Programmation Orientée Objet (illustrée en Python & C#)

[Mise à jour le : 4/7/2023]

- **Sources**
  - **Documentation** sur Python.org : fonctions natives (built-in)
- **Outils**
  - IDE : Visual Studio
  - UML for Python : pynsource
- **Lectures connexes**
  - **Real Python**
    - Object-Oriented Programming (OOP) With Python
- **Mots-clés** : objet, état, comportement, instance de classe, classe, attribut, constructeur, méthode, méthode spéciale, propriété, association, héritage, polymorphisme.

> Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

and	continue	finally	is	raise
as	**def**	for	lambda	return
assert	del	from	None	True
async	elif	global	nonlocal	try
await	else	if	not	while
break	except	import	or	with
class	False	in	**pass**	yield

- Fonctions natives **(built-in)**[1] utilisées dans les exemples : **print**(), **str**()

Introduction

Dans les langages informatiques, il est possible de stocker et de manipuler des **objets**[2] en mémoire comme autant d'ensembles de couples **attribut/valeur**. Un objet possède donc un **nom** qui le rend unique, un **état** caractérisé par la valeur de ses attributs et des **comportements** lui permettant de changer d'état.

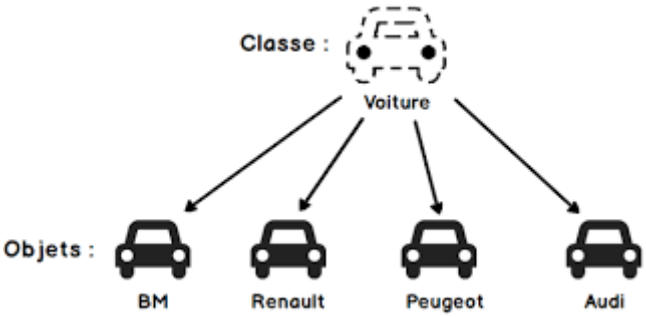
OBJET = ÉTAT + COMPORTEMENTS

Exemple
Pour décrire la voiture ci-dessus sous la forme d'un objet logiciel (très simplifié), on peut :
- la **nommer** maRenaultClio,
- préciser son **état** par “couleur ← rouge”, “année ← 2019”, vitesse ← 30 etc,
- lui affecter les **comportements** “avancer”, “reculer”, etc.

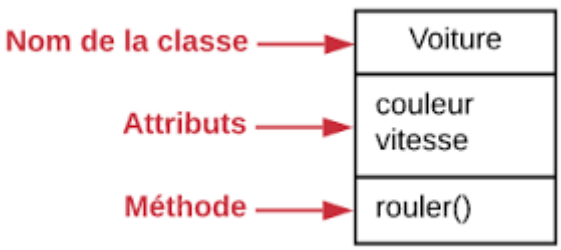
> Une Renault Clio est une voiture au même titre qu'une Ford Fiesta, une Fiat Tipo ou une Citroën C3. Une nouvelle structure de données voit le jour en Orienté Objet : la **classe**.

1. La classe

> Une **classe** permet de définir un ensemble d'objets ayant des caractéristiques communes. C'est un “**moule à objets**”. Les **objets** sont des **instances** de classe.



On représente une classe avec un symbole du langage **UML**[3] comme ci-dessous.



1.1 Déclarer une classe

- Python
- C#

Mot-clé

- En **Python**, on utilise le mot-clé ***class*** pour déclarer une classe.

Syntaxe

*.py

```
class nomClass:
    ''' docstring'''
    pass # Évite qu'une erreur ne se produise lorsque la classe est
vide
```

Exemple
Création d'une classe “Monstre” vide !

*.py

```
class Monstre:
    pass # classe sans attributs ni méthodes
```

Mot-clé

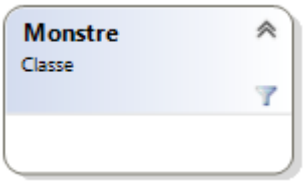
- En **C#**, on utilise le mot-clé ***class*** pour déclarer une classe.

Syntaxe

*.cs

```
class nomClass {
      // corps de la classe
}
```

Exemple
Création d'une classe “Monstre” vide !



*.cs

```
class Monstre {

}
```

Définir une classe consiste à déclarer les **attributs**[4] et les **méthodes**[5] caractérisant les **objets**, **instances**[6] de la classe.

1.2 Les attributs

En **p**rogrammation **o**rientée **o**bjet (**POO**), un **attribut** fait référence à une **variable** ou une donnée qui est associée à une classe ou à un objet. Les attributs représentent les caractéristiques d'un objet et définissent son état.
Un attribut peut être associé à l'objet (attribut d'**instance**) ou à la classe (attribut **statique**).

1.2.1 Les attributs d'instance

Les attributs d'instance ou attributs sont des **variables d'instance** accessibles depuis toutes les méthodes de la classe où elles sont définies.

- Python
- C#

Mots-clés
En Python, contrairement à d'autres langages comme C#, il n'est pas nécessaire d'avoir déclaré à l'avance les attributs de l'objet (ou leur type). Il sont créés automatiquement la première fois qu'on leur donne une valeur dans le constructeur **__init__**. Le mot-clé **self** est la référence à l'instance en cours. Cet argument doit être placé en premier. self va correspondre à l'instance créé par la classe (voir le paragraphe “Créer et manipuler des objets”).

Remarque : En python les **attributs sont publics par défaut**. Il faut les faire précéder de __ pour les rendre privés et ainsi respecter le principe d'encapsulation de l'OO.

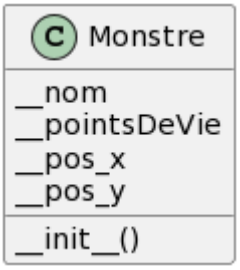
Syntaxe

*.py

```
def __init__(self,argument_1,...,argument_n): # __init__ est ce qui se
```

```
rapproche le plus d'un constructeur
   self.__attribut_1 = argument_1 # ou self.attribut_1 pour que
attribut_1 soit public
   ...
   self.__attribut_n = argument_n # ou self.attribut_n pour que
attribut_n soit public
```

Exemple
Création d'une classe “Monstre” contenant trois attributs, la position du monstre en x et y sur l'écran et son nombre de points de vie (pointsDeVie).



