La Programmation Orientée Objet

Introduction

Dans les débuts de l'informatique, les programmes étaient écrits de manière linéaire : une suite d'instructions exécutées les unes après les autres. Cette approche fonctionnait bien pour des programmes courts, mais à mesure que les logiciels sont devenus plus gros et plus complexes, elle a montré ses limites. Les développeurs avaient de plus en plus de mal à organiser leur code, à le maintenir et à le faire évoluer sans tout casser.

C'est dans ce contexte, vers les années 1960, que la programmation orientée objet est apparue. L'idée était de mieux représenter les éléments d'un programme en s'inspirant du monde réel. Plutôt que d'écrire des suites d'instructions, on a commencé à créer des objets : **de petites unités** regroupant à la fois des données (leurs **caractéristiques**) et des comportements (leurs **actions**).

Cette nouvelle façon de penser le code a permis de résoudre plusieurs problèmes importants :

- le code est plus facile à comprendre, car chaque objet représente une idée concrète ;
- il devient plus simple de réutiliser certaines parties du programme ;
- la maintenance et les évolutions sont plus faciles, car on peut modifier un objet sans casser le reste.

Le concept d'objet est apparu pour la première fois avec le langage **Simula**, développé entre 1962 et 1967 par **Ole-Johan Dahl** et **Kristen Nygaard** à l'Institut norvégien de calcul d'Oslo. Ce langage avait été imaginé à l'origine pour faciliter la simulation de systèmes complexes, comme ceux qu'on trouve dans l'industrie ou la recherche scientifique.

Simula introduisait deux idées fondamentales : la **classe** et l'**objet**. Grâce à ces notions, il devenait possible de regrouper au même endroit les **données** (ce qu'un objet possède) et les **comportements** (ce qu'il sait faire). Cette approche a profondément changé la manière de concevoir les programmes, car elle permettait de créer un lien plus clair entre le monde réel et sa représentation informatique.

Les objets pouvaient ainsi contenir leur propre état interne et les opérations capables de le modifier. Chaque objet devenait une unité indépendante et cohérente, capable de collaborer avec d'autres sans que leurs fonctionnements internes soient mélangés.

Dans les années 1970, les idées introduites par Simula se sont largement diffusées grâce à l'apparition de **Smalltalk**, un langage développé au centre de recherche **Xerox PARC** par **Alan Kay**, **Dan Ingalls** et **Adele Goldberg**. Smalltalk fut le premier langage à appliquer de manière complète la programmation orientée objet : tout y était considéré comme **un objet** capable d'échanger des messages avec d'autres objets.

Cette approche, à la fois radicale et élégante, montrait qu'un programme pouvait être conçu entièrement autour d'interactions entre objets. Elle posait les bases d'un modèle clair et unifié qui allait influencer durablement la conception des langages de programmation.

Smalltalk donna sa forme la plus aboutie aux notions d'**héritage**, de **polymorphisme** et d'**encapsulation**, trois piliers de la programmation orientée objet. Ces principes permirent d'écrire des programmes plus souples, mieux structurés et plus proches de la logique du monde réel.

La **programmation orientée objet** répondait à plusieurs besoins essentiels du développement logiciel moderne. Elle offrait avant tout une façon efficace de gérer la complexité en découpant les programmes en **petites entités autonomes**, plus faciles à comprendre et à modifier. Cette organisation rendait le code plus clair et plus simple à maintenir au fil du temps.

Grâce à **l'encapsulation**, chaque objet pouvait protéger ses données internes en ne montrant à l'extérieur que ce qui était nécessaire. Les autres parties du programme interagissaient ainsi avec lui uniquement par des **interfaces claires**, sans dépendre de son fonctionnement interne.

Le **principe d'héritage** permettait, quant à lui, de **réutiliser le code** déjà existant. Une nouvelle classe pouvait hériter des caractéristiques d'une autre et les compléter sans tout réécrire, ce qui évitait la duplication et assurait une meilleure cohérence dans l'ensemble du programme.

Enfin, le **polymorphisme** apportait une grande **souplesse** : il devenait possible de manipuler différents types d'objets de la même manière, tant qu'ils partageaient un même comportement. Cela rendait les programmes plus flexibles, plus évolutifs et plus faciles à adapter aux changements futurs.

