Structures de données

Programmation orientée objet

Objectifs pédagogiques :

- ✓ Écrire la définition d'une classe
- ✓ Accéder aux attributs et méthodes d'une classe.

Jusqu'à présent, le paradigme (famille) de programmation que nous avons utilisé est celui de la programmation procédurale. Il repose sur l'utilisation de fonctions et procédures afin de décomposer un problème complexe en sous-problèmes plus simples. Il existe une façon plus intuitive et plus proche du raisonnement humain permettant de coder un programme : il s'agit de la programmation orientée objet (POO). Le paradigme de la POO repose sur la définition et l'interaction de briques logicielles appelées *objets*. En POO, un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne... Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations avec d'autres objets. L'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités souhaitées et ainsi de mieux résoudre des problème complexes. En POO, l'étape de modélisation revêt une importance majeure. C'est en effet elle qui permet de transcrire les éléments du réel sous forme virtuelle.

1. Les concepts clés de la POO

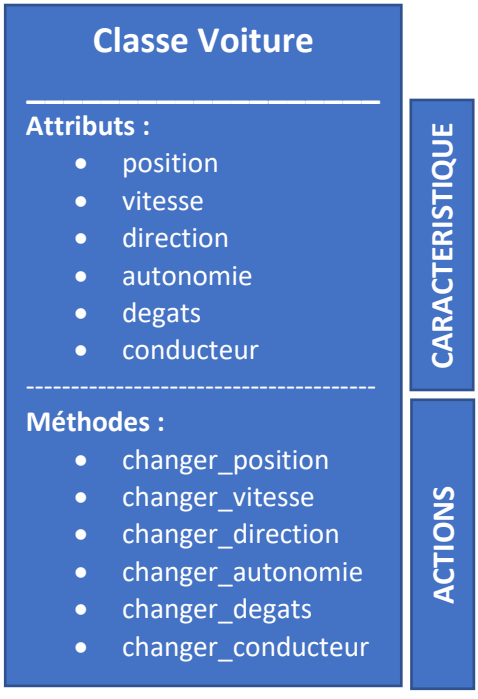
1.1. Qu'est-ce que la POO ?

La POO consiste en la création et l'utilisation d'**objets**, qui se composent d'**attributs** et de **méthodes** liés à des **classes**.

- Les **attributs** renvoient aux variables contenues dans l'objet : ce sont les données.
- Les **méthodes** sont les éléments qui permettent d'interagir avec les attributs. Les méthodes sont des fonctions internes à une classe.
- La notion de **classe** fait quant à elle référence aux « familles » qui définissent les objets. En clair, chaque objet est une instance d'une classe, c'est-à-dire qu'il hérite des attributs et des méthodes génériques de sa classe.

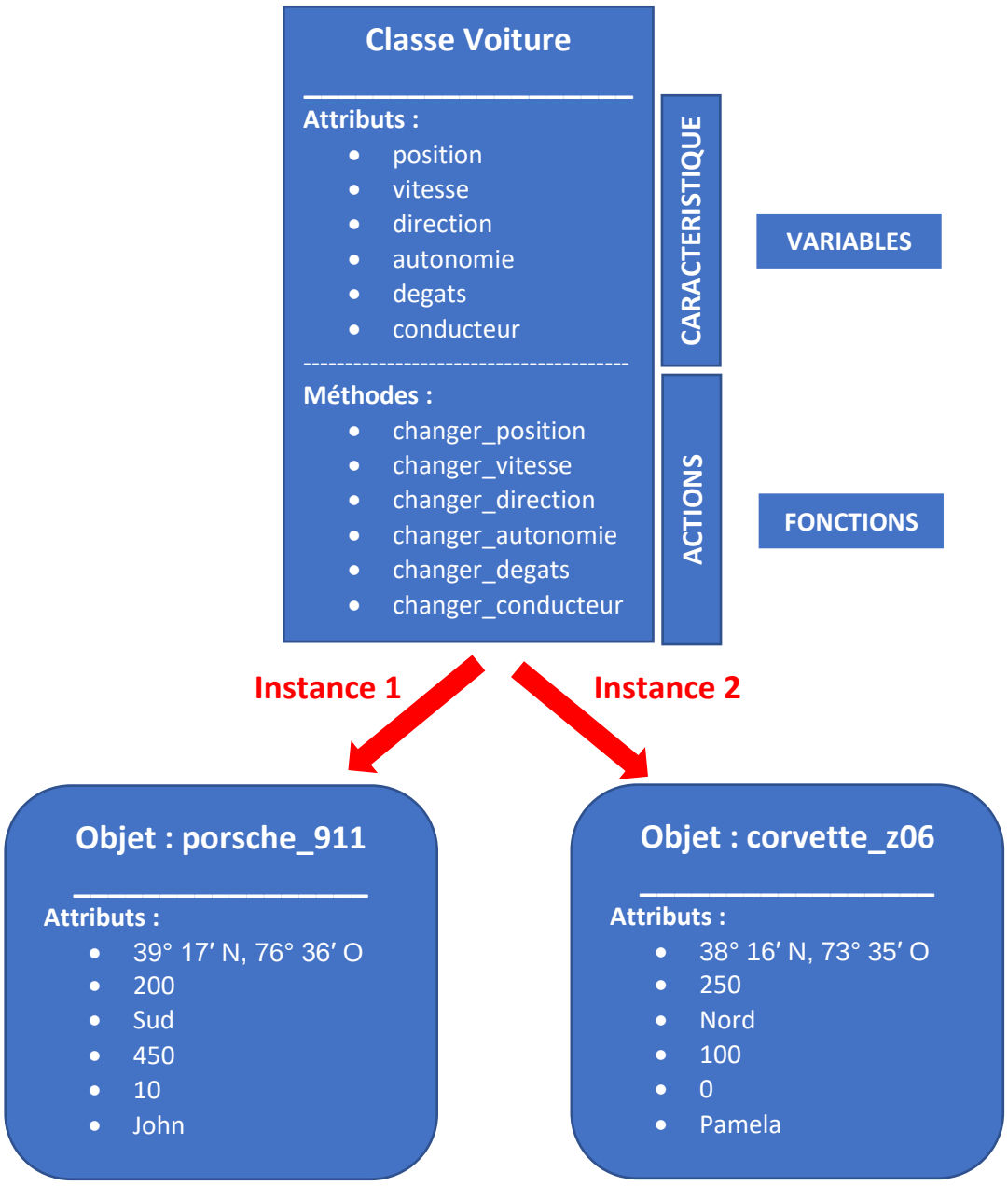
1.2. Notion de classe

Une manière simple d'aborder la notion de classe est de considérer celle-ci comme une « usine » permettant de fabriquer des objets possédant des caractéristiques (attributs) et des actions exécutables (méthodes) communes.
La classe *Voiture* ci-contre regroupe un ensemble de caractéristiques communes à toutes les voitures d'un jeu vidéo.
Cette « usine » permettra de fabriquer (instancier) des objets « voitures » qui se distinguerons les uns des autres par des caractéristiques propres différentes (vitesse, direction …) et sur lesquels on pourra agir d'une certaine façon (changer_position, changer_vitesse …).



