunion avec son équipe.")

class Vendeur(Employe):
    def __init__(self, nom, salaire, objectif_vente):
        super().__init__(nom, salaire)
        self.objectif_vente = objectif_vente

    def vendre(self):
        print(f"{self.nom} réalise une vente.")
```

```python
personne.dire_bonjour()
employe.dire_bonjour()
employe.travailler()
manager.dire_bonjour()
manager.organiser_reunion()
vendeur.dire_bonjour()
vendeur.vendre()
```

Cela permet de réutiliser le code existant de la classe parente dans la classe dérivée, tout en ajoutant des fonctionnalités spécifiques à la classe dérivée.

```
super().nom_de_la_methode(args)
```

Lorsque vous définissez un constructeur dans une classe dérivée, vous pouvez utiliser super().__init__(args) pour appeler le constructeur de la classe parente et initialiser les attributs hérités de la classe parente. Cela vous évite de réécrire le code d'initialisation déjà présent dans la classe parente.

```python
employes = [manager, vendeur]

for employe in employes:
    employe.dire_bonjour()

    if isinstance(employe, Manager):
        employe.organiser_reunion()
    elif isinstance(employe, Vendeur):
        employe.vendre()
```

Les methodes d'instance

Les méthodes d'instance sont liées à des instances spécifiques de la classe. Elles peuvent accéder aux attributs de l'instance via le paramètre self. Les méthodes d'instance sont les plus couramment utilisées dans la programmation orientée objet et sont définies sans aucun décorateur particulier.

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

    def dire_bonjour(self):
        print(f"Bonjour, je m'appelle {self.nom}.")

personne = Personne("Alice")
personne.dire_bonjour()
```

Les methodes de classe

Les méthodes de classe sont associées à la classe elle-même plutôt qu'à une instance spécifique de la classe. Elles sont généralement utilisées pour effectuer des opérations qui concernent la classe dans son ensemble plutôt que des instances individuelles. Les méthodes de classe sont définies à l'aide du décorateur @classmethod et prennent cls comme premier paramètre, qui fait référence à la classe elle-même.

```python
class Personne:
    nombre_personnes = 0

    def __init__(self, nom):
        self.nom = nom
        Personne.nombre_personnes += 1

    @classmethod
    def compter_personnes(cls):
        return cls.nombre_personnes
```

```python
personne1 = Personne("Alice")
personne2 = Personne("Bob")
personne3 = Personne("Eve")


nombre_total_personnes = Personne.compter_personnes()
print("Nombre total de personnes :", nombre_total_personnes)
```

Les methodes static

Les méthodes statiques sont des méthodes qui n'ont pas accès aux instances de la classe et ne prennent pas self ou cls comme paramètres. Elles sont généralement utilisées pour des opérations qui ne dépendent pas de l'état de l'instance ou de la classe. Les méthodes statiques sont définies à l'aide du décorateur @staticmethod.

```python
class Personne:
    count=5
    def __init__(self, nom):
        self.nom = nom

    @staticmethod
    def hello(nom):
        print("hello ",nom)

Personne.hello("Mouun")
```

Class abstraite

Interfaces et classes abstraites :
Les interfaces définissent un ensemble de méthodes qu'une classe doit implémenter.
En Python, les interfaces sont implémentées en utilisant des classes abstraites.
Une classe abstraite est une classe qui ne peut pas être instanciée directement, mais qui sert de base pour d'autres classes.
En Python, les classes abstraites sont créées en utilisant le module abc et en dérivant de la classe ABC (Abstract Base Class).
Les méthodes abstraites sont des méthodes déclarées dans une classe abstraite mais sans implémentation. Les sous-classes doivent les implémenter.

Notez que si une classe dérivée ne redéfinit pas la méthode abstraite de la classe abstraite, une erreur sera levée lors de l'instanciation de cette classe. Cela garantit que toutes les classes dérivées implémentent bien la méthode abstraite.

Class abstraite

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def dire_bonjour(self):
        pass

class Etudiant(Personne):
    def dire_bonjour(self):
        print(f"Bonjour, je suis l'étudiant {self.nom}.")

class Enseignant(Personne):
    def dire_bonjour(self):
        print(f"Bonjour, je suis l'enseignant {self.nom}.")
```

```python
personne1 = Etudiant("Alice")
personne2 = Enseignant("Bob")

personne1.dire_bonjour()
personne2.dire_bonjour()
```

Class abstraite

Interfaces et classes abstraites :
Les interfaces définissent un ensemble de méthodes qu'une classe doit implémenter.
En Python, les interfaces sont implémentées en utilisant des classes abstraites.
Une classe abstraite est une classe qui ne peut pas être instanciée directement, mais qui sert de base pour d'autres classes.
En Python, les classes abstraites sont créées en utilisant le module abc et en dérivant de la classe ABC (Abstract Base Class).
Les méthodes abstraites sont des méthodes déclarées dans une classe abstraite mais sans implémentation. Les sous-classes doivent les implémenter.

Notez que si une classe dérivée ne redéfinit pas la méthode abstraite de la classe abstraite, une erreur sera levée lors de l'instanciation de cette classe. Cela garantit que toutes les classes dérivées implémentent bien la méthode abstraite.

Methode speciale

Les méthodes spéciales, également appelées méthodes du double soulignement (ou dunder methods) en Python, sont des méthodes prédéfinies avec des noms spéciaux entourés de double soulignement. Elles permettent de définir le comportement spécial des objets dans des situations particulières

__init__: La méthode __init__ est utilisée pour l'initialisation d'un objet dès sa création.
__str__: La méthode __str__ est utilisée pour afficher une représentation lisible de l'objet sous forme de chaîne de caractères.
__repr__: La méthode __repr__ est utilisée pour fournir une représentation détaillée de l'objet, souvent utilisée pour le débogage.
__len__: La méthode __len__ est utilisée pour obtenir la longueur ou la taille d'un objet.
__getitem__: La méthode __getitem__ est utilisée pour accéder aux éléments d'un objet en utilisant l'opérateur d'indexation [ ].
__iter__ et __next__: Ces méthodes sont utilisées pour définir un itérable et un itérateur personnalisés.

Methodes speciales

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

    def __str__(self):
        return f"Personne : {self.nom}"

    def __repr__(self):
        return f"Personne(nom='{self.nom}')"

    def __len__(self):
        return len(self.nom)

    def __getitem__(self, index):
        return self.nom[index]

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < len(self.nom):
            char = self.nom[self._index]
            self._index += 1
            return char
        else:
            raise StopIteration
```

```python
personne = Personne("Alice")
print(personne)                 # Affiche "Personne : Alice"
print(repr(personne))           # Affiche "Personne(nom='Alice')"
print(len(personne))            # Affiche 5
print(personne[2])              # Affiche 'i'

for char in personne:
    print(char)                  # Affiche chaque caractère du nom "Alice"
```