23_python_poo

December 15, 2023

1 Python Programation Orientée Objet

Python est un langage orienté objet. Cela signifie qu'il fournit tous les éléments classiques de la programmation orientée objet (POO) : les classes et les objets.

La POO est un paradigme de programmation qui permet de structurer un programme en combinant des briques de base appelées objets. Il s'agit de rassembler en un tout cohérent des données et les traitements qui s'y appliquent. La POO est un niveau d'abstraction supplémentaire qui permet de mieux gérer la complexité des programmes, tout comme les fonctions le sont pour les instructions.

1.1 Objet

Un objet est une entité qui regroupe des données et des traitements qui lui sont applicables. Un objet possède une identité, un état et un comportement.

1.1.1 Identité

L'identité d'un objet est un identifiant unique qui le distingue des autres objets. En Python, l'identité d'un objet est accessible via la fonction id().

1.1.2 Etat

L'état d'un objet est défini par ses données, appelées attributs. Les attributs d'un objet peuvent être de n'importe quel type Python : des types de base, des collections, des instances de classes, etc.

1.1.3 Comportement

Le comportement d'un objet est défini par ses méthodes. Une méthode est une fonction qui s'applique à l'objet et qui peut utiliser ses attributs.

1.1.4 Conclusion

Un objet est donc une structure de données qui regroupe des données et les traitements qui s'y appliquent. Les données sont représentées par les attributs et les traitements par les méthodes.

1.2 Classe

Les classe sont les modèles des objets (blueprint). Une classe définit la structure et le comportement des objets qui en sont issus.

1.2.1 Définition

Une classe est définie par le mot-clé class suivi du nom de la classe et de deux points. Le nom de la classe doit commencer par une majuscule.

1.2.2 Attributs

Les attributs d'une classe sont définis dans une méthode spéciale appelée constructeur. Le constructeur est une méthode qui porte le nom **init**(). Cette méthode est appelée automatiquement lors de la création d'un objet.

1.2.3 Méthodes

Les méthodes sont des fonctions qui s'appliquent aux objets. Elles sont définies dans la classe et prennent comme premier paramètre une référence à l'objet en cours de traitement. Par convention, ce paramètre s'appelle `self`.

1.3 Définition d'une classe

Une classe est définie par le mot-clé class suivi du nom de la classe et de deux points. Le nom de la classe doit commencer par une majuscule.

```
class Point:
    pass
```

1.4 Attributs

Les attributs d'une classe sont des variables propres a chaque objet. Il sont définis à l'intérieur de la classe.

```
class Point:
    x = 0
    y = 0
```

> **Note**: Les attributs en Python sont publics. Il n'y a pas de notion de visibilité (private, protected, public) comme en Java ou en C++. **Note**: Par convention, les attributs privés sont précédés d'un underscore.

[ ]:
```
class Point:
    # private variable
    _private = "I'm private by convetion"

    # class attribute
    x = 10
    y = 0
```

1.5 Les attributs/méthodes privées

En python, comme il n'y pas de notion de visibilité, il n'y a pas de variables / méthodes privées. Cependant, par convention, les attributs / méthodes privés sont précédés d'un underscore. Ces variables ne sont pas sensée être utilisées en dehors de la classe.

Parfois on peut voir des variables/methodes privée qui commencent avec deux underscores. Ces variables sont appelées “mangled”. Elles sont utilisées pour éviter les conflits de noms lors de l’héritage.

1.6 Creation d’un objet

Pour créer un objet, on utilise le nom de la classe suivi de parenthèses. Cela appelle le `constructeur` de la classe qui crée l’objet.

[ ]:
```
# Creation of two instances of the Point class
p1 = Point()
p2 = Point()

print(p1.x)
print(p1.y)
p1.x = 20
p1.y = 20
print(p1.x)
print(p1.y)

p1._private = "Not so private ..."
print(p1._private)
```

1.7 Acceder aux attributs d’un objet

En python, pour accéder aux attributs d’un objet, on utilise le point `.`

[ ]:
```
point1 = Point()
point2 = Point()
# Modification of attributes
point1.x = 5
point1.y = 4

point2.x = 3
point2.y = 6

print(Point.calc_dist(point1, point2))

# access attributes
print(f"P1 valeur en X={p1.x}; Y={p1.y}")
print(f"P2 valeur en X={p2.x}; Y={p2.y}")

p1._private = "Antonio"
print(p1._private)
```

1.8 Methods

Les méthodes sont des fonctions qui s’appliquent aux objets. Elles sont définies dans la classe et prennent comme premier paramètre une référence à l’objet en cours de traitement. Par convention,

ce paramètre s'appelle `self`.

[ ]:
```python
class Room:
    length = 0.0
    width = 0.0

    def calculate_perimeter(self):
        return 2*self.length+2*self.width

    def calculate_area(self):
        return self.length * self.width

    def print_area(self, name=""):
        print(self)
        print(name, self.calculate_area())

# Creation of an instance of the Room class
nomades_main_room = Room()
nomades_small_room = Room()

nomades_main_room.width = 40
nomades_main_room.length = 60
nomades_small_room.width = 23
nomades_small_room.lenght = 20

print(nomades_main_room)
print(nomades_main_room.calculate_perimeter())
print(nomades_small_room.calculate_perimeter())
nomades_main_room.print_area("Nomades main course room for pse")
```

1.9 Mot clé self

Le mot clé self est une convention en python. Il est utilisé pour faire référence à l'objet en cours de traitement. Il est passé automatiquement en premier paramètre de chaque méthode. Il est possible de le renommer mais il est fortement déconseillé de le faire. Le premier paramètre d'une méthode est toujours une référence à l'objet en cours de traitement `self`.

