6 : Polymorphisme et héritage multiple

6.1: Polymorphisme

Le **polymorphisme** est un autre concept clé de la programmation.

Il permet d'utiliser une seule interface (fonction ou méthode) en fonction des types des objets.

Par exemple, la fonction *built-in* `len` est polymorphe : appliquée à des `str`, elle renvoie le nombre de caractères ; appliquée à des `list`, elle renvoie le nombre d'éléments de la liste.

In [1]:
```
print("résultat de la fonction len appliquée à 'coucou' : ", len("coucou"))
print("résultat de la fonction len appliquée à [1, 2, 'coucou'] : ", len([1, 2, 'coucou']
```

```
résultat de la fonction len appliquée à 'coucou' :  6
résultat de la fonction len appliquée à [1, 2, 'coucou'] :  3
```

6.1.1 : Polymorphisme en programmation fonctionnelle

Il est possible de rendre des fonction polymorphes, par exemple en utilisant des arguments optionnels ou en se basant sur le type des arguments.

Exemple : avec des arguments optionnels

In [2]:
```
def aire_rectangle(longueur, largeur = None):
    if largeur is None:  # sans largeur, on suppose que le rectangle est en fait un carre
        return longueur**2
    else:
        return longueur * largeur

print("aire_rectangle(2) = ", aire_rectangle(2))
print("aire_rectangle(2, 3) = ", aire_rectangle(2, 3))
```

```
aire_rectangle(2) =  4
aire_rectangle(2, 3) =  6
```

Exemple : en fonction du type des arguments (comme dans les fonctions *built-in* `len` , `sum` , etc.)

Ici, on va étendre la fonction factorielle aux réels : on définit la fonction Gamma

$$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t}\, dt,$$

et on admet que, pour tout $n \in \mathbb{N}$, on a $\Gamma(n+1) = n!$.

Si l'argument d'entrée x est entier, on renverra la valeur de $x!$; sinon, on renverra $\Gamma(x)$.

In [3]:
```
import numpy as np

def factorielle(x):
    if isinstance(x, int):
        return np.math.factorial(x)
    else:
        return np.math.gamma(x + 1)

print(factorielle(5))
print(factorielle(5.0001))
```

```
120
120.02047526743465
```

6.1.2 : Polymorphisme en programmation orientée objet

Les cas qui nous intéressent le plus relèvent de l'utilisation de polymorphisme en programmation orientée objet, c'est-à-dire dans les classes.

Exemple : une hiérarchie de classes représentant des véhicules

On commence par une classe générale `Véhicule`.

In [4]:
```
class Véhicule:

    def __init__(self):
        self.est_allumé = True
        self.__vitesse = 10

    def get_vitesse(self):
        return self.__vitesse

    def déplacement(self):
        if self.est_allumé and self.__vitesse > 0:
            print("Le véhicule se déplace")
```

In [5]:
```
v = Véhicule()
v.déplacement()
```

```
Le véhicule se déplace
```

On introduit ensuite le polymorphisme en spécialisant la méthode `déplacement` : d'abord, un véhicule maritime va naviguer.

In [6]:
```
class VéhiculeMaritime(Véhicule):

    def naviguer(self):
        print("Le véhicule maritime navigue")

    def déplacement(self):
        if self.est_allumé and self.get_vitesse() > 0:
            self.naviguer()
```

In [7]:
```
vm = VéhiculeMaritime()
vm.déplacement()
```

```
Le véhicule maritime navigue
```

Ensuite, un véhicule terrestre va rouler.

In [8]:
```
class VéhiculeTerrestre(Véhicule):

    def rouler(self):
        print("Le véhicule terrestre roule")

    def déplacement(self):
        if self.est_allumé and self.get_vitesse() > 0:
            self.rouler()
```

In [9]:
```
vt = VéhiculeTerrestre()
vt.déplacement()
```

```
Le véhicule terrestre roule
```

Et enfin, un véhicule aérien va voler.

In [10]:
```
class VéhiculeAérien(Véhicule):

    def voler(self):
        print("Le véhicule aérien vole")

    def déplacement(self):
        if self.est_allumé and self.get_vitesse() > 0:
            self.voler()
```

In [11]:
```
va = VéhiculeAérien()
va.déplacement()
```

```
Le véhicule aérien vole
```

On a donc trois classes filles, chacune représentant un type de véhicule différent, et chacune ayant sa propre méthode `déplacement`.

Question : Comment faire pour créer un véhicule qui soit à la fois maritime, terrestre et aérien ?

Réponse : Avec de l'héritage multiple !

6.2 : Introduction à l'héritage multiple

Jusqu'à présent, on a parlé d'héritage simple : on a une classe mère et une classe fille (ou plusieurs).

On peut étendre ces notions au cas de l'**héritage multiple** : une classe fille descendant de plusieurs classes mère.