*.py

```
class Monstre:
    def __init__(self, nom, x=0, y=0, pv=100):
        self.__nom = nom
        self.__pointsDeVie = pv
        self.__pos_x = x
        self.__pos_y = y
```

Mot-clé

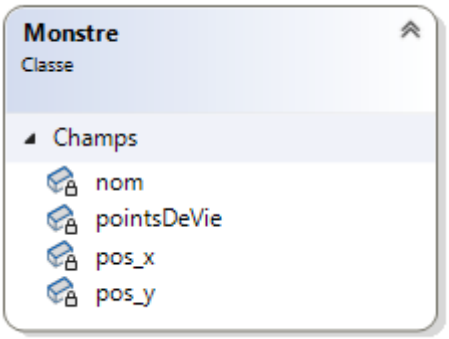
- **Le C# incite à respecter le principe d'encapsulation**. Les **attributs** sont **privés par défaut**. On peut les déclarer publics avec le mot-clé ***public*** pour y accéder directement à partir d'un objet, mais ce n'est pas une bonne pratique. Le respect du principe d'encapsulation se fait avec des getters et des setters.

Syntaxe

*.cs

```
class nomClasse {
    private type nomAttribut_1; # le mot private est optionnel
    ...
    private type nomAttribut_n;
}
```

Exemple
Création d'une classe “Monstre” contenant trois attributs, la position du monstre en x et y sur l'écran et son nombre de points de vie (pointsDeVie).



*.cs

```
class Monstre
{
    private int nom;
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;
}
```

1.2.2 Les attributs statiques

Un attribut statique est une **variable définie au niveau de la classe plutôt qu'au niveau de l'instance (objet)**. Cela signifie que sa valeur est partagée par toutes les instances de la classe et peut être accédée et modifiée sans avoir besoin d'instancier la classe.

- Python
- C#

Exemple
L'attribut de l'exemple ci-dessous est utilisé pour compter le nombre de monstres en vie.

*.py

```
class Monstre
   __nombreDeMonstres = 0 # Cet attribut est statique car non précédé
de self.
```

Mot-clé
En C#, on utilise le mot-clé **static** pour déclarer un attribut de classe.

Exemple
L'attribut de l'exemple ci-dessous est utilisé pour compter le nombre de monstres en vie.

*.cs

```
private static int nombreDeMonstres = 0;
```

1.3 Les méthodes

> En **P**rogrammation **O**rientée **O**bjet (**POO**), une **méthode** est une **fonction ou un sous-programme membre d'une classe**. Une méthode peut être:
> - un **constructeur** qui se définit comme une méthode spéciale utilisée pour initialiser les objets créés à partir de cette classe. Il ne renvoie aucune valeur;
> - une **méthode d'instance**, n'agissant que sur un seul objet (instance de la classe) à la fois;
> - une **méthode statique** ou **méthode de classe** appartient à la classe plutôt qu'à une instance. Elle est indépendante de toute instance de la classe (objet) et peut être appelée directement à partir de la classe.

- Python
- C#

> **Il n'est pas possible de surcharger une méthode en Python**. À la place, on peut donner des valeurs par défaut aux arguments. Contrairement aux autres langages, l'objet sur lequel la méthode agit doit être nommé explicitement en premier paramètre.

1.3.1 Le constructeur

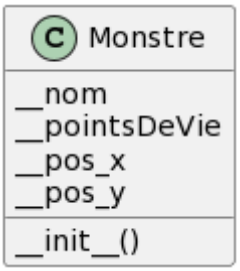
Mot-Clé
En Python, comme on l'a vu dans le paragraphe “Attributs” ci-dessus, le constructeur a pour nom **__init__** et nécessite ***self*** comme premier paramètre.

Synthaxe (Rappel)

*.py

```
def __init__(self,argument_1,...,argument_n): # __init__ est ce qui se rapproche le plus d'un constructeur
    self.__attribut_1 = argument_1 # ou self.attribut_1 pour que attribut_1 soit public
    ...
    self.__attribut_n = argument_n # ou self.attribut_n pour que attribut_n soit public
```

Exemple
Création d'une classe “Monstre” contenant trois attributs, la position du monstre en x et y sur l'écran et son nombre de points de vie (pointsDeVie).



*.py

```
class Monstre: # Remarque : la classe est déclarée 1 fois
    def __init__(self, nom, x=0, y=0, pv=100):
        self.__nom = nom
        self.__pointsDeVie = pv
        self.__pos_x = x  # ou self.pos_x pour que pos_x soit publique
        self.__pos_y = y
```

1.3.2 Méthode d'instance

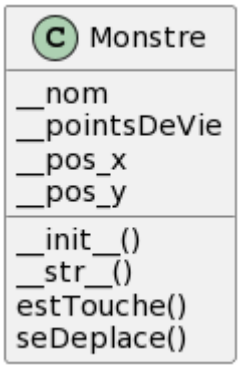
Mot-clé
Les méthodes d'instance étant des **routines membres de classe**, elles sont définies par le mot-clé ***def***.

Syntaxe

*.py

```
class nomClass:
    def nomMethode_1(self,argument_1,...,argument_n):
        pass # Évite qu'une erreur ne se produise lorsque la méthode est
vide
```

Exemple
Création d'une classe “Monstre” contenant trois attributs (+ le constructeur) et trois méthodes.



monstres.py

```
class Monstre:
```

```
    # Les attributs est la méthode __init__ ont été présentées ci-
dessus.

    def seDeplace(self, x, y):
        self.__pos_x = x
        self.__pos_y = y

    def estTouche(self, degats):
        self.__pointsDeVie -= degats

    def __str__(self):  # Redéfinition de la méthode print
        return str(self.__nom) + ' : pos_x=' + str(self.__pos_x) + '
pos_y=' + str(self.__pos_y)+' points de vie=' + str(self.__pointsDeVie)
```

Le mot-clé **self** est **INDISPENSABLE** dès qu'il s'agit de faire référence à l'objet lui-même.
Toutes les méthodes portant sur l'**objet** se doivent de le recevoir en paramètre.
Remarque : pour simplifier sa déclaration dans cette première approche, la classe Monstre ne répond pas au concept de MASQUAGE des données. Voir le paragraphe “Concept d'ENCAPSULATION”.

1.3.3 Méthode de classe

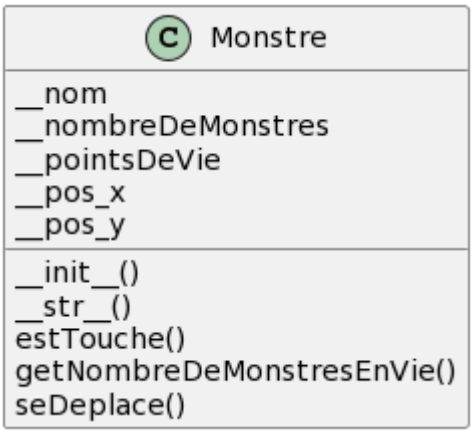
Mot-clé
En Python, le décorateur **@staticmethod** est utilisé pour définir une méthode statique dans une classe.