1.10 Constructor

Le constructeur est une méthode spéciale appelée `__init__()`. Cette méthode est appelée automatiquement lors de la création d'un objet. Elle permet d'initialiser les attributs de l'objet. Il est possible de donner des arguments au constructeur afin de pouvoir initialiser les attributs avec des valeurs différentes.

[ ]:
```python
class GeoPoint:
    def __init__(self, lat=0, lng=0):
        self._lat = lat
        self._lng = lng
```

```
        self._dist = lat - lng

    def set_lat(self, lat):
        if lat >= -90 and lat <= 90:
            self._lat = lat
        else:
            print("provide right latitude")
    def set_lng(self, lng):
        self._lng = lng

    def get_lat(self):
        return self._lat
    def get_lng(self):
        return self._lng

    def print_point(self):
        print(f"GeoPoint: ({self._lat}, {self._lng})")
    def __str__(self):
        return f"GeoPoint: ({self._lat}, {self._lng})"

# Creation of an instance of the GeoPoint class
p1 = GeoPoint(10, 20)
p1.set_lat(88)
print(p1)

p2 = GeoPoint(lng=12)
```

Ici, on définit un constructeur avec la fonction `__init__`. Le constructeur prend en arguments trois paramètres `self`, `lat`, `lng`, qui correspondent aux attributs de la classe. Le paramètre `self` est obligatoire et correspond à l'objet en cours de traitement. `self` est automatiquement passé lors de l'appel de la méthode et définis la classe actuelle.

les paramètres `lat` et `lng` sont optionnels. Si on ne les fournit pas, ils prennent la valeur par défaut 0. le constructeur les assigne aux attributs `self.lat` et `self.lng`. Ici, on utilise le mot clé `self` pour assigner les attributs a l'objet actuel `p1` oou `p_nomades`.

> **Note**: Les attributs `self.lat` et `self.lng` sont maintenant des attributs de la classe, même s'il ne sont pas définis dans la classe. Possibilité de les utilisé dans les autres fonction de classe (comme dans `print_point()`).

[ ]:
```
class GeoPointStr:
    def __init__(self, lat=0, lng=0):
        self.lat = lat
        self.lng = lng
    def __repr__(self):
        return f"Point: ({self.lat}, {self.lng})"

ps1 = GeoPointStr(49.19103, 6.13562)
```

```
print(p1)
print(ps1)
```

2 Python POO - Héritage

L'héritage est un mécanisme qui permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe hérite des attributs et des méthodes de la classe existante. On appelle la classe existante la classe mère ou la super-classe et la nouvelle classe la classe enfant ou la sous-classe.

2.1 Définition

Pour définir une classe enfant, on place le nom de la classe mère entre parenthèses après le nom de la classe enfant.

```
# define a superclass
class super_class:
    # attributes and method definition

# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

2.1.1 Exemple 1: Animal et chien

[ ]:
```
class Animal:
    # attribute and method of the parent class
    name = ""
    def eat(self):
        print("I can eat")
    def talk(self):
        print("Try to make some noise")
# inherit from Animal
class Dog(Animal):
    # new method in subclass
    def talk(self):
        # access name attribute of superclass using self
        print(self.name, "Woof")
    def walk(self):
        print("I walk to the park")
class Cat(Animal):
    def talk(self):
        print(self.name, "Meow")

# create an object of the subclass
labrador = Dog()
# access superclass attribute and method
```

```
labrador.name = "Rohu"
labrador.eat()
labrador.talk()

golden = Dog()
golden.name = "Max"
golden.walk()
```

2.2 is-a (est-un) relationship

L'héritage est utilisé pour modéliser une relation "est-un". De ce fait, on procède a un héritage seulement s'il y a une relation "est-un" entre la classe mère et la classe enfant.

- Un chien est un animal
- Une voiture est un véhicule
- Un carré est un rectangle
- Une pommme est un fruit

2.3 has-a (a-un) relationship

L'héritage n'est pas utilisé pour modéliser une relation "a-un". la realtion "a-un" est modélisée par la composition. On parle de composition lorsqu'un objet contient un autre objet.

- Un ordinateur a un processeur
- Un ordinateur a un disque dur
- Un ordinateur a une carte graphique

Ici, le processeur est un objet qui est attribut de l'objet ordinateur. On parle de composition.

2.3.1 Exemple 2: Polygone et triangle

Un polygone est une figure géométrique fermée plane d'au moin 3 cotés. On peut donc définir la classe `Polygon`:

[ ]:
```
class Polygon:
    def __init__(self, nb_of_sides):
        self.n = nb_of_sides
        self.sides = [0 for i in range(nb_of_sides)]
    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in␣
 ↪range(self.n)]
    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
    def info(self):
        print("I'm a polygon")

    def __repr__(self):
        return f"Polygone of {self.n} sides"
    def __str__(self):
```

