

Course: Python language

Ali ZAINOUL <ali.zainoul.az@gmail.com>

for IBM - Needemand
July 17, 2024



- 1 Programmation Orientée Objet
 - Concepts fondamentaux de la POO
 - Les Six principes de la POO
 - Définition d'une classe d'objet
 - Définition d'un objet (état, comportement, identité)
 - Notions de classe d'objet, d'objet (instance), d'attribut et de méthode
 - Notion de classe
 - Notion d'attribut
 - Notion de méthode
 - Notion de spécificateurs d'accès
 - Spécificateurs d'accès en Python
 - Notion de Name Mangling en Python
 - Encapsulation des données
 - Instanciation d'objets, fonction `isinstance()`
 - Constructeur (`__init__`)
 - Attributs et méthodes

- Le paramètre `self`
- Surcharge d'affichage (`__str__`)
- Surcharge d'opérateurs
- Propriété, accesseur et mutateur, et décorateur
 - Propriété, accesseur et mutateur, et décorateur
 - Décorateur
- Espaces de noms global, de l'objet, de la classe
- Variable de classe
- Constructeur à nombre d'arguments arbitraire (`*args, **kwargs`)
- Association / Agrégation / Composition
 - Association
 - Agrégation
 - Composition
 - Différences entre Association, Agrégation et Composition
 - Différentes relations
- Notion d'Héritage de classe (généralisation)
 - Principe de l'héritage
 - Types d'héritage
 - Problème du diamant

- Fonctions `issubclass()`, `super()` et méthode `mro()`
- Fonction `super()`
- La méthode `mro()` et l'attribut `__mro__`

■ Notion d'Abstraction

- Classes abstraites
- Les Interfaces

■ Notion de Polymorphisme

- Compile Time Polymorphism
- Runtime Polymorphism

**** Programmation Orientée Objet ****

Le paradigme Orienté Objet

Définition : La programmation orientée objet (POO) est l'un des principes fondamentaux de la programmation informatique, dont les origines remontent aux années 1970 avec les langages Simula et Smalltalk. Le principe a rapidement pris de l'ampleur avec la création du langage C++, qui est une extension du langage C, dans le but de rendre les logiciels plus robustes.

Principes fondamentaux de la POO

Une aide mnémotechnique utile pour les **six principes fondamentaux** de la programmation orientée objet est ACOPIE.

- *Abstraction*
- *Class*
- *Object*
- *Polymorphism*
- *Inheritance*
- *Encapsulation*

Objet

- En informatique, un objet est un conteneur symbolique autonome qui contient des informations et des fonctions/méthodes liées à un sujet, manipulé dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel.

Classe

- La classe est une structure de données spéciale dans les langages de programmation orientés objet.
- Elle décrit la structure interne des données et définit les méthodes qui s'appliqueront aux objets de la même famille (la même classe) ou de même type.
- Elle fournit des méthodes pour créer des objets dont la représentation sera donc celle donnée par la classe génératrice. Les objets sont alors appelés instances de la classe. C'est pourquoi les attributs d'un objet sont également appelés variables d'instance et les messages sont appelés opérations ou méthodes d'instance.

Encapsulation

- L'encapsulation consiste à regrouper des données et des méthodes qui les manipulent en une seule entité appelée *capsule*.
- Cela permet d'organiser le code, de limiter l'accès à certaines parties depuis l'extérieur de la capsule et de disposer d'un code robuste.

Abstraction

- L'abstraction est l'un des concepts clés des langages de programmation orientés objet. Son objectif principal est de gérer la complexité en cachant les détails inutiles à l'utilisateur.
- L'abstraction est le processus de représentation d'un objet réel sous forme de modèle informatique. Cela implique essentiellement d'extraire les variables pertinentes attachées aux objets que nous voulons manipuler et de les placer dans un modèle informatique approprié.

Héritage

- Il s'agit d'un mécanisme de transmission de toutes les méthodes d'une classe dite "mère" à une autre appelée "filles" et ainsi de suite.

Polymorphisme

- Le nom du polymorphisme vient du grec et signifie "plusieurs formes". Cette caractéristique est l'un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est lié aux méthodes des objets.

Définition d'une classe d'objet

- En Python, une classe est un modèle permettant de créer des objets.
- Elle définit la structure et le comportement des objets qui en sont des instances.
- Une classe contient des attributs (variables) et des méthodes (fonctions).
- Les méthodes sont des fonctions définies à l'intérieur de la classe et agissent sur les objets de cette classe.
- L'utilisation de classes permet une meilleure organisation du code et favorise la réutilisabilité.
- Les classes sont largement utilisées pour modéliser des entités du monde réel ou abstrait.
- Documentation : [Python Classes](#)

Template d'une classe d'objet

```
class nameClass:  
# Constructor __init__ and attributes (members)  
# Methods of class nameClass
```

Définition d'un objet (état, comportement, identité)

- **État** : Un objet a un état qui est défini par ses attributs. Ces attributs représentent les caractéristiques de l'objet à un moment donné.
- **Comportement** : Un objet a un comportement défini par ses méthodes. Les méthodes représentent les actions ou opérations que l'objet peut effectuer.
- **Identité** : Chaque objet a une identité unique qui le distingue des autres objets. L'identité est généralement attribuée par le système.

Définition d'un objet

Définition: Un **objet** est une **instance** d'une **classe**.