L'adoption progressive de la programmation orientée objet dans l'industrie s'explique aussi par la façon dont elle facilite le **travail en équipe**. En définissant des **interfaces précises** et en séparant clairement les **responsabilités** de chaque composant, plusieurs développeurs peuvent collaborer sur un même projet sans se gêner mutuellement. Chacun peut se concentrer sur une partie spécifique du système, en ayant la garantie que son code interagit correctement avec celui des autres grâce aux contrats établis entre les objets.

Cette **modularité naturelle** est particulièrement précieuse dans les projets de grande taille, où la coordination entre équipes est souvent complexe. En structurant le code en modules indépendants et cohérents, la programmation orientée objet rend le développement plus organisé et plus prévisible.

De plus, la **correspondance intuitive** entre les objets logiciels et les éléments du **monde réel** améliore la compréhension du système par l'ensemble de l'équipe. Les développeurs et les experts métier peuvent ainsi échanger plus facilement, parler le même langage et réduire les risques d'erreurs dans la définition ou la mise en œuvre des besoins.

Python a été pensé comme un langage **pragmatique** et **multi-paradigme**, c'est-à-dire capable de s'adapter à plusieurs styles de programmation. Dès ses premières versions, il a intégré naturellement la **programmation orientée objet**, sans en faire une obligation stricte.

Contrairement à des langages comme **Java** ou **Smalltalk**, qui imposent une approche entièrement fondée sur les objets, Python a conservé une **grande liberté** : on peut y écrire du code **procédural**, **fonctionnel** ou **orienté objet**, selon ce qui convient le mieux au problème à résoudre. Cette flexibilité permet d'adopter une approche simple pour les petits scripts comme une architecture plus élaborée pour les projets complexes.

Ce choix illustre la **maturité** du paradigme objet à l'époque de la création de Python. Il n'était plus considéré comme la solution unique à tous les problèmes, mais comme un **outil puissant** à utiliser de manière réfléchie, en fonction du contexte et des besoins du projet.

Vocabulaire

Dans le monde de la programmation orientée objet, le vocabulaire joue un rôle essentiel pour bien comprendre comment les programmes sont conçus et organisés. Chaque terme technique a un sens précis

et correspond à une fonction particulière dans la structure d'un logiciel. Maîtriser ce langage permet de mieux appréhender la logique qui relie les différents éléments d'un programme et de comprendre comment ils interagissent entre eux.

Une **classe** peut être vue comme un **modèle** ou un **plan de fabrication**. Elle décrit la structure générale que suivront tous les objets créés à partir d'elle. C'est dans la classe qu'on définit les **données** (appelées **attributs**) que les objets contiendront, ainsi que les **actions** (appelées **méthodes**) qu'ils pourront accomplir. En somme, la classe sert de base commune : elle définit ce qu'un objet sait et ce qu'il peut faire, avant même que celui-ci n'existe réellement dans le programme.

La classe `str` correspond au type d'objet utilisé pour manipuler les chaînes de caractères en Python.

Lorsqu'on écrit :

```
prenom = "Alice"  # on a instancié un objet de type 'str' et on l'a
initialisé avec la valeur 'Alice'
```

Deux étapes se produisent en même temps : **l'instanciation** et **l'initialisation**.

L'**instanciation** désigne le moment où un **objet** est créé à partir d'une **classe**. C'est à cet instant que Python réserve un espace en mémoire pour ce nouvel objet et lui donne une existence concrète dans le programme. L'objet devient alors une instance de la classe `str`.

L'**initialisation**, elle, correspond à la phase où cet objet reçoit ses premières valeurs. Ici, la chaîne de caractères `"Alice"` devient le contenu initial de l'objet. On dit que c'est son **état de départ**, celui qu'il possède dès sa création.

Dans l'exemple suivant, on fait apparaître explicitement la création de l'objet :

```
nom = str('Smith')  # on a instancié un objet de type 'str' et on l'a
initialisé avec la valeur 'Smith'
```

Ici, l'appel à `str()` fait directement intervenir le **constructeur** de la classe, c'est-à-dire la fonction spéciale qui sert à créer de nouveaux objets de ce type. En lui passant la valeur `'Smith'`, on demande à Python de fabriquer une nouvelle **instance** de `str` et de l'initialiser immédiatement avec cette donnée.

