Dunder Functions

Les "dunder functions" (méthodes spéciales en Python) sont des méthodes dont le nom est encadré par deux double underscores (d'où le terme "dunder", qui vient de "double underscore"). Ces méthodes spéciales sont utilisées pour définir des comportements spécifiques pour les objets de classe et sont invoquées implicitement dans certaines situations par l'interpréteur Python.

Voici quelques points importants à retenir sur les dunder functions :

1. **Syntaxe** : Les dunder functions ont un nom spécifique précédé et suivi de deux underscores, par exemple :

```
__init__
__repr__
__add__
```

2. **Fonctionnalité** : Chaque dunder function a une fonctionnalité spécifique qui lui est associée. Par exemple, `__init__` est utilisé pour initialiser un nouvel objet, `__repr__` pour obtenir une représentation sous forme de chaîne de caractères de l'objet, `__add__` pour définir l'addition entre deux objets, etc.

3. **Invocation** : Les dunder functions sont invoquées automatiquement dans des situations particulières. Par exemple, lorsque vous créez une nouvelle instance d'une classe en utilisant `obj = MaClasse()`, Python appelle automatiquement la méthode `__init__` pour initialiser l'objet.

4. **Surcharge** : Vous pouvez surcharger (redéfinir) les dunder functions dans vos propres classes pour modifier le comportement par défaut. Cela vous permet d'adapter le comportement de vos objets aux besoins spécifiques de votre application.

5. **Fonctionnalités avancées** : Les dunder functions permettent d'implémenter des fonctionnalités avancées telles que la surcharge d'opérateurs, la gestion du cycle de vie des objets, la conversion de types, l'itération sur des objets, etc.

Exemples

1. init et new :

Utilisez `__new__` pour la création d'un nouvel objet et `__init__` pour initialiser cet objet avec des valeurs par défaut. Veillez à bien comprendre la différence entre ces deux méthodes.

Exemple :

```
class MaClasse:
    def __new__(cls):
        print("__new__ est appelé")
        instance = super().__new__(cls)
        return instance

    def __init__(self):
        print("__init__ est appelé")

# Création d'une instance de MaClasse
objet = MaClasse()
```

Résultat:

```
__new__ est appelé
__init__ est appelé
```

Dans cet exemple, **new** est appelé en premier pour créer une nouvelle instance de la classe. Ensuite, **init** est appelé pour initialiser cette instance.

2. repr et str :

Définissez `__repr__` pour une représentation détaillée de l'objet, souvent utilisée pour le débogage. Utilisez `__str__` pour une représentation plus lisible destinée aux utilisateurs finaux.

Exemple :

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

    def __repr__(self):
        return f"MaClasse({self.x}, {self.y})"

    def __str__(self):
        return f"({self.x}, {self.y})"

# Création d'une instance de MaClasse
objet = MaClasse(10, 20)

# Affichage de l'objet
print(repr(objet))  # Utilisation de __repr__
print(str(objet))   # Utilisation de __str__
```

Résultat :

```
MaClasse(10, 20)
(10, 20)
```

Dans cet exemple, **repr** est utilisé pour obtenir une représentation détaillée de l'objet, tandis que **str** est utilisé pour obtenir une représentation plus lisible de l'objet.

3. Conversion de types :

Utilisez `__int__`, `__float__`, `__complex__` pour définir la conversion de votre objet en types numériques. Cela rend votre objet plus flexible et compatible avec les opérations mathématiques standard.

Exemple `__int__` :

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

    def __int__(self):
        return int(self.x)

# Création d'une instance de MaClasse
objet = MaClasse(3.14)

# Conversion en entier en utilisant int()
entier = int(objet)

# Affichage du résultat
print(entier)  # Affiche : 3
```

Dans cet exemple, **int** est utilisé pour définir la conversion de l'objet en un entier en utilisant la fonction int().

Exemple `__float__` :

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

    def __float__(self):
        return float(self.x)

# Création d'une instance de MaClasse
objet = MaClasse(3)
```

```
# Conversion en float en utilisant float()
flottant = float(objet)

# Affichage du résultat
print(flottant)  # Affiche : 3.0
```

4. Comparaison :

Implémentez les méthodes de comparaison telles que `__eq__`, `__lt__`, `__gt__`, etc., pour permettre une comparaison logique entre vos objets. Cela est utile lors du tri ou de la recherche dans des collections d'objets.

Exemple `__eq__` (Égalité) :

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

    def __eq__(self, other):
        return self.x == other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(5)
objet2 = MaClasse(5)

# Comparaison d'égalité entre les deux objets
print(objet1 == objet2)  # Affiche : True
```

Dans cet exemple, **eq** est utilisé pour définir le comportement de l'opérateur d'égalité (==). Il retourne True si les attributs x des deux objets sont égaux, et False sinon.