Syntaxe

*.py

```
@staticmethod
def nomMethode(argument_1,...,argument_n):
    pass # Évite qu'une erreur ne se produise lorsque la méthode est
vide
```

Exemple
Création d'une méthode statique getNombreDeMonstresEnVie() dans la classe Monstre pour connaître le nombre de monstres en vie.



*.py

```
class Monstre:
    __nombreDeMonstres = 0 # Attribut statique

    # Modification du constructeur pour incrémenter __nombreDeMonstres
    def __init__(self, nom, x=0, y=0, pv=100):
        self.__nom = nom
        self.__pointsDeVie = pv # exemple self.__posx
        self.__pos_x = x   # Un attributs (où variable d'instance)
        self.__pos_y = y   # est rendu privé par : __
        Monstre.__nombreDeMonstres += 1

    @staticmethod
    def getNombreDeMonstresEnVie():
        print(f"Le nombre de Monstres en vie est
:{Monstre.__nombreDeMonstres}")

    # Les autres méthodes ont été présentées précédemment.
```

1.3.1 Constructeur

Mot-clé
Le constructeur se doit d'être public, c'est-à-dire accessible à partir d'un objet. Pour cela, on utilise le mot-clé ***public***.

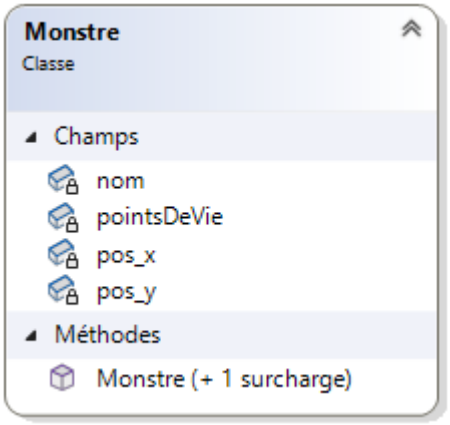
Syntaxe

*.cs

```
class nomClass {
  public nomClasse(parametre_1,...,parametre_n) {
    private Attribut_1 = parametre_1;
    ...
    private Attribut_n = parametre_n;
  }
```

```
}
```

Exemple
Constructeurs de la classe “Monstre” contenant trois attributs, la position du monstre en x et y sur l'écran et son nombre de points de vie (pointsDeVie).



*.cs

```cs
class Monstre {
    private string nom;
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;

    // Constructeur surchargé
    public Monstre(string lenom)
    {
        nom = lenom;
        pos_x = 0;
        pos_y = 0;
        pointsDeVie = 100;
    }

    public Monstre(string lenom, int x=0, int y=0, int pv = 100)
    {
        nom = lenom;
        pos_x = x;
        pos_y = y;
        pointsDeVie = pv;
    }
}
```

> **Surcharger une méthode** revient à en **créer une nouvelle** dont la **signature** se différencie de la précédente uniquement **par la liste et la nature des arguments**.

1.3.2 Méthode d'instance

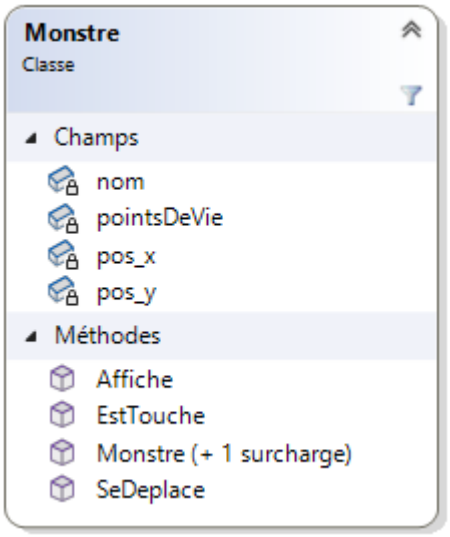
Mot-clé
Les méthodes d'instance appartenant à l'interface de la classe sont rendues publiques avec le mot-clé **public**.

Syntaxe

*.cs

```
class nomClass {
  public(ou private) type nomMethode(parametre_1,...,parametre_n) {
    // Corps de la méthode d'instance
  }
}
```

Exemple
Création d'une classe “Monstre” contenant trois attributs (+ le constructeur surchargé) et trois méthodes.



*.cs

```
class Monstre {
    // Les attributs est le constructeur ont été présentés ci-dessus.

    public void SeDeplace(int x, int y)
    {
        pos_x = x;
        pos_y = y;
    }

    public void EstTouche(int degats)
    {
        pointsDeVie -= degats;
    }
```

```
    public void Affiche()
    {
        Console.WriteLine($"{nom} : x={pos_x}, y={pos_y}, points de
vie={pointsDeVie}");
    }
}
```

1.3.3 Méthode de classe (ou statique)

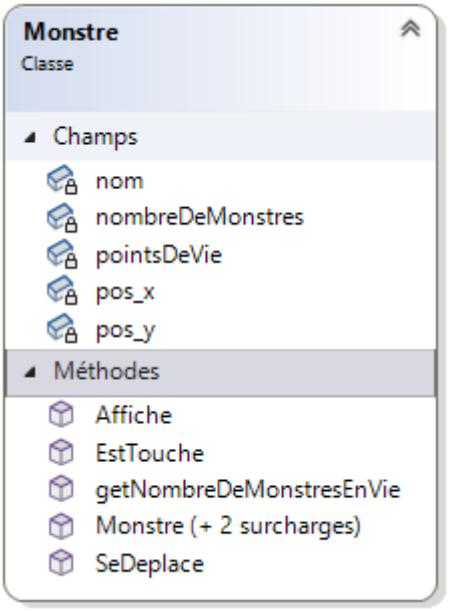
Mot-clé
Une méthode de classe est créée avec le mot-clé **static**.

Syntaxe

*.cs

```
class nomClass {
  public(ou private) static type
nomMethode(parametre_1,...,parametre_n) {
    // Corps de la méthode de classe
  }
}
```

Exemple
Création d'une méthode statique getNombreDeMonstresEnVie() dans la classe Monstre pour connaître le nombre de monstres en vie.