Le résultat obtenu est le même que dans la forme abrégée avec les guillemets, mais cette version montre de manière plus **explicite** les étapes internes : on appelle une classe (`str`) comme une fonction, et elle renvoie un nouvel objet de son propre type.

Le mot **instance** est souvent utilisé comme synonyme d'**objet**, mais il insiste sur le lien hiérarchique entre un objet concret et la **classe** dont il provient. Ici, la variable `nom` fait donc référence à **une instance de la classe** `str`, c'est-à-dire un objet particulier représentant la chaîne de caractères `'Smith'`.

Le troisième exemple met en évidence la différence entre **instanciation** et **initialisation** :

```
adresse = str()  # on a instancié un objet de type 'str'
```

Ici, un objet de type `str` est bel et bien créé : Python réserve de la mémoire pour cette nouvelle chaîne de caractères. Cependant, aucune valeur n'est fournie au moment de sa création. L'objet existe donc réellement, il possède une adresse en mémoire et peut être utilisé dans le programme, mais son contenu textuel est simplement vide.

Cette distinction entre la **création de l'objet** (instanciation) et la **mise en place de ses valeurs initiales** (initialisation) est essentielle en programmation orientée objet. Elle permet de mieux comprendre comment un langage comme Python gère le **cycle de vie d'un objet**, depuis sa naissance jusqu'à sa configuration et son utilisation.

Dans ce contexte, le terme **type** est équivalent à la notion de **classe**. Dire que `prenom` est de type `str` revient à dire que `prenom` **fait référence à une instance** de la classe `str`. Cette correspondance illustre le système de types unifié de Python : qu'il s'agisse d'un type dit "primitif" comme `int` ou `str`, ou d'une classe que l'on définit soi-même, tout repose sur le même mécanisme orienté objet. Chaque objet possède ainsi un type qui détermine **les opérations qu'il accepte** et **la manière dont il réagit** lorsqu'on interagit avec lui.

Notre propre type

En Python, jusqu'à présent, on a manipulé des **types intégrés** comme les entiers (`int`), les chaînes de caractères (`str`) ou les listes (`list`). Mais le langage permet aussi de **créer ses propres types**, grâce au mot-clé `class`. Cette possibilité constitue l'un des fondements de la **programmation orientée objet**, car elle permet de représenter des concepts du monde réel ou des structures logiques qui n'existent pas directement dans le langage.

```
class Fraction:  # je viens de créer un nouveau type
    pass
```

En écrivant `class Fraction:`, on déclare un **nouveau type** appelé **Fraction**. Cela indique à Python que l'on souhaite définir une **nouvelle catégorie d'objets**, avec ses propres **données** et **comportements**. Le deux-points (`:`) marque le début du bloc de définition, comme pour les fonctions, les boucles ou les conditions.

À l'intérieur, on trouve pour l'instant l'instruction `pass`. Cette instruction spéciale ne fait rien, mais elle sert de **bouchon temporaire** : Python exige qu'un bloc ne soit jamais vide, et `pass` permet justement de laisser un espace réservé pour du code à venir.

En ce qui concerne la **convention d'écriture**, les noms de classes commencent toujours par une **majuscule**. Ce n'est pas une obligation stricte, mais c'est une règle de style universellement suivie en Python. Elle permet d'identifier immédiatement une classe au premier coup d'œil : lorsqu'on lit `Fraction`, on comprend qu'il s'agit d'un **type**, et non d'une variable ou d'une fonction (qui, elles, commencent par une minuscule). Cette cohérence visuelle rend le code plus clair et plus facile à maintenir.

Pour l'instant, la classe `Fraction` existe bien dans le programme, mais elle ne contient encore **aucune donnée ni comportement**. On peut toutefois déjà créer des objets à partir d'elle, par exemple :

```
ma_fraction = Fraction()
```

Cet appel crée une **instance vide** de la classe `Fraction`. L'objet existe en mémoire, mais il ne possède encore ni attributs ni méthodes, donc aucun rôle concret. Les prochaines étapes consisteront à **ajouter des attributs** pour stocker des informations (comme le numérateur et le dénominateur) et des **méthodes** pour définir ce que cet objet sait faire (comme additionner deux fractions ou les afficher).