```
        return self.__repr__()
```

Cette classe, a un atribut de classe `n` qui represente le nombre de côtés du polygone, ainsi qu'un attribut d'instance `sides` qui est une liste des longueurs des côtés du polygone.

Un `Triangle` est un polygone de 3 côtés. On peut donc définir la classe `Triangle` qui hérite de la classe `Polygon`. Grâce à l'héritage, la classe `Triangle` hérite de l'attribut de classe `n` et de l'attribut d'instance `sides` de la classe `Polygon`. On a pas besoin de les redéfinir (**Code reusability**) dans la classe `Triangle`.

La classe `Triangle` peut être définie comme suit :

[ ]:
```
class Rectangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,2)

    def findArea(self):
        super().inputSides()
        a, b = self.sides
        return a*b
    def info(self, parent=False):
        if parent:
            super().info()
        else:
            print("I'm a Rectangle")


class Square(Rectangle):
    def __init__(self):
        Polygon.__init__(self, 1)
        super().info(True)
    def findArea(self):
        return self.sides[0]**2
    def info(self):
        print("I'm a square, a special rectangle")
```

Nous avons définis la classe Triangle qui hérites de la classe Polygon. Le constructeur de la classe Triangle, appelle le constructeur de la classe Polygon pour initialiser les attributs de la classe Triangle. On peut accéder aux attributs/méthodes de la classe Polygon grâce au mot clé `super`. Il est nécéssaire ici d'appeler la classe parente, car on spécifie le nombre de coté du Triangle.

[ ]:
```
t = Square()
t.inputSides()
t.dispSides()
print(t.findArea())
t.info()
print(t)
```

Method overriding

Lors de l'héritage, il est possible de redéfinir les méthodes de la classe parente dans la classe enfant. On parle de `method overriding`. Cela permet de modifier le comportement d'une méthode héritée.

[ ]:
```python
class Animal:
    # attributes and method of the parent class
    name = ""
    def eat(self):
        print("I can eat")
# inherit from Animal
class Dog(Animal):
    # override eat() method
    def eat(self):
        print("I like to eat bones")
class Cat(Animal):
    pass
# create an object of the subclass
labrador = Dog()
# call the eat() method on the labrador object
labrador.eat()
c = Cat()
c.eat()
```

2.4 Le mot clé super

Si on a besoin d'utiliser une méthode de la classe parente, on peut utiliser le mot clé `super`. Ce mot clé désigne la classe parente.

[ ]:
```python
class Animal:
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):
    def possibilities(self):
        super().eat()

    # override eat() method
    def eat(self):
        # call the eat() method of the superclass using super()
        print("eat in dog")
        super().eat()

# create an object of the subclass
labrador = Dog()
labrador.eat()
```

2.5 Conclusion Héritage

L'héritage est un mécanisme qui permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe hérite des attributs et des méthodes de la classe existante. On appelle la classe existante la classe mère ou la super-classe et la nouvelle classe la classe enfant ou la sous-classe.

- L'héritage est utilisé pour modéliser une relation "est-un".
- L'hériage permet de réutiliser le code existant (classe parente).

3 Python POO - Héritage multiple

En python, une classe peut hériter de plusieurs classes. On parle d'héritage multiple. Pour définir une classe qui hérite de plusieurs classes, on place les noms des classes mères entre parenthèses après le nom de la classe enfant.

```
class sub_class(super_class1, super_class2, ...):
    # attributes and method of super_class1
    # attributes and method of super_class2
    # attributes and method of sub_class
```

3.1 Exemple 1: chien, animal et mammifère

Une Chauve-souris est un animal à ailes et un mammifère. On peut donc définir les classes `WingedAnimal` et `Mammal` et la classe `Bat` qui hérite de ces deux classes.

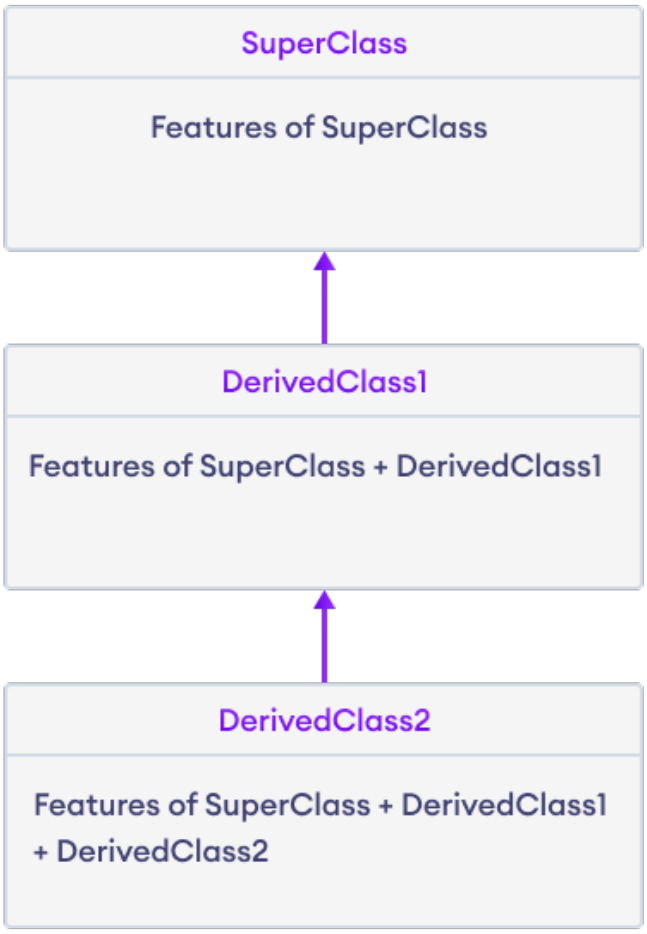
[ ]:
```
class WingedAnimal:
    def __init__(self, species):
        self.species = species