*.cs

```
class Monstre {
```

```
    private string nom;
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;

    private static int nombreDeMonstres = 0;

    public static void getNombreDeMonstresEnVie()
    {
        Console.WriteLine($"Nombre de monstres en vie :
{Monstre.nombreDeMonstres}");
    }

    public Monstre(string lenom)
    {
        nom = lenom;
        pos_x = 0;
        pos_y = 0;
        pointsDeVie = 100;
        Monstre.nombreDeMonstres += 1;
    }

    public Monstre(string lenom, int x=0, int y=0, int pv = 100)
    {
        nom = lenom;
        pos_x = x;
        pos_y = y;
        pointsDeVie = pv;
        Monstre.nombreDeMonstres += 1;
    }

    // Les autres méthodes ont été présentées ci-dessus.
 }
```

2. Instancier (créer) et manipuler des objets

> Instancier une classe consiste à **créer un objet sur son modèle**. Entre classe et objet, il y a, en quelque sorte, le même rapport qu'entre type et variable.

2.1 Instanciation (création) d'un objet et initialisation de ses attributs

- Python
- C#

Mot-clé

La création d'une variable de type objet (en abrégé d'un objet) est identique à celle des types standards du langage python : elle passe par une simple **affectation**.

Rappel: La méthode spéciale __init__ est appelée lors de la création de l'objet. Les valeurs passées en paramètre sont affectées aux attributs de l'objet par l'intermédiaire des arguments de __init__.

Syntaxe

*.py

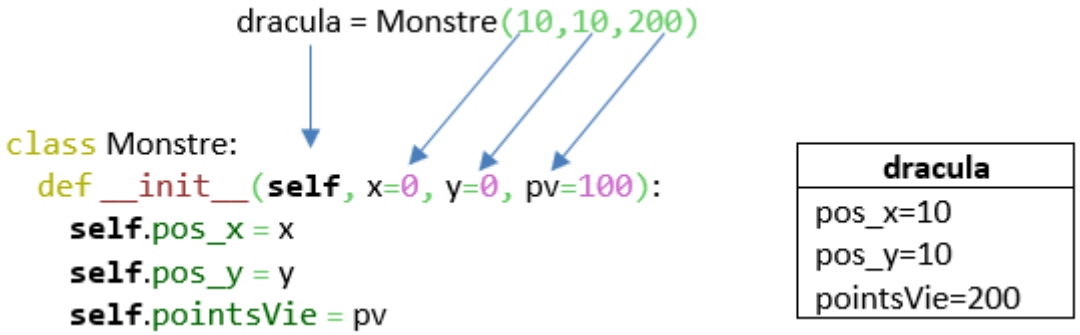
```
nomObjet = nomClasse(paramètre_1, ...,paramètre_n)
```

Exemple

*.py

```
# Simplification de la classe Monstre vue précédemment
# Limitée à trois attributs publics
class Monstre:
    def __init__(self, x=0, y=0, pv=100):
        self.pos_x = x
        self.pos_y = y
        self.pointsDeVie = pv

# Création de l'objet <dracula> de type <Monstre>
dracula = Monstre(10,10,200)
```

Illustration



> Les attributs n'étant pas privés dans cet exemple, il est possible d'y accéder indépendamment des méthodes de la classe. Ceci n'est pas souhaitable, car on ne répond plus au concept d'**ENCAPSULATION**, un des trois piliers de la POO avec l'**héritage** et le **polymorphisme**.

Mot-clé

Syntaxe

*.cs

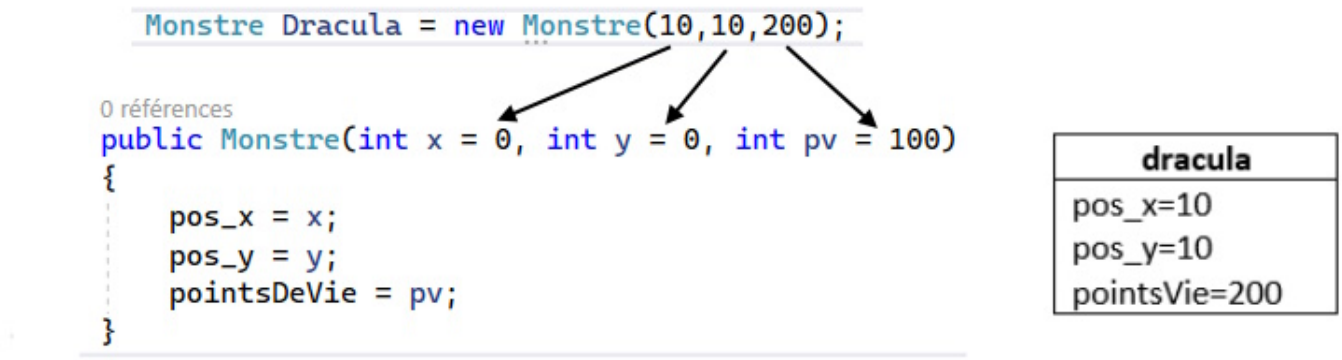
```
nomClasse nomObjet = nomClasse(paramètre_1, ...,paramètre_n)
```

Exemple

*.cs

```
// Simplification de la classe Monstre vue précédemment
class Monstre
{
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;

   public Monstre(int x=0, int y=0, int pv = 100)
    {
        pos_x = x;
        pos_y = y;
        pointsDeVie = pv;
    }
}
public class Jeu
{
    public static void Main()
    {
        Monstre dracula = new Monstre(10,10,200); // Création de
l'objet <dracula> de type <Monstre>
    }
}
```

Illustration



2.2 Manipulation d'un objet

> Les objets sont manipulés par l'intermédiaire de leurs **méthodes**.

- Python
- C#

Syntaxe

*.py

```
nomObjet.nomMethode(paramètre_1, ...,paramètre_n)
```

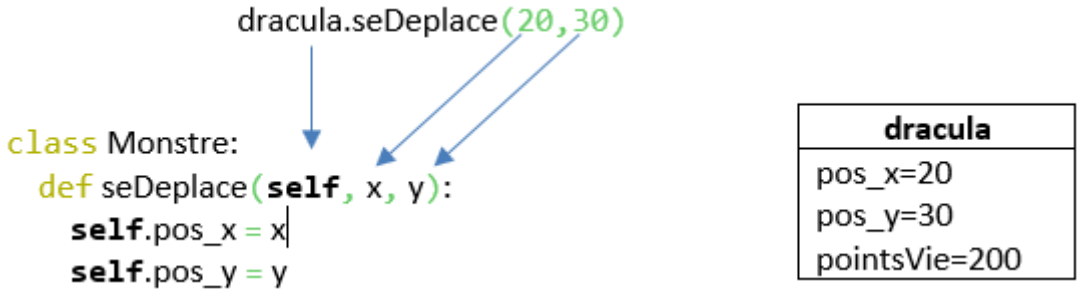
Exemple

*.py

```
class Monstre:
    def __init__(self, x=0, y=0, pv=100):
        self.__pos_x = x
        self.__pos_y = y
        self.__pointsDeVie = pv

    def seDeplace(self, x, y):
        self.__pos_x = x
        self.__pos_y = y

    def estTouche(self, degats):
        self.__pointsDeVie -= degats

    def __str__(self):  # Redéfinition de la méthode print
        return 'pos_x=' + str(self.__pos_x) + ' pos_y=' +
str(self.__pos_y)+' points de vie=' + str(self.__pointsDeVie)

# Jeu
```

```
dracula = Monstre(10, 10, 200)
dracula.seDeplace(20, 30)
dracula.estTouche(6)
print(f'Dracula est touché : {dracula}') # Résultat - Dracula est
touché : posx=20 posy=30 points de vie=194
```