Les Attributs

Dans la version suivante, la classe `Fraction` contient désormais deux **attributs de classe** :

```
class Fraction:
    numerateur = 1
    denominateur = 1
```

Ces deux variables, `numerateur` et `denominateur`, appartiennent directement à la **classe** elle-même, et non à un objet particulier. Cela signifie qu'elles sont partagées par **toutes les instances** créées à partir de cette classe.

Lorsqu'on écrit :

```
tiers = Fraction()  # On instancie un objet de type 'Fraction'
```

on crée une **instance** concrète de la classe, c'est-à-dire un objet en mémoire construit d'après le "plan" défini par la classe. La classe représente donc le modèle abstrait, tandis que l'instance correspond à un exemplaire réel. Les **parenthèses** après `Fraction` déclenchent le processus de création de l'objet. La variable `tiers` référence désormais cet objet en mémoire, de type `Fraction`.

On peut ensuite accéder aux attributs de l'objet grâce à la **notation pointée** :

```
print(tiers.numerateur)  # affiche 1
print(tiers.denominateur)  # affiche 1
```

Ces deux instructions montrent que l'objet `tiers` **hérite** automatiquement des attributs définis au niveau de la classe. La notation avec le point (`.`) est la manière standard en Python pour accéder aux données ou aux comportements d'un objet.

Si l'on crée plusieurs objets à partir de la même classe, ils partageront tous ces mêmes valeurs :

```
tiers = Fraction()
quart = Fraction()
```

```
print(tiers.numerateur)  # 1
print(quart.numerateur)  # 1
```

Dans cet état, les deux objets `tiers` et `quart` possèdent les mêmes valeurs pour `numerateur` et `denominateur`, car ils se réfèrent tous deux aux **mêmes attributs de classe**.

Ce comportement devient évident lorsqu'on modifie la valeur d'un attribut au niveau de la classe :

```
Fraction.numerateur = 5
print(tiers.numerateur)  # 5
print(quart.numerateur)  # 5
```

Tous les objets voient alors la nouvelle valeur, car l'attribut appartient à la classe et non à chaque instance.

Cette situation montre la **limite des attributs de classe** : ils conviennent lorsque la valeur doit être partagée entre tous les objets (par exemple, une constante ou un compteur global), mais pas lorsque chaque objet doit avoir ses propres données.

Le Mappingproxy

Chaque classe et chaque objet en Python possède un attribut spécial appelé `__dict__`. Cet attribut contient la liste des **noms des attributs** et leurs **valeurs associées**, sous la forme d'un **dictionnaire**.

Si on examine celui de la classe `Fraction` :

```
print(Fraction.__dict__)
```

Python affiche :

```
mappingproxy({'__module__': '__main__',
              'numerateur': 1,
              'denominateur': 1,
              '__dict__': <attribute '__dict__' of 'Fraction' objects>,
              '__weakref__': <attribute '__weakref__' of 'Fraction'
objects>,
              '__doc__': None})
```

Le résultat montre un objet un peu particulier : un `mappingproxy`. Ce n'est pas un vrai dictionnaire, mais une **vue en lecture seule** du dictionnaire interne de la classe. Cela signifie qu'on peut le consulter, mais pas le modifier directement. Ce mécanisme protège la structure interne de la classe contre toute modification involontaire.

Dans ce `mappingproxy`, on retrouve nos deux attributs définis dans la classe, `numerateur` et `denominateur`, chacun associé à la valeur `1`. Les autres éléments affichés (`__module__`, `__weakref__`,

etc.) sont ajoutés automatiquement par Python lors de la création de la classe. Pour l'instant, ils ne sont pas importants à comprendre.

Voyons maintenant ce qu'il en est pour une instance :

```
print(tiers.__dict__ )
# {}
```

Le résultat est un dictionnaire **vide** : `{}`. Cela veut dire que l'objet `tiers` ne possède **aucun attribut qui lui appartient vraiment**. Il existe bien en mémoire, mais toutes ses données viennent encore de la **classe**.

Quand on écrit :

```
tiers.numerateur
```

Python suit un **ordre de recherche précis** :

1. Il regarde d'abord dans le dictionnaire de l'objet (`tiers.__dict__`).
2. Comme celui-ci est vide, il continue sa recherche dans celui de la **classe** (`Fraction.__dict__`).

C'est ainsi que Python trouve la valeur `1` pour `tiers.numerateur`. Ce comportement montre que, tant qu'un objet n'a pas ses propres attributs, il utilise ceux de la classe dont il provient.