1.3. Notion d'objet

Imaginons que vous participiez au codage d'un jeu vidéo dans lequel des personnages peuvent évoluer dans un environnement ouvert. Parmi les actions possibles qui leurs sont offertes, il peuvent monter dans une voiture afin de l'utiliser pour se déplacer. On vous confie le développement de la classe Voiture qui servira à «créer » (instancier) des objets « voitures » dans le jeu.



Un **objet** créé à partir d'une classe est une **instance** de cette **classe**. Dans l'exemple ci-dessous, les objets *porsche_911* et *corvette_z06* sont deux objets de type *Voiture* c'est-à-dire deux instances différentes de cette classe. Une classe contient «les schémas directeurs » permettant de fabriquer un objet selon un cahier des charges précis. Les **caractéristiques** (**attributs**) de chacun des objets instanciés pourront être modifiés en faisant appel aux **fonctions internes** (**méthodes**) de la classe à laquelle ils appartiennent.

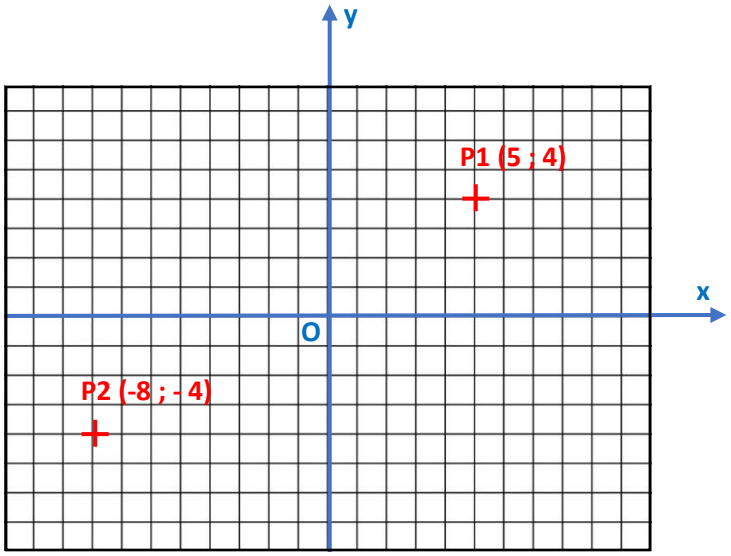
Q1. Qu'est-ce qu'une classe ?
Q2. Qu'est-ce qu'une instance de classe ?
Q3. Qu'est-ce qu'un attribut d'instance ?
Q4. Qu'est-ce qu'une méthode ?

2. Les classes et les objets en Python

2.1. Déclaration d'une classe : notions de constructeur et d'attributs

Afin d'illustrer la notion de classe et d'objet simplement, nous pouvons partir d'un exemple permettant de modéliser un point géométrique. En effet, on peut considérer un point dans un repère cartésien orthonormé comme un objet caractérisé par ses cordonnées. On peut donc voir les points A et B de l'exemple ci-dessous comme deux instances d'une classe *Point* ayant pour attributs les coordonnées x et y et comme méthodes des fonctions internes permettant de réaliser un certains nombre d'actions sur, ou à partir, de ces attributs (déplacer un point, mesurer une distance entre deux points…).

■ **Les points P1 et P2 sont caractérisés par leurs coordonnées :**



■ **1ère Implémentation d'une classe *Point* : constructeur**

Une **classe** est un ensemble incluant des **variables** ou **attributs** et des **fonctions** ou **méthodes**. Les attributs sont des variables accessibles depuis toute méthode de la classe où elles sont définies. En Python, les classes sont des types modifiables. Le mot-clé **class** permet de créer sa propre classe, suivi du nom de cette classe. Par convention, le nom identifiant une classe (qu'on appelle aussi son identifiant) débute par une majuscule. Le bloc d'instructions indenté situé après le symbole « : » définit le corps de la classe.

```
Entrée [1]: class Point:
                """Définition d'un point géométrique"""
                # Constructeur
                def __init__(self, X, Y):
                    self.x = X
                    self.y = Y
```

Lors de la création d'un objet il est souvent nécessaire d'initialiser les valeurs des attributs de l'instance : on utilise pour cela un **constructeur**. Un constructeur n'est rien d'autre qu'une méthode, sans valeur de retour, qui porte un nom imposé par le langage Python : **__init__()**. Ce nom est constitué de **init** entouré avant et après par **__** (**deux fois** le symbole **underscore** **_**, qui est le tiret sur la touche **8**). Cette méthode sera appelée lors de la création de l'objet. Le constructeur peut disposer d'un nombre quelconque de paramètres (ici x et y) mais avec toujours à minima **self.**

Les paramètres (ici X et Y) du constructeur sont des variables locales, comme c'est habituellement le cas pour une fonction. Les attributs (ici x et y) de l'objet sont quant à eux dans l'espace de noms de l'instance. Les attributs se distinguent facilement car ils ont le mot **self** placé devant eux. Le paramètre **self** désigne l'instance de la classe qui sera créée (self est une convention et pas un mot clé à proprement parlé).

Remarque : lors de la phase de construction d'un programme, il est possible de créer une classe vide de la manière suivante.

```
class classe_vide:
    pass
```

■ **Instanciation d'un objet point_1 de type *Point* :**

On vient de définir ci-dessus une classe ***Point***. On peut dès à présent s'en servir pour créer des objets de ce type, par instanciation.

❶ Créons un nouvel objet et mettons la référence à cet objet dans la variable `point_1` :

```
Entrée [2]: point_1 = Point(5,4) ❶

Entrée [3]: print(point_1) ❷
<__main__.Point object at 0x000002118668F488>

Entrée [4]: print(type(point_1)) ❸
<class '__main__.Point'>
```

❷ Le message renvoyé par Python indique que `point_1` contient une référence à une instance de la classe ***Point***, qui est définie elle-même au niveau principal du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale : 0x000002118668F488.

❸ L'objet (instance) `point_1` est bien de type ***Point***.