    def fly(self):
        print('{} flies'.format(self.species))

class Mammal:
    def __init__(self, species):
        self.species = species

    def feed_young_with_milk(self):
        print('{} feeds young with milk'.format(self.species))

class Bat(WingedAnimal, Mammal):
    def __init__(self):
        Mammal.__init__(self, "Bat")
        WingedAnimal.__init__(self, "Bat2")
```

[ ]:
```
b1 = Bat()
b1.fly()
b1.feed_young_with_milk()
```

4 Python POO - Multi-level inheritance

En python, une classe peut hériter d'une classe qui hérite elle-même d'une autre classe. On parle d'héritage multi-niveau.

```
class SuperClass:
    # Super class code here

class DerivedClass1(SuperClass):
    # Derived class 1 code here

class DerivedClass2(DerivedClass1):
    # Derived class 2 code here
```



Source

[ ]:
```
class Rectangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,2)
        print("Rectangle created")

    def findArea(self):
```

```
        a, b = self.sides
        area = a * b
        print('The area of the rectangle is %0.2f' %area)

    def info(self):
        print("This is a rectangle")

class Square(Rectangle):
    def __init__(self):
        Polygon.__init__(self,1)

    def findArea(self):
        print('The area of the square is %0.2f' %(self.sides[0]**2))