Essayons maintenant de **modifier** l'attribut `denominateur` de l'objet `tiers` :

```
tiers.denominateur = 3
```

Cette instruction ne change **pas** la classe, mais modifie directement la **structure interne** de l'instance `tiers`. Python crée un **nouvel attribut d'instance** appelé `denominateur` et lui assigne la valeur `3`.

Le `mappingproxy` de la classe `Fraction`, lui, **reste exactement le même** :

```
print(Fraction.__dict__)
mappingproxy({'__module__': '__main__',
              'numerateur': 1,
              'denominateur': 1,
              '__dict__': <attribute '__dict__' of 'Fraction' objects>,
              '__weakref__': <attribute '__weakref__' of 'Fraction'
objects>,
              '__doc__': None})
```

On voit bien que `denominateur` vaut toujours `1` au niveau de la classe.

C'est un comportement essentiel de Python : si l'assignation modifiait directement la classe, **toutes les autres instances** de `Fraction` auraient vu leur valeur changer également, ce qui rendrait impossible

d'avoir des objets avec des données différentes.

À présent, quand on lit à nouveau `tiers.denominateur`, Python trouve immédiatement la valeur dans le dictionnaire de l'objet, sans avoir besoin de consulter la classe :

```
print(tiers.denominateur)
# 3
```

Et si on regarde le dictionnaire interne de `tiers`, on y voit clairement cette nouvelle donnée :

```
print(tiers.__dict__)
# {'denominateur': 3}
```

Cette situation illustre ce qu'on appelle le **masquage d'attribut**.

L'objet `tiers` possède maintenant son **propre attribut** `denominateur`, qui **masque** celui défini dans la classe. Les deux existent en même temps, mais dans des **espaces de noms différents**. Python donne toujours la priorité à l'attribut de l'instance lorsqu'il en trouve un du même nom.

Si on crée un nouvel objet sans modifier quoi que ce soit :

```
moitie = Fraction()
print(moitie.denominateur)  # 1
```

on obtient bien `1`, car `moitie` n'a pas encore d'attribut `denominateur` dans son propre dictionnaire. Python va donc le chercher directement dans la classe.

Dans la version suivante, la classe `Fraction` contient une méthode appelée `dire_bonjour` :

```
class Fraction:
    numerateur = 1
    denominateur = 1

    def dire_bonjour():
        return "Bonjour tout le monde"

tiers = Fraction()
```

Une **méthode** est simplement une **fonction** que l'on écrit à l'intérieur d'une classe.

La façon de la définir est identique à celle d'une fonction normale : on utilise le mot-clé `def`, suivi du nom de la méthode, des parenthèses, puis du corps de la méthode indenté. Ici, `dire_bonjour()` renvoie simplement le texte `"Bonjour tout le monde"`.

Mais une méthode ne se comporte pas toujours de la même manière selon la façon dont on y accède. Python distingue deux cas :

- l'accès **depuis la classe** elle-même,
- et l'accès **depuis une instance** créée à partir de cette classe.

Regardons le premier cas :

```
print(Fraction.dire_bonjour)
# <function __main__.Fraction.dire_bonjour()>
```

Lorsqu'on passe par la classe, Python nous renvoie une **fonction**. C'est la définition brute de la méthode, telle qu'elle est enregistrée dans la classe. À ce stade, il s'agit juste d'un objet fonction stocké dans la classe, au même titre que `numerateur` et `denominateur`.

Voyons maintenant ce qui se passe lorsqu'on y accède via une instance :

```
print(tiers.dire_bonjour)
# <bound method Fraction.dire_bonjour of <__main__.Fraction object at
0x...>>
```

Le résultat est différent. Cette fois, Python affiche une **méthode liée** (*bound method*). Cela signifie que la fonction a été **automatiquement associée** à l'objet `tiers`. Autrement dit, la même fonction définie dans la classe est maintenant connectée à une instance précise.

Cette différence entre les deux affichages prépare à comprendre un concept fondamental : **lorsqu'une méthode est appelée à partir d'un objet, Python établit un lien spécial entre cette méthode et l'instance concernée.** Ce lien explique certains comportements que l'on découvrira dans la suite.

