MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

*Liberté*
*Égalité*
*Fraternité*

**VOIE GÉNÉRALE**

2DE | 1RE | TLE

*Numérique et sciences informatiques*

ENSEIGNEMENT
SPÉCIALITÉ

# VOCABULAIRE DE LA PROGRAMMATION OBJET

## SOMMAIRE



Retrouvez éduscol sur

## Introduction

Le programme de terminale mentionne la programmation objet dans les rubriques « Structures de données » et « Langages et programmation ». Pour autant, il est clairement précisé que des concepts importants de la programmation objet ne sont pas à aborder (comme l'héritage et le polymorphisme). Ainsi, même s'il est possible d'approfondir certains concepts en fonction des projets proposés aux élèves, une présentation complète de ce paradigme de programmation n'est pas à réaliser[1]. Avec l'introduction du vocabulaire, les élèves peuvent « distinguer sur des exemples les paradigmes impératif, fonctionnel et objet. » Quant au choix d'un paradigme par l'élève, capacité dont la maîtrise sous-entend une pratique importante de la programmation, celui-ci peut par exemple être discuté au début de la mise en œuvre d'un projet.

Une fois le vocabulaire précisé, la programmation objet peut être utilisée pour varier les implémentations d'une structure de données. Il permet également d'éclairer certains termes utilisés en classe de première. Ces motivations permettent d'orienter la présentation faite aux élèves ainsi que d'estimer les compétences exigibles en évaluation.

## Liens avec les usages de première

En première, les élèves ont déjà été confrontés aux notions de classes et de méthodes. En effet, le type «`list` Python» régulièrement manipulé permet d'illustrer ces notions :

```
>>> ma_liste = [12, 24, 28]
>>> type(ma_liste)
<class 'list'>
```

Ainsi, l'affichage du type d'une variable fait référence à une classe (ici `list`) et présente le mot-clé correspondant (`class`). De plus, l'usage du mot méthode (par exemple `append`) était régulièrement utilisé sans vraiment être expliqué et les élèves sont désormais habitués à la syntaxe suivante :

```
>>> ma_liste.append(31)
>>> ma_liste
[12, 24, 28, 31]
```

Ces usages, spécifiques à la programmation objet, peuvent être explicités en inspectant par exemple la méthode randint de la bibliothèque random que les élèves ont été amenés à utiliser. Le code suivant :

```
import random
import inspect

print(inspect.getsource(random.randint))
```

1. Voir notamment la dernière partie pour des précisions sur l'encapsulation.

permet d'accéder au code source de la méthode randint en affichant en console :

```
    def randint(self, a, b):
        """Return random integer in range [a, b], including both end
points.
        """
        return self.randrange(a, b+1)
```

On y retrouve la documentation de la méthode ainsi qu'un mot-clé self qui peut surprendre (la méthode randint attendant uniquement deux paramètres a et b). On peut également inspecter toute la bibliothèque random avec print(inspect.getsource(random.randint)) afin d'exhiber les mots-clés class ainsi que `__init__` pour évoquer le constructeur[2].

## *Présentation du vocabulaire*

### La notion d'objet

Dans un premier temps, on peut avoir une discussion avec les élèves autour du concept d'objet, que l'on définit par des attributs et des méthodes qui lui sont associées. Par exemple, il semble tout naturel de définir un nouvel objet rectangle par ses attributs longueur et largeur. Les méthodes `calcul_perimetre` et `calcul_aire` semblent tout aussi naturelles. En revanche, ça n'a pas de sens de définir une méthode permettant l'addition entre deux rectangles. On peut alors, sur plusieurs exemples, présenter l'intérêt des classes dans le sens d'une limitation des attributs et des méthodes qui lui sont appliquées (par exemple, la méthode `append` existe pour les listes mais pas pour les tuples). Cela prend tout son sens dans la partie traitant des structures de données.

### Les mots-clés à présenter

Comme observé en inspectant le module `random`, la création d'une classe se fait à l'aide du mot-clé `class`.