Examples

Exemple - (1/3)

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name          # Member or attribute 1
4         self.age = age            # Member or attribute 2
5
6 # Creating objects via constructor __init__
7 person1 = Person("Ali", 30)      # Object (instance of class) 1
8 person2 = Person("Alison", 25)  # Object (instance of class) 2
9
10 # Displaying identity
11 print(f"{person1.name} is {person1.age} years old")
12 print(f"{person2.name} is {person2.age} years old")
```

Exemple - (2/3)

```
1 class Animal:
2     def __init__(self, name, species):
3         self.name = name          # Member or attribute 1
4         self.species = species    # Member or attribute 2
5
6     def display_info(self):        # Method of class Animal
7         print(f"{self.name} is a {self.species}")
8
9 # Creating objects via constructor __init__
10 lion = Animal("Simba", "lion")    # Object (instance of class) 1
11 snake = Animal("Kaa", "snake")    # Object (instance of class) 2
12
13 # Calling methods of the class Animal
14 lion.display_info()
15 snake.display_info()
```

Exemple - (3/3)

```
1 class Car:
2     def __init__(self, brand, model):
3         self.brand = brand          # Member or attribute 1
4         self.model = model          # Member or attribute 2
5         self.speed = 0              # Member or attribute 3
6
7     def accelerate(self, increment): # Method 1 of class Car
8         self.speed += increment
9         print(f"{self.brand} {self.model} accelerates to {self.speed} km/h")
10
11    def brake(self, decrement):      # Method 2 of class Car
12        self.speed -= decrement
13        print(f"{self.brand} {self.model} brakes to {self.speed} km/h")
14
15    # Creating objects
16    my_car = Car("Porsche", "Cayenne") # Object (instance of class) 1
17
18    # Calling methods of the class Car
19    my_car.accelerate(20)
20    my_car.brake(10)
```

Définition d'une classe

Définition : Une **classe** est une **structure de données** fondamentale en **programmation orientée objet** qui **encapsule** les **attributs** et les **méthodes**.

Ces éléments définissent les **caractéristiques** et le comportement d'un **objet**, permettant ainsi la création d'**instances** spécifiques de cette classe.

Exemples

- Voir l'exemple (1/3) pour plus de détails ;
- Voir l'exemple (2/3) pour plus de détails ;
- Voir l'exemple (3/3) pour plus de détails.

Définition d'un attribut

Définition : un **attribut**, également appelé **membre de classe**, est une **variable** déclarée à l'**intérieur d'une classe** qui a pour but de **stocker des données spécifiques** à cette classe.

Les attributs sont ainsi des **caractéristiques** ou des **propriétés d'un objet** qui sont définies dans la classe et qui peuvent être utilisées pour décrire l'état ou les **caractéristiques de l'objet**.

Exemples

- Voir l'exemple (1/3) pour plus de détails ;
- Voir l'exemple (2/3) pour plus de détails ;
- Voir l'exemple (3/3) pour plus de détails.

Définition d'une méthode

Définition : Une **méthode** est une fonction ou un bloc de code **associé à une classe** ou à un **objet**. Elle permet de définir le **comportement** de l'**objet** ou de la **classe**.

- **Méthode d'instance:** Appelée par une instance de la classe, agit sur les attributs de cette instance.
- **Méthode statique:** Appelée par la classe elle-même, indépendante des instances, souvent utilisée pour des fonctionnalités utilitaires.
- **Méthode de classe:** Appelée par la classe, agit sur les attributs de la classe plutôt que sur les instances.

Table: Comparison of Regular, Static, and Class Methods

	Regular Method	Static Method	Class Method
Definition Syntax	Defined without decorators	Defined with <code>@staticmethod</code> decorator	Defined with <code>@classmethod</code> decorator
First Argument	<code>self</code> (instance)	None	<code>cls</code> (class)
Access to Attributes/Methods	Can access instance attributes and methods directly	No implicit access to instance attributes/methods	Can access class-level attributes and methods via <code>cls</code>
Instance Specific	Requires an instance	Not tied to any instance	Not tied to any instance
Class Specific	Cannot access class-level attributes/methods	Cannot access class-level attributes/methods	Can access class-level attributes/methods via <code>cls</code>
Inheritance	Inherited by subclasses	Not inherited by subclasses	Inherited by subclasses
Usage	Use when instance-specific behavior is needed	Use when the method does not depend on instance or class attributes/methods	Use when class-level behavior is needed, such as factory methods or alternative constructors
Example	<code>def method(self, arg):</code>	<code>@staticmethod</code> <code>def method(arg):</code>	<code>@classmethod</code> <code>def method(cls, arg):</code>

Exemples

- Voir l'exemple (1/4) pour plus de détails ;
- Voir l'exemple (2/4) pour plus de détails ;
- Voir l'exemple (3/4) pour plus de détails ;
- Voir l'exemple (4/4).

Les méthodes Getters & Setters

Définition : Les **getters** et les **setters** sont des méthodes utilisées **respectivement** pour **accéder** et **modifier** les valeurs des **membres** (attributs / variables) d'une **classe**.

Signature des Getters & Setters sans type de retour

- **Les getters retournent** la valeur d'un membre et ont la signature :

```
def getVar(self):  
    return self.var
```

- **Les setters modifient** la valeur d'un membre et ont la signature :

```
def setVar(self, _var):  
    self.var = _var
```

Exemple

```
1 class Person:
2     def __init__(self, name, age):
3         self._name = name      # PROTECTED Member or attribute 1
4         self._age = age        # PROTECTED Member or attribute 2
5
6     # Getter for name
7     def get_name(self):
8         return self._name
9
10    # Setter for name
11    def set_name(self, name):
12        self._name = name
13
14    # Getter for age
15    def get_age(self):
16        return self._age
17
18    # Setter for age
19    def set_age(self, age):
20        self._age = age
21
22    # Creating objects via constructor __init__
23    person1 = Person("Alison", 25) # Object or instance of class 1
24
25    # Displaying identity
26    print(f"{person1.get_name()} is {person1.get_age()} years old")
```

Signature des Getters & Setters avec type de retour

- **Les getters retournent** la valeur d'un membre et ont la signature :

```
def getVar(self) -> typeName:  
    return self.var
```

- **Les setters modifient** la valeur d'un membre et ont la signature :