# Creating an instance of the Square class
s = Square()
# Prompting the user to enter the side of the square
s.inputSides()
# Displaying the sides of the square
s.dispSides()
# Calculating and printing the area of the square
s.info()
```

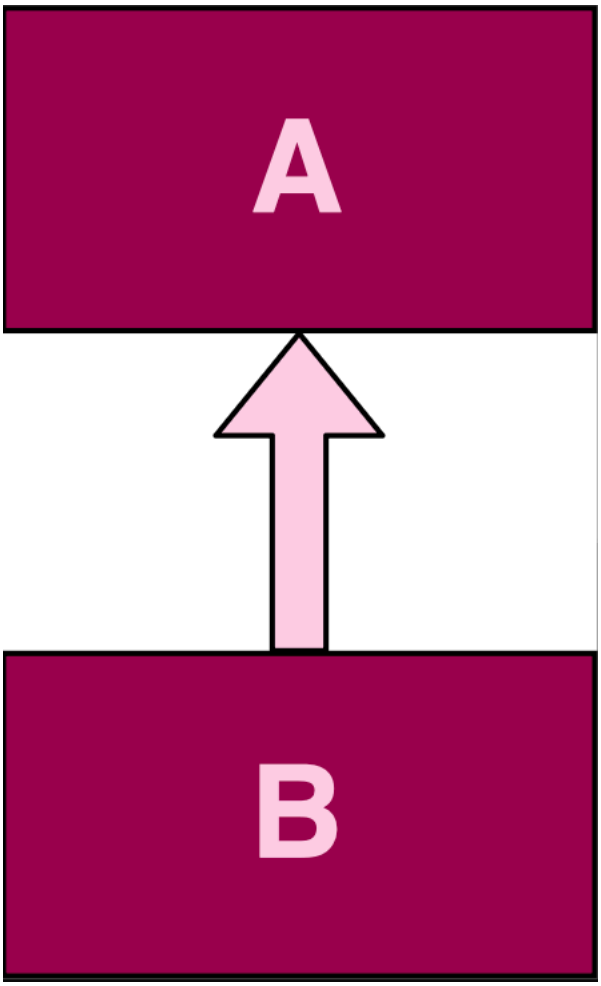
[ ]:
```
s.info()
print(isinstance(s, Bat))
print(s.__class__.__bases__)
print(issubclass(Square, Polygon))
print(s.__class__.__bases__[0].__bases__[0])
```

5 Python POO - MRO (Method Resolution Order)

Lorsqu'une classe hérite de plusieurs classes, il est possible que les classes mères héritent elles-mêmes d'autres classes. On parle d'héritage multiple. Dans ce cas, il faut définir l'ordre dans lequel les méthodes des classes mères sont appelées. Cet ordre est appelé l'ordre de résolution des méthodes (MRO).

La règle est simple, la méthode appellée sera, la première méthode trouvée dans la liste des classes mères. Cette liste est définie par l'ordre dans lequel les classes mères sont passées lors de la définition de la classe enfant. Si deux classes mère sont au même niveau dans la hiérarchie, c'est la première qui est appelée (celle de gauche).

5.1 Heritage simple



Source

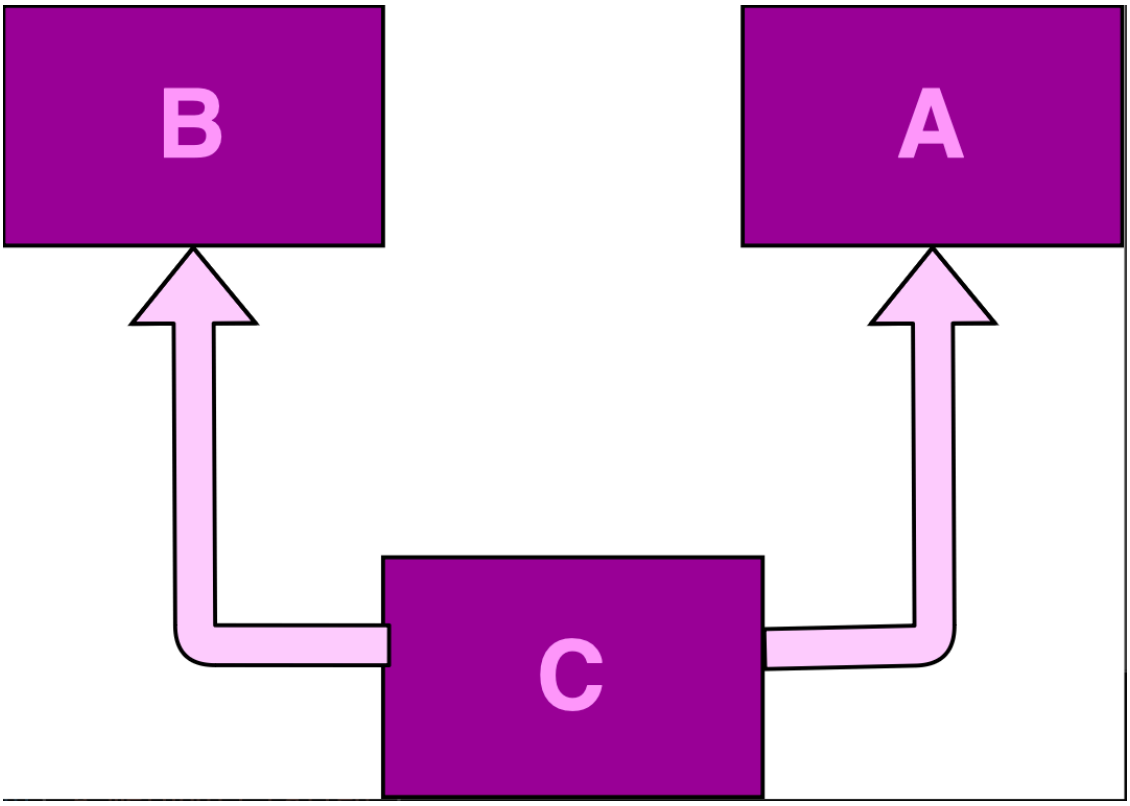
[64]:
```
class A:
  def method(self):
    print("A.method() called")

class B(A):
  def method(self):
    print("B.method() called")