Illustration
Objet dracula après l'exécution de : dracula.seDeplace(20, 30)



Syntaxe

*.cs

```
nomObjet.nomMethode(paramètre_1, ...,paramètre_n)
```

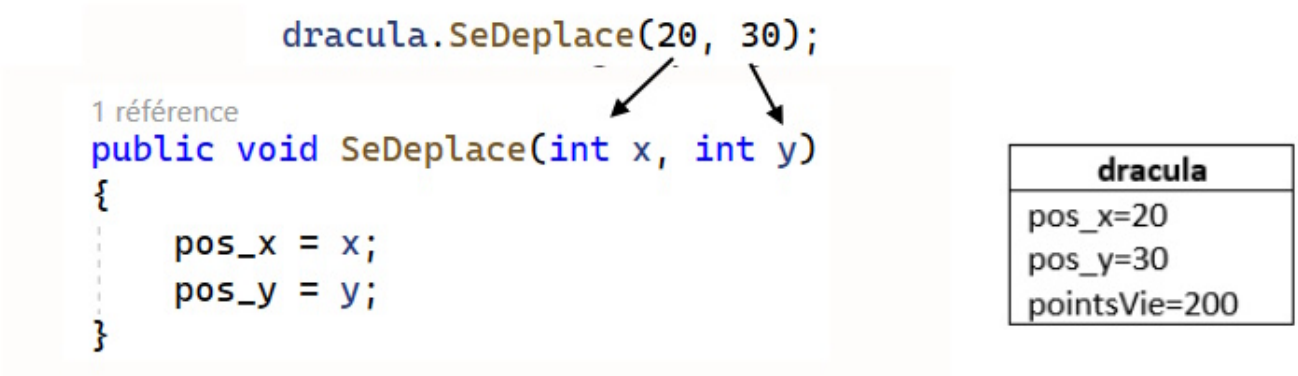
Exemple

*.cs

```
class Monstre
{
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;

    public Monstre(int x = 0, int y = 0, int pv = 100)
    {
        pos_x = x;
        pos_y = y;
        pointsDeVie = pv;
    }

    public void SeDeplace(int x, int y)
    {
        pos_x = x;
        pos_y = y;
    }

    public void EstTouche(int degats)
    {
        pointsDeVie -= degats;
```

```
    }

    public void Affiche()
    {
        Console.WriteLine($"x={pos_x}, y={pos_y}, points de
vie={pointsDeVie}");
    }
}

public class Jeu
{
    public static void Main()
    {
        Monstre dracula = new Monstre(10,10,200);
        dracula.SeDeplace(20, 30);
        dracula.EstTouche(6);
        Console.Write("Dracula est touché ");
        dracula.Affiche(); // Résultat - Dracula est touché x=20, y=30,
points de vie=194
    }
}
```

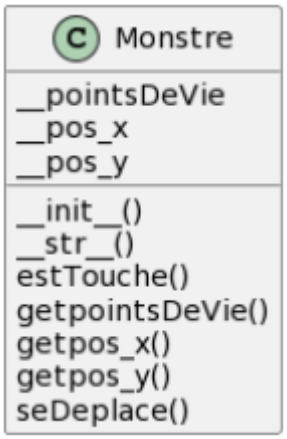
Illustration
Objet dracula après l'exécution de : dracula.SeDeplace(20, 30)



3. Principe d'encapsulation

> En programmation, l'**encapsulation** désigne le principe de **regrouper des données brutes avec un ensemble de routines** permettant de les lire ou de les manipuler. Ce principe est souvent accompagné du **masquage** de ces données brutes afin de s'assurer que l'utilisateur ne contourne pas l'interface qui lui est destinée. Wikipédia

- Python
- C#



En Python, pour **MASQUER** les attributs, c'est-à-dire les rendre **privés** afin qu'ils ne soient accessibles qu'aux méthodes de la classe, on fait précéder leur nom d'un double soulignement __.

Exemple
Les **attributs** de la classe “Monstre” sont **masqués**. Il n'est pas possible d'y accéder directement. Il faut passer par les methodes de la classe ou des méthodes d'accès appelées **accesseur**(ou **getteur**) et **mutateur**(ou **setteur**).

monstres.py

```
class Monstre: # Simplifiée
    def __init__(self, x=0, y=0, pv=100):
        self.__pos_x = x    # Les attributs (où variables d'instances)
        self.__pos_y = y    # sont rendus privés lorsqu'ils sont
précédés de __
        self.__pointsDeVie = pv

    def getpos_x(self):  # Accesseur (getteur)
        return self.__pos_x

    def getpos_y(self):  # Accesseur (getteur)
        return self.__pos_y

    def getpointsDeVie(self):  # Accesseur (getteur)
        return self.__pointsVie

    def seDeplace(self, x, y):
        self.__pos_x = x
        self.__pos_y = y

    def estTouche(self, degats):
        self.__pointsDeVie -= degats

    def __str__(self):  # Redéfinition de la méthode print
        return 'posx=' + str(self.__pos_x) + ' posy=' +
str(self.__pos_y)+' points de vie=' + str(self.__pointsDeVie)
```

- **Accesseur (getteur)**

Les méthodes *getpos_x()*, *getpos_y()*, *getpointVie()* sont appelées **accesseurs** ou getteur. Elles exposent les attributs privés de l'objet. Il s'agit de fonction permettant de consulter la valeur des attributs de l'objet. Le nom de ces fonctions commence par get.

- **Mutateur (setteur)**

Pour accéder aux attributs privés de l'objet dans le but de les modifier, on construit des **mutateurs** ou setteurs.

Exemple
Mutateur accédant aux points de vie dans la classe Monstre.