Cependant, lors de la création de la classe, on ne connait pas à l'avance le nom des objets qui seront créés. D'où la nécessité de représenter le futur objet créé à l'aide d'un mot-clé `self`.

Par exemple :

```
class Rectangle:

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

Le constructeur `__init__` permet de définir des attributs au futur objet créé ainsi que de leur affecter les valeurs désirées.

2. Même si sémantiquement, c'est un initiateur, on s'autorise cet écart de langage pour garder uniquement le terme constructeur avec les élèves...

Pour créer une instance d'une classe (autre expression pour désigner un objet), on fait suivre le nom de la classe de la liste des arguments attendus par `__init__` :

```
rectangle1 = Rectangle(6, 4)
```

Le code présent dans le constructeur est exécuté et le mot-clé self dans le constructeur représente l'instance de la classe en cours de création. On peut ensuite vérifier que notre instance de classe possède bien les deux attributs :

```
>>> rectangle1.longueur
6
>>> rectangle1.largeur
4
```

Les méthodes sont définies à la suite, dans la création de la classe :

```
    def calcule_aire(self):
        return self.longueur * self.largeur
```

Dans la définition d'une méthode, le premier argument, appelé self par convention, représente l'objet sur lequel la méthode est appliquée. Voici un exemple à l'aide de la console, où l'on retrouve les usages des méthodes rencontrées en première sur les listes :

```
>>> rectangle1.calcule_aire()
24
```

Quelques exemples simples permettent ainsi de présenter le vocabulaire de la programmation objet utile en terminale.

À ce stade, un élément n'a pas été évoqué : qu'en est-il de la modification de la valeur d'un attribut d'une instance de la classe ?

En Python, il est tout à fait possible de modifier directement un attribut, par exemple :

```
>>> rectangle1.longueur = 10
```

Ceci dit, notamment pour des raisons de « sécurité » du code, cette pratique est déconseillée. La dernière partie de ce document revient plus en détail sur cette notion d'encapsulation des données.

## Autres éléments intéressants à présenter

### Pour préparer la présentation des listes chainées

En gardant à l'esprit la motivation de la présentation de la programmation objet, on peut également proposer des mises en situation où un attribut est lui-même une instance de classe.

Si l'on considère cet exemple basique :

```
class Chien :

    def __init__(self, nom, age, race):
        self.nom = nom
        self.age = age
        self.race = race


class Personne :

    def __init__(self, prenom, nom, age, chien):
        self.prenom = prenom
        self.nom = nom
        self.age = age
        self.chien = chien


chien1 = Chien(«Medor», 2, «Berger Allemand»)
chien2 = Chien(«Brutus», 4, «Chihuahua»)

personne1 = Personne("Alain", "Térieur", 42, chien1)
```

Il n'est pas évident que les élèves voient l'intérêt d'utiliser la variable `chien1`, référençant une instance de la classe `Chien`, plutôt que la chaine de caractères «Medor». On peut alors exposer l'intérêt de conserver toutes les informations à notre disposition, avec par exemple :

```
>>> personne1.chien.race
Berger Allemand
```

De plus, chien1 possède d'autres attributs, notamment son âge qui évolue. Et il peut changer de maître. Outre une écriture très visuelle, cela permet également de préparer une présentation ultérieure des listes chainées.

### Les méthodes spéciales

Les méthodes spéciales ne sont pas au programme de NSI. Cependant, si dans un premier temps nous avons complété la classe `Chien` par exemple ainsi :

```
    def affiche_informations(self):

        return «Ce chien de la race « + self.race + « a pour nom « + self.
nom +
            « et pour age « + str(self.age) + " an(s)."
```

Nous pouvons alors afficher les informations relatives à une instance de classe grâce à cette méthode :