b = B()
b.method()
```

```
B.method() called
```

Ici, le cas est simple, car car `B override` la méthode `method` du coup il appel son instance de la methode `method`

> **Note**: le MRO ici est : `B -> A -> Object`

5.2 Heritage multiple



Source

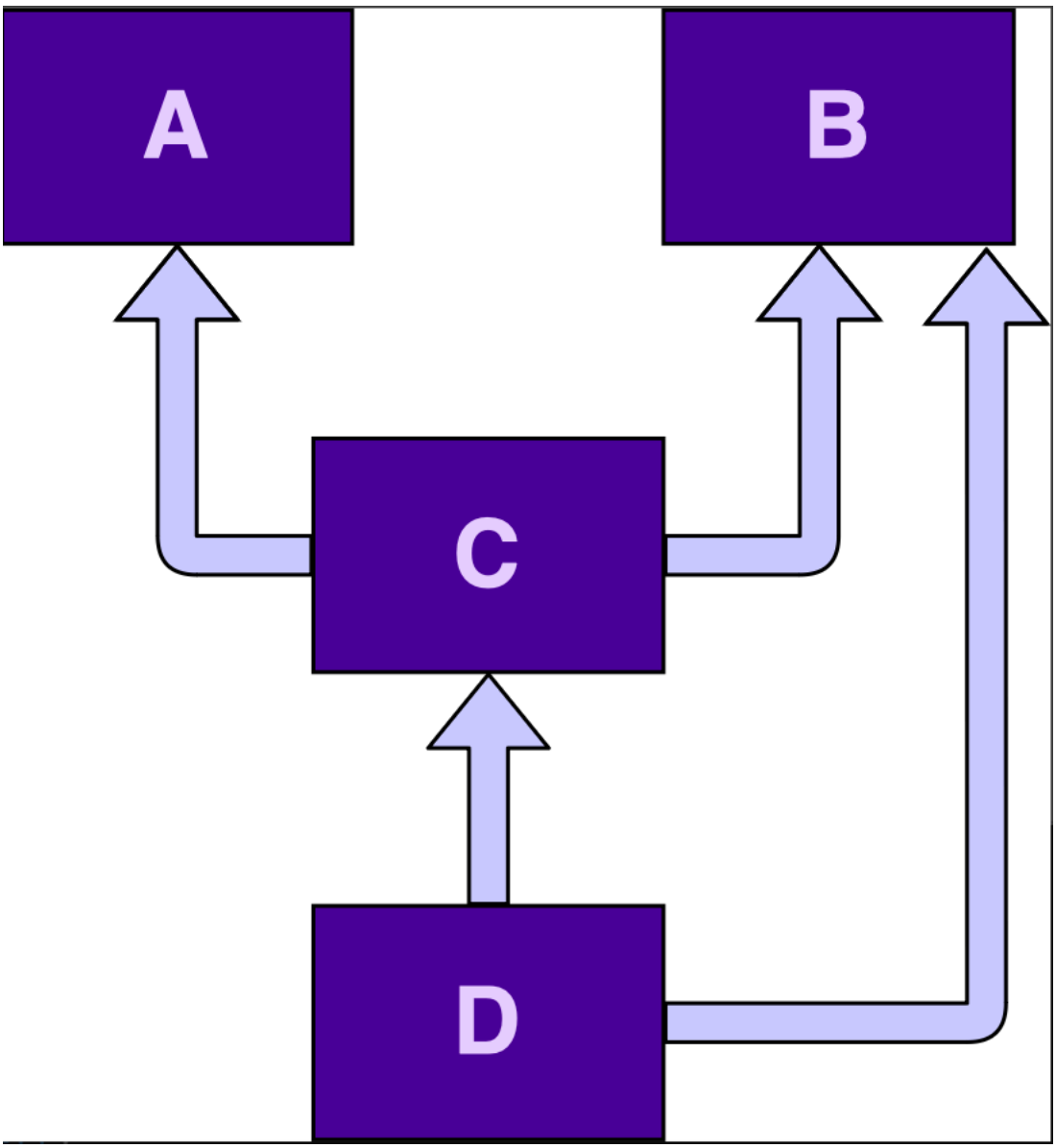
[70]:
```
class A:
  def method(self):
    print("A.method() called")

class B:
  def method2(self):
    print("B.method() called")

class C(A, B):
    pass

c = C()
c.method()
```

```
A.method() called
```

Ici, la classe `C` hérite de la classe A et de la classe `B`. La fonction n'est pas définie dans `C` du coup elle va regarder dans `A` puis dans `B` et enfin dans `object`. Car `A` est définie avant `B` dans la classe `C`.

> **Note**: le MRO ici est : `C` -> `A` -> `B` -> `Object`

Heritage Tricky



Source

[74]:
```
class A:
  def method(self):
    print("A.method() called")
class B:
  def method(self):
    print("B.method() called")

class C(A, B):
  pass

class D(C, B):
  pass
```

```
d = D()
d.method()
```

```
A.method() called
```

Malgré le fait qu'ici `D`est directement connecté avec `B`, `C`est d'abord définis dans `D` du coup il va chercher dans `C`. Comme `C`ne contient pas la méthode, on va regarder dans les parents de `C` on aura donc `A` qui lui contient la fonction.

> **Note**: le MRO ici est : `D` -> (`C` -> `A` -> `B`) -> `B` -> `Object`

6 Python POO - Polymorphisme

Le polymorphisme est un mécanisme qui permet de traiter de la même façon des objets de types différents mais liés. Le polymorphisme est un des trois piliers de la programmation orientée objet avec l'encapsulation et l'héritage.

6.1 exemple 1: Garage

On aimerais stocker dans un garage privé des véhicules privés et non publics. Voicis comment on pourrait gérer ca grace au polymorphisme.

[84]:
```
class PrivateVehicle:
    def __init__(self, brand, model, color):
        self.brand = brand
        self.model = model
        self.color = color

    def display(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Color:", self.color)

class Car(PrivateVehicle):
    def __init__(self, brand, model, color, doors):
        PrivateVehicle.__init__(self, brand, model, color)
        self.doors = doors

    def display(self):
        PrivateVehicle.display(self)
        print("Doors:", self.doors)
class golf_cart(Car):
    pass

class Motorcycle(PrivateVehicle):
    def __init__(self, brand, model, color, seats):
        PrivateVehicle.__init__(self, brand, model, color)
        self.seats = seats
```

```
    def display(self):
        super().display()
        print("Seats:", self.seats)
```

[77]:
```
class PublicVehicles:
    def __init__(self, year, color):
        self.year = year
        self.color = color

    def display(self):
        print("Year:", self.year)
        print("Color:", self.color)

class Autobus(PublicVehicles):
    def __init__(self, year, color, seats):
        PublicVehicles.__init__(self, year, color)
        self.seats = seats

    def display(self):
        PublicVehicles.display(self)
        print("Seats:", self.seats)

class Taxi(PublicVehicles):
    def __init__(self, year, color, busy):
        PublicVehicles.__init__(self, year, color)
        self.busy = busy

    def display(self):
        PublicVehicles.display(self)
        print("Busy:", self.busy)
```

[89]:
```
class PrivateGarage:
    def __init__(self):
        self.vehicles = []

    def add_vehicle(self, vehicle):
        if isinstance(vehicle, PrivateVehicle):
            self.vehicles.append(vehicle)

    def display(self):
        for vehicle in self.vehicles:
            print(type(vehicle))
            vehicle.display()
```

[90]:
```
c1 = Car("Peugeot", "308", "blue", 5)
m1 = Motorcycle("Triumph", "Speed Triple", "white", 2)
a1 = Autobus(2015, "Orange", 50)
```