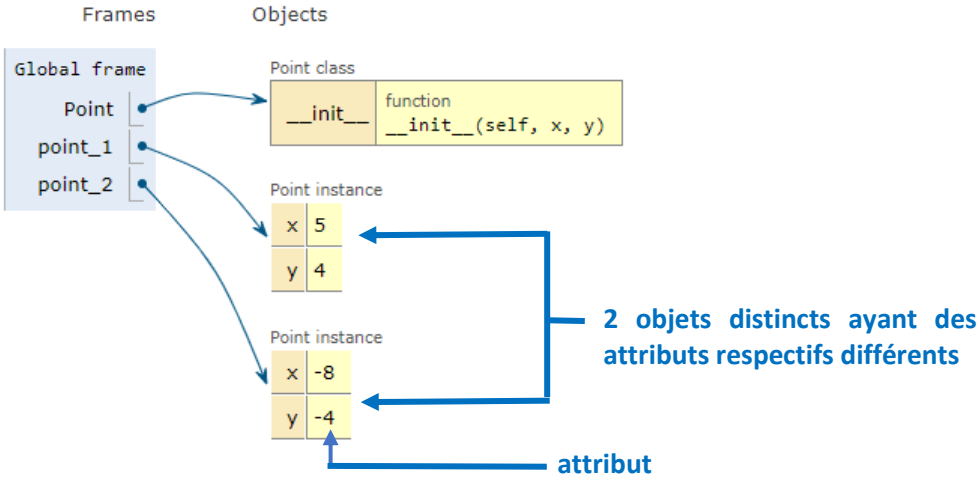
■ **Instanciation d'un objet point_2 de type *Point* :**

```
Entrée [5]: point_2 = Point(-8,-4)

Entrée [6]: print(point_2)
<__main__.Point object at 0x00000287F21C7AC8>

Entrée [7]: print(type(point_2))
<class '__main__.Point'>
```

On remarque que l'adresse mémoire allouée à l'instance `point_2` est bien différente de celle de `point_1` : il s'agit bien de deux objets distincts.



Q5. Comment définie-t-on une classe en Python ?

Q6. Qu'est-ce qu'un constructeur ? Comment le code-t-on en Python ?

Q7. Que désigne le mot self ?

Q8. Comment instancie-t-on un objet en Python ?

■ **Accès aux valeurs des attributs des objets de type *Point* :**

On affiche la valeur d'un attribut d'un objet (instance) avec la syntaxe suivante : nom_objet.nom_attribut

```
Entrée [8]: print(point_1.x)
            print(point_1.y)
```

```
5
4
```

```
Entrée [9]: print(point_2.x)
            print(point_2.y)
```

```
-8
-4
```

Q9. Comme accède-t-on aux attributs d'un objet instancié ?

2.2. Attributs de classe vs attributs d'instance

Dans l'exemple précédent, x et y sont des attributs propres à chaque objet instancié. Si on créé plusieurs objets de type *Point*, les attributs x et y ne seront pas forcément identiques. Il est cependant possible de créer des attributs propres non plus à chaque objet mais à la classe elle-même : on parle alors d'attributs de classe. Dans l'exemple ci-dessous, ***compteur_objet*** est un attribut de la classe ***Point***. À chaque fois qu'on crée un objet de type ***Point***, l'attribut de classe ***compteur_objet*** s'incrémente de 1. Il peut être utile dans certains cas de figures, d'avoir des attributs de classe, quand tous les objets doivent partager certaines données identiques.

```
Entrée [10]: class Point:
                 """Définition d'un point géométrique"""
                 compteur_objet = 0
                 # Constructeur
                 def __init__(self, X, Y):
                     self.x = X
                     self.y = Y
                     Point.compteur_objet += 1
```

L'attribut de classe ***compteur_objet*** s'incrémente d'une unité à chaque fois que le constructeur est appelé pour créer une nouvelle instance de classe.

```
Entrée [11]: point_1 = Point(5,4)
```

```
Entrée [12]: print(Point.compteur_objet)
```

```
1
```

```
Entrée [13]: point_2 = Point(-8,-4)
```

```
Entrée [14]: print(Point.compteur_objet)
```

```
2
```

L'attribut de classe ***compteur_objet*** n'est pas propre à une instance mais bien à toute la classe.

Q10. Quelle est la différence entre un attribut de classe et un attribut d'instance ?

2.3. Les méthodes

Une méthode est une fonction interne à une classe utilisant les attributs des instances et / ou agissant sur eux. Dans l'exemple ci-dessous la méthode ***deplacer()*** permet de modifier la valeur des attributs x et y associés à une instance de la classe ***Point***.

```
Entrée [15]:
class Point:
    """Définition d'un point géométrique"""
    # Constructeur
    def __init__(self, X, Y):
        self.x = X
        self.y = Y
    # Méthode
    def deplacer(self,dx,dy):
        self.x = self.x + dx
        self.y = self.y + dy

Entrée [16]:
point_3 = Point(1,1)

Entrée [17]:
point_3.deplacer(2,3)

Entrée [18]:
print(point_3.x)

3

Entrée [19]:
print(point_3.y)

4
```

Q11. Quel paramètre devra à minima contenir une méthode ?

2.4. La notion d'encapsulation

Certains langages orientés objet (C++ par exemple) mettent en place des attributs dits *privés* dont l'accès est impossible depuis l'extérieur de la classe. Ceux-ci existent afin d'éviter qu'un utilisateur n'aille perturber ou casser quelque chose dans la classe. Les arguments auxquels l'utilisateur a accès sont quant à eux dits *publics*. En Python, il n'existe pas d'attributs privés comme dans d'autres langages orientés objet. L'utilisateur a accès à tous les attributs quels qu'ils soient, même s'ils contiennent devant eux 2 caractères *underscore* ! Comme nous allons le voir ci-après, la présence de 2 *underscores* devant les noms d'attributs est un signe clair que l'utilisateur ne doit pas y toucher. Toutefois, cela n'est qu'une convention, et comme dit ci-dessus l'utilisateur d'une classe peut tout de même modifier ces attributs.

Remarque : le fait de mettre 2 *undescores* devant le nom d'un attribut d'une instance s'appelle le ***name mangling***, ou encore **substantypage** ou déformation de nom en français. C'est un mécanisme qui transforme le nom ***self.__attribut*** à l'intérieur de la classe en ***instance._NomClasse__attribut*** à l'extérieur de la classe.

Le concept d'**encapsulation** est un concept très utile de la POO. Il permet en particulier d'éviter une modification accidentelle des données d'un objet (valeurs de ses attributs) par un utilisateur de la classe dont il est issu. En effet, il n'est alors pas possible d'agir directement sur les données (attributs) d'un objet. Il est alors nécessaire de passer par ses méthodes qui jouent le rôle d'interface obligatoire.

Le changement de direction d'une voiture est une bonne analogie à ce type de situation. En effet si le conducteur (utilisateur) souhaite faire tourner son véhicule à gauche, il va agir sur le volant (interface / méthode) afin de modifier l'angle (valeur) des roues (attributs) avants. Le conducteur (utilisateur) n'agit donc pas directement sur les roues (attributs) de sa voiture mais par l'intermédiaire du volant (interface / méthode) pour des questions de sécurité.

Q12. Comment peut-on signifier à un utilisateur de ne pas modifier directement depuis l'extérieur d'une classe certains attributs d'instance ?