*.py

```
# A ajouter dans la classe Monstre
    def setpointsDeVie(self, pv):  # Mutateur (setter)
        self.__pointsDeVie = pv

# Accès à l'attribut __pointsVie
dracula.setpointsDeVie(400)
```

> En C#, les attributs sont **MASQUES (PRIVES) par défaut**. Ils ne sont accessibles qu'aux méthodes de l'objet ou à des **propriétés**.
> Une propriété (property) est un membre d'une classe qui expose une valeur ou un comportement spécifique de l'objet de cette classe. Elle peut être utilisée pour lire ou écrire une valeur d'une manière encapsulée, en fournissant un accès contrôlé aux données internes de la classe.

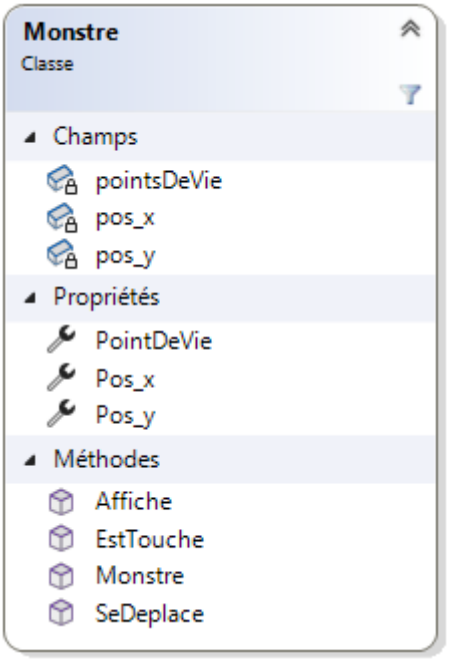
Mots-clés
Les mots-clés **get** et **set** spécifient les accesseurs (accessors) de la propriété, permettant de lire et écrire la valeur, respectivement.

Syntaxe

*.cs

```
public type NomPropriete { get; set; }
```

Exemple
Les attributs de la classe “Monstre” sont masqués. Pour y accéder sans passer par les méthodes de l'objet, on définit les **propriétés** *Pos_x*, *Pos_y* et *PointDeVie*.



*.cs

```
class Monstre
{
    private int pointsDeVie;
    private int pos_x; // position du monstre en (x,y)
    private int pos_y;

    public int Pos_x
    {
        get { return pos_x; }
    }

    public int Pos_y
    {
        get { return pos_y; }
    }

    public int PointDeVie
    {
        get { return pointsDeVie; }
        set { pointsDeVie = value; }
    }

    public void SeDeplace(int x, int y)
    {
        pos_x = x;
        pos_y = y;
    }

    public void EstTouche(int degats)
    {
        pointsDeVie -= degats;
```

```
    }

    public void Affiche()
    {
        Console.WriteLine($"{nom} : x={pos_x}, y={pos_y}, points de
vie={pointsDeVie}");
    }
}
```

4. Association entre classes

> **Association de classes**
>
> *"La manière privilégiée pour permettre à **deux classes de communiquer par envoi de messages** consiste à ajouter aux attributs primitifs de la première un **attribut référent** de la seconde. La classe peut donc, tout à la fois, contenir des attributs et se constituer en nouveau type d'attribut. C'est grâce à ce mécanisme de typage particulier que, partout dans son code, la première classe pourra faire appel aux méthodes disponibles de la seconde." Bersini*

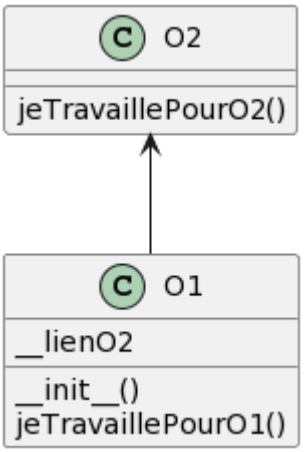
4.1 Association unidirectionnelle

4.1.1 Exemple 1: Association unidirectionnelle simple - lien d'association

- Python
- C#

Lecture du **diagramme de classes** : chaque objet O1 est associé à un objet O2. La réciproque n'est pas vraie.
O1 peut envoyer un message à O2 par le biais de l'association car :

- il possède un objet (référent) de type O2 dans ses champs
- cet objet est initialisé avec O2 lors de la création de O1



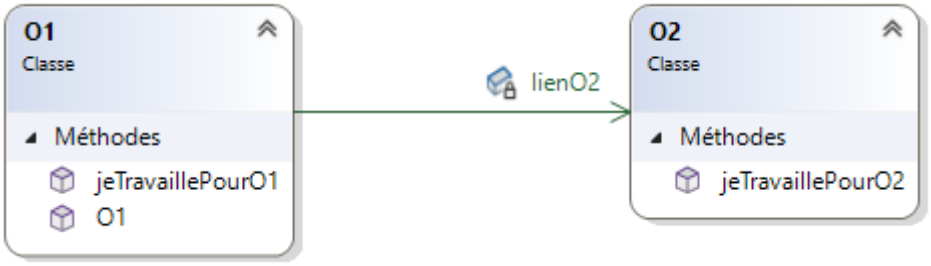
exemple1.py

```
class O2:
    def jeTravaillePourO2(self):
        print("Je travaille pour O2")

class O1:
    # lien d'association de O1 --> O2
    def __init__(self, O2):
        self.__lienO2 = O2

    def jeTravaillePourO1(self):
        print("Je travaille pour O1 et demande à O2 de travailler")
        self.__lienO2.jeTravaillePourO2()

# Programme
objO2 = O2()
objO1 = O1(objO2)
objO1.jeTravaillePourO1()
```

Résultat
Je travaille pour O1 et demande à O2 de travailler
Je travaille pour O2

Lecture du **diagramme de classes** : chaque objet O1 est associé à un objet O2. La réciproque n'est pas vraie.
O1 peut envoyer un message à O2 par le biais de l'association car :

- il possède un objet (référent) de type O2 dans ses champs
- cet objet est initialisé avec O2 lors de la création de O1



*.cs

```
    class O1
    {
        // Champs
        private O2 lienO2; // Lien d'association (fort et permanent)
avec un objet O2

        // Constructeur
        public O1(O2 objO2) // Passage de l'objet O2 à lier avec
l'objet O1
        {
            lienO2 = objO2;
        }

        // Méthode d'instance
        public void jeTravaillePourO1()
        {
            Console.WriteLine("Je travaille pour O1 et demande à O2 de
travailler");
            lienO2.jeTravaillePourO2();
        }
    }

    class O2
    {
        public void jeTravaillePourO2()
        {
            Console.WriteLine("Je travaille pour O2");
        }
    }

    public class Program
    {
        public static void Main()
        {
            O2 objO2 = new O2(); // Création de l'objet O2
            O1 objO1 = new O1(objO2); // Création de l'objet O1
initialisé avec O2
            objO1.jeTravaillePourO1(); // L'objet O1 envoie un message
à l'objet O2
        }
```