```
>>> chien1.affiche_informations()
Ce chien de la race Berger Allemand a pour nom Medor et pour age 2 ans.
```

Il peut être légitime de vouloir obtenir cet affichage avec l'instruction print(chien1), instruction qui, par défaut, renvoie un affichage précisant notamment l'adresse mémoire dans laquelle est stockée la valeur de la variable :

```
>>> print(chien1)
<__main__.Chien object at 0x10a98af60>
```

Une méthode est prédéfinie pour définir cela : `__str__` et il suffit de la modifier pour obtenir l'affichage voulu. Ainsi, en renommant la méthode affiche_informations en `__str__`, on obtient exactement le résultat souhaité.

Indépendamment de cette considération d'affichage, il peut être intéressant de présenter dans certains cas des méthodes spéciales comme `__eq__` ou `__add__` (il ne semble pas pertinent de tester l'égalité ou d'ajouter deux chiens...).

Considérons par exemple une classe Point avec deux instances de cette classe :

```
class Point:

    def __init__(self, abscisse, ordonnee):
        self.abscisse = abscisse
        self.ordonnee = ordonnee

point1 = Point(3,4)
point2 = Point(3,4)
```

L'affichage suivant peut surprendre :

```
>>> point1 == point2
False
```

En effet, par défaut, l'opérateur == compare les identifiants et donc ne renvoie ici vrai que si les 2 opérandes de part et d'autre du signe '==' désignent la même instance :

```
>>> point1
<__main__.Point at 0x10a931550>
>>> point2
<__main__.Point at 0x10a931668>
```

Cependant ceci ne correspond pas à la représentation que nous nous faisons de l'égalité de deux points. Pour éviter cet écueil, il suffit d'ajouter à la classe Point la méthode __eq__ qui est appelée en cas de test d'égalité :

```
    def __eq__(self, autrePoint):

        return self.abscisse == autrePoint.abscisse and
            self.ordonnee == autrePoint.ordonnee
```

Permettant d'obtenir le comportement attendu :

```
>>> point1 == point2
True
```

D'autre part, l'utilisation ponctuelle de la méthode __add__ peut mener à un retour sur le programme de première concernant l'opérateur '+' au comportement différent suivant les types de variables utilisées :

```
>>> [1, 3, 4] + [5, 6]
[1, 3, 4, 5, 6]
```

On peut retrouver un comportement identique avec la somme de deux vecteurs, qui est un nouveau vecteur : (on note que bien des élèves de terminale NSI n'ont pas rencontré de vecteurs depuis la classe de seconde) :

```
class Vecteur:

    def __init__(self, abscisse, ordonnee):
        self.abscisse = abscisse
        self.ordonnee = ordonnee

    def __add__(self, autre_vecteur):
        return Vecteur(self.abscisse + autre_vecteur.abscisse, self.
ordonnee
            + autre_vecteur.ordonnee)

    def __str__(self):
        return «Vecteur de coordonnées
            («+str(self.abscisse)+»,»+str(self.ordonnee)+»)»


v1 = Vecteur(1, 2)
v2 = Vecteur(-3, 4)
```

```
>>> v3 = v1 + v2
>>> print(v3)
Vecteur de coordonnées (-2, 6)
```

On retrouve bien un comportement qui rappelle l'exemple donné ci-dessus sur les listes.

Enfin, il peut être intéressant de présenter la fonction (bien nommée) isinstance, qui renvoie un booléen. Elle peut être présentée comme une généralisation de la fonction type utilisée sur les types de base Python.

Par exemple :

```
>>> isinstance(v3, Vecteur)
True
>>> isinstance(v3, Point)
False
```

Elle peut notamment permettre un peu de programmation défensive, par exemple en précisant dans la méthode __add__ ci-dessus que le second paramètre doit bien être un vecteur :