■ Définition d'attributs privés

On réalise la pseudo-protection des attributs de notre classe ***Point*** grâce à l'utilisation d'attributs privés. Pour avoir des attributs privés, leur nom doit débuter par __ (**deux fois** le symbole **underscore** _, qui est le tiret du bas sur la touche 8).

```
Entrée [20]: class Point:
                 """Définition d'un point géométrique"""
                 # Constructeur (avec attributs privés)
                 def __init__(self, X, Y):
                     self.__x = X
                     self.__y = Y
                 # Méthode
                 def deplacer(self,dx,dy):
                     self.__x = self.__x + dx
                     self.__y = self.__y + dy
```

Il n'est alors plus possible de faire appel aux attributs **__x** et **__y** depuis l'extérieur de la classe **Point** comme le montre l'exemple ci-dessous.

```
Entrée [21]: point_4 = Point(4,5)
             print(point_4.__x)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-12a34bf8c323> in <module>
      1 point_4 = Point(4,5)
----> 2 print(point_4.__x)

AttributeError: 'Point' object has no attribute '__x'
```

Il faut donc disposer de méthodes qui vont permettre par exemple de modifier ou d'afficher les informations associées à ces variables.

■ Accesseurs et mutateurs

Parmi les différentes méthodes que comporte une classe, on distingue souvent :

- les **constructeurs** (en anglais builder) qui permettent l'initialisation des valeurs des attributs d'instance ;
- les **accesseurs** (en anglais *accessor*) qui fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses attributs (généralement privés) sans les modifier ;
- les **mutateurs** (en anglais *mutator*) qui modifient l'état d'un objet, donc les valeurs de certains de ses attributs.

On rencontre souvent dans la littérature, l'utilisation de noms de la forme **get_attribut()** pour les accesseurs et **set_attribut()** pour les mutateurs, y compris dans des programmes dans lesquels les noms de variable sont francisés. Par exemple, pour la classe ***Point*** on peut définir les méthodes suivantes :

Entrée [22]:
```
class Point:
    """Définition d'un point géométrique"""
    # Constructeur
    def __init__(self, X, Y):
        self.__x = X
        self.__y = Y
    # Accesseurs
    def get_x(self):
        return self.__x
    def get_y(self):
        return self.__y
    # Mutateurs
    def set_x(self,X):
        self.__x = X
    def set_y(self,Y):
        self.__y = Y
    # Méthodes classiques
    def deplacer(self,dx,dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy
    def afficher(self) :
        print("(x = ",self.get_x(), ";", "y = ",self.get_y(),")")
```

Entrée [23]:
```
point_5 = Point(10,20)
print(point_5.get_x())
print(point_5.get_y())
point_5.afficher()
```

```
10
20
(x =  10 ; y =  20 )
```

Entrée [24]:
```
point_5.set_x(30)
point_5.set_y(50)
point_5.afficher()
```

```
(x =  30 ; y =  50 )
```

Entrée [25]:
```
point_5.deplacer(5,5)
point_5.afficher()
```

```
(x =  35 ; y =  55 )
```

Q13. Qu'est qu'un accesseur et qu'est-ce qu'un mutateur ?
Q14. Selon vous, quel est le principal inconvénient de leur utilisation ?

■ La classe property

De manière générale, une syntaxe avec des *getters* et *setters* du côté utilisateur surcharge la lecture du code et complique son implémentation.

```
point = Point(0,0)
```

```
# Pratique conforme à l'esprit de Python
print(point.x)
print(point.y)
```

```
# Pratique non conforme à l'esprit de Python
print(point.get_x())
print(point.get_x())
```

La méthode Python compatible est plus « douce » à lire, on parle aussi de « *syntactic sugar* » ou littéralement en français « *sucre syntaxique* ». De plus, à l'intérieur de la classe, il faut définir un *getter* et un *setter* pour chaque attribut, ce qui multiple les lignes de code.

Si on souhaite contrôler l'accès, la modification (voire la destruction) de certains attributs stratégiques d'une instance, Python met à disposition une classe spéciale nommée **property**. Celle-ci permet de combiner le maintien de la syntaxe lisible **instance.attribut**, tout en utilisant en filigrane des fonctions pour accéder, modifier, voire détruire l'attribut (à l'image des *getters* et *setters* évoqués ci-dessus). Pour faire cela, on utilise la fonction Python interne **property()** qui crée un objet (ou instance) **property** :

```
attribut = property(fget=accesseur, fset=mutateur, fdel=destructeur)
```

Les arguments passés à **property()** sont systématiquement des méthodes dites *callback*, c'est-à-dire des noms de méthodes que l'on a définies précédemment dans notre classe, mais on ne précise ni argument, ni parenthèse, ni **self**.

Avec la ligne de code précédente, **attribut** est un objet de type *property* qui fonctionne de la manière suivante à l'extérieur de la classe :

- l'instruction **instance.attribut** appellera la méthode **.accesseur()**.
- l'instruction **instance.attribut = valeur** appellera la méthode **.mutateur()**.
- l'instruction **del instance.attribut** appellera la méthode **.destructeur()**

Entrée [26]:
```
class Point:
    """Définition d'un point géométrique"""
    # Constructeur
    def __init__(self, X, Y):
        self.x = X
        self.y = Y
    # Accesseurs
    def get_x(self):
        return self.__x
    def get_y(self):
        return self.__y
    # Mutateurs
    def set_x(self,X):
        self.__x = X
    def set_y(self,Y):
        self.__y = Y

    # Méthodes classiques
    def deplacer(self,dx,dy):
        self.x = self.x + dx
        self.y = self.y + dy
    def afficher(self) :
        print("(x = ",self.x, ";", "y = ",self.y,")")

    x = property(fget=get_x, fset=set_x)
    y = property(fget=get_y, fset=set_y)
```

Entrée [27]:
```
point = Point(0,0)
```

Entrée [28]:
```
print(point.x)
print(point.y)
```

```
0
0
```

Entrée [29]:
```
point.x = 5
point.y = 10
```

Entrée [30]:
```
print(point.x)
print(point.y)
```

```
5
10
```

Entrée [31]:
```
del point.x
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-31-75d2fc1f0174> in <module>
----> 1 del point.x

AttributeError: can't delete attribute
```

Cette exécution montre qu'à chaque appel de **self.x** ou **self.y** on va utiliser les méthodes accesseur ou mutateur. Comme il n'y a pas de méthode destructeur (passée avec l'argument fdel), on ne pourra pas détruire les attributs x ou y : un **del point.x** (ou **del point.y**) conduit à une erreur de ce type : AttributeError: can't delete attribute.
Dans le constructeur, la commande **self.x = X** (ou **self.y = Y**) va appeler automatiquement la méthode **.set_x()** (ou **.set_y()**).

Remarque :
Il existe une autre syntaxe considérée comme plus élégante pour mettre en place les objets *property*. Il s'agit des *décorateurs* **@property**, **@attribut.setter** et **@attribut.deleter**. Toutefois, la notion de décorateur va bien au-delà de ce cours. Vous pouvez néanmoins consulter le site PROGAMIZ pour aller plus loin.

