

Méthodes statiques et Méthodes de classe

En Python, les **méthodes statiques** et les **méthodes de classe** constituent deux formes particulières de méthodes que l'on rattache à une classe plutôt qu'à une instance. Elles participent toutes deux à la structuration et à la modularité du code orienté objet, mais leur rôle et leur mode de fonctionnement diffèrent selon le contexte d'utilisation.

Une **méthode statique** est une méthode définie à l'intérieur d'une classe, mais qui n'interagit ni avec les attributs d'instance (**self**), ni avec les attributs de classe (**cls**). Elle se comporte comme une fonction autonome rangée dans la classe pour des raisons de cohérence conceptuelle.

On l'emploie lorsqu'une opération logique est liée à la classe sur le plan sémantique, sans dépendre de son état interne. Le décorateur **@staticmethod** indique explicitement à l'interpréteur que la méthode ne reçoit aucun paramètre implicite, et qu'elle doit être appelée comme une simple fonction, soit depuis la classe, soit depuis une instance.

À l'inverse, une **méthode de classe**, définie avec le décorateur **@classmethod**, reçoit automatiquement en premier paramètre la classe elle-même (généralement nommée **cls**) au lieu de l'instance. Elle est utile lorsqu'on souhaite manipuler ou modifier des données partagées par toutes les instances, ou encore lorsqu'on veut proposer plusieurs manières de construire des objets à partir d'une même définition de classe. Autrement dit, là où la méthode statique représente une fonction indépendante logiquement rattachée à la classe, la méthode de classe incarne un comportement collectif propre à la classe entière.

L'intérêt combiné de ces deux mécanismes réside dans la **clarté organisationnelle** qu'ils apportent. On peut ainsi regrouper, au sein d'une même classe, des opérations d'ordres différents : celles qui concernent les objets particuliers (méthodes d'instance), celles qui concernent la classe dans son ensemble (méthodes de classe) et celles qui sont purement utilitaires (méthodes statiques). Cette hiérarchie de comportements rend le code plus expressif : chaque méthode indique, par sa nature même, la portée de son action et le niveau de données qu'elle manipule.