```
def setVar(self, _var: typeName) -> None:  
    self.var = _var
```

Exemple

```
1 class BankAccount:
2     def __init__(self, account_holder: str, balance: float):
3         self._account_holder = account_holder # PROTECTED Member or attribute 1
4         self._balance = balance # PROTECTED Member or attribute 2
5
6     # Getter for account holder
7     def get_account_holder(self) -> str:
8         return self._account_holder
9
10    # Setter for account holder
11    def set_account_holder(self, account_holder: str):
12        self._account_holder = account_holder
13
14    # Getter for balance
15    def get_balance(self) -> float:
16        return self._balance
17
18    # Setter for balance
19    def set_balance(self, balance: float):
20        self._balance = balance
21
22    # Creating objects via constructor __init__
23    account1 = BankAccount("John Doe", 1000.0) # Object (instance of class) 1
24
25    # Displaying identity
26    print(f"{account1.get_account_holder()} has a balance of ${account1.get_balance()}")
```


Spécificateurs d'accès en Python

- En Python, il existe trois spécificateurs d'accès: `public`, `_protected` et `__private`.
- Les attributs ou méthodes `public` peuvent être accessibles de n'importe où dans le programme.
- Les attributs ou méthodes `_protected` sont accessibles dans la classe où ils sont définis et dans les sous-classes.
- Les attributs ou méthodes `__private` ne peuvent être accessibles que dans la classe où ils sont définis.
- Les spécificateurs d'accès sont définis par des conventions plutôt que par des fonctionnalités de langage strictes.

Spécificateurs d'accès en Python - Exemple 1

```
1 class MyClass:
2     def __init__(self):
3         self.public_attribute = 42
4         self._protected_attribute = "protected"
5         self.__private_attribute = "private"
6
7     def access_attributes(self):
8         print("Public attribute:", self.public_attribute)
9         print("Protected attribute:", self._protected_attribute)
10        print("Private attribute:", self.__private_attribute)
11
12 # Creating an object of MyClass
13 obj = MyClass()
14
15 # Accessing attributes
16 obj.access_attributes()
```

Spécificateurs d'accès en Python - Exemple 2

```
1 class MyClass:
2     def __init__(self):
3         self.public_attribute = 42
4         self._protected_attribute = "protected"
5         self.__private_attribute = "private"
6
7     def access_attributes(self):
8         print("Public attribute:", self.public_attribute)
9         print("Protected attribute:", self._protected_attribute)
10        print("Private attribute:", self.__private_attribute)
11
12    def _protected_method(self):
13        print("This is a protected method")
14
15    def __private_method(self):
16        print("This is a private method")
17
18 # Creating an object of MyClass
19 obj = MyClass()
20
21 # Accessing attributes and methods
22 obj.access_attributes()
23 obj._protected_method()
24 # obj.__private_method() # This will raise an error
```

Name Mangling en Python

- Le **name mangling** est un mécanisme utilisé en Python pour rendre les attributs ou méthodes d'une classe plus difficiles à accéder à partir de l'extérieur de la classe.
- Cela est réalisé en ajoutant le nom de la classe comme préfixe au nom de l'attribut ou de la méthode, précédé par deux traits de soulignement.
- Le **name mangling** est déclenché lorsqu'un nom d'attribut ou de méthode commence par deux traits de soulignement `__`.
- Python renomme alors cet attribut ou cette méthode en y ajoutant le nom de la classe, précédé par un trait de soulignement.
- En utilisant le "name mangling", les attributs et méthodes d'une classe peuvent être rendus moins accessibles depuis l'extérieur de la classe, bien que cela ne les rende pas complètement privés.

Name Mangling en Python - Exemple illustratif

```
1 class MyClass:
2     def __init__(self):
3         self.__private_attritube = 420
4
5     def __private_method(self):
6         print("This is a private method")
7
8 # Creating an instance of the class
9 obj = MyClass()
10
11 # Accessing the private attribute from outside the class
12 # print(obj.__private_attritube) # This will raise an AttributeError
13
14 # Calling the private method from outside the class
15 # obj.__private_method() # This will also raise an AttributeError
16
17 # Accessing the private attribute using name mangling
18 print(obj._MyClass__private_attritube) # This will print 420
19
20 # Calling the private method using name mangling
21 obj._MyClass__private_method() # This will print "This is a private method"
```

Principe de l'encapsulation

Définition : L'encapsulation est un mécanisme qui permet de cacher les détails d'**implémentation d'une classe** aux **autres classes**.

Il s'agit d'un concept clé en **programmation orientée objet** qui **protège les données** d'une classe et **garantit leur intégrité**.

Objectifs de l'encapsulation

- Réduction des erreurs et des conflits de noms.
- Amélioration de la sécurité.
- Facilitation de la maintenance du code.
- Modification facile du code sans affecter les autres parties du programme.

Exemple d'encapsulation

Supposons que nous avons une classe **BankAccount** qui contient des données sensibles telles que le solde du compte et le numéro de compte bancaire. Afin de protéger ces données, nous pouvons les déclarer comme `protected` ou `private` et utiliser des méthodes `public` pour y accéder et les modifier, par exemple `get_balance()` ou encore `deposit()`.

De cette manière, les autres classes ne peuvent pas accéder directement aux données sensibles de la classe `BankAccount`, mais doivent passer par les méthodes publiques qui garantissent que les données sont manipulées de manière sûre et sécurisée.

Conclusion

- En encapsulant les données de cette manière, nous assurons que:
 - les informations sensibles ne sont pas accessibles depuis l'extérieur de la classe ;
 - leur manipulation est effectuée de manière contrôlée.
- Cela permet de **garantir l'intégrité des données** et d'améliorer la **sécurité de notre programme**.

Instanciation d'objets

- En Python, les objets sont des instances de classes.
- Pour instancier un objet, utilisez le nom de la classe suivi de parenthèses.
- L'instance est créée en appelant le constructeur de la classe, généralement défini par la méthode `__init__()`.
- Exemple d'instanciation :

