Les méthodes magiques ou Dunder methods

Introduction

Les « méthodes magiques », souvent appelées « dunder methods » car leurs noms sont entourés de deux underscores, sont des points d'extension que Python invoque automatiquement pour intégrer nos objets aux mécanismes du langage.

Elles ne sont pas destinées à être appelées directement par l'utilisateur, mais par l'interpréteur lorsqu'une opération donnée survient.

Par exemple, l'initialisation d'un objet utilise `__init__`, l'addition peut solliciter `__add__`, la comparaison `__lt__`, l'itération `__iter__`, et l'obtention d'une représentation textuelle `__repr__` ou `__str__`.

Grâce à elles, on ne « colle » pas des fonctions autour d'une classe, on rend l'objet nativement compatible avec les opérations usuelles de Python, ce qui améliore la lisibilité, le débogage et l'interopérabilité.

`__repr__` ou la représentation textuelle

On pose une classe simple, volontairement minimale, qui servira de fil rouge :

```
class Person:
    def __init__(self, nom: str, prenom: str, age: int) -> None:
        self.nom = nom
        self.prenom = prenom
        self.age = age
```

Si on instancie puis on affiche une instance dans l'interpréteur interactif,

```
alice = Person("Alice", "Smith", 19)
print(alice)
# <__main__.Person object at 0x...>
```

on obtient généralement une forme peu informative qui ressemble à `<__main__.Person object at 0x...>`.

Cette sortie est la représentation par défaut fournie par `object`, et elle n'aide ni à comprendre l'état interne de l'objet, ni à déboguer une collection d'objets.

On pourrait aussi utiliser `print(alice.__dict__)` pour afficher les attributs d'un objet, mais cela ne rend pas le code plus lisible.

```
print(alice.__dict__)
# {'nom': 'Alice', 'prenom': 'Smith', 'age': 19}
```

C'est exactement le rôle de `__repr__` de corriger cela.

La méthode `__repr__` doit renvoyer une chaîne de caractères qui décrit l'objet de manière non ambiguë.

La convention en Python veut que cette représentation soit « non-ambiguë et, si raisonnable, évaluable », c'est-à-dire qu'elle ressemble à une expression valide permettant de reconstruire l'objet.

En pratique, l'objectif principal reste le débogage : `__repr__` est utilisé par la fonction `repr(obj)`, par l'affichage de l'objet en console interactive, par les conteneurs quand ils listent leurs éléments, et très souvent dans les journaux applicatifs. On écrit donc `__repr__` pour soi-même et pour ses outils, afin de voir rapidement les attributs-clés sans ouvrir un débogueur.

On l'implémente pas à pas sur notre classe. On commence par une version claire, compacte, et fidèle aux valeurs internes :

```
class Person:
    def __init__(self, nom: str, prenom: str, age: int) -> None:
        self.nom = nom
        self.prenom = prenom
        self.age = age

    def __repr__(self) -> str:
        return f"Person(nom={self.nom!r}, prenom={self.prenom!r}, age=
{self.age!r})"
```

Le suffixe `!r` dans les f-strings force l'utilisation de `repr()` pour chaque champ, ce qui garantit l'ajout de guillemets autour des chaînes et une représentation sans ambiguïté des valeurs.

Cette écriture évite des surprises comme des espaces manquants ou des chaînes non citées, et elle reste correcte même si un attribut contient des caractères spéciaux.

On observe maintenant le bénéfice immédiat. En console, `pers` s'affiche avec ses attributs :

```
alice = Person("Alice", "Smith", 19)
print(alice)
# Person(nom='Alice', prenom='Smith', age=19)
```

Le même avantage se retrouve lorsqu'on imprime des collections. Sans `__repr__`, une liste de personnes afficherait des adresses mémoire illisibles. Avec `__repr__`, on lit directement le contenu :

```
groupe = [
    Person("Dupont", "Alice", 30),
    Person("Martin", "Bob", 22)
]
print(groupe)
# [Person(nom='Dupont', prenom='Alice', age=30), Person(nom='Martin',
prenom='Bob', age=22)]
```

On précise le périmètre d'utilisation. On écrit `__repr__` dès qu'un objet représente une entité métier que l'on va manipuler, inspecter, trier, journaliser ou tester.

On le privilégie dans tous les contextes pédagogiques et professionnels où l'on veut comprendre rapidement l'état d'un objet au milieu d'un flux d'exécution.

On évite d'y exposer des secrets ou des champs volumineux : par exemple, on ne place pas un jeton d'API ou un long binaire dans `__repr__`. On limite le contenu à ce qui caractérise l'objet et permet de raisonner sur son état, ici `nom`, `prenom` et `age`.