Q15. Compétez la classe Point avec une méthode permettant de calculer la distance euclidienne entre deux points.
Rappel : si A(x_A ; y_A) et B(x_B ; y_B) alors distance = $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

3. Pour aller plus loin : héritage de classe, opérateurs et polymorphisme

3.1. Principe de l'héritage de classe

En programmation, l'héritage est la capacité d'une classe d'hériter des propriétés d'une classe pré-existante. On parle de **classe mère** et de **classe fille**. En Python, l'héritage peut être multiple lorsqu'une classe fille hérite de plusieurs classes mères. En Python, lorsque l'on veut créer une classe héritant d'une autre classe, on ajoutera après le nom de la classe fille le nom de la ou des classe(s) mère(s) entre parenthèses :

```
class Mere1:
    # contenu de la classe mère 1


class Mere2:
    # contenu de la classe mère 2


class Fille1(Mere1):
    # contenu de la classe fille 1


class Fille2(Mere1, Mere2):
    # contenu de la classe fille 2
```

Dans l'exemple précédent, la classe **Fille1** hérite de la classe **Mere1** et la classe **Fille2** hérite des deux classes **Mere1** et **Mere2**. Dans le cas de la classe **Fille2**, on parle d'héritage multiple.
En programmation orientée objet, "hériter" signifie "avoir également accès à". Lorsqu'on dit qu'un objet "hérite" des attributs et des méthodes de la classe qui l'a défini, cela signifie que l'objet peut utiliser ces attributs et ces méthodes c'est-à-dire qu'il y a accès.

Illustrons cette notion d'héritage sur un exemple simple.

■ Classe mère Personne

Entrée [36]:
```
class Personne:
    # Constructeur
    def __init__(self, fname, lname, bdate):
        # Attributs d'instance
        self.firstName = fname
        self.lastName = lname
        self.birthDate = bdate
    #Méthode
    def printIdentity(self):
        print("{} {} {}".format(self.firstName,self.lastName,self.birthDate ))
```

Entrée [37]:
```
# Instanciation d'un objet de type Personne
perso1 = Personne("John","DOE","20/02/2000")
```

Entrée [38]:
```
# La méthode printIdentity() s'applique sur une instance de la classe Personne
perso1.printIdentity()
```

```
John DOE 20/02/2000
```

Remarque : Python permet aux arguments des fonctions / méthodes d'avoir une valeur par défaut, si la fonction / méthode est appelée sans l'argument il a la valeur par défaut. On parle alors d'arguments optionnels nommés.

Exemple d'arguments optionnels nommés dans le constructeur :

Entrée [36]:
```
class Personne:
    # Constructeur
    def __init__(self, fname="unknow", lname="unknow", bdate="unknow"):
        # Attributs d'instance
        self.firstName = fname
        self.lastName = lname
        self.birthDate = bdate
    #Méthode
    def printIdentity(self):
        print("{} {} {}".format(self.firstName,self.lastName,self.birthDate ))
```

Entrée [37]:
```
perso1 = Personne()
perso1.printIdentity()
```

```
unknow unknow unknow
```

Entrée [38]:
```
perso1 = Personne(lname="DOE",bdate="20/02/2020")
perso1.printIdentity()
```

```
unknow DOE 20/02/2020
```

■ Classe fille Eleve

Entrée [39]:
```
# La classe Eleve est une classe fille de la classe mère Personne
class Eleve(Personne) :
    pass
```

Entrée [40]:
```
# Instanciation d'un objet de type Eleve
eleve1 = Eleve("Laure","DINATEUR","07/07/2002")
```

Entrée [41]:
```
# La méthode printIdentity() héritée de la classe Personne s'applique sur une instance de la classe Eleve
eleve1.printIdentity()
```

```
Laure DINATEUR 07/07/2002
```

Q16. Pourquoi peut-on dire que la classe **Eleve** est une classe fille de la classe mère **Personne**. Qu'est-ce-que cela implique ?

3.2. Constructeurs et héritage

Par défaut, la sous-classe hérite de toutes les variables et fonctions de la classe parent et notamment de sa fonction `__init__()`). Il arrive souvent néanmoins que l'on souhaite « étendre » le constructeur de la classe fille en lui intégrant par exemple un nouveau paramètre tout en conservant l'héritage du constructeur de la classe mère.

Par exemple, on peut envisager un constructeur spécifique à la classe **Eleve** qui en plus des paramètres fname, lname et bdate du constructeur de la classe mère **Personne** introduit le nouveau paramètre croom (classe scolaire de l'élève). Pour conserver l'héritage de la fonction `__init__()` du parent, il est nécessaire de faire dans le constructeur de l'enfant un appel explicite à la fonction `__init__()` du parent tel qu'illustré dans l'exemple ci-après.
Exemple :

Entrée [42]:
```
class Personne:
    # Constructeur
    def __init__(self, fname, lname, bdate):
        # Attributs d'instance
        self.firstName = fname
        self.lastName = lname
        self.birthDate = bdate
    #Méthode
    def printIdentity(self):
        print("{} {} {}".format(self.firstName,self.lastName,self.birthDate ))

class Eleve(Personne) :
    # Constructeur
    def __init__(self, fname, lname, bdate,croom):
        # Conservation de l'héritage du constructeur de la classe Personne
        Personne.__init__(self, fname, lname, bdate)
        # Attribut d'instance
        self.classRoom = croom
    #Méthode
    def printClassRoom(self) :
        print(self.classRoom)
```

```
Entrée [43]: # Instanciation d'un objet de type Eleve
             eleve2 = Eleve("Cyprien","MAILLET","06/06/2003","1A")

Entrée [44]: # La méthode printIdentity() héritée de la classe Personne s'applique sur une instance de la classe Eleve
             eleve2.printIdentity()

Cyprien MAILLET 06/06/2003

Entrée [45]: # La méthode printClassRoom() s'applique sur une instance de la classe Eleve
             eleve2.printClassRoom()

1A
```

Le code ci-dessus a permis d'étendre la fonction `__init__()` de la classe mère **Personne** en ajoutant un nouvel argument (`croom`) dans le constructeur de la classe fille **Eleve**.

Remarque : en l'absence de l'instruction `Personne.__init__(self, fname, lname, bdate)` faisant référence au constructeur de la classe mère dans celui de la classe fille, l'héritage au niveau des constructeurs mère / fille aurait été « cassé » : celui de la classe fille aurait en effet « écrasé » celui de la classe mère (surcharge).

Q17. Si une classe fille possède son propre constructeur `__init__()`, quelle précaution doit-ont prendre pour ne pas casser l'héritage avec le constructeur de la classe mère ?

3.3. Tests d'héritage

La fonction `isinstance()` permet de tester le type d'une instance, c'est-à-dire le type d'une objet. Elle permet de savoir si un objet appartient à une certaine classe ou pas. On va lui passer en arguments l'objet dont on souhaite tester le type et le type qui doit servir de test. Cette fonction renverra `True` si l'objet est bien du type passé en second argument ou si il est d'un sous-type dérivé de ce type ou `False` sinon.
La fonction `issubclass()` permet de tester l'héritage d'une classe, c'est-à-dire permet de savoir si une classe hérite bien d'une autre classe ou pas. On va lui passer en arguments la classe à tester ainsi qu'une autre classe dont on aimerait savoir si c'est une classe mère de la classe passée en premier argument ou pas. La fonction renverra `True` si c'est le cas ou `False` sinon. Ces deux fonctions sont très utiles pour tester rapidement les liens hiérarchiques entre certaines classes et objets et peuvent aider à comprendre plus facilement un programme complexe en POO que l'on doit étudier.

