Introduction à la POO en Python

Définition :

La programmation orientée objet (POO) est un paradigme qui utilise des objets et des classes.

Concepts principaux :

- Classe : Modèle pour créer des objets.
- Objet : Instance d'une classe.
- Méthodes : Fonctions définies dans une classe.
- Attributs : Variables associées à une classe/objet.

Création de classes et constructeurs

```python
class Personne:
    def __init__(self):
        print("Le constructeur est lancé")

unePersonne = Personne()
print(unePersonne)
```

Points importants :

- Le constructeur __init__ s'exécute automatiquement lors de la création de l'objet.
- Un objet est une instance de la classe.

Attributs d'objets

```python
class Point:
    """Un joli commentaire"""

p = Point()
p.x = 3.0
p.y = 12
print(p.x, p.y)
print(p.__doc__)
```

Attributs :

- Les variables p.x et p.y sont définies dynamiquement pour l'objet.
- Les classes peuvent avoir une **documentation** accessible via __doc__.

Initialisation d'attributs

Attributs dans le constructeur :

- Initialisation directe des propriétés avec des paramètres.

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

unePersonne = Personne("KORRI", "Ilyas", "Clichy", 47)
print(unePersonne.nom, unePersonne.prenom,  unePersonne.age)
```

Valeurs par défaut

Valeurs par défaut :

Les paramètres peuvent avoir des valeurs par défaut dans __init__

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

unePersonne = Personne("Samuel", "BOUHENIC")
moi = Personne("Ilyas")
print(unePersonne.nom, moi.nom)
```

Méthodes dans les classes

```python
class Personne:
    def __init__(self, nom="KORRI", prenom="Ilyas"):
        self.nom = nom
        self.prenom = prenom

    def afficher(self):
        print(f"Nom : {self.nom}, Prénom : {self.prenom}")

moi = Personne()
moi.afficher()
```

Méthodes :
Les fonctions dans une classe permettent d'effectuer des opérations sur les objets.

Calcul dans une méthode

```python
class Personne:
    def __init__(self, nom, prenom, note1, note2, note3):
        self.nom = nom
        self.prenom = prenom
        self.note1 = note1
        self.note2 = note2
        self.note3 = note3

    def afficher_moyenne(self):
        self.moyenne = (self.note1 + self.note2 + self.note3) / 3
        print(f"La moyenne de {self.prenom} est {self.moyenne}")

unePersonne = Personne("Dupont", "Bernard", 8, 9, 19)
unePersonne.afficher_moyenne()
```

Attribut calculé : La moyenne est calculée et stockée dans un attribut.

Méthode avec une Valeur de Retour

```python
class Rectangle:
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur

    def calculer_surface(self):
        return self.longueur * self.largeur

mon_rectangle = Rectangle(10, 5)
surface = mon_rectangle.calculer_surface()
print(f"La surface du rectangle est : {surface}")
```

Valeur retournée: utilisée ultérieurement dans le programme.

Surcharge de __str__

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

    def __str__(self):
        return f"Nom: {self.nom}, Prénom: {self.prenom}"

unePersonne = Personne("KORRI", "Ilyas")
print(unePersonne)
```

__str__ : Permet de personnaliser l'affichage de l'objet.

Utilisation de __getattr__

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

    def __getattr__(self, attr):
        return f"L'attribut '{attr}' n'existe pas dans cet objet."

moi = Personne("KORRI", "Ilyas")
# Accès à un attribut inexistant
print(moi.age)  # Affiche : L'attribut 'age' n'existe pas dans cet objet.
```

Si l’attribut existe (nom ou prenom), il est retourné normalement.
Si l’attribut n’existe pas (age), __getattr__ est exécuté pour gérer l’erreur.

.

La Méthode Spéciale __repr__

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

    # def __str__(self):
    #     return f"Nom: {self.nom}, Prénom: {self.prenom}"

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

# Création de l'objet
moi = Personne("KORRI", "Ilyas", 30)

print(moi)  # Appelle automatiquement __repr__
```

L’Attribut Spécial __dict__

Il contient un **dictionnaire** qui stocke les attributs de l’objet sous forme de paires clé-valeur.