```
1 # Définition de la classe
2 class MyClass:
3     def __init__(self, x):
4         self.x = x
5
6 # Instanciation de l'objet
7 obj = MyClass(10)
```

Fonction isinstance()

- La fonction `isinstance()` est utilisée pour vérifier le type d'un objet.
- Elle prend deux arguments : l'objet à vérifier et le type ou la classe attendu(e).
- Elle renvoie `True` si l'objet est une instance de la classe spécifiée, sinon `False`.
- Exemple d'utilisation :

```
1 # Définition de la classe
2 class MyClass:
3     pass
4
5 # Instanciation de l'objet
6 obj = MyClass()
7
8 # Vérification du type de l'objet
9 if isinstance(obj, MyClass):
10     print("Obj est une instance de MyClass")
11 else:
12     print("Obj n'est pas une instance de MyClass")
```

Constructeur (`__init__`)

- Le constructeur `__init__()` est une méthode spéciale utilisée pour initialiser les objets lors de leur création, il est appelé automatiquement lorsque vous instanciez la classe pour créer un nouvel objet.
- Le premier paramètre de la méthode `__init__()` est conventionnellement nommé `self` et fait référence à l'objet lui-même.
- Vous pouvez définir des attributs d'instance dans le constructeur en utilisant la notation `self.nom_attribut`.
- Exemple :

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6 person1 = Person("Alice", 30) # Création d'un objet de type Person
```

Attributs et méthodes

- Les attributs sont des variables liées à une classe ou à un objet.
- Les méthodes sont des fonctions définies à l'intérieur d'une classe et sont utilisées pour effectuer des opérations sur les objets de cette classe.
- Les attributs et les méthodes sont accessibles à travers l'instance de la classe à l'aide de la notation pointée (par exemple, `objet.attribut` ou `objet.methode()`).
- Exemple :

```
1 class Circle:
2     def __init__(self, radius):
3         self.radius = radius
4
5     def area(self):
6         return 3.14 * self.radius ** 2
7
8 # Création d'un objet de type Circle
9 circle1 = Circle(5)
10 print("Area of circle:", circle1.area())
```

Le paramètre `self`

- En Python, `self` est utilisé comme premier paramètre dans la définition de méthodes de classe.
- Il fait référence à l'instance de la classe elle-même.
- Lorsque vous appelez une méthode sur un objet, Python passe automatiquement l'objet lui-même en tant que premier argument à la méthode, donc vous ne devez pas spécifier `self` lors de l'appel de la méthode.
- Exemple :

```
1 class Circle:
2     def __init__(self, radius):
3         self.radius = radius
4
5     def area(self):
6         return 3.14 * self.radius ** 2
7
8 # Création d'un objet de type Circle
9 circle1 = Circle(5)
10 print("Area of circle:", circle1.area())
```

Surcharge d'affichage (`__str__`)

- La méthode spéciale `__str__()` est utilisée pour retourner une représentation de chaîne de caractères de l'objet.
- Elle est invoquée lorsque la fonction `str()` ou la méthode `print()` est appelée sur l'objet.
- En surchargeant cette méthode, vous pouvez spécifier comment vous souhaitez que votre objet soit affiché sous forme de chaîne de caractères.

Surcharge d'affichage (__str__) - Exemple

■ Exemple :

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __str__(self):
7         return f"Point({self.x}, {self.y})"
8
9 # Utilisation
10 p = Point(3, 4)
11 print(str(p)) # Output: Point(3, 4)
```


Surcharge d'opérateurs

- En Python, vous pouvez surcharger de nombreux opérateurs pour définir le comportement des objets de vos classes.
- Voici une liste des opérateurs pouvant être surchargés avec leur utilisation typique :
 - `__add__()` : Addition (+)
 - `__sub__()` : Soustraction (-)
 - `__mul__()` : Multiplication (*)
 - `__truediv__()` : Division (/)
 - `__floordiv__()` : Division entière (//)
 - `__mod__()` : Modulo (%)
 - `__pow__()` : Puissance (**)

Surcharge d'opérateurs (suite)

- `__eq__()` : Égalité `==`
- `__ne__()` : Différent `!=`
- `__gt__()` : Supérieur strict `>`
- `__ge__()` : Supérieur ou égal `>=`
- `__le__()` : Inférieur ou égal `<=`
- `__and__()` : Et logique `&`
- `__or__()` : Ou logique `|`
- `__xor__()` : Ou exclusif `^`
- `__invert__()` : Inversion de bits `~`

Documentation : [Surcharge d'opérateurs en Python](#)

Surcharge d'opérateurs (`__eq__`)

- La méthode spéciale `__eq__()` est utilisée pour comparer l'égalité entre deux objets.
- Elle est invoquée lorsque l'opérateur d'égalité (`==`) est utilisé entre deux objets.
- En surchargeant cette méthode, vous pouvez définir votre propre logique d'égalité entre les objets de votre classe.
- La méthode devrait renvoyer `True` si les objets sont égaux et `False` sinon.

Surcharge d'opérateurs (__eq__) - Exemple

■ Exemple :

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __eq__(self, other):
7         return self.x == other.x and self.y == other.y
8
9 p1 = Point(3, 4)
10 p2 = Point(4, 3)
11 print(p1 == p2) # Output: False
```

Surcharge d'opérateurs (`__add__`)

- La méthode spéciale `__add__()` est utilisée pour définir le comportement de l'opérateur d'addition `+` pour les objets de votre classe.
- Elle est invoquée lorsque l'opérateur d'addition est utilisé entre deux objets.
- En surchargeant cette méthode, vous pouvez définir votre propre logique pour l'addition des objets de votre classe.