Exemple :

```
# Instanciation d'un objet de type Personne
perso1 = Personne("John","DOE","20/02/2000")

# Instanciation d'un objet de type Eleve
eleve1 = Eleve("Laure","DINATEUR","07/07/2002","1B")

print(isinstance(perso1,Personne))
print(isinstance(perso1,Eleve))
print(isinstance(eleve1,Personne))
print(isinstance(eleve1,Eleve))
```

```
True
False
True
True
```

La fonction `isinstance()` permet d'observer que l'instance perso1 est une personne mais pas un élève alors que l'instance eleve1 est une personne et un élève (héritage).

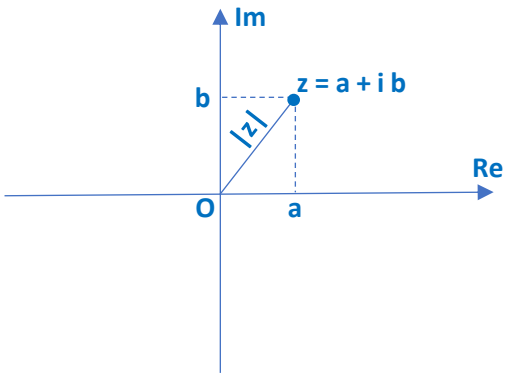
```
print(issubclass(Eleve,Personne))
print(issubclass(Personne,Eleve))
```

```
True
False
```

La fonction `issubclass()` permet d'observer que **Eleve** est une sous-classe (classe fille) de Personne (classe mère) mais pas l'inverse.

3.4. Notion d'opérateur de classe

Quand on manipule des classes, dans certain cas de figure on est obligé de redéfinir les propriétés associées à un opérateur basique tel que « + » par exemple. Illustrons cela sur la classe **Complexe** suivante, permettant d'instancier des nombres complexes.



Un nombre complexe z est de la forme **z = a + i b** avec a et b ∈ ℝ et i tel que $i^2 = -1$.

- a : partie réelle
- b : partie imaginaire
- $|z| = \sqrt{a^2 + b^2}$: module

Addition de 2 nombres complexes :

- $z_1 = a_1 + i\, b_1$
- $z_2 = a_2 + i\, b_2$
- $z_3 = z_1 + z_2 = (a_1 + a_2) + i\,(b_1 + b_2)$

On voit ici que l'opérateur « + » pour les nombres complexes n'est pas aussi trivial que celui utilisé pour additionner deux nombres réels.

On peut définir une classe **Complexe** permettant d'instancier des nombres complexes. Il faudra en particulier définir l'opérateur « + » avec la règle vue précédemment et la méthode permettant de calculer le module d'un nombre complexe

Le programme suivant contient une classe définissant un nombre complexe. La méthode ajoute définit ce qu'est une addition entre nombres complexes et la méthode module permet de calculer le module.

Entrée [46]:
```
from math import sqrt

class Complexe:
    # Constructeur
    def __init__(self, a = 0, b = 0):
        self.a = a
        self.b = b
    # Méthodes
    def module(self):
        return sqrt((self.a)**2 + (self.b)**2)
    def ajoute(self, z):
        return Complexe(self.a + z.a, self.b + z.b)
```

Entrée [47]:
```
z1 = Complexe(0, 1)
```

Entrée [48]:
```
z2 = Complexe(1, 0)
```

Entrée [49]:
```
z3 = z1.ajoute(z2)
print(z3.a,z3.b)
```

```
1 1
```

Entrée [50]:
```
print(z3.module())
```

```
1.4142135623730951
```

Toutefois, on aimerait bien écrire simplement z3 = z1 + z2 au lieu de z3 = z1.ajoute(z2) car cette syntaxe est plus facile à lire et surtout plus intuitive. Le langage Python offre cette possibilité. Il existe en effet des méthodes *clés* dont l'implémentation définit ce qui doit être fait dans le cas d'une addition, d'une comparaison, d'un affichage, … A l'instar du constructeur, toutes ces méthodes clés, qu'on appelle des *opérateurs*, sont encadrées par deux blancs soulignés, leur déclaration suit invariablement le même schéma. Voici celui de l'opérateur __add__ qui décrit ce qu'il faut faire pour une addition.

```
class nom_class:
    def __add__(self, autre):
        # corps de l'opérateur
        return ...   # nom_classe
```

nom_classe est une classe. L'opérateur __add__ définit l'addition entre l'instance self et l'instance autre et retourne une instance de la classe nom_classe.

Le programme suivant reprend le précédent de manière à ce que l'addition de deux nombres complexes suive dorénavant une syntaxe correcte et naturelle.

Entrée [51]:
```
from math import sqrt

class Complexe:
    # Constructeur
    def __init__(self, a = 0, b = 0):
        self.a = a
        self.b = b
    # Méthodes
    def module(self):
        return sqrt((self.a)**2 + (self.b)**2)
    def __add__(self, z):
        return Complexe(self.a + z.a, self.b + z.b)
```

Entrée [52]:
```
z1 = Complexe(0, 1)
```

Entrée [53]:
```
z2 = Complexe(1, 0)
```

Entrée [54]:
```
z3 = z1 + z2
print(z3.a,z3.b)
```

```
1 1
```

Chaque opérateur possède sa méthode-clé associée. L'opérateur += , différent de + est associé à la méthode-clé __iadd__.

```
class nom_class :
    def __iadd__ (self, autre) :
        # corps de l'opérateur
        return self
```

nom_classe est une classe. L'opérateur __iadd__ définit l'addition entre l'instance self et l'instance autre. L'instance self est modifiée pour recevoir le résultat. L'opérateur retourne invariablement l'instance modifiée self.

On peut ainsi étoffer la classe Complexe à l'aide de l'opérateur __iadd__.

Entrée [55]:
```
from math import sqrt

class Complexe:
    # Constructeur
    def __init__(self, a = 0, b = 0):
        self.a = a
        self.b = b
    # Méthodes
    def module(self):
        return sqrt((self.a)**2 + (self.b)**2)
    def __add__(self, z):
        return Complexe(self.a + z.a, self.b + z.b)
    def __iadd__(self, z):
        self.a += z.a
        self.b += z.b
        return self
```

Entrée [56]:
```
z1 = Complexe(0, 1)
```

Entrée [57]:
```
z2 = Complexe(1, 0)
```

Entrée [58]:
```
z1 += z2
print(z1.a,z1.b)
```

```
1 1
```

Un autre opérateur souvent utilisé est __str__ qui permet de redéfinir l'affichage d'un objet lors d'un appel à l'instruction print.

```
class nom_class :
    def __str__ (self) :
        # corps de l'opérateur
        return...
```

nom_classe est une classe. L'opérateur __str__ construit une chaîne de caractères qu'il retourne comme résultat de façon à être affiché.