Exemple `__lt__` (Infériorité) :

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

    def __lt__(self, other):
        return self.x < other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(3)
objet2 = MaClasse(5)

# Comparaison d'infériorité entre les deux objets
print(objet1 < objet2)  # Affiche : True
```

Exemple `__gt__` (Supériorité) :

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

    def __gt__(self, other):
        return self.x > other.x

# Création de deux instances de MaClasse
objet1 = MaClasse(3)
objet2 = MaClasse(5)

# Comparaison de supériorité entre les deux objets
print(objet1 > objet2)  # Affiche : False
```

Dans cet exemple, `__gt__` est utilisé pour définir le comportement de l'opérateur de comparaison de supériorité (>). Il retourne True si l'attribut x de l'objet 1 est supérieur à celui de l'objet 2, et False sinon.

5. Itération :

Si votre objet représente une séquence ou une collection, implémentez `__iter__` et `__next__` pour le rendre itérable. Cela permet d'utiliser des boucles for et d'autres fonctionnalités d'itération.

Exemples :

```
class MaClasse:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.limit:
            self.current += 1
            return self.current
        else:
            raise StopIteration

# Création d'une instance de MaClasse
obj = MaClasse(5)

# Utilisation de l'objet dans une boucle for
for valeur in obj:
    print(valeur)
```

```
#Résultat au terminal :
1
2
3
4
5
```

Dans cet exemple, nous définissons une classe MaClasse avec les dunder functions `__iter__` et `__next__`. Lorsque la méthode `__iter__` est appelée, elle retourne l'objet lui-même (ce qui permet à l'objet d'être son propre itérateur). La méthode `__next__` définit la logique pour générer les éléments de l'itérateur. À chaque appel de `__next__`, l'objet génère la valeur suivante jusqu'à ce qu'elle atteigne la limite spécifiée. Lorsque la limite est atteinte, une exception StopIteration est levée, signalant à Python que l'itération est terminée.

L'utilisation de ces dunder functions permet à l'objet MaClasse d'être utilisé dans une boucle for ou avec d'autres structures d'itération en Python.

6. Gestion de la taille :

Implémentez `__len__` pour permettre à votre objet d'être utilisé avec la fonction len() et de fournir des informations sur sa taille ou sa longueur.

Exemples :

```
class MaListe:
    def __init__(self, elements):
        self.elements = elements

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

# Création d'une instance de MaListe
liste = MaListe([1, 2, 3, 4, 5])

# Utilisation de len() sur l'instance
print(len(liste))  # Utilisation de __len__
```

Résultat:

```
5
```

Dans cet exemple, **len** est utilisé pour permettre l'utilisation de la fonction len() sur des instances de la classe MaListe, ce qui renvoie le nombre d'éléments dans la liste.

7. Cycle de vie de l'objet :

Utilisez `__del__` avec précaution, car il est appelé lorsque l'objet est supprimé de la mémoire. Évitez de vous appuyer sur cette méthode pour la gestion des ressources, car le moment de sa déclenchement n'est pas garanti.

Exemple :

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

    def __del__(self):
        print(f"L'objet de la classe MaClasse avec le nom '{self.nom}' a été
détruit.")

# Création d'une instance de MaClasse
objet = MaClasse("MonObjet")

# Suppression explicite de l'objet
del objet
```

8. Autres méthodes spéciales :

Explorez d'autres méthodes spéciales en fonction des besoins spécifiques de votre objet. Par exemple, `__call__` pour rendre l'objet appelable comme une fonction, `__getitem__` pour permettre l'indexation de l'objet comme une liste, etc.

More :

1. `__name__` :

- La dunder function `__name__` est utilisée pour accéder au nom d'une fonction, d'une classe, d'une méthode ou d'un module en Python.
- Elle retourne le nom de l'objet sous forme de chaîne de caractères.
- Utile pour obtenir dynamiquement le nom de l'objet à des fins de documentation, de débogage ou d'introspection.

```
def ma_fonction():
    pass

print(ma_fonction.__name__)  # Affiche : ma_fonction
```

2. `__doc__` :

- La dunder function `__doc__` est utilisée pour accéder à la documentation d'une fonction, d'une classe, d'une méthode ou d'un module en Python.
- Elle retourne la docstring associée à l'objet sous forme de chaîne de caractères.
- Utile pour fournir des informations sur l'objet aux utilisateurs et aux développeurs.

```
def ma_fonction():
    '''Cette fonction ne fait rien de spécial.'''
    pass

print(ma_fonction.__doc__)  # Affiche : Cette fonction ne fait rien de spécial.
```

3. `__class__` :

- La dunder function `__class__` est utilisée pour accéder à la classe d'une instance en Python.
- Elle retourne la classe à laquelle l'instance appartient.
- Utile pour accéder aux attributs et méthodes de la classe à partir d'une instance.

```
class MaClasse:
    pass

objet = MaClasse()
print(objet.__class__)  # Affiche : <class '__main__.MaClasse'>
```

4. `__dict__` :

- La dunder function **dict** est utilisée pour accéder au dictionnaire d'attributs d'un objet en Python.
- Elle retourne un dictionnaire contenant les attributs de l'objet et leurs valeurs.
- Utile pour inspecter dynamiquement les attributs d'un objet et pour effectuer des opérations de manipulation d'attributs.

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

objet = MaClasse(5, 10)
print(objet.__dict__)  # Affiche : {'x': 5, 'y': 10}
```