Surcharge d'opérateurs (__add__) - Exemple

■ Exemple :

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __add__(self, other):
7         return Point(self.x + other.x, self.y + other.y)
8
9 # Utilisation
10 p1 = Point(1, 2)
11 p2 = Point(3, 4)
12 result = p1 + p2
13 print(result.x, result.y) # Output: 4 6
```

Propriété, accesseur et mutateur

- Les **propriétés** (`property()`) en Python offrent un moyen de contrôler l'accès aux données d'un objet de manière encapsulée et cohérente. Elles sont définies à l'aide de la fonction spéciale `property()`, permettant d'associer des accesseurs et des mutateurs à un attribut.
 - Un **accesseur** est une méthode permettant de récupérer la valeur d'un attribut.
 - Un **mutateur** est une méthode permettant de modifier la valeur d'un attribut.
- L'utilisation de propriétés garantit l'encapsulation des données et permet d'appliquer des validations ou des transformations lors de l'accès ou de la modification des attributs. Cela contribue à la robustesse et à la cohérence des classes en facilitant le contrôle précis de l'accès aux données.

Fonction spéciale property - Exemple 1

- Voici un exemple de définition d'une propriété avec des accesseurs et des mutateurs :

```
1 class Person:
2     def __init__(self, name):
3         self._name = name          # _name: private member
4
5     def get_name(self):
6         return self._name          # get_name: getter
7
8     def set_name(self, value):      # set_name: setter
9         if isinstance(value, str):
10            self._name = value
11        else:
12            raise TypeError("Name must be a string")
13    # property function
14    name = property(get_name, set_name)
15
16 # Usage of property
17 person = Person("Alice")           # Instantiation of Person
18 print(person.name)                 # Usage of the accessor <==> Getter
19 person.name = "Bob"                # Usage of Mutator <==> Setter
20 print(person.name)
```


Fonction spéciale property - Exemple 1 - Suite

- Dans cet exemple, la classe `Person` définit une propriété `name` avec un accesseur `get_name()` et un mutateur `set_name()`. L'utilisation de la propriété permet d'accéder à l'attribut `_name` de manière contrôlée et encapsulée.

Fonction spéciale property - Exemple 2

■ Voici un autre exemple de définition d'une propriété:

```
1 # circle.py
2 class Circle:
3     def __init__(self, radius):
4         self.__radius = radius
5
6     def _get__radius(self):
7         print("Get radius")
8         return self.__radius
9
10    def _set__radius(self, value):
11        print("Set radius")
12        self.__radius = value
13
14    def _del__radius(self):
15        print("Delete radius")
16        del self.__radius
17
18    radius = property(
19        fget=_get__radius,
20        fset=_set__radius,
21        fdel=_del__radius,
22        doc="The radius property."
23    )
```

Fonction spéciale property - Exemple 2 - Suite

■ Suite :

```
1 >>> from circle import Circle
2 >>> circle = Circle(42.0)
3 >>> circle.radius
4 Get radius
5 42.0
6 >>> circle.radius = 100.0
7 Set radius
8 >>> circle.radius
9 Get radius
10 100.0
11 >>> del circle.radius
12 Delete radius
13 >>> circle.radius
14 Get radius
15 Traceback (most recent call last):
16 ...
17 AttributeError: 'Circle' object has no attribute '__radius'
18 >>> help(circle)
19 Help on Circle in module __main__ object:
20
21 class Circle(builtins.object)
22     ...
23     | radius
24     |     The radius property.
```

Fonction spéciale property - Exemple 3

■ Voici le même exemple avec la notion de decorator (décorateurs) :

```
1 # circle.py
2
3 class Circle:
4     def __init__(self, radius):
5         self.__radius = radius
6
7     @property
8     def radius(self):
9         """The radius property."""
10        print("Get radius")
11        return self.__radius
12
13    @radius.setter
14    def radius(self, value):
15        print("Set radius")
16        self.__radius = value
17
18    @radius.deleter
19    def radius(self):
20        print("Delete radius")
21        del self.__radius
```

Espaces de noms global, de l'objet, de la classe

- L'espace de noms global contient les noms définis au niveau du module.
- L'espace de noms de l'objet contient les attributs propres à chaque instance de la classe.
- L'espace de noms de la classe contient les attributs partagés par toutes les instances de la classe.
- L'accès aux attributs se fait en utilisant la notation pointée.

Variable de classe

- Une variable de classe est une variable définie au niveau de la classe et partagée par toutes les instances de la classe.
- Elle est définie en dehors des méthodes de la classe.
- Elle est accessible à partir de n'importe quelle instance de la classe ou même de la classe elle-même.

Exemple d'utilisation des variables de classe

- Exemple complet d'utilisation.

Constructeur à nombre d'arguments arbitraire

(*args, **kwargs)

- Les paramètres `*args` et `**kwargs` permettent de définir des constructeurs à nombre d'arguments arbitraire.
- `*args` permet de recevoir un nombre variable d'arguments positionnels.
- `**kwargs` permet de recevoir un nombre variable d'arguments nommés.
- Cela offre une grande flexibilité lors de l'instanciation d'objets.

Exemple d'utilisation de *args et **kwargs dans un constructeur

```
1 class MyClass:
2     def __init__(self, *args, **kwargs):
3         for arg in args:
4             print("Arg:", arg)
5         for key, value in kwargs.items():
6             print("Keyword arg - {}: {}".format(key, value))
7
8 # Instanciation de l'objet avec différents types d'arguments
9 obj1 = MyClass(1, 2, 3, name="Alice", age=30)
10 obj2 = MyClass("Hello", name="Bob", age=25)
```

Association

- L'association est une relation entre deux classes en Python.
- Elle permet à des objets de chaque classe de collaborer entre eux.
- Les objets de chaque classe peuvent exister indépendamment les uns des autres.

Exemple d'association

```
1 class Player:
2     def __init__(self, name):
3         self.name = name
4
5 class Team:
6     def __init__(self, team_name):
7         self.team_name = team_name
8         self.players = []
9
10    def add_player(self, player):
11        self.players.append(player)
12
13 # Create objects
14 player1 = Player("John")
15 player2 = Player("Alice")
16 team = Team("Team A")
17
18 # Associate objects
19 team.add_player(player1)
20 team.add_player(player2)
```

Agrégation

- L'agrégation est une technique de construction de classes en Python.
- L'agrégation consiste à ajouter des objets d'autres classes comme attributs de la classe courante.
- Elle permet de créer des relations entre les classes et de réutiliser du code de manière efficace.

Exemple d'agrégation