Entrée [59]:
```
from math import sqrt

class Complexe:
    # Constructeur
    def __init__(self, a = 0, b = 0):
        self.a = a
        self.b = b
    # Méthodes
    def module(self):
        return sqrt((self.a)**2 + (self.b)**2)
    def __add__(self, z):
        return Complexe(self.a + z.a, self.b + z.b)
    def __iadd__(self, z):
        self.a += z.a
        self.b += z.b
        return self
    def __str__(self):
        if self.b == 0:
            return "%f" % (self.a)
        elif self.b > 0:
            return "%f + %f i" % (self.a, self.b)
        else:
            return "%f - %f i" % (self.a, -self.b)
```

Entrée [60]:
```
z1 = Complexe(0, 1)
```

Entrée [61]:
```
z2 = Complexe(1, 0)
```

Entrée [62]:
```
z3 = z1 + z2
print(z3)
```

```
1.000000 + 1.000000 i
```

Il existe de nombreux opérateurs qu'il est possible de définir.

`__int__(self)`, `__float__(self)`, `__complex__(self)`	Ces opérateurs implémente la conversion de l'instance `self` en entier, réel ou complexe.
`__add__(self,x)`, `__div__(self,x)`, `__mul__(self,x)` `__sub__(self,x)`, `__pow__(self,x)`, `__lshift__(self, x)`, `__rshift__(self, x)`	Opérateurs appelés pour les opérations `+`, `/`, `*`, `-`, `**`, `<`, `<`
`__iadd__(self,x)`, **`__idiv__(self,x)`**, **`__imul__(self,x)`**, `__isub__(self,x)`, `__ipow__(self,x)`, `__ilshift__(self, x)` `__irshift__(self, x)`	Opérateurs appelés pour les opérations `+=`, `/=`, `*=`, `-=`, `**=`, `<<=`, `>>=`

La liste complète des opérateurs Python est accessible à Operators.

3.5. Le polymorphisme

"Polymorphisme" signifie littéralement "plusieurs formes". Dans le cadre de la programmation orientée objet, le polymorphisme est un concept qui fait référence à la capacité d'une variable, d'une fonction / méthode ou d'un objet à prendre plusieurs formes, c'est-à-dire à sa capacité de posséder plusieurs définitions différentes.

Pour bien comprendre le concept de polymorphisme, imaginons qu'on définisse une classe nommée Animaux qui possède des fonctions comme seNourrir(), seDeplacer(), etc.

Entrée [63]:
```
class Animaux:
    def __init__(self, nom):
        self.animal_nom = nom
    def seNourrir(self):
        pass
    def seDeplacer(self):
        pass
```

Entrée [64]:
```
class Chien(Animaux):
    def seDeplacer(self):
        print("Courir")
class Oiseau(Animaux):
    def seDeplacer(self):
        print("Voler")
class Poisson(Animaux):
    def seDeplacer(self):
        print("Nager")
```

Entrée [65]:
```
animal1 = Chien("Rotweiler")
animal1.seDeplacer()
animal2 = Oiseau("Aigle")
animal2.seDeplacer()
animal3 = Poisson("Requin")
animal3.seDeplacer()
```

```
Courir
Voler
Nager
```

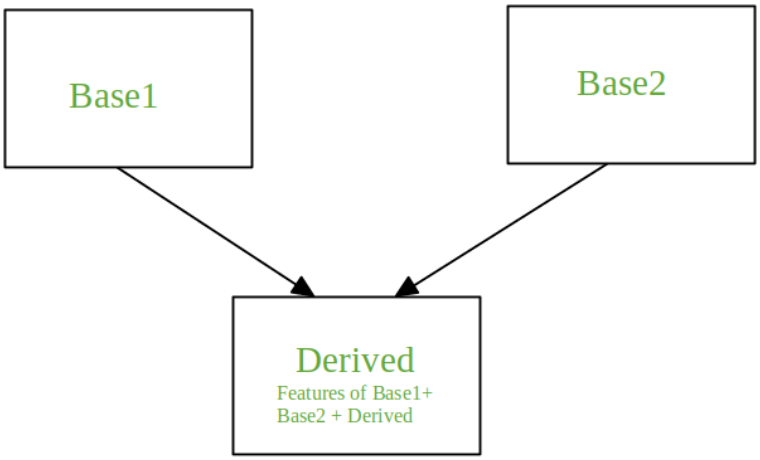
Les trois sous-classes Chien , Oiseau et Poisson vont par défaut hériter des membres de leur classe mère Animaux et notamment de la méthode seDeplacer(). Ici, chacune des sous-classes va implémenter cette méthode différemment. Ceci est un exemple typique de polymorphisme : plusieurs sous-classes héritent d'une méthode d'une classe de base qu'ils implémentent de manière différente.

Remarque :
Le polymorphisme permet également d'obtenir un code plus clair, plus lisible et plus cohérent : on va pouvoir par exemple fournir des définitions de fonctions vides dans une classe de base afin de laisser des sous-classes implémenter (définir) ces fonctions de différentes manières.

3.6. Héritage multiple

L'héritage est le mécanisme permettant de réutiliser le code (on parle de factorisation) car une classe (classe enfant) peut dériver les propriétés d'une autre classe (classe parent). Il fournit également la transitivité, c.-à-d. si la classe **Daughter** hérite de **Mother**, alors toutes les sous-classes de **Daughter** hériteraient également de **Mother**.

Lorsqu'une classe est dérivée de plusieurs classes de base, elle est appelée héritage multiple. La classe dérivée hérite de toutes les fonctionnalités du scénario de base.



Syntax:

```
Class Base1:
       Body of the class

Class Base2:
     Body of the class

Class Derived(Base1, Base2):
     Body of the class
```

4. A vous de jouer …

4.1. Du rectangle au parallélépipède

Q18. Ecrire en langage Python une classe **Rectangle**, permettant d'instancier un rectangle doté des attributs longueur et largeur.
Q19. Créer une méthode **perimetre()** permettant de calculer le périmètre d'un rectangle et une méthode **surface()** permettant de calculer sa surface.

Exemple d'affichage pour Q18 et Q19 :

```
Entrée [67]: rectangle1 = Rectangle(10,5)

Entrée [68]: print(rectangle1)

Rectangle (10 cm x 5 cm) : Surface de 50 cm2 ; Périmètre de 30 cm
```

Q20. Créer une classe fille Parallelepipede héritant de la classe Rectangle et dotée en plus d'un attribut hauteur et d'une méthode volume() permettant de calculer le volume d'un parallélépipède.