`__add__` ou l'addition

Partons de la même classe `Person` que précédemment, mais on souhaite **additionner** deux personnes. Bizarre non ?

```
alice = Person("Alice", "Smith", 19)
john = Person("John", "Doe", 25)

print(alice + john)
# TypeError: unsupported operand type(s) for +: 'Person' and 'Person'
```

Python lève une erreur, car il ne sait pas comment additionner deux objets `Person`.

L'opérateur `+` fonctionne très bien pour des entiers, des flottants ou des chaînes de caractères, car il a été codé pour. Mais il n'a aucune idée de ce que cela signifie pour une classe que nous avons nous-mêmes créée.

Pour résoudre ce problème, on doit apprendre à Python ce que veut dire `+` dans ce contexte.

C'est le rôle de la méthode magique `__add__`.

Python appelle automatiquement cette méthode lorsqu'il rencontre l'opérateur `+` entre deux objets.

Ainsi, `alice + john` revient à écrire `alice.__add__(john)`.

Si cette méthode n'existe pas dans la classe, l'interpréteur ne sait pas quoi faire et renvoie l'erreur qu'on vient de voir.

On ajoute donc la méthode `__add__` dans la classe `Person` :

```
class Person:
    def __init__(self, nom: str, prenom: str, age: int) -> None:
        self.nom = nom
        self.prenom = prenom
        self.age = age

    def __repr__(self) -> str:
        return f"Person(nom={self.nom!r}, prenom={self.prenom!r}, age=
{self.age!r})"
```

```
    def __add__(self, other: 'Person') -> int:
        return self.age + other.age
```

> Voir plus bas l'explication de l'annotation `'Person'` dans la définition de la méthode `__add__`.

Cette méthode prend deux paramètres :

- `self` : la première personne (`alice` dans notre exemple)
- `other` : la seconde personne (`john`)

Elle retourne la somme des deux âges. Désormais, si on relance le code :

```
alice = Person("Alice", "Smith", 19)
john = Person("John", "Doe", 25)

print(alice + john)
```

Python exécute automatiquement :

```
alice.__add__(john)
```

et affiche :

```
44
```

L'opération `+` a donc maintenant un sens pour la classe `Person`.

Grâce à `__add__`, on a défini ce que "l'addition" de deux personnes signifie dans notre programme : ici, la somme de leurs âges.

On remarque que rien n'empêche de choisir une autre interprétation. Si on voulait, `__add__` pourrait combiner les prénoms, créer un nouvel objet ou même renvoyer une phrase. Mais dans tous les cas, l'idée centrale reste la même : avec `__add__`, on enseigne à Python comment utiliser `+` entre deux objets d'une même classe.

Pourquoi `'Person'` comme type de `other` ?

Lorsqu'on écrit la méthode suivante :

```
def __add__(self, other: 'Person') -> int:
    return self.age + other.age
```

le type `'Person'` est mis entre guillemets. Ce détail n'est pas anodin : il s'agit d'une **annotation de type différée**, aussi appelée **Forward Reference** (référence anticipée).

En Python, au moment où l'interpréteur lit la définition de la classe `Person`, celle-ci **n'est pas encore complètement connue**. Autrement dit, lorsqu'on définit les méthodes à l'intérieur de la classe, le nom `Person` n'existe pas encore comme type utilisable pour l'annotation. Si on écrivait simplement :

```
def __add__(self, other: Person) -> int:
```

Python lèverait une erreur, car il ne reconnaîtrait pas encore le symbole `Person` au moment où il lit la signature de la méthode.

Pour contourner ce problème, on place le nom du type entre guillemets :

```
'Person'
```

De cette façon, Python ne cherche pas immédiatement à évaluer ce nom comme une variable existante. Il le garde sous forme de chaîne de caractères jusqu'à ce que la classe soit entièrement créée. L'interpréteur saura ensuite interpréter correctement cette annotation lorsqu'elle sera utilisée par les outils de typage (comme **mypy**, **VS Code**, ou **Pyright**).

En résumé, on met `'Person'` entre quotes pour indiquer à Python :

> "Ce type n'existe pas encore au moment où tu lis ce code, mais il existera une fois la classe entièrement définie."