```
1 class Engine:
2     def __init__(self, horsepower):
3         self.horsepower = horsepower
4
5 class Car:
6     def __init__(self, model, engine):
7         self.model = model
8         self.engine = engine
9
10 engine = Engine(200)
11 car = Car("Toyota Camry", engine)
```

Composition

- La composition est une technique de construction de classes en Python.
- La composition consiste à inclure des objets d'autres classes en tant qu'attributs de la classe courante et à les initialiser dans le constructeur.
- Elle permet de créer des relations entre les classes et de réutiliser du code de manière efficace.

Exemple de composition

```
1 class Engine:
2     def __init__(self, horsepower):
3         self.horsepower = horsepower
4
5 class Car:
6     def __init__(self, model):
7         self.model = model
8         self.engine = Engine(200)
9
10 car = Car("Toyota Camry")
```

Différences entre Association, Agrégation et Composition

■ Association :

- C'est une relation entre deux classes.
- Les objets de chaque classe peuvent exister indépendamment les uns des autres.

■ Agrégation :

- C'est une relation tout-partie.
- L'objet de la classe agrégée peut exister indépendamment de l'objet de la classe conteneur.

■ Composition :

- Relation tout-ou-rien.
- C'est une forme plus forte d'agrégation.
- L'objet de la classe composée ne peut pas exister sans l'objet de la classe conteneur.

Concepts objets, et diagrammes de classes - lien avec la POO

- La figure ci-dessous explicite les diverses notions de relation existantes:







Class Diagram Relationship Type	Notation
Association	
Inheritance	
Realization/ Implementation	
Dependency	
Aggregation	
Composition	

Figure: relationships

Principe de l'héritage

Définition: L'héritage est un principe fondamental de la programmation orientée objet (POO) qui permet à **une classe d'hériter des caractéristiques (attributs et méthodes) d'une autre classe.**

- La classe qui hérite est appelée **sous-classe** ou **classe dérivée**.
- La classe dont on hérite est appelée **classe de base** ou **classe parente**.

Référence 1 : Documentation Python - Héritage

Référence 2 : d'avantage de documentation sur l'héritage.

Les Types d'héritage en Python

L'Héritage en Python se décompose en différents types qui déterminent comment une classe dérive des propriétés d'une autre.

- **Héritage Simple:** Une classe hérite des propriétés d'une seule autre classe de base.
- **Héritage Multiple:** Une classe hérite des propriétés de plusieurs classes de base. Héritage Multiple
- **Héritage Hiérarchique:** Plusieurs classes dérivent des propriétés d'une seule classe de base.
- **Héritage Hybride:** Combinaison d'héritage multiple et hiérarchique.

Exemple d'Héritage Simple en Python

```
1 # Simple inheritance
2 class Animal:
3     def speak(self):
4         print("Animal speaks")
5
6 class Dog(Animal):
7     def bark(self):
8         print("Dog barks")
9
10 # Derived class usecase
11 dog_instance = Dog()
12 dog_instance.speak()
13 dog_instance.bark()
```

Exemple d'Héritage Multiple en Python

```
1 # Multiple inheritance
2 class Animal:
3     def speak(self):
4         print("Animal speaks")
5
6 class Flyable:
7     def fly(self):
8         print("Can fly")
9
10 class Bird(Animal, Flyable):
11     pass
12
13 # Derived class usecase
14 bird_instance = Bird()
15 bird_instance.speak()
16 bird_instance.fly()
```

Exemple d'Héritage Hiérarchique en Python

```
1 # Hierarchical Inheritance
2 class Animal:
3     def speak(self):
4         print("Animal speaks")
5
6 class Cat(Animal):
7     def purr(self):
8         print("Cat purrs")
9
10 class Tiger(Cat):
11     def roar(self):
12         print("Tiger roars")
13
14 # Derived class usecase
15 tiger_instance = Tiger()
16 tiger_instance.speak()
17 tiger_instance.purr()
18 tiger_instance.roar()
```

Exemple d'Héritage Hybride en Python

```
1 class Animal:
2     def speak(self):
3         print("Animal speaks")
4
5 class Bird(Antimal):
6     def fly(self):
7         print("Bird flies")
8
9 class Fish(Antimal):
10    def swim(self):
11        print("Fish swims")
12
13 class Amphibian(Bird, Fish):
14     def jump(self):
15         print("Amphibian jumps")
16
17 # Derived class use case
18 amphibian_instance = Amphibian()
19 amphibian_instance.speak() # Output: Animal speaks
20 amphibian_instance.fly()  # Output: Bird flies
21 amphibian_instance.swim()  # Output: Fish swims
22 amphibian_instance.jump()  # Output: Amphibian jumps
```

Problème du diamant

- En programmation orientée objet, le problème du diamant survient lorsqu'une classe hérite de deux classes qui elles-mêmes héritent d'une même classe.
- Cela crée une ambiguïté lors de l'appel de méthodes ou d'attributs qui sont définis dans la classe parente commune.
- Python gère le problème du diamant en utilisant l'ordre de résolution des attributs mro (Method Resolution Order), qui spécifie l'ordre dans lequel les classes sont explorées pour trouver les attributs.

Problème du diamant - Suite

- L'ordre de résolution des attributs est déterminé par l'algorithme C3 de Python, qui garantit la cohérence et la prévisibilité dans la résolution des méthodes et des attributs.
- Malgré cela, il est recommandé de concevoir des hiérarchies de classes de manière à éviter le problème du diamant, lorsque cela est possible, en utilisant des modèles de conception appropriés.

Problème du diamant - Exemple