Exemple d'affichage pour Q20 :

```
Entrée [70]: parallelepipede1 = Parallelepipede(10,5,7)

Entrée [71]: print(parallelepipede1)

Parallélépipède (10 cm x 5 cm x 7 cm) : Surface de base de 50 cm2 ; Hauteur de 7 cm ; Volume de 350 cm3
```

4.2. Compte bancaire

Q21. Créer une classe Python nommée CompteBancaire qui représente un compte bancaire ayant pour attributs :

- numeroCompte
- prenom
- nom
- solde

Pour cela on créera un constructeur ayant comme paramètres ceux référencés ci-dessus.

Exemple d'affichage :

```
Entrée [73]: compte1 = CompteBancaire(7777777, "John", "DOE", 10000)

Entrée [74]: print(compte1)

          N° compte : 7777777    | Prénom : John         | NOM : DOE     | Solde : 10000
```

Q22. Créer les méthodes suivantes et s'assurer de leur bon fonctionnement :

- méthode versement() qui gère les versements ;
- méthode retrait() qui gère les retraits ;

4.3. Mini-jeu de combat (Ready player one ?)

Q23. Créer puis tester en Python une classe **Player** ayant les caractéristiques suivantes :

> Attributs :

- pseudo — pseudo du personnage incarné par le joueur
- health — points de vie du joueur
- attack — points d'attaque du joueur

> Méthodes :

- get_pseudo — retourne la valeur de l'attribut pseudo
- get_health — retourne la valeur de l'attribut health
- get_attack — retourne la valeur de l'attribut attack
- damage — inflige un nombre de points de dégâts
- attack_player — permet à player1 d'attaquer player2 en lui infligeant des dégâts
- __str(self)__ — méthode spéciale affichant les attributs d'un joueur via print(instance)

Exemple : 2 instances de la classe **Player** combattant l'une contre l'autre

```
Entrée [78]: player1 = Player("Obi-Wan",30,5)
             player2 = Player("Anakin",40,10)

Entrée [79]: print(player1)
             print(player2)

          Player : Obi-Wan        | Points de vie : 30    | Points d'attaque : 5
          Player : Anakin         | Points de vie : 40    | Points d'attaque : 10

Entrée [80]: player2.attack_player(player1)

          Anakin attaque Obi-Wan

Entrée [81]: print(player1)
             print(player2)

          Player : Obi-Wan        | Points de vie : 20    | Points d'attaque : 5
          Player : Anakin         | Points de vie : 40    | Points d'attaque : 10

Entrée [82]: player1.attack_player(player2)

          Obi-Wan attaque Anakin

Entrée [83]: print(player1)
             print(player2)

          Player : Obi-Wan        | Points de vie : 20    | Points d'attaque : 5
          Player : Anakin         | Points de vie : 35    | Points d'attaque : 10
```

Q24. Créer puis tester en Python une classe **Weapon** ayant les caractéristiques suivantes :

> Attributs :

- name — nom de l'arme
- damage — nombre de points de dégâts infligés par l'arme

> Méthodes :

- get_name — retourne le nom de l'arme
- get_damage_amount — retourne le nombre de points de dégâts infligés par l'arme
- __str(self)__ — méthode spéciale affichant les attributs de l'arme via print(instance)

Exemple : instanciation de 3 instances de la classe **Weapon**

```
Entrée [85]: weapon1 = Weapon("Poing",1)
             print(weapon1)
Arme : Poing    | Dégâts infligés : 1

Entrée [86]: weapon2 = Weapon("Pied",2)
             print(weapon2)
Arme : Pied     | Dégâts infligés : 2

Entrée [87]: weapon3 = Weapon("Sabre",10)
             print(weapon3)
Arme : Sabre    | Dégâts infligés : 10
```

Q25. Modifier et adapter la classe **Player** de manière à ce qu'un joueur puisse utiliser une arme instanciée par la classe **Weapon**. Pour cela créer dans la classe **Player** :

> Attribut :

weapon — par défaut self.weapon = None

> Méthodes :

set_weapon — change l'arme du joueur

has_weapon — vérifie si un joueur à une arme

Modifier en conséquence la méthode attack_player de la classe **Player** pour que si un joueur à une arme, les points de dégâts infligés par l'arme s'ajoutent aux points d'attaque du joueur.

Exemple : 2 instances de la classe **Player** combattant l'une contre l'autre avec une arme

```
Entrée [92]: player1 = Player("Obi-Wan",30,5)
             player2 = Player("Anakin",40,10)
             print(player1)
             print(player2)
             weapon1 = Weapon("Sabre Jedi",10)
             weapon2 = Weapon("Sabre Sith",15)
             player1.set_weapon(weapon1)
             player2.set_weapon(weapon2)
             player2.attack_player(player1)
             print(player1)
             print(player2)
             player1.attack_player(player2)
             print(player1)
             print(player2)
Player : Obi-Wan        | Points de vie : 30    | Points d'attaque : 5
Player : Anakin         | Points de vie : 40    | Points d'attaque : 10
Anakin attaque Obi-Wan
Player : Obi-Wan        | Points de vie : 5     | Points d'attaque : 5
Player : Anakin         | Points de vie : 40    | Points d'attaque : 10
Obi-Wan attaque Anakin
Player : Obi-Wan        | Points de vie : 5     | Points d'attaque : 5
Player : Anakin         | Points de vie : 25    | Points d'attaque : 10
```

Q26. Créer une classe **Warrior** qui est une classe fille de **Player**. La sous-classe **Warrior** contiendra l'attribut supplémentaire armor dont la valeur correspondra à des points d'armures. A chaque attaque d'un adversaire, une instance de la classe **Warrior** perdra 1 point d'armure. Une fois les points d'armures épuisés, les points de vie de l'instance de la classe **Warrior** seront décrémentés.

> Attributs :

- pseudo — pseudo du personnage incarné par le joueur
- health — points de vie du joueur
- attack — points d'attaque du joueur
- armor — points d'armure (nouveau attribut)

Attributs de la classe mère **Player**

> Méthodes

- damage — à adapter (surcharger)
- blade — réinitialise les points d'armure
- __str(self)__ — à adapter (surcharger)

Exemple : 1 instance Warrior et 1 instance Player combattent l'une contre l'autre.

```
Entrée [91]: warrior1 = Warrior("Stormtrooper", 10, 1, 5 )
             print(warrior1)
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 5

Entrée [92]: player3 = Player("Han Solo",15,5)
             print(player3)
Player : Han Solo       | Points de vie : 15    | Points d'attaque : 5

Entrée [93]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 4

Entrée [94]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 3

Entrée [95]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 2

Entrée [96]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 1

Entrée [97]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 10    | Points d'attaque : 1   | Armure : 0

Entrée [98]: player3.attack_player(warrior1)
             print(warrior1)
Han Solo attaque Stormtrooper
Player : Stormtrooper   | Points de vie : 5     | Points d'attaque : 1   | Armure : 0

Entrée [99]: warrior1.blade(5)
             print(warrior1)
Armure rechargée
Player : Stormtrooper   | Points de vie : 5     | Points d'attaque : 1   | Armure : 5
```