Regardons d'abord ce qu'il se passe lorsque l'on appelle la méthode depuis la classe :

```
print(Fraction.dire_bonjour())
# 'Bonjour tout le monde'
```

Ici, tout fonctionne normalement. La méthode `dire_bonjour()` est appelée **directement depuis la classe**, donc Python exécute la fonction telle qu'elle a été définie, sans rien ajouter.

Mais si on fait la même chose avec une **instance**, le résultat change :

```
print(tiers.dire_bonjour())
```

Python affiche alors une erreur :

```
TypeError: Fraction.dire_bonjour() takes 0 positional arguments but 1 was
given
```

Traduction: Fraction.dire_bonjour() prend 0 paramètres, mais un lui a été donné.

À première vue, le message semble étrange : on n'a pourtant **rien mis entre les parenthèses**. Alors pourquoi Python dit-il qu'on a "donné un argument" à la fonction ?

On peut reproduire exactement le même type d'erreur avec une fonction ordinaire :

```
def toto():
    return "toto"

toto("titi")
```

Résultat :

```
TypeError: toto() takes 0 positional arguments but 1 was given
```

Ici aussi, la fonction `toto()` ne prend **aucun paramètre**, mais on lui en a **fourni un**.

Alors pourquoi `dire_bonjour()` nous dit qu'on lui a donné un argument !!??

La réponse se cache dans le fonctionnement interne du langage. Lorsqu'une méthode est appelée à travers une instance, ici `tiers`, **Python envoie automatiquement cette instance comme premier argument à la méthode**. Autrement dit, il exécute en réalité quelque chose comme :

```
tiers.dire_bonjour(tiers)
```

Cette étape est **automatique** et **invisible à l'œil nu**.

C'est pour cela que l'erreur indique qu'un argument a été transmis alors que vous n'en avez pas fourni explicitement. La méthode `dire_bonjour()` n'attend aucun paramètre, mais Python lui en a quand même passé un (l'objet `tiers`).

C'est exactement ce que Python fait dans le cas de `tiers.dire_bonjour()` : il fournit un argument de manière implicite, l'objet qui appelle la méthode (`tiers`).

Dans la nouvelle version suivante, la méthode `dire_bonjour()` a été modifiée pour **accepter un paramètre** :

```
class Fraction:
    numerateur = 1
    denominateur = 1
```

```
    def dire_bonjour(obj):  # obj = tiers par exemple
        return "Bonjour tout le monde"
```

Grâce à ce changement, on peut maintenant appeler la méthode depuis une instance sans provoquer d'erreur :

```
tiers = Fraction()
print(tiers.dire_bonjour())  # Bonjour tout le monde
```

Cette fois, Python ne se plaint plus, car la méthode **attend bien un paramètre**, et ce paramètre correspond à **l'objet qui l'appelle**. Lorsqu'on écrit `tiers.dire_bonjour()`, Python traduit automatiquement cet appel en :

```
Fraction.dire_bonjour(tiers)
```

Autrement dit, **l'instance `tiers` est passée en argument à la méthode**, sans qu'on ait besoin de l'écrire soi-même.

Ce mécanisme est une règle interne du langage : à chaque fois qu'une méthode est appelée à partir d'un objet, **Python envoie cet objet comme premier paramètre** à la fonction correspondante.

Dans l'exemple ci-dessus, on a choisi d'appeler ce paramètre `obj` pour bien montrer ce qui se passe : `obj` **représente l'objet qui a fait l'appel**, ici, `tiers`.

Ainsi, pendant l'exécution, l'interpréteur exécute en réalité :

```
def dire_bonjour(obj):  # obj = tiers
    return "Bonjour tout le monde"
```

L'objet `tiers` est donc passé automatiquement à `obj`.

Dans l'exemple suivant, la méthode `quotient()` vient donner tout son sens au paramètre que nous avons ajouté :

```
class Fraction:
    numerateur = 1
    denominateur = 1

    def quotient(obj):  # obj = tiers ou quart selon l'appel
        return obj.numerateur / obj.denominateur

tiers = Fraction()
tiers.denominateur = 3
```

```
quart = Fraction()
quart.denominateur = 4

print(tiers.quotient())  # 0.3333...
print(quart.quotient())  # 0.25
```

Ici, Python applique toujours le même principe : lorsqu'on appelle `tiers.quotient()`, l'interpréteur traduit automatiquement cet appel en :

```
Fraction.quotient(tiers)
```

Cela signifie que l'objet `tiers` est transmis comme premier argument à la méthode et devient le paramètre `obj`. Ainsi, à l'intérieur de la méthode, `obj` **représente l'objet qui a fait l'appel**.

