4 : Relations entre objets

Maintenant que nous avons créé des objets, la prochaine étape est la gestion des interactions entre ces objets.
Nous allons voir les différentes façons avec lesquelles un objet peut interagir avec un autre objet, qu'il soit de la même classe ou non.

Remarque : Dans tous les cas, les objets vont utiliser leur **interface publique** pour interagir entre eux. Il est donc primordial que cette interface publique (les variables non précédées de __) soit claire et bien documentée.

Nous allons décrire quatre types de relations :

- les **interactions** entre objets de même classe,
- les **associations simples** (utilisation d'une classe dans une autre),
- la **composition** (une classe est conceptuellement incluse dans une autre),
- et l'**agrégation** (deux classes sont fortement inter-dépendantes).

4.1 : Interactions entre objets de même classe

Dans l'utilisation d'une classe, il est habituel d'implémenter des interactions entre instances de cette même classe. D'ailleurs, nous en avons déjà implémenté sans le mentionner explicitement !
On peut par exemple :

- modifier l'instance courante ou renvoyer une nouvelle instance via une méthode,
- passer une instance de la classe en argument d'une méthode.

Exemple de modification de l'instance courante et de renvoi d'une nouvelle instance, avec la classe `complexe` :

In [21]:

```
class complexe:
    def __init__(self, reel, imag):
        self.__reel = reel
        self.__imag = imag

    def rotation_90(self):
        # modification de l'instance courante
        old_reel = self.__reel
        self.__reel = - self.__imag
        self.__imag = old_reel

    def conjugué(self):
        # renvoi d'une nouvelle instance
        return complexe(self.__reel, - self.__imag)

    def __str__(self):
        return ("partie réelle = " + str(self.__reel) +
                " ; partie imaginaire = " + str(self.__imag))
```

In [22]:

```
z = complexe(2, 1)
print(z)
z.rotation_90()
print(z)
z_bar = z.conjugué()
print(z_bar)
```

```
partie réelle = 2 ; partie imaginaire = 1
partie réelle = -1 ; partie imaginaire = 2
partie réelle = -1 ; partie imaginaire = -2
```

Exemple de passage d'une instance de classe en argument d'une méthode de la classe, avec la classe `voiture` :

In [24]:

```
class voiture:
    def __init__(self, marque, modèle):
        self.__marque = marque
        self.__modèle = modèle
        self.__est_accidentée = False

    def get_marque(self):
        return self.__marque

    def get_modèle(self):
        return self.__modèle

    def accident(self, other):
        print("accident entre une ", self.__marque, self.__modèle,
              " et une ", other.get_marque(), other.get_modèle(), " !")
        self.__est_accidentée = True
        other.__est_accidentée = True

    def print_état(self):
        if self.__est_accidentée:
            print("la ", self.__marque, self.__modèle, " est accidentée :(")
        else:
            print("la ", self.__marque, self.__modèle, " n'est pas accidentée :)")
```

In [25]:

```
peugeot_108 = voiture("peugeot", "108")
renault_clio = voiture("renault", "clio")
peugeot_108.print_état()
renault_clio.print_état()
peugeot_108.accident(renault_clio)
peugeot_108.print_état()
renault_clio.print_état()
```

```
la  peugeot 108  n'est pas accidentée :)
la  renault clio  n'est pas accidentée :)
accident entre une  peugeot 108  et une  rena
ult clio  !
la  peugeot 108  est accidentée :(
la  renault clio  est accidentée :(
```

4.2 : Associations simples

On parle d'**association simple** lorsqu'une classe en utilise une autre, en particulier dans son comportement.
C'est une définition très générale, qui peut généralement s'exprimer en disant **"utilise un"**.

Exemples : On parlera d'association simple lorsqu'une classe :

- crée une instance d'une autre classe ou une autre instance d'elle-même,
- renvoie une instance d'une autre classe dans une méthode,
- a besoin d'un argument venant d'une autre classe dans une de ses méthodes.