```
    def __add__(self, autre_vecteur):
        assert isinstance(autre_vecteur, Vecteur)
        return Vecteur(self.abscisse + autre_vecteur.abscisse, self.
ordonnee
            ↪ + autre_vecteur.ordonnee)
```

## À propos de l'encapsulation

Comme évoqué en introduction, le paradigme de programmation objet n'est pas à présenter dans son ensemble aux élèves. Une question se pose quant à la présentation de l'encapsulation des données (c'est-à-dire le principe de ne rendre accessibles et modifiables les attributs d'instance uniquement à l'aide de méthodes). En effet, en préfixant un attribut par un double underscore, celui-ci ne semble plus directement accessible (et donc modifiable, on appelle cela un attribut privé) :

```
class Exemple:

    def __init__(self, attribut1):
        self.__attribut1 = attribut1
```

```
>>> objet1 = Exemple(«test»)
>>> objet1.__attribut1
AttributeError: 'Exemple' object has no attribute '__attribut1'
```

Cependant, en inspectant la variable `objet1`, on se rend compte que Python lui a affecté un attribut `_Exemple__attribut1`, à la fois accessible et modifiable à l'extérieur de la classe :

```
>>> objet1._Exemple__attribut1 = "modification"
```

En ajoutant par exemple une méthode renvoie_attribut1 (que l'on a pu nommer ici get_attribut1 pour respecter les conventions de la programmation objet, mais celles-ci ne semblent pas nécessaires en Terminale) à notre classe :

```
    def renvoie_attribut1(self):
        return self.__attribut1
```

On obtient alors en console :

```
>>> objet1.renvoie_attribut1()
"modification"
```

Ainsi, en Python, l'encapsulation ne reste que purement théorique, et un utilisateur d'une classe peut toujours accéder à n'importe quel attribut d'instance (et le modifier). Il n'est donc pas nécessaire de présenter la notion d'attribut privé, ceci n'apportant rien de plus dans les applications de la programmation objet du programme de NSI.

Sans entrer dans des présentations aussi détaillées avec les élèves, une discussion peut être intéressante à mener avec eux sur le sujet : le langage Python simplifie l'accès aux attributs (ainsi que leur modification en dehors de la classe), en donnant une souplesse lorsque les classes sont utilisées par exemple dans l'implémentation de structures de données. Cependant, autant un accès simple aux attributs d'instance n'est pas très gênant, autant laisser l'utilisateur les modifier à sa guise peut être dangereux.

Pour insister sur la distinction des rôles entre le développeur d'une classe et un utilisateur (qui de plus, est souvent la même personne dans le cadre de l'enseignement NSI), l'aspect modulaire est pertinent à travailler et on peut aller plus loin qu'un usage éditeur/console pour distinguer le développeur d'une classe et son utilisateur : si un fichier nommé= `mesClasses.py`= contient le code de la classe `Exemple` ci-dessus , dans un fichier Python du même dossier, il suffit alors d'écrire le code suivant pour reproduire l'exemple ci-dessus :

```
import mesClasses

objet1 = mesClasses.Exemple("test")
```

Une fois cette distinction faite, et si un élève modifie un attribut d'instance directement, on peut alors lui demander de passer par une méthode (ce qu'on appelle un « setter », sans forcément utiliser ce vocabulaire) et sensibiliser les élèves au fait que le degré de sécurisation du code dépend notamment du choix du langage. Si plus tard ils sont amenés à produire un code sécurisé en programmation objet empêchant l'utilisateur de modifier les attributs d'instance sans passer par une méthode d'instance, ils devront choisir un langage de programmation conçu ainsi, c'est-à-dire un autre langage que Python…

Enfin, même si les exemples de ce document n'évoquent pas ces sujets, on pense à associer des tests unitaires à une classe, ainsi qu'une documentation.

En particulier, si l'on reprend un exemple de la partie précédente :

```
class Vecteur:

    """Classe permettant de représenter un vecteur du plan"""

    def __init__(self, abscisse, ordonnee):
        self.abscisse = abscisse
        self.ordonnee = ordonnee
```

On accède ainsi à la documentation de la classe :

```
>>> Vecteur.__doc__
"Classe permettant de représenter un vecteur du plan"
```