```
class Switch:

    def __init__(self, nbPort,administrable ):
        self.nbPort = nbPort
        self.administrable = administrable

objet = Switch(24, True)
print(objet.__dict__)
```

Exemple avec plusieurs objets

```python
class Ordinateur:
    def __init__(self, marque, prix):
        self.marque = marque
        self.prix = prix

    def afficher(self):
        print(f"Marque : {self.marque}, Prix : {self.prix}€")

mac = Ordinateur("Macbook", 1700)
acer = Ordinateur("Acer", 3000)
mac.afficher()
acer.afficher()
```

Multiples instances : Chaque objet a ses propres valeurs d'attribut

Méthodes conditionnelles

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

    def isCoder(self):
        if self.codeur:
            print(f"{self.nom} est codeur")
        else:
            print(f"{self.nom} n'est pas codeur")

ilyas = Personne("KORRI", True)
ilyas.isCoder()
```

Méthodes basées sur une condition : Actions différenciées selon les attributs

Attribut de classe

```python
class Outil:
    outils_crees = 0  # Attribut de classe

    def __init__(self, nom):
        self.nom = nom
        Outil.outils_crees += 1

outil1 = Outil("Marteau")
outil2 = Outil("Tournevis")
print(Outil.outils_crees)  # 2
```

Permet par exemple de compter le nombre d’objet créés

Le décorateur @classmethod

```python
class Outil:
    outils_crees = 0  # Attribut de classe

    def __init__(self, nom):
        self.nom = nom
        Outil.outils_crees += 1

    @classmethod
    def outils_total(cls):
        return cls.outils_crees

outil1 = Outil("Marteau")
outil2 = Outil("Tournevis")
print(Outil.outils_total())  # 2
```

Gestion des Exceptions dans les Méthodes

Les méthodes peuvent inclure des blocs try/except pour une gestion robuste des erreurs.

```
class Calculatrice:
    def diviser(self, a, b):
        try:
            return a / b
        except ZeroDivisionError:
            return "Erreur : Division par zéro"

calc = Calculatrice()
print(calc.diviser(10, 0))  # Erreur : Division par zéro
```

Créer un objet dans une classe

```python
class Adresse:
    def __init__(self, rue, ville, code_postal):
        self.rue = rue
        self.ville = ville
        self.code_postal = code_postal

class Etudiant:
    def __init__(self, nom, prenom, rue, ville, code_postal):
        self.nom = nom
        self.prenom = prenom
        # Création d'un objet Adresse lors de l'instanciation d'un étudiant
        self.adresse = Adresse(rue, ville, code_postal)

    def afficher_etudiant(self):
        return f"""
{self.prenom} {self.nom}
{self.adresse.rue}
{self.adresse.code_postal} {self.adresse.ville}"""

etudiant = Etudiant("Dupont", "Alice", "12 rue des Lilas", "Paris", "75012")
print(etudiant.afficher_etudiant())
```

Gestion des Types dans un Constructeur

La méthode **isinstance** permet de vérifier les types des arguments

```
class IoTDevice:
    def __init__(self, device_id: str):
        if not isinstance(device_id, str):
            print(f"""device_id n'est pas un str, \
    c'est un {type(device_id).__name__}""")
        self.device_id: str = device_id

device = IoTDevice("device123")  # OK
print(device.__class__.__name__)
device = IoTDevice(123)   # Affiche un message d'erreur
```

Gestion des Types dans un Constructeur

La méthode **isinstance** permet de vérifier les types des arguments

```
class IoTDevice:
    def __init__(self, device_id: str):
        if not isinstance(device_id, str):
            print(f"""device_id n'est pas un str, \
    c'est un {type(device_id).__name__}""")
        self.device_id: str = device_id

device = IoTDevice("device123")  # OK
print(device.__class__.__name__)
device = IoTDevice(123)   # Affiche un message d'erreur
```

Valider avant de créer l'objet avec une fonction

On peut effectuer une validation des arguments avant d'appeler le constructeur avec une fonction.

```python
class IoTDevice:
    def __init__(self, device_id: str):
        self.device_id: str = device_id

def create_device(device_id):
    if not isinstance(device_id, str):
        print(f"Le type device_id: {device_id} est invalide")
        return None
    return IoTDevice(device_id)

device1 = create_device("device123")  # OK
device2 = create_device(123) # Retourne None au lieu de lever une exception

if device1:
    print(device1.__dict__)  # Affiche les attributs de device1
if device2:
    print(device2.__dict__)  # Ne s'exécute pas car device2 est None
```

Méthode de classe pour encapsuler la création

On peut créer une méthode de classe pour valider les types avant la création.

```python
class IoTDevice:
    def __init__(self, device_id: str):
        self.device_id: str = device_id

    @classmethod
    def create(cls, device_id):
        if not isinstance(device_id, str):
            print(f"Le type de device_id: {device_id} est invalide")
            return None
        return cls(device_id)

device1 = IoTDevice.create("device123")  # OK
device2 = IoTDevice.create(123)  # Retourne None
```