Exemple de création d'instance, avec la classe `voiture` :

In [27]:

```
class voiture:
    def __init__(self, marque, modèle):
        self.__marque = marque
        self.__modèle = modèle

    def __str__(self):
        return "marque : " + self.__marque + " ; modèle : " + self.__modèle


peugeot_208 = voiture("peugeot", "208")
renault_clio = voiture("renault", "clio")
print(peugeot_208)
print(renault_clio)
```

```
marque : peugeot ; modèle : 208
marque : renault ; modèle : clio
```

In [28]:

```
class usine:
    def __init__(self, marque):
        self.__marque = marque

    def construire(self, modèle):
        return voiture(self.__marque, modèle)


usine_peugeot = usine("peugeot")
peugeot_208 = usine_peugeot.construire("208")
peugeot_308 = usine_peugeot.construire("308")
print(peugeot_208)
print(peugeot_308)
```

```
marque : peugeot ; modèle : 208
marque : peugeot ; modèle : 308
```

Ici, la classe `usine` utilise la classe `voiture` : c'est un cas d'association simple. L'intérêt de cette approche est de n'avoir besoin que du modèle de la voiture, et non de la marque, pour la créer : en effet, l'usine ne va construire que des voitures d'une marque donnée.

Exemple mathématique d'un argument venant d'une autre classe, avec la classe `complexe` :

In [30]:

```
class complexe:
    def __init__(self, module, argument):
        self.__module = module
        self.__argument = argument % (2 * np.pi)  # % représente le modulo

    def get_module(self):
        return self.__module

    def get_argument(self):
        return self.__argument

    def __str__(self):
        argument_degrés = self.__argument * 180 / np.pi
        return ("module = " + str(self.__module) +
                " ; argument (°) = " + str(argument_degrés))


z1 = complexe(1, 4.3)
z2 = complexe(2.7, 15.6)
print(z1)
print(z2)
```

```
module = 1 ; argument (°) = 246.37185190625422
module = 2.7 ; argument (°) = 173.81416040408422
```

In [31]:

```
class rotation_2d:
    def __init__(self, angle):
        self.__angle = angle

    def appliquer(self, z):
        nouvel_angle = z.get_argument() + self.__angle
        return complexe(z.get_module(), nouvel_angle)


z1 = complexe(1, 0)
print(z1)
rotation_135 = rotation_2d(3 * np.pi / 4)
z2 = rotation_135.appliquer(z1)
print(z2)
rotation_270 = rotation_2d(3 * np.pi / 2)
z3 = rotation_270.appliquer(z2)
print(z3)
```

```
module = 1 ; argument (°) = 0.0
module = 1 ; argument (°) = 135.0
module = 1 ; argument (°) = 45.0
```

Encore une fois, on parle ici d'association simple car la méthode `appliquer` de la classe `rotation_2d` prend un argument de type `complexe`, i.e. une instance de la classe `complexe`.
Ensuite, les méthodes publiques `get_module` et `get_argument` de la classe `complexe` sont utilisées.

4.3 : Agrégation et composition

On parle d'**agrégation** lorsque plusieurs classes sont inter-dépendantes, mais peuvent exister séparément.
On parle de **composition** lorsqu'une classe est complètement dépendante d'une autre, i.e. n'a pas de raison d'être hormis comme attribut d'une autre classe.
En général, on parlera d'agrégation ou de composition lorsque la relation entre deux classes peut être résumée par **"possède un"**.

Intérêt : Ces paradigmes permettent de décomposer la définition d'un objet complexe en plusieurs définitions d'objets plus simples.
Par exemple, on pourra créer des classes correspondant aux roues d'une voiture, à ses portes, à son réservoir, etc., plutôt que de mettre tous ces attributs dans la classe `voiture`.
La classe `voiture` en sera ainsi simplifiée.

4.3.1 : Exemple de **composition**, avec la classe `voiture`

In [10]:

```
class porte:
    def __init__(self, type):
        self.__type = type

    def __str__(self):
        return "porte de type " + self.__type

    def __del__(self):
        print("porte de type ", self.__type, "détruite")

class réservoir:
    def __init__(self, capacité):
        self.__capacité = capacité

    def __str__(self):
        return "réservoir de capacité " + str(self.__capacité) + "L"

    def __del__(self):
        print("réservoir de capacité ", str(self.__capacité), "L détruit")
```

In [40]:

```
class voiture:
    def __init__(self, nb_portes, capacité_réservoir):
        self.créer_portes(nb_portes)
        self.créer_réservoir(capacité_réservoir)
    def __str__(self):
        return ("voiture avec " + str(self.__nb_portes) +
                " portes, un réservoir de " + str(self.__capacité_réservoir) + "L")

    def créer_réservoir(self, capacité_réservoir):
        self.__capacité_réservoir = capacité_réservoir
        self.__réservoir = réservoir(capacité_réservoir)
    def afficher_réservoir(self):
        print(self.__réservoir)

    def créer_portes(self, nb_portes):
        self.__nb_portes = nb_portes
        if nb_portes == 3:
            self.__portes = [porte("avant"), porte("avant"), porte("coffre")]
        elif nb_portes == 5:
            self.__portes = [porte("avant"), porte("avant"),
                             porte("arrière"), porte("arrière"), porte("coffre")]
    def afficher_portes(self):
        print("la voiture a ", self.__nb_portes, " portes :")
        for porte in self.__portes:
            print(porte)
```