Depuis Python 3.7, on peut aussi activer ce comportement automatiquement pour tout le fichier en écrivant tout en haut du script :

```
from __future__ import annotations
```

Avec cette instruction, il n'est plus nécessaire de mettre des guillemets : Python retardera automatiquement l'évaluation de toutes les annotations. On pourrait alors écrire :

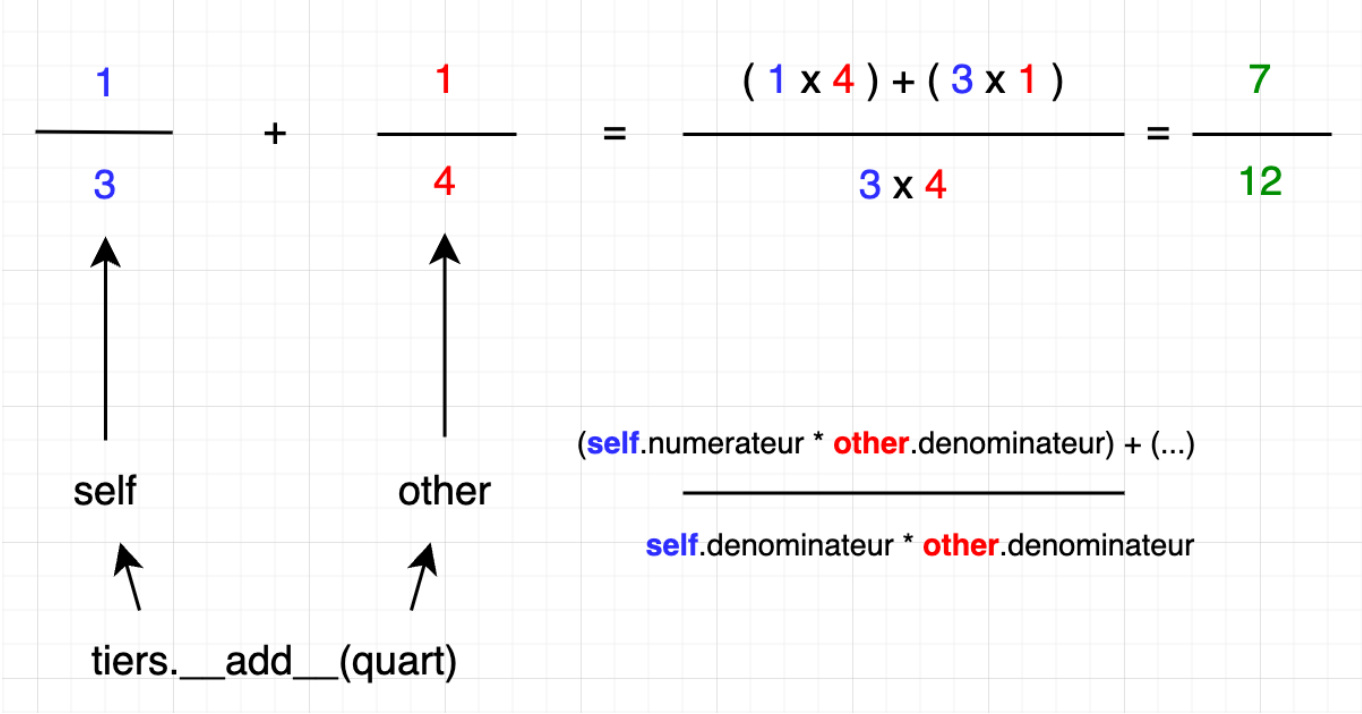
```
def __add__(self, other: Person) -> int:
    return self.age + other.age
```

Mais dans une optique pédagogique, utiliser `'Person'` permet de montrer explicitement ce mécanisme de **référence anticipée**, essentiel pour comprendre comment Python lit et interprète les annotations de type.

Exercice

Reprenez la classe **`Fraction`** et implémentez la méthode `__add__` pour la classe `Fraction`.

Petit aide:



Attention : une fraction **PLUS** une autre fraction retourne une **NOUVELLE** fraction.

Correction

```
class Fraction:
    def __init__(self, numerateur: int, denominateur: int) -> None:
        self.numerateur = numerateur
        self.denominateur = denominateur
        self.simplifier()

    def __repr__(self) -> str:
        return f"Fraction(numerateur={self.numerateur!r}, denominateur=
{self.denominateur!r})"

    def __add__(self, other: 'Fraction') -> 'Fraction':
        new_num = (self.numerateur * other.denominateur) +
(self.denominateur * other.numerateur)
        new_den = self.denominateur * other.denominateur
        return Fraction(new_num, new_den)

    def simplifier(self) -> None:
        pgcd = Fraction.pgcd(self.numerateur, self.denominateur)
        self.numerateur = self.numerateur // pgcd
        self.denominateur = self.denominateur // pgcd

    @staticmethod
    def pgcd(a: int, b: int) -> int:
        if b == 0:
            return a
        return Fraction.pgcd(b, a % b)
```

`__eq__` ou l'égalité

Reprenons l'exercice sur la gestion de tâches que vous avez fait, et de la liste de tâches, où chaque élément est une instance de la classe `Tache`. On souhaite vérifier qu'une tâche n'existe pas déjà dans la liste avant de l'ajouter. Intuitivement, on pourrait écrire :

```
if tache in liste_taches:
    print("Cette tâche existe déjà")
```

Mais en réalité, cette vérification ne se comporte pas comme on pourrait le croire.