Résultat
Je travaille pour O1 et demande à O2 de travailler
Je travaille pour O2

4.1.2 Exemple 2 - Association unidirectionnel - lien de composition

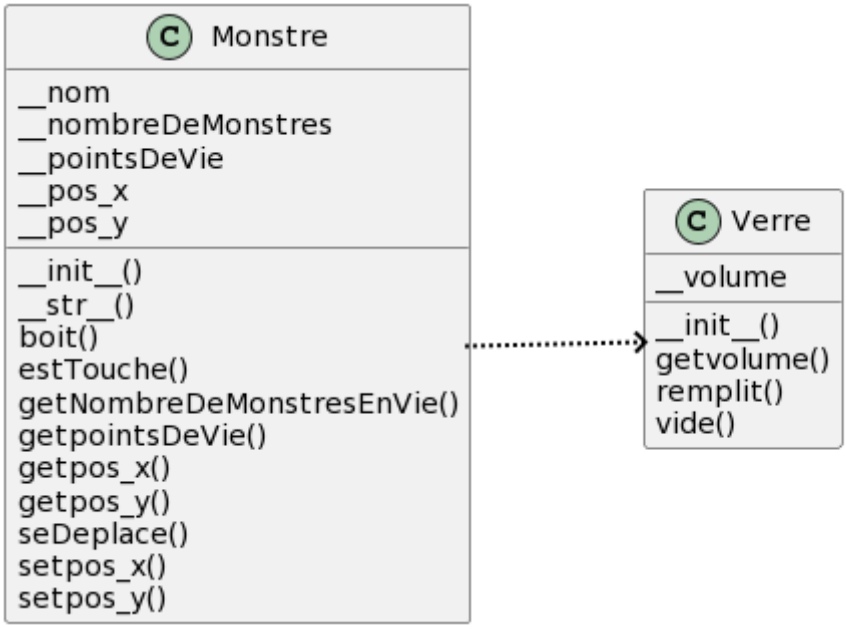
- Python
- C#

4.1.3 Exemple 3 - Association unidirectionnel avec passage d'un paramètre - lien de dépendance

Dracula boit un verre de sang ! A chaque gorgé, Dracula boit 10cl de sang. Le contenu du verre diminue de 10cl.

- Python
- C#

Lecture du diagramme de classe : l'objet verreDeSang est passé **provisoirement** à l'objet Dracula à l'aide de sa méthode boit. Contrairement à l'exemple précédent l'objet verreDeSang n'est pas associé à un lien permanent dans la classe Monstre. En effet, Dracula n'aura pas toujours un verre à la main !



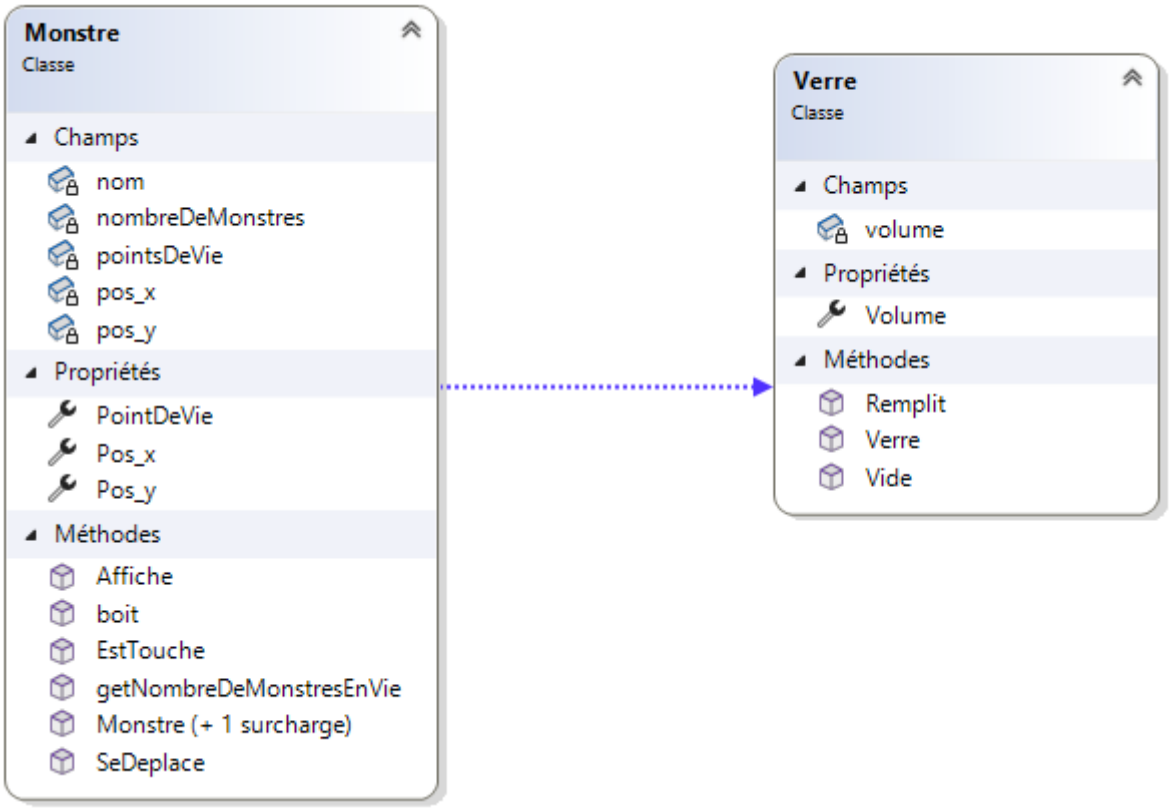
exemple3.py

```
class Verre:
    def __init__(self, vol):
        self.__volume = vol

    def remplit(self, vol):
```

```
        self.__volume += vol

    def vide(self, vol):
        self.__volume = self.__volume - vol

    def getvolume(self):  # Accesseur (getteur)
        return self.__volume

class Monstre:
    # Le code est limité à l'ajout de la méthode boit dans la version précédente
    # Ici le lien d'association est faible et provisoire verre n'existe plus à la sortie de la méthode boit
    def boit(self, vol, verre):
        verre.vide(vol)

# Programme
# Un vampire sert un verre de sang à Dracula
verreDeSang = Verre(50)
print(f'Le verre contient {verreDeSang.getvolume()} cl de sang') # 1
dracula = Monstre("Dracula", verreDeSang)
# Dracula voit le verre et se déplace
dracula.seDeplace(20, 30)
# mais reçoit un projectile
dracula.estTouche(6)
print(f'{dracula} : touché !') # 2
Monstre.getNombreDeMonstresEnVie() # 3
# Toujours en vie il boit
dracula.boit(10, verreDeSang)
print(f'Il reste {verreDeSang.getvolume()} cl dans le verre de sang')
#4
```