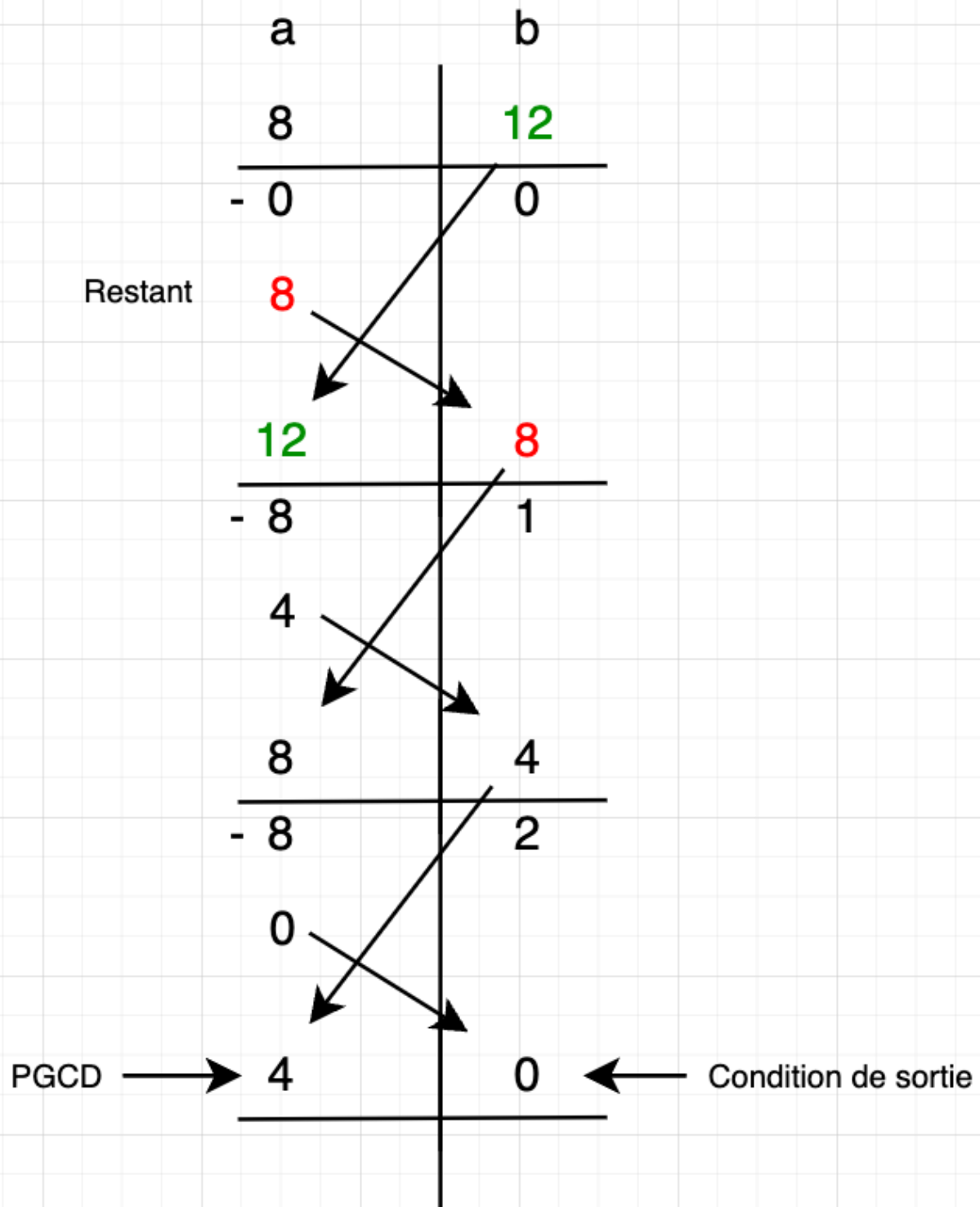
Exemple d'utilisation de méthodes statiques

On continue avec notre classe **Fraction**. Que peut-on faire avec une fraction ? **On peut la simplifier !**

Simplifier une fraction revient à la réduire à une forme équivalente où le numérateur et le dénominateur n'ont plus de facteur commun autre que 1. Par exemple, la fraction **8/12** peut être simplifiée en **2/3**, car les deux nombres sont divisibles par 4.

Le nombre 4 est ici le **plus grand commun diviseur (PGCD)** de 8 et 12.

Le calcul du PGCD repose sur un principe mathématique simple appelé **algorithme d'Euclide**. Cet algorithme consiste à diviser le plus grand des deux nombres par le plus petit, puis à recommencer le calcul avec le diviseur et le reste obtenu. L'opération se répète jusqu'à ce que le reste soit nul. À ce moment-là, le dernier diviseur non nul est le PGCD.



Cette méthode s'exprime naturellement à l'aide d'une **fonction récursive**.

Une fonction récursive est une fonction qui s'appelle elle-même pour résoudre un sous-problème plus simple du même type. C'est un modèle particulièrement adapté aux calculs répétitifs comme ici, mais il faut toujours veiller à définir une **condition d'arrêt** (ou **condition de sortie**), c'est-à-dire une situation dans laquelle la fonction cesse de s'appeler elle-même. Sans cela, la récursion serait infinie et provoquerait une erreur d'exécution.

Dans le cas du calcul du PGCD, la condition d'arrêt est atteinte lorsque le second nombre (b) devient nul : à ce moment-là, le premier (a) est le résultat final.

```
class Fraction:
    def __init__(self, numerateur, denominateur):
        self.numerateur = numerateur
        self.denominateur = denominateur

    def quotient(self):
        return self.numerateur / self.denominateur

    def pgcd(self, a, b) :
        if b == 0:
            return a
        return self.pgcd(b, a % b)
```

On remarque que la méthode s'appelle elle-même tant que le reste `a % b` n'est pas nul. Dès que `b` devient égal à zéro, la fonction renvoie `a`, ce qui constitue la condition de sortie de la récursion.