In [41]:

```
renault_twingo = voiture(3, 35)
print(renault_twingo)
renault_twingo.afficher_portes()
renault_twingo.afficher_réservoir()
```

```
porte de type  coffre détruite
porte de type  avant détruite
porte de type  avant détruite
réservoir de capacité  35 L détruit
voiture avec 3 portes, un réservoir de 35L
la voiture a  3  portes :
porte de type avant
porte de type avant
porte de type coffre
réservoir de capacité 35L
```

In [42]:

```
peugeot_508 = voiture(5, 62)
print(peugeot_508)
peugeot_508.afficher_portes()
peugeot_508.afficher_réservoir()
```

```
voiture avec 5 portes, un réservoir de 62L
la voiture a  5  portes :
porte de type avant
porte de type avant
porte de type arrière
porte de type arrière
porte de type coffre
réservoir de capacité 62L
```

Dans ce cas, les classes `réservoir` et `porte` dépendent de la classe `voiture` via une relation de composition.
En effet, les instances de ces classes ne sont créées qu'à l'intérieur de la classe `voiture`.

Si on détruit l'instance de `voiture`, les instances de `réservoir` et `porte` seront aussi détruites.

In [43]:

```
peugeot_508 = None
```

```
porte de type  coffre détruite
porte de type  arrière détruite
porte de type  arrière détruite
porte de type  avant détruite
porte de type  avant détruite
réservoir de capacité  62 L détruit
```

4.3.2 : Exemple d'**agrégation**, avec les classes `complexe` et `vecteur_2d`

On va commencer par définir une classe `complexe` et une classe `vecteur_2d`, puis on rajoutera des méthodes de conversion entre ces deux classes.

In [44]:

```
class complexe:
    def __init__(self, module, argument):
        self.__module = module
        self.__argument = argument % (2 * np.pi)
        self.__reel = self.calcul_reel()
        self.__imag = self.calcul_imag()

    def calcul_reel(self):
        return self.__module * np.cos(self.__argument)

    def calcul_imag(self):
        return self.__module * np.sin(self.__argument)
```

In [45]:

```
class vecteur_2d:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

Maintenant que nous avons les deux classes `complexe` et `vecteur_2d`, nous allons leur ajouter des méthodes :

- une méthode `to_vecteur_2d` dans la classe `complexe`, qui renverra un vecteur de $\mathbb{R}^2$ à partir des parties réelle et imaginaire d'un nombre complexe,
- une méthode `to_complexe` dans la classe `vecteur_2d`, qui renverra un complexe à partir des coordonnées d'un vecteur de $\mathbb{R}^2$.

In [46]:

```
class complexe:
    def __init__(self, module, argument):
        self.__module = module
        self.__argument = argument % (2 * np.pi)
        self.__reel = self.calcul_reel()
        self.__imag = self.calcul_imag()

    def calcul_reel(self):
        return self.__module * np.cos(self.__argument)

    def calcul_imag(self):
        return self.__module * np.sin(self.__argument)

    def to_vecteur_2d(self):
        return vecteur_2d(self.__reel, self.__imag)

    def __str__(self):
        argument_degrés = self.__argument * 180 / np.pi
        return ("module = " + str(self.__module) +
                " ; argument (°) = " + str(argument_degrés))
```

In [47]:

```
class vecteur_2d:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def to_complexe(self):
        module = np.sqrt(self.__x**2 + self.__y**2)
        if self.__x > 0:
            argument = np.arctan(self.__y / self.__x)
        else:
            argument = - np.pi
        return complexe(module, argument)

    def __str__(self):
        return ("coordonnées du vecteur : x = " + str(self.__x) +
                ", y = " + str(self.__y))
```

In [48]:

```
z = complexe(1.2, np.pi / 2)
print(z)
V = z.to_vecteur_2d()
print(V)

print("")

z = complexe(6.3, 3 * np.pi / 4)
print(z)
V = z.to_vecteur_2d()
print(V)
```

```
module = 1.2 ; argument (°) = 90.0
coordonnées du vecteur : x = 7.34788079488411
9e-17, y = 1.2

module = 6.3 ; argument (°) = 135.0
coordonnées du vecteur : x = -4.4547727214752
49, y = 4.454772721475249
```

In [49]:

```
V = vecteur_2d(- 2.3, 0)
print(V)
z = V.to_complexe()
print(z)

print("")

V = vecteur_2d(3.2, -3.2)
print(V)
z = V.to_complexe()
print(z)
```

```
coordonnées du vecteur : x = -2.3, y = 0
module = 2.3 ; argument (°) = 180.0

coordonnées du vecteur : x = 3.2, y = -3.2
module = 4.525483399593905 ; argument (°) = 3
15.0
```

Ici, les deux classes `complexe` et `vecteur_2d` interagissent fortement entre elles grâce aux méthodes de conversion. En effet, mathématiquement, un nombre complexe et un vecteur de $\mathbb{R}^2$ représentent le même objet.
On parle donc ici d'**agrégation**, car ces deux classes peuvent exister indépendamment l'une de l'autre, mais qu'elles sont tout de même fortement liées.

4.4 : Architecture MVC

Pour les longs codes, l'architecture MVC (Modèle, Vue, Contrôleur), rendue possible par la programmation orientée objet, peut nous aider.
On va distinguer 3 type d'objets ou méthodes, en fonction de s'ils sont de type Modèle, Contrôleur ou Vue.

- Les objets appartenant au **Modèle** : ce sont les conteneurs de données. Ils n'ont pas de méthodes associées, à part des méthodes accès (section suivante), ils sont donc très facile à sérialiser. Par exemple, la classe `voiture`.
- Les classes de type **Contrôleur** : ces classes permettent de modifier et d'utiliser les "modèles". Par exemple, une classe `démarrage` ou `réparation` pour la voiture,
- Les classe de type **Vue** : elles gèrent les entrées/sorties, notamment graphiques.

Plus simplement, on met les données dans une classe, et les actions sur ces données (en général, des méthodes) dans une autre classe.

Exemple de géométrie et d'algèbre linéaire: un droite (sous espace vectoriel de $\mathbb{R}^2$) et ses transformations linéaires.

In [25]:

```
# modèle

class droite():
    def __init__(self, a, b):  # equation ax + by = 0 d'une droite passant par l'origine
        self.a = a
        self.b = b
        if self.b == 0:
            raise ValueError("b ne doit pas être nul dans le constructeur de la classe droite")
        self.base = [1.0, -self.a / self.b]

    def point(self):  # génère un point sur la droite
        alpha = np.random.uniform(-10, 10)
        x = self.base[0] * alpha
        y = self.base[1] * alpha
        return x, y
```

In [26]:

```
# contrôleur

class rotation():
    def __init__(self, angle):
        self.angle = angle

    def applique(self, d):
        t = self.angle
        new_base = [
            np.cos(t) * d.base[0] - np.sin(t) * d.base[1],
            np.sin(t) * d.base[0] + np.cos(t) * d.base[1]
        ]
        new_droite = droite(1.0, -new_base[1] / new_base[0])
        return new_droite
```

In [27]:

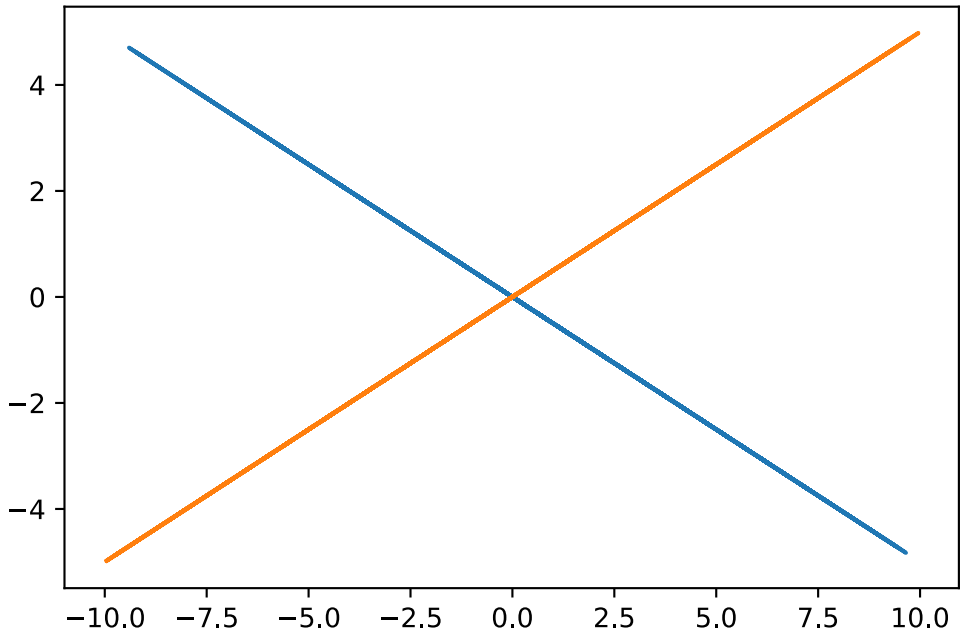
```
# vue