Résultat
1. Le verre contient 50 cl de sang
2. Dracula : pos_x=20 pos_y=30 points de vie=94 : touché !
3. Le nombre de monstres en vie est : 1
4. Il reste 40 cl dans le verre de sang

Lecture du diagramme de classe : l'objet verreDeSang est passé **provisoirement** à l'objet Dracula à l'aide de sa méthode boit. Contrairement à l'exemple précédent l'objet verreDeSang n'est pas associé à un lien permanent dans la classe Monstre. En effet, Dracula n'aura pas toujours un verre à la main !



exemple3.cs

```
namespace Enfer
{
    class Verre
    {
        private int volume;

        public Verre(int vol) { volume = vol; }
        public void Remplit(int vol) { volume += vol; }
        public void Vide(int vol) { volume -= vol; }
        public int Volume { get { return volume; } }
    }

    class Monstre
    {
    /*
      Le code est limité à l'ajout de la méthode boit dans la version précédente
      Ici le lien d'association est faible et provisoire verre n'existe plus à la sortie de la méthode boit
    */
        public void boit(int vol, Verre verre)
        {
            verre.Vide(vol);
        }
    }
```

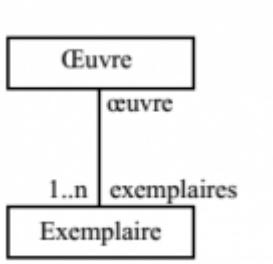
```
     public class Program
    {
        public static void Main()
        {
            // Un vampire sert un verre de sang à Dracula
            Verre verreDeSang = new Verre(50);
            Console.WriteLine($"Le verre contient
{verreDeSang.Volume}cl de sang");
            Monstre dracula = new Monstre("Dracula", 10, 10, 200);
            dracula.Affiche();
            // Dracula voit le verre et se déplace
            dracula.SeDeplace(20, 30);
            // mais reçoit un projectile
            dracula.EstTouche(6);
            dracula.Affiche("est touché");
            // Toujours en vie il boit
            dracula.boit(10, verreDeSang);
            Console.WriteLine($"Il reste {verreDeSang.Volume}cl dans le
verre de sang");
        }
    }
}
```

Résultat
1. Le verre contient 50 cl de sang
2. Dracula : pos_x=20 pos_y=30 points de vie=94 : touché !
3. Le nombre de monstres en vie est : 1
4. Il reste 40 cl dans le verre de sang

> **Communication possible entre objets**
>
> *“Deux objets pourront communiquer si les deux classes correspondantes possèdent entre elles une liaison de type **association** (exemple 1), **composition** (exemple 2) ou **dépendance** (exemple 3)., la force et la durée de la liaison allant décroissant avec le type de liaison. La communication sera dans les deux premiers cas possibles, quelle que soit l'activité entreprise par le premier objet, alors que dans le troisième cas elle se déroulera uniquement durant l'exécution des seules méthodes du premier objet, qui recevront de façon temporaire un référent du second.”* “Bersini

4.2 Association bidirectionnelle

Lecture du **diagramme de classes** : chaque oeuvre est **associée** à un ensemble d'exemplaires (1..n) et chaque exemplaire est associé à une et une seule oeuvre.



- Python
- C#

Exemple 1

5. Héritage et polymorphisme

5.1 Héritage

> L'héritage est l'un des principes clés de la POO et permet de créer de nouvelles classes (appelées classes dérivées ou sous-classes) à partir d'une classe existante (appelée classe de base ou superclasse).

- Python
- C#

Syntaxe

```
class nomClasse(nomClasseMère) # nomClasse hérite de nomClasseMère
   pass # Évite qu'une erreur ne se produise lorsque la méthode est vide
```

Mot-clé

Syntaxe

*.cs

```

```

Exemple

*.cs

```

```

5.2 Polymorphisme

> Le polymorphisme est un concept clé de la programmation orientée objet (POO) qui permet à des objets d'une même hiérarchie de classes de se comporter différemment en fonction du contexte. En d'autres termes, il permet à des objets de types différents d'être traités de manière uniforme en utilisant des méthodes communes, tout en offrant des implémentations spécifiques pour chaque type d'objet.

- Python
- C#

Exemple

expoly.py

```
# Dans l'exemple défini plus haut
# Redéfinition de la méthode str héritée de object
    def __str__(self):
        return "Elève : " + self.__prenom + " " + self.__nom + " " +
str(self.__age) + " " + "ans"
```

Mot-clé

Syntaxe

*.cs

```

```

Exemple

*.cs

```

```

Résumé

- La POO repose sur l'abstraction, l'**encapsulation**, l'**héritage** et le **polymorphisme**.
- Une **classe** permet de définir un ensemble d'objets ayant des caractéristiques communes. C'est un “**moule à objets**”. Les **objets** sont des **instances** de classe.

- Définir une classe consiste à définir les attributs et les méthodes caractérisant les objets instances de la classe.
- Les **attributs** sont déclarés **privés** afin d'en interdire l'accès à l'extérieure de la classe. C'est ce qu'on appelle l'**encapsulation**.
- A finir

Pour aller plus loin

- **Real Python**
  - Object-Oriented Programming (OOP) in Python 3
  - What Is Python's "Self" Argument, Anyway?
  - Getters and Setters: Manage Attributes in Python
  - Inheritance and Composition: A Python OOP Guide
  - Object-Oriented Programming in Python vs Java

1)

Fonctions toujours disponibles.

2)

En informatique, un objet est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. C'est le concept central de la programmation orientée objet (POO).Wikipédia

3)

Le Langage de Modélisation Unifié, de l'anglais **Unified Modeling Language**, est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une **méthode normalisée** pour visualiser la conception d'un système. Il est couramment utilisé en développement logiciel et en **conception orientée objet**.

4)

Les attributs représentent l'**état** de l'objet.

5)

Les méthodes sont les **fonctions** membres des classes. Elles représentent le **comportement** de l'objet. Ici on se limite aux méthodes d'instance.

6)

En programmation orientée objet, on appelle instance d'une classe un objet avec un **comportement** et un **état**, tous deux définis par la classe.