Quand Python exécute la ligne :

```
return obj.numerateur / obj.denominateur
```

il doit aller chercher les valeurs de `numerateur` et `denominateur`. Pour cela, il suit une règle de recherche très précise :

1. **Python commence par regarder dans le mappingproxy de l'objet** (`obj.__dict__`).
    - Dans le cas de `tiers`, ce mappingproxy contient `{'denominateur': 3}`.
    - Python y trouve donc `denominateur = 3`.
2. **S'il ne trouve pas un attribut dans le mappingproxy de l'objet**, Python remonte dans celui de la **classe `Fraction`** (`Fraction.__dict__`).
    - Il y trouve `numerateur = 1`, car cet attribut n'a pas été redéfini dans l'objet `tiers`.

Ainsi, pour `tiers.quotient()`, Python fait :

- `obj.numerateur` → trouvé dans `Fraction.__dict__` → valeur 1
- `obj.denominateur` → trouvé dans `tiers.__dict__` → valeur 3 Résultat : `1 / 3 = 0.3333`

De la même manière, pour `quart.quotient()`, Python suit le même ordre :

- `obj.numerateur` → trouvé dans `Fraction.__dict__` → valeur 1
- `obj.denominateur` → trouvé dans `quart.__dict__` → valeur 4 Résultat : `1 / 4 = 0.25`

Ce comportement montre comment Python organise la recherche des attributs : il commence toujours par **l'objet qui appelle la méthode**, puis, s'il ne trouve pas l'attribut demandé, il **remonte vers la classe**. C'est ce mécanisme de recherche hiérarchique qui permet à chaque instance d'avoir ses propres valeurs tout en conservant celles définies par la classe comme valeurs par défaut.

Dans la dernière version suivante, la méthode `quotient()` est désormais écrite de la manière standard utilisée en Python :

```
class Fraction:
    numerateur = 1
    denominateur = 1

    def quotient(self):  # self = tiers ou quart
        return self.numerateur / self.denominateur
```

Le mot **`self`** remplace simplement le paramètre `obj` que nous utilisions jusqu'ici. Il ne s'agit pas d'un mot-clé réservé, mais d'une **convention universelle** en Python. Chaque fois qu'une méthode est appelée sur un objet, Python passe automatiquement cet objet en premier paramètre, et par convention on le nomme `self`.

Ainsi, lorsque l'on écrit :

```
tiers = Fraction()
tiers.denominateur = 3
print(tiers.quotient())
```

Python exécute en réalité :

```
Fraction.quotient(tiers)
```

et attribue la valeur de `tiers` à `self`.

À l'intérieur de la méthode, `self` représente donc **l'objet sur lequel la méthode agit**. Cela permet d'accéder à ses données internes (`self.numerateur`, `self.denominateur`) et de manipuler son état indépendamment des autres instances.

L'utilisation de `self` rend ainsi le code plus clair, plus lisible et plus conforme aux conventions du langage. Elle symbolise le lien direct entre la méthode et l'objet qui l'utilise : chaque instance exécute la même méthode, mais sur **ses propres données**.

L'initialisation

Jusqu'à présent, chaque fois qu'on voulait créer une fraction différente, on procédait en deux étapes :

```
tiers = Fraction()
tiers.denominateur = 3

quart = Fraction()
quart.denominateur = 4
```

```
deux_cinquieme = Fraction()
deux_cinquieme.numerateur = 2
deux_cinquieme.denominateur = 5
```

D'abord, on **instancie** la classe `Fraction`, ce qui crée un objet avec les valeurs par défaut (`numerateur = 1`, `denominateur = 1`). Ensuite, on **modifie manuellement** ces valeurs pour adapter chaque fraction à ce qu'on veut représenter.

Ce procédé fonctionne, mais il est un peu lourd et peu pratique : il oblige à créer l'objet avant de le configurer. Ce serait plus logique de pouvoir **fournir directement les valeurs souhaitées dès la création** de la fraction.

C'est exactement le rôle de la méthode spéciale `__init__`. Cette méthode permet de **donner des valeurs initiales** à une instance au moment où elle est créée. Elle "prépare" l'objet juste après son instanciation et permet d'éviter d'avoir à modifier ses attributs manuellement après coup.