import matplotlib.pyplot as plt


class plot_droite():
    def plot(self, d):
        x = np.zeros(100)
        y = np.zeros(100)
        for i in range(0, 100):
            x[i], y[i] = d.point()
        plt.plot(x, y)
```

In [28]:

```
dr = droite(1.0, 2.0)
ro = rotation(np.pi / 2.0)
dr2 = ro.applique(dr)
p = plot_droite()
p.plot(dr)
p.plot(dr2)
```



4.5 : TD

4.5.1 : Exercice 1 - Agrégation entre une classe `Polynôme` et une classe `Complexe`

En repartant de la classe `Polynôme` du TD numéro 2, on souhaite écrire une classe pour représenter un polynôme a coefficients complexes.
On rappelle ci-dessous la classe `Polynôme`. Que faut-t-il modifier dans cette classe ? Quelles méthodes et surcharges faut-t-il implémenter dans la classe `Complexe` ?

In [3]:

```
class Polynome:

    def __init__(self):
        self.__deg = 0
        self.__coeffs = []

    def get_deg(self):
        return self.__deg

    def __set_deg(self, p):
        self.__deg = p

    def get_coeffs(self):
        return self.__coeffs

    def get_coeff(self, i):
        return self.__coeffs[i]

    def __set_coeffs(self, list_co):
        self.__coeffs = list_co

    def set_coeff(self, i, x):
        if i > 0 and i < self.__deg + 1:
            self.__coeffs[i] = x
        else:
            print("problème")
```

```
    def __str__(self):
        chain = str(self.get_coeff(0))
        for i in range(1, self.get_deg() + 1):
            chain = chain + " + " + str(self.get_coeff(i)) + " x**" + str(i)
        return chain

    def construction(self, q, list_co):
        if q + 1 == len(list_co):
            self.__set_deg(q)
            self.__set_coeffs(list_co)
        else:
            print("deg faux")

    def __convert(self, q):
        res = Polynome()
        list_new_co = [0.0 for i in range(0, q + 1)]
        list_new_co[0:self.get_deg() + 1] = self.get_coeffs()
        res.construction(q, list_new_co)
        return res

    def addition(self, other):
        res = Polynome()

        if self.get_deg() > other.get_deg():
            convert = other.__convert(self.get_deg())
            sum_coeffs = [self.get_coeff(i) + convert.get_coeff(i) for i in range(0, self.get_deg() + 1)]
            res.construction(self.get_deg(), sum_coeffs)

        elif self.get_deg() < other.get_deg():
            convert = self.__convert(other.get_deg())
            sum_coeffs = [other.get_coeff(i) + convert.get_coeff(i) for i in range(0, other.get_deg() + 1)]
            res.construction(other.get_deg(), sum_coeffs)
        else:
            sum_coeffs = [other.get_coeff(i) + self.get_coeff(i) for i in range(0, other.get_deg() + 1)]
            res.construction(other.get_deg(), sum_coeffs)

        return res

    def eval_p(self, x):
        res = 0.0
        for i in range(0, self.get_deg() + 1):
            res += self.get_coeff(i) * x**i
        return res
```

4.5.2 : Exercice 2 - Géométrie et architecture MVC

1) Écrire une classe `Triangle`. Quels sont les attributs de cette classe ? Proposer des méthodes et opérateurs potentiels.
2) Écrire une classe `Cercle`. Quels sont les attributs de cette classe ? Proposer des méthodes et opérateurs potentiels.
3) Écrire une classe `Affichage` qui permet de visualiser un ou deux objets géométriques.
4) Écrire une classe `Translation` qui translate un objet géométrique.