Ce calcul servira ensuite à simplifier la fraction en divisant le numérateur et le dénominateur par leur PGCD commun.

```
class Fraction:
    def __init__(self, numerateur, denominateur):
        self.numerateur = numerateur
        self.denominateur = denominateur

    def quotient(self):
        return self.numerateur / self.denominateur

    def simplifier(self):
        pgcd = self.pgcd(self.numerateur, self.denominateur)
        self.numerateur = self.numerateur // pgcd
        self.denominateur = self.denominateur // pgcd

    def pgcd(self, a, b) :
        if b == 0:
            return a
        return self.pgcd(b, a % b)
```

`//` est l'opérateur de division entière. En effet, ce sont des fractions, donc on ne garde que la partie entière.

Ainsi on peut écrire :

```
a = Fraction(8, 12)
a.simplifier()
print(a.__dict__)
# {'numerateur': 2, 'denominateur': 3}
```

On peut aller plus loin, en effet, en mathématiques, les calculs sont identiques pour des fractions simplifiées et non simplifiées.

Donc au moment de construire notre objet fraction, on peut la simplifier directement.

```
class Fraction:
    def __init__(self, numerateur, denominateur):
        self.numerateur = numerateur
        self.denominateur = denominateur
        self.simplifier() # ici

    ...
```

```
a = Fraction(8, 12) # La simplification est effectuée à l'instanciation
print(a.__dict__)
# {'numerateur': 2, 'denominateur': 3}
```

Posons le problème suivant : J'ai besoin de connaître le pgcd de 128 et 212 par exemple.

Pour cela je vais faire:

```
tiers = Fraction(128, 212)
print(tiers.pgcd(128, 212)) # 4
```

On voit ici que je crée une fraction, puis je fais appel à la méthode `pgcd` de la fraction. **Je crée un objet dont je n'aurais aucune utilité.**

L'idéal serait de pouvoir faire appel à la méthode `pgcd` de la classe fraction directement, sans avoir à instancier une fraction.

C'est là que les **méthodes statiques** entrent en jeu.

Les méthodes statiques sont des méthodes qui sont définies sur une classe, et qui peuvent être appelées sans avoir à créer une instance de cette classe.

Pour cela, on va transformer `pgcd` en **méthode statique** en utilisant le mot-clé `@staticmethod`.

En Python, un décorateur est une annotation placée au-dessus d'une fonction ou d'une méthode qui en modifie la façon dont elle est enregistrée ou appelée. On y reviendra en détail plus tard.

Ici, le décorateur `@staticmethod` indique que la méthode ne reçoit ni `self` (instance) ni `cls` (classe) et qu'elle se comporte comme une fonction autonome rangée dans la classe pour des raisons de cohérence conceptuelle.

```
class Fraction:
    def __init__(self, numerateur: int, denominateur: int) :
```

```

        self.numerateur = numerateur
        self.denominateur = denominateur
        self.simplifier()

    def quotient(self) -> float:
        return self.numerateur / self.denominateur

    def simplifier(self) :
        pgcd = Fraction.pgcd(self.numerateur, self.denominateur) #
attention ici
        self.numerateur = self.numerateur // pgcd
        self.denominateur = self.denominateur // pgcd

    @staticmethod # ici
    def pgcd(a: int, b: int) :
        if b == 0:
            return abs(a)
        return Fraction.pgcd(b, a % b) # attention ici

```

On a déplacé l'appel dans `simplifier` vers `Fraction.pgcd(...)` pour souligner que le calcul ne dépend pas de l'état interne de l'objet. On peut désormais invoquer le PGCD directement sur la classe sans instancier de fraction, ce qui répond exactement au besoin énoncé.

```
Fraction.pgcd(128, 212) # 4
```

On conserve ainsi un code plus clair et mieux organisé : la simplification utilise une fonction mathématique pure, rangée à l'endroit logique où on s'attend à la trouver, tout en évitant la création d'objets inutiles.