```
t1 = Taxi(2018, "black", True)
cg = golf_cart("1", "3", "green", 0)

g = PrivateGarage()
g.add_vehicle(c1)
g.add_vehicle(m1)
g.add_vehicle(a1)
g.add_vehicle(t1)
g.add_vehicle(cg)


g.display()
```

```
<class '__main__.Car'>
Brand: Peugeot
Model: 308
Color: blue
Doors: 5
<class '__main__.Motorcycle'>
Brand: Triumph
Model: Speed Triple
Color: white
Seats: 2
<class '__main__.golf_cart'>
Brand: 1
Model: 3
Color: green
Doors: 0
```

7 Python POO - Encapsulation

L'encapsulation est un mécanisme qui permet de regrouper des données et les traitements qui s'y appliquent en une seule entité. Cette entité est appelée classe. L'encapsulation permet de cacher les données et de protéger l'intégrité des données en empêchant leur accès direct.

7.1 Getters et setters

Les getters et les setters sont des méthodes qui permettent d'accéder aux attributs d'un objet. Les getters permettent d'accéder aux attributs et les setters permettent de modifier les attributs.

[ ]:
```
class Point:
    def __init__(self, x, y):
        self._x = 0
        self._y = 0

    # getter method
    def get_x(self):
        return int(self._x)
```

```
    def get_y(self):
        return self._y

    # setter method
    def set_x(self, x):
        if x < 0:
            raise ValueError("x must be positive")
        if not isinstance(x, (int, float)):
            raise TypeError("x must be a number")
        assert(x > 0)
        self._x = x

    def set_y(self, y):
        if y < 0:
            raise ValueError("x must be positive")
        if not isinstance(y, (int, float)):
            raise TypeError("x must be a number")
        self._y = y
```

[ ]:
```
p1 = Point(5, 6)

print(p1._x)
print("x", p1.get_x())
print("y", p1.get_y())

p1.set_x(-10)
p1.set_y(20)

print("x1", p1.get_x())
print("y1", p1.get_y())

#p1.set_x(-10)
#p1.set_y("-20")
```

8 Python POO - Annotations

Les annotations de classes permettent de définir des comportements différents sur certaines methods de classes.

8.1 Annotations - @staticmethod

- Le décorateur @staticmethod est utilisé pour définir une méthode statique dans une classe.
- Une méthode statique est une méthode qui n'a pas de référence implicite à l'instance de la classe (pas de premier argument self).
- Elle peut être appelée à partir de la classe elle-même, sans nécessiter une instance de cette classe.

- Une méthode statique est souvent utilisée pour des fonctionnalités qui sont indépendantes de toute instance ou de la classe elle-même.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @staticmethod
    def distance(p1, p2):
        return math.sqrt((p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2)
```

8.2 Annotations - @classmethod

- Le décorateur @classmethod est utilisé pour définir une méthode de classe dans une classe.
- Une méthode de classe prend comme premier argument la classe elle-même, conventionnellement appelé cls.
- Elle peut être appelée à partir de la classe elle-même, sans nécessiter une instance de cette classe.
- Une méthode de classe est souvent utilisée pour effectuer des opérations qui concernent la classe dans son ensemble.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def from_tuple(cls, t):
        return cls(t[0], t[1])
```

8.3 Annotations - @property

- Le décorateur @property est utilisé pour définir une méthode comme une propriété d'une classe.
- Il permet d'accéder à la méthode comme si elle était un attribut de la classe, sans nécessiter d'appel explicite.
- Une propriété peut être utilisée pour obtenir (getter) ou définir (setter) la valeur d'un attribut, ou pour effectuer des opérations supplémentaires lors de l'accès à un attribut.

[ ]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def distance_from_origin(self):
        import math
        return math.sqrt(self.x ** 2 + self.y ** 2)
```

```
    @distance_from_origin.setter
    def distance_from_origin(self, d):
        ratio = d / self.distance_from_origin
        self.x *= ratio
        self.y *= ratio

    @distance_from_origin.deleter
    def distance_from_origin(self):
        self.x = 0
        self.y = 0

p = Point(3, 4)
print(p.x, p.y)
print(p.distance_from_origin)
p.distance_from_origin = 2.5
print(p.x, p.y)
```

8.4 Annotations - @abstractmethod

- Le décorateur @abstractmethod est utilisé pour définir une méthode abstraite dans une classe abstraite.
- Une méthode abstraite est une méthode qui doit être implémentée dans les classes dérivées, mais qui n'a pas d'implémentation concrète dans la classe abstraite.
- Elle sert de contrat pour les sous-classes et garantit que toutes les sous-classes fournissent une implémentation de cette méthode.
- Une classe abstraite est déclarée en utilisant le module abc (Abstract Base Classes).