```
1 class A:
2     def my_method(self):
3         print("my_method called from class A")
4
5 class B(A):
6     def my_method(self):
7         print("my_method called from class B")
8         A.my_method(self)
9
10 class C(A):
11     def my_method(self):
12         print("my_method called from class C")
13         A.my_method(self)
14
15 class D(C, B):
16     def my_method(self):
17         print("my_method called from class D")
18         C.my_method(self)
19         B.my_method(self)
20
21 instance_of_class_D = D()
22 instance_of_class_D.my_method() # Output: DCABA
23 print(D.mro())                 # Output: DCBAo
```

Problème du diamant - Solution

```
1 class A:
2     def my_method(self):
3         print("my_method called from class A")
4
5 class B(A):
6     def my_method(self):
7         print("my_method called from class B")
8         A.my_method(self)
9
10 class C(A):
11     def my_method(self):
12         print("my_method called from class C")
13         A.my_method(self)
14
15 class D(C, B):
16     def my_method(self):
17         print("my_method called from class D")
18         C.my_method(self)
19         B.my_method(self)
20
21 instance_of_class_D = D()
22 instance_of_class_D.my_method() # Output: DCBA
23 print(D.mro())                 # Output: DCBAo
```

Fonctions `issubclass()`, `super()` et méthode `mro()`

- Les fonctions `issubclass()`, `super()` et la méthode `mro()` sont des outils puissants de la programmation orientée objet en Python.
- Elles permettent de gérer l'héritage des classes et d'accéder à la hiérarchie d'héritage.
- La fonction `issubclass()` permet de vérifier si une classe est une sous-classe d'une autre.
- La fonction `super()` permet d'accéder aux méthodes et attributs de la classe parente.
- La méthode `mro()` (Method Resolution Order) mro retourne l'ordre de résolution des méthodes pour une classe.

Exemples d'utilisation

```
1 class Animal:
2     def speak(self):
3         return "Animal speaks"
4
5 class Dog(Animal):
6     def speak(self):
7         return "Dog barks"
8
9 class Labrador(Dog):
10    def speak(self):
11        return "Labrador barks"
12
13 # Utilisation de isinstance()
14 print(isinstance(Dog, Animal)) # Output: True
15
16 # Utilisation de super()
17 labrador = Labrador()
18 print(super(Labrador, labrador).speak()) # Output: Dog barks
19
20 # Utilisation de mro()
21 print(Labrador.mro()) # Output: [Labrador, Dog, Animal, object]
```

La fonction `super()` en Python

- En Python, la fonction `super()` est utilisée pour accéder aux méthodes et aux attributs de la classe parente dans les sous-classes.
- Elle est souvent utilisée lorsque vous redéfinissez une méthode dans une sous-classe et que vous souhaitez toujours appeler la version de la méthode de la classe parente.

Accès aux attributs de la classe parente

- L'accès aux attributs de la classe parente se fait en utilisant `super()` dans le constructeur de la sous-classe.

```
1 class Parent:
2     def __init__(self, x):
3         self.x = x
4
5 class Child(Parent):
6     def __init__(self, x, y):
7         super().__init__(x) # Appel du constructeur de la classe parente
8         self.y = y
9
10 child = Child(10, 20)
11 print(child.x) # Accès à l'attribut de la classe parente
```

Accès aux méthodes de la classe parente

- Pour accéder à une méthode de la classe parente dans une sous-classe, utilisez `super()` suivi du nom de la méthode et des arguments appropriés.

```
1 class Parent:
2     def method(self):
3         print("Method from Parent class")
4
5 class Child(Parent):
6     def method(self):
7         super().method() # Appel de la méthode de la classe parente
8         print("Method from Child class")
9
10 child = Child()
11 child.method()
```


Super en héritage multiple

- Lorsqu'il y a héritage multiple, `super()` suit l'ordre de la résolution des méthodes (MRO) pour appeler les méthodes des classes parentes dans un ordre cohérent.

Utilisation dans l'abstraction et le polymorphisme

- Super est souvent utilisée dans la mise en œuvre de l'abstraction et du polymorphisme pour créer des hiérarchies de classes avec des comportements cohérents et des méthodes redéfinies selon les besoins.

Méthode de Résolution des Ordres (MRO) en Python

- La méthode de résolution des ordres (MRO) est utilisée pour déterminer l'ordre dans lequel les méthodes sont recherchées et appelées dans une hiérarchie de classes.
- La MRO est calculée automatiquement pour chaque classe en utilisant l'algorithme C3.
- La MRO peut être accédée en utilisant la méthode `mro()` ou l'attribut spécial `__mro__`.
- La méthode `mro()` renvoie une liste ordonnée des classes dans l'ordre de recherche des méthodes, tandis que l'attribut `__mro__` renvoie un tuple.

Exemple d'utilisation de la MRO

```
1 class A:
2     pass
3
4 class B(A):
5     pass
6
7 class C(A):
8     pass
9
10 class D(C, B):
11     pass
12
13 print(D.mro())
14 # Output:
15 # [<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
16 print(D.__mro__)
17 # Output:
18 # (<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

Notion d'abstraction

- L'**abstraction** est un concept fondamental de la programmation orientée objet (POO).
- Elle consiste à masquer les détails d'implémentation d'un objet et à se concentrer uniquement sur son comportement ou ses caractéristiques essentielles.
- En Python, l'abstraction est souvent réalisée en définissant des classes abstraites et en utilisant des méthodes abstraites pour définir une interface commune.

Classes abstraites en Python

- En Python, une classe abstraite est une classe qui ne peut pas être instanciée directement et qui sert de modèle pour d'autres classes.
- On utilise le module `abc` (Abstract Base Classes) pour créer des classes abstraites.
- Pour déclarer une classe abstraite, on utilise le décorateur `@abstractmethod` pour marquer les méthodes abstraites, c'est-à-dire les méthodes qui doivent être implémentées par les classes dérivées.