Grâce à `__init__`, on pourra créer directement une fraction déjà configurée :

```
tiers = Fraction(1, 3)
quart = Fraction(1, 4)
deux_cinquieme = Fraction(2, 5)
```

Ainsi, au moment même où Python crée l'objet, il reçoit les valeurs du numérateur et du dénominateur, et les enregistre dans ses attributs internes. On dit alors que l'objet est **initialisé** dès sa création, il possède déjà toutes les données dont il a besoin pour fonctionner.

Dans cette nouvelle version, la classe `Fraction` introduit une méthode spéciale appelée `__init__` :

```
class Fraction:
    numerateur = 1
    denominateur = 1

    def __init__(self, numerateur, denominateur):
        self.numerateur = numerateur
        self.denominateur = denominateur

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

Cette méthode est automatiquement exécutée **au moment où l'on crée une nouvelle instance**. C'est elle qui permet de **donner des valeurs de départ** à l'objet, sans avoir à les modifier manuellement juste après sa création.

Par exemple :

```
tiers = Fraction(1, 3)
quart = Fraction(1, 4)
```

Lorsque Python lit la ligne `tiers = Fraction(1, 3)`, il exécute en réalité deux opérations :

1. Il **crée un nouvel objet vide** de type `Fraction` ;
2. Il **appelle automatiquement** la méthode `__init__` pour **initialiser** cet objet avec les valeurs données.

À ce moment-là, le paramètre `self` prend toute son importance. Lorsqu'on écrit :

```
self.numerateur = numerateur
self.denominateur = denominateur
```

on indique explicitement à Python **dans quel mappingproxy** (c'est-à-dire dans quel dictionnaire d'instance) il faut créer les clés `numerateur` et `denominateur`.

Concrètement :

- `self` représente **l'objet en cours de création** (par exemple `tiers` ou `quart`) ;
- `self.numerateur = numerateur` demande à Python de **créer la clé `'numerateur'` dans le dictionnaire de l'objet `self`** (`self.__dict__`) et d'y stocker la valeur reçue en argument ;
- `self.denominateur = denominateur` fait exactement la même chose pour la clé `'denominateur'`.

Cela veut dire que chaque objet `Fraction` possède désormais **ses propres attributs** stockés dans son propre dictionnaire d'instance. Ainsi, `tiers` a son propre couple `(1, 3)` et `quart` le sien `(1, 4)`, totalement indépendants l'un de l'autre.

En résumé, `self` indique à Python **sur quel objet** il doit travailler. Sans `self`, Python ne saurait pas où enregistrer les données reçues en paramètres lors de l'appel au constructeur.

En conséquence, les attributs **`numerateur`** et **`denominateur`** définis au niveau de la classe **ne sont plus jamais utilisés**. Leur présence dans le mappingproxy de la classe devient donc inutile, car aucun objet n'ira les consulter.

On peut alors simplifier la définition de la classe en les retirant complètement :

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

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

Dès lors, chaque objet Fraction possède ses propres attributs indépendants, et la classe ne conserve que la structure, c'est-à-dire le plan, qui permet de les créer et de les initialiser.

Conclusion

Le couple `self` et `__init__` forme le cœur du fonctionnement orienté objet en Python.

La méthode `__init__` intervient immédiatement après la création d'un objet. C'est elle qui assure son **initialisation**, c'est-à-dire la mise en place de ses premiers attributs et de leurs valeurs. Elle permet ainsi de construire des objets déjà prêts à l'emploi, sans devoir modifier manuellement leurs données après leur création.

Le paramètre `self`, quant à lui, joue un rôle fondamental : il **désigne l'objet sur lequel la méthode agit**. Lorsque Python appelle une méthode sur une instance, il transmet automatiquement cette instance en premier argument. Grâce à `self`, chaque objet peut accéder à ses propres données, indépendamment des autres instances de la même classe.

Ensemble, `__init__` et `self` donnent à chaque instance son **identité propre** :

- `__init__` définit **ce que contient l'objet au moment de sa création** ;
- `self` permet à l'objet de **se reconnaître lui-même** et de manipuler ses propres informations.

C'est cette combinaison qui fait de chaque instance une entité autonome, capable de stocker, modifier et utiliser ses données sans interférer avec celles des autres.