[ ]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.r = radius
    def area(self):
        return 3.1415*r**2
```

```
r = Rectangle(2,3)
print(r.area())
c = Circle(10)
print(c.area())
```

8.5 Annotations - @dataclass

- Le décorateur @dataclass est utilisé pour créer rapidement une classe de données (data class) en Python.
- Il génère automatiquement des méthodes spéciales telles que **init**, **repr**, **eq**, etc., à partir des annotations de type des attributs de classe.
- Les classes de données sont principalement utilisées pour stocker des données et fournissent une implémentation prédéfinie pour les méthodes spéciales.
- Le module dataclasses est requis pour utiliser ce décorateur, qui est disponible à partir de Python 3.7.

[ ]:
```
from dataclasses import dataclass

@dataclass
class Point():
    x
    y

p1 = Point()
print(p1)
```

9 Python POO - Operator Overloading

L'overloading d'opérateur est un mécanisme qui permet de redéfinir le comportement des opérateurs (+, -, *, /, <, >, ==, !=, etc.) pour les objets d'une classe. On parle aussi de surcharge d'opérateur.

9.1 Magic functions

Les magic functions en python sont les fonctions qui commencnet et terminent par deux underscores`__x__`. Elles sont appelées automatiquement par python dans certaines situations. Par exemple, la fonction `__init__()` est appelée automatiquement lors de la création d'un objet.

Python utilise les magic function lorqu'un operateur est appliqué, une magic function est appelée. Par exemple, lorsqu'on applique l'opérateur `+` à deux objets, la fonction `__add__()` est appelée.

[91]:
```
x = 10

print(x + 5)
print(x.__add__(5))
```

```
15
15
```

9.1.1 Quelque magic functions

- `__init__()` : appelée lors de la création d'un objet
- `__str__()` : appelée lors de l'appel de la fonction `str()`
- `__len__()` : appelée lors de l'appel de la fonction `len()`
- `__add__()` : appelée lors de l'opération `+`

exemple 1: + operator overloading On peut définir une classe `Point` qui représente un point dans un plan. On peut définir l'opérateur `+` pour ajouter deux points. Pour cela, on définit la méthode `__add__()` qui prend comme paramètre deux points et qui retourne un nouveau point dont les coordonnées sont la somme des coordonnées des deux points.

[99]:
```python
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return "Point({0}, {1})".format(self.x, self.y)
    @staticmethod
    def add(p1, p2):
        x = p1.x + p2.x
        y = p1.y + p2.y
        return Point(x, y)
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)
    def __mul__(p1, p2):
        x = p1.x * p2.x
        y = p1.y * p2.y
        return Point(x, y)

p1 = Point(1, 2)
p2 = Point(2, 3)
p3 = Point(5, 10)
print(Point.add(p1,p2))
print(p1)
print((p1+p2))
print(p1+p2*p3)
```

```
Point(3, 5)
Point(1, 2)
Point(3, 5)
Point(11, 32)
```

Ici, on a définis la méthode `__add__()` qui permet d'ajouter deux points. Cette méthode prend comme paramètre un point `other`. Elle retourne un nouveau point dont les coordonnées sont la somme des coordonnées de ses propres points `self` et des points de `other`.

9.1.2 list of operators and their magic functions

Operateur	Expression	Magic function
Addition	p1 + p2	p1.__add__(p2)
Soustraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Puissance	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor division	p1 // p2	p1.__floordiv__(p2)
Modulo	p1 % p2	p1.__mod__(p2)
Bitwise left shift	p1 « p2	p1.__lshift__(p2)
Bitwise right shift	p1 » p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

9.2 Overloading comparison operators

On peut définir les opérateurs de comparaison pour comparer deux points. Pour cela, on définit les méthodes `__lt__()`, `__le__()`, `__eq__()`, `__ne__()`, `__gt__()` et `__ge__()`.

[103]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # overload < operator
    def __lt__(pleft, pright):
        return pleft.age < pright.age

    def younger(self, p2):
        return self.age < p2.age

p1 = Person("Alice", 20)
p2 = Person("Bob", 30)

print(p1 < p2)
print(p2 < p1)
print(p1.younger(p2))
```

```
True
False
True
```

Note: Ici, nous avons de comparer les `Person`en fonction de leur age.

9.2.1 list of comparison operators and their magic functions

Operateur	Expression	Magic function
Plus petit que	p1 < p2	p1.__lt__(p2)
Plus petit ou égal à	p1 <= p2	p1.__le__(p2)
Égal à	p1 == p2	p1.__eq__(p2)
Différent de	p1 != p2	p1.__ne__(p2)
Plus grand que	p1 > p2	p1.__gt__(p2)
Plus grand ou égal à	p1 >= p2	p1.__ge__(p2)

9.3 Overloading unary operators

On peut définir les opérateurs unaire `-` et `+` pour changer le signe des coordonnées d'un point. Pour cela, on définit les méthodes `__neg__()` et `__pos__()`.

9.3.1 list of unary operators and their magic functions

Operateur	Expression	Magic function
Moins unaire	-p1	p1.__neg__()
Plus unaire	+p1	p1.__pos__()