Exemple d'utilisation de méthodes de classe

On souhaite créer des fractions **à partir d'une chaîne de caractères** comme `"128/212"`. Intuitivement, on écrit une méthode qui découpe la chaîne, convertit les nombres, puis renvoie une nouvelle `Fraction` déjà simplifiée.

```

class Fraction:
    def __init__(self, numerateur: int, denominateur: int) :
        self.numerateur = numerateur
        self.denominateur = denominateur
        self.simplifier()

    ...

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

```

```
def creer_depuis_chaine(self, s: str):
    numerateur_str, denominateur_str = s.split('/')
    return Fraction(int(numerateur_str), int(denominateur_str))

# Problème : on doit instancier "pour rien"
tmp = Fraction(1, 1)
fraction = tmp.creer_depuis_chaine("128/212")
```

Ici, on définit la méthode sur l'instance. Elle renvoie bien une **nouvelle** fraction... mais elle oblige à créer une **fraction temporaire** juste pour appeler la méthode, ce qui est inutile et confus : on n'a pas besoin de **self**.

On voit le même problème qu'avec **pgcd** avant sa transformation : on crée un objet sans utilité pour accéder à une opération qui ne dépend pas d'un état d'instance.

L'instruction

```
numerateur_str, denominateur_str = s.split('/')
```

illustre ce qu'on appelle le **déballage de données** (ou *destructuring assignment* en anglais).

Lorsqu'on écrit **s.split('/')**, Python renvoie une **liste** contenant deux éléments : la partie avant le / et la partie après.

Par exemple :

```
s = "128/212"
print(s.split('/'))
# ['128', '212']
```

En écrivant

```
numerateur_str, denominateur_str = s.split('/')
```

on demande à Python de **décomposer** cette liste directement en deux variables. Le premier élément ('128') est affecté à **numerateur_str**, et le second ('212') à **denominateur_str**.

C'est une écriture très fréquente et pratique en Python : elle permet d'extraire plusieurs valeurs d'un seul coup, sans passer par des index comme **parties[0]** ou **parties[1]**.

Une **méthode de classe** (décorée avec **@classmethod**) reçoit la **classe** en premier paramètre implicite, conventionnellement nommé **cls**. Elle convient parfaitement aux **constructeurs alternatifs** : on fabrique une instance à partir d'un autre format de données, sans créer d'objet temporaire.

```
class Fraction:
    def __init__(self, numerateur: int, denominateur: int) :
```

```

        self.numerateur = numerateur
        self.denominateur = denominateur
        self.simplifier()

    ...

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

    @classmethod
    def depuis_chaine(cls, s: str):
        numerateur_str, denominateur_str = s.split('/')
        return cls(int(numerateur_str), int(denominateur_str))

```

Désormais, on crée directement la fraction, de manière claire et expressive, **au niveau de la classe** :

```
f = Fraction.depuis_chaine("128/212")
```

- On n'a plus d'objet temporaire inutile : la construction est **directe** et **lisible**.
- La méthode est sémantiquement à la bonne place : c'est un **constructeur alternatif** qui retourne une instance de `cls`.
- La solution reste **ouverte à l'extension** : on peut ajouter d'autres constructeurs alternatifs cohérents, par exemple `depuis_float`, `depuis_tuple`, ou des « raccourcis » utiles :

```

@classmethod
def nulle(cls) -> 'Fraction':
    return cls(0, 1)

@classmethod
def unitaire(cls) -> 'Fraction':
    return cls(1, 1)

```

Comme on l'a fait pour `pgcd` en le convertissant en **méthode statique** parce qu'il ne dépendait pas d'un état d'objet, on convertit ici la création « à partir d'une chaîne » en **méthode de classe** parce qu'elle **construit** justement une **nouvelle instance** de la classe. On choisit ainsi le bon « niveau » d'appartenance de la méthode en fonction de ce qu'elle manipule : l'instance, la classe, ou aucun des deux.