Lorsqu'on écrit `if tache in liste_taches:`, Python parcourt chaque élément de la liste et vérifie un par un avec `==`. Autrement dit, il exécute successivement :

```
tache == liste_taches[0]
tache == liste_taches[1]
tache == liste_taches[2]
...
```

et ainsi de suite, jusqu'à trouver une égalité vraie ou atteindre la fin de la liste.

Si deux objets `Tache` possèdent exactement les mêmes attributs, par exemple le même titre et la même date limite, Python les considère quand même comme **différents**, sauf si on lui apprend comment les comparer.

Cela s'explique par le comportement par défaut de Python : lorsqu'aucune méthode spéciale de comparaison n'est définie, il ne compare pas les valeurs internes, mais **les identités mémoire** des objets, autrement dit le résultat de la fonction `id()`.

On peut le constater avec un exemple simple :

```
class Tache:
    def __init__(self, titre, description, date_limite):
        self.titre = titre
        self.description = description
        self.date_limite = date_limite

t1 = Tache("Ranger le bureau", "Faire un peu d'ordre")
t2 = Tache("Ranger le bureau", "Faire un peu d'ordre")

print(t1 == t2)  # False
print(id(t1), id(t2))  # 14050592, 14050675
```

Bien que `t1` et `t2` aient exactement le même contenu, Python renvoie `False`, car ce sont deux objets distincts en mémoire. `id(t1)` et `id(t2)` montrent deux adresses différentes : pour lui, ils ne sont pas

“égaux”, même si leurs valeurs le sont.

C’est ici qu’intervient la méthode magique `__eq__`. Cette méthode est invoquée automatiquement lorsque l’on utilise l’opérateur `==`. Elle permet de définir **le critère d’égalité logique** entre deux instances.

On peut alors enseigner à Python ce que signifie “deux tâches identiques” dans notre contexte. Par exemple, on peut décider qu’une tâche est identique à une autre si son **titre** et sa **date limite** sont les mêmes :

```
class Tache:
    def __init__(self, titre: str, description: str, date_limite: str) ->
None:
        self.titre = titre
        self.description = description
        self.date_limite = date_limite

    def __eq__(self, other: 'Tache') -> bool:
        if not isinstance(other, Tache):
            return False
        return self.titre == other.titre and self.date_limite ==
other.date_limite
```

Désormais, la comparaison se base **sur les valeurs internes**, pas sur les adresses mémoire :

```
t1 = Tache("Ranger le bureau", "Faire un peu d’ordre", "2025-10-19")
t2 = Tache("Ranger le bureau", "Faire un peu d’ordre", "2025-10-19")

print(t1 == t2)  # True
```

Et, conséquence directe, la condition suivante fonctionnera comme prévu :

```
liste_taches = [t1]

if t2 in liste_taches:
    print("Cette tâche existe déjà")
```

Grâce à notre méthode `__eq__`, cette vérification devient pertinente : Python sait désormais que deux objets sont considérés égaux si leurs valeurs internes correspondent, et non plus s’ils partagent la même adresse mémoire.

Sans cette méthode, la vérification d’existence d’une tâche dans la liste ne fonctionnerait jamais correctement, car Python comparerait uniquement les identités des objets, pas leur contenu réel.

`__mul__` et `__rmul__`

Revenons à notre exemple de fraction. On souhaite pouvoir multiplier une **fraction** par un **entier**, par exemple :

Si l'on exécute :

```
f = Fraction(3, 4)
print(f * 2)
```

Python renvoie une erreur :

```
TypeError: unsupported operand type(s) for *: 'Fraction' and 'int'
```

L'interpréteur ne sait pas comment multiplier une `Fraction` avec un `int`. Vous l'avez deviner, c'est ici qu'intervient la méthode magique `__mul__`.

La méthode `__mul__` est appelée automatiquement lorsque Python rencontre l'opérateur `*` avec l'objet placé **à gauche** de l'opération. Ainsi, `f * 2` déclenche `f.__mul__(2)`.