Exemple : cas d'un hydravion

In [12]:
```
class Hydravion(VéhiculeAérien, VéhiculeMaritime, VéhiculeTerrestre):
    pass
```

In [13]:
```
hy = Hydravion()
hy.naviguer()
hy.rouler()
hy.voler()
hy.déplacement()
```

```
Le véhicule maritime navigue
Le véhicule terrestre roule
Le véhicule aérien vole
Le véhicule aérien vole
```

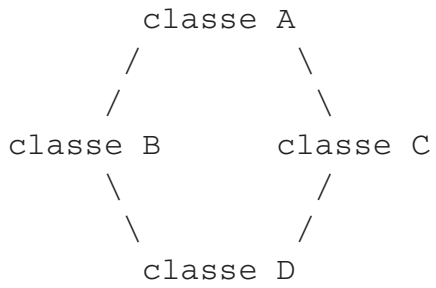
6.3 : *MRO* (Method Resolution Order)

On voit, dans l'exemple précédent, que la méthode `déplacement` de la classe `Hydravion` appelle celle de la classe `VéhiculeAérien`.

Ceci est dû au *MRO* (Method Resolution Order), un algorithme interne à Python qui va déterminer dans quelle classe mère regarder pour trouver les méthodes des classes filles.

Si une classe est déclarée avec `class Fille(Mère1, Mère2)`, les méthodes non déclarées dans `Fille` seront d'abord prises dans `Mère1` puis dans `Mère2`.

Exemple : cas simple de *MRO*, avec une forme en diamant



In [14]:

```
class A:
    def __init__(self):
        print("entrée dans A.__init__()")
        print("sortie  de  A.__init__()")

class B(A):
    def __init__(self):
        print("entrée dans B.__init__()")
        super().__init__()
        print("sortie  de  B.__init__()")

class C(A):
    def __init__(self):
        print("entrée dans C.__init__()")
        super().__init__()
        print("sortie  de  C.__init__()")

class D(B, C):
    def __init__(self):
        print("entrée dans D.__init__()")
        super().__init__()
        print("sortie  de  D.__init__()")
```

On peut afficher le *MRO* avec la méthode `mro()` (ou l'attribut `__mro__`).

On voit ci-dessous que les classes `B` et `C` héritent de `A`, et que `D` hérite de `B` puis `C`, qui héritent elles-mêmes de `A`.

In [15]:
```
print(B.mro())
print(C.mro())
print(D.__mro__)
```

```
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>)
```

Il y a des subtilités : lorsqu'on appelle le constructeur de la classe mère de `D` avec `super().__init__()`, on voit que les constructeurs de `B` et `C` sont appelés tous les deux, l'un après l'autre.

Par ailleurs, le constructeur de `A` n'est appelé qu'une seule fois alors qu'a priori, il devrait être appelé deux fois : par ceux des classes `B` et `C`. Ici, le *MRO* a compris que `A` était mère de `B` et de `C`, et qu'il n'y a donc besoin d'appeler le constructeur qu'une seule fois.

In [16]:
```
d = D()
```

```
entrée dans D.__init__()
entrée dans B.__init__()
entrée dans C.__init__()
entrée dans A.__init__()
sortie  de  A.__init__()
sortie  de  C.__init__()
sortie  de  B.__init__()
sortie  de  D.__init__()
```

Si on veut appeler directement le constructeur de `A` sans passer par celui de `C`, on doit le faire explicitement.

In [17]:

```
class A:
    def __init__(self):
        print("entrée dans A.__init__()")
        print("sortie  de  A.__init__()")

class B(A):
    def __init__(self):
        print("entrée dans B.__init__()")
        A.__init__(self)
        print("sortie  de  B.__init__()")

class C(A):
    def __init__(self):
        print("entrée dans C.__init__()")
        A.__init__(self)
        print("sortie  de  C.__init__()")

class D(B, C):
    def __init__(self):
        print("entrée dans D.__init__()")
        super().__init__()
        print("sortie  de  D.__init__()")
```

In [18]:

```
d = D()
```

```
entrée dans D.__init__()
entrée dans B.__init__()
entrée dans A.__init__()
sortie  de  A.__init__()
sortie  de  B.__init__()
sortie  de  D.__init__()
```

6.4 : TD

6.4.1 : Héritage simple et polymorphisme

On va s'intéresser à des nombres complexes particuliers : les complexes de module 1 et les imaginaires purs. On les représentera sous forme exponentielle.

1. Proposer une classe mère `Complexe` et des classes filles `Complexe1` et `ImaginairePur` . Quels sont leurs attributs ?
2. Proposer une surcharge de l'opérateur `*` pour la classe mère.
3. Modifier cette surcharge le cas échéant pour les deux classes filles.

6.4.2 : Héritage multiple : les polygones

On reprend le TD du cours 5 sur l'héritage. Cette fois, on va proposer de l'héritage multiple.

1. Proposer une hiérarchie de classes pour représenter des polygones. On s'intéressera notamment aux polygones réguliers centrés en zéro et dont un point est (1, 0).
2. Proposer des implémentations de ces classes. Pour la hiérarchie de quadrilatères, on pourra développer des méthodes pour les tailles des diagonales et l'angle qu'elles forment. Pour certains polygones particuliers, on pourra calculer périmètre et aire.