On implémente cette méthode dans la classe :

```
class Fraction:
    def __init__(self, numerateur: int, denominateur: int) -> None:
        self.numerateur = numerateur
        self.denominateur = denominateur

    def __repr__(self) -> str:
        return f"{self.numerateur}/{self.denominateur}"

    def __mul__(self, other: int) -> "Fraction":
        if isinstance(other, int):
            return Fraction(self.numerateur * other, self.denominateur)
        raise TypeError("Multiplication possible uniquement avec un
entier")
```

Désormais, la multiplication `f * 2` fonctionne :

```
f = Fraction(3, 4)
print(f * 2)  # 6/4
```

Python appelle `f.__mul__(2)` et crée une nouvelle fraction dont le numérateur est multiplié par 2.

Mais que se passe-t-il si l'on inverse les opérandes ?

```
print(2 * f)
```

Cette fois, Python renvoie à nouveau :

```
TypeError: unsupported operand type(s) for *: 'int' and 'Fraction'
```

La raison est simple : dans `2 * f`, l'objet de gauche est un entier (`int`).

Python commence donc par appeler `int.__mul__(f)`.

Or, la classe `int` n'a aucune idée de ce qu'est un `Fraction`. Quand l'opération échoue, Python **n'essaie pas automatiquement** l'opération inverse, à moins qu'on lui dise comment faire.

C'est le rôle de la méthode magique `__rmul__`.

`__rmul__` (**Right Multiply**) est invoquée lorsque l'objet de gauche ne sait pas gérer la multiplication. Autrement dit, si `2 * f` échoue, Python appellera `f.__rmul__(2)`.

On l'ajoute donc à la classe :

```
class Fraction:
    def __init__(self, numerateur: int, denominateur: int) -> None:
        self.numerateur = numerateur
        self.denominateur = denominateur

    def __repr__(self) -> str:
        return f"{self.numerateur}/{self.denominateur}"

    def __mul__(self, other: int) -> "Fraction":
        if isinstance(other, int):
            return Fraction(self.numerateur * other, self.denominateur)
        raise TypeError("Multiplication possible uniquement avec un
entier")

    def __rmul__(self, other: int) -> "Fraction":
        return self.__mul__(other)
```

Et tous les méthodes magiques **Rights** sont implémentées en utilisant les méthodes magiques **Lefts**.

Ainsi :

```
f = Fraction(3, 4)
print(f * 2)  # 6/4
print(2 * f)  # 6/4
```

Les deux expressions produisent le même résultat. Dans le second cas (`2 * f`), Python essaie d'abord `int.__mul__(f)` (qui échoue), puis se rabat sur `Fraction.__rmul__`.

Donc

1. `f * 2` → Python appelle `f.__mul__(2)`
2. `2 * f` → Python tente `int.__mul__(f)`, échoue, puis appelle `f.__rmul__(2)`

Ces deux méthodes magiques permettent donc de rendre la classe `Fraction` **symétrique** dans son comportement face à la multiplication, quelle que soit la position de la fraction dans l'opération.

C'est un principe que l'on retrouve dans beaucoup de classes numériques personnalisées : on implémente `__mul__` et `__rmul__` ensemble pour que la multiplication soit cohérente dans les deux sens.

Conclusion

Les méthodes magiques, ou *dunder methods*, incarnent l'un des aspects les plus élégants du langage Python : sa capacité à rendre la logique métier **aussi naturelle que le langage lui-même**.

Elles permettent d'intégrer ses propres classes au cœur du fonctionnement du langage, sans jamais sacrifier la lisibilité. Grâce à elles, des opérations comme `alice + john`, `tache in liste_taches`, ou `2 * f` cessent d'être de simples instructions : elles deviennent des expressions de sens, où le code raconte ce qu'il fait.

Ce pouvoir réside dans l'idée que l'on n'invente pas un nouveau vocabulaire pour chaque classe, mais qu'on enseigne à Python comment parler le nôtre. On ne dit plus "appelle une méthode pour additionner deux fractions" ; on écrit simplement `f1 + f2`. On ne crée pas une fonction `comparer_taches(t1, t2)` ; on laisse l'opérateur `==` traduire l'intention.

Cette explicite immédiate rend le code plus lisible, plus fluide, et plus proche de la pensée humaine. On lit du Python comme on lit une phrase. C'est là toute la beauté de ces méthodes spéciales : elles transforment nos objets en citoyens à part entière du langage, capables de dialoguer naturellement avec ses structures, ses opérateurs et ses conventions, tout en préservant cette philosophie fondamentale qui guide Python depuis ses débuts : **la clarté prime toujours sur la complexité cachée.**