Classes abstraites - Exemple

```
1 from abc import ABC, abstractmethod
2
3 class Shape(ABC):
4     @abstractmethod
5     def area(self):
6         pass
7
8     @abstractmethod
9     def perimeter(self):
10        pass
11
12 class Rectangle(Shape):
13     def __init__(self, length, height):
14         self.length = length
15         self.height = height
16
17     def area(self):
18         return self.length * self.height
19
20     def perimeter(self):
21         return 2 * (self.length + self.height)
```

Classes abstraites - Exemple - Suite

```
1 class Circle(Shape):
2     def __init__(self, radius):
3         self.radius = radius
4
5     def area(self):
6         return 3.14 * self.radius ** 2
7
8     def perimeter(self):
9         return 2 * 3.14 * self.radius
10
11 # Example usage
12 rectangle = Rectangle(5, 4)
13 circle = Circle(3)
14
15 print("Rectangle Area:", rectangle.area())      # Output: Rectangle Area: 20
16 print("Rectangle Perimeter:", rectangle.perimeter())  # Output: Rectangle Perimeter: 18
17 print("Circle Area:", circle.area())            # Output: Circle Area: 28.26
18 print("Circle Perimeter:", circle.perimeter())    # Output: Circle Perimeter: 18.84
```


Notion: Interface

- En programmation orientée objet, une interface est un ensemble de méthodes qu'une classe doit implémenter.
- Les interfaces définissent un contrat ou une spécification pour les classes qui les implémentent.
- En Python, il n'y a pas de mot-clé pour définir explicitement une interface, mais on utilise souvent des classes abstraites pour ce but.
- Les classes qui implémentent une interface doivent fournir des définitions pour toutes les méthodes déclarées dans cette interface.

Utilisation des classes abstraites comme interfaces

- En Python, on utilise souvent des classes abstraites pour définir des interfaces.
- On utilise le module `abc` (Abstract Base Classes) pour créer des interfaces.
- Les classes qui implémentent une interface héritent de la classe abstraite correspondante et fournissent des implémentations pour toutes les méthodes abstraites définies dans cette classe.

Interfaces - Exemple

```
1 from abc import ABC, abstractmethod
2
3 class PaymentGateway(ABC):
4     @abstractmethod
5     def process_payment(self, amount):
6         pass
7
8     @abstractmethod
9     def refund_payment(self, transaction_id):
10         pass
11
12 class PayPal(PaymentGateway):
13     def process_payment(self, amount):
14         print("Processing payment of amount", amount, "via PayPal")
15
16     def refund_payment(self, transaction_id):
17         print("Refunding payment for transaction ID", transaction_id, "via PayPal")
```

Interfaces - Exemple - Suite

```
1 class Stripe(PaymentGateway):
2     def process_payment(self, amount):
3         print("Processing payment of amount", amount, "via Stripe")
4
5     def refund_payment(self, transaction_id):
6         print("Refunding payment for transaction ID", transaction_id, "via Stripe")
7
8 # Example usage
9 paypal = PayPal()
10 stripe = Stripe()
11
12 paypal.process_payment(100)    # Output: Processing payment of amount 100 via PayPal
13 stripe.process_payment(200)   # Output: Processing payment of amount 200 via Stripe
14
15 paypal.refund_payment("123456") # Output: Refunding payment for transaction ID 123456 via PayPal
16 stripe.refund_payment("789012") # Output: Refunding payment for transaction ID 789012 via Stripe
```

Classes abstraites vs Interfaces

Table: Différences et similitudes entre les classes abstraites et les interfaces

Classes abstraites	Interfaces
Peut contenir des méthodes concrètes en plus des méthodes abstraites	Ne peut contenir que des méthodes abstraites
Peut avoir des méthodes normales	Ne peut pas avoir de méthodes normales
Les sous-classes peuvent fournir des implémentations pour certaines méthodes et laisser d'autres méthodes abstraites	Les classes qui implémentent une interface doivent fournir des implémentations pour toutes les méthodes déclarées dans cette interface
Peut avoir des variables de classe et d'instance	Ne peut pas avoir de variables de classe ou d'instance
Héritage multiple possible	Héritage multiple possible
Utilisation du module <code>abc</code> pour définir	Utilisation du module <code>abc</code> pour définir

Remarques sur les classes abstraites

1. Une classe dérivée d'une classe abstraite peut redéfinir que certaines méthodes abstraites, dans ce cas, elle devient elle-même abstraite, et par conséquent l'on ne peut plus l'instancier.
2. Une classe peut-elle dériver de plusieurs classes abstraites ?
Oui, Python permet l'héritage multiple, donc une classe peut hériter de plusieurs classes abstraites.
3. Une classe abstraite peut-elle dériver d'une autre classe abstraite ?
Oui, une classe abstraite peut hériter d'une autre classe abstraite en Python.

Remarques sur les interfaces

4. Une classe peut-elle dériver de plusieurs interfaces ?

Oui, Python ne supporte pas nativement les interfaces, mais une classe peut implémenter plusieurs classes contenant uniquement des méthodes abstraites, ce qui est souvent utilisé comme des interfaces.

5. Une classe peut-elle dériver d'une ou plusieurs classes abstraites ET interfaces ?

Oui, une classe peut hériter d'une ou plusieurs classes abstraites et implémenter plusieurs interfaces.

6. Une interface peut-elle hériter d'une interface ou d'une classe abstraite ?

Oui, en Python, une interface (une classe ne contenant que des méthodes abstraites) est représentée par une classe abstraite, de ce fait, une interface peut hériter d'une autre interface ou d'une classe abstraite.

Polymorphisme

- Le polymorphisme est un concept clé de la programmation orientée objet (POO).
- Il fait référence à la capacité d'un objet à prendre plusieurs formes ou à se comporter de différentes manières en fonction du contexte dans lequel il est utilisé.
- En Python, le polymorphisme peut être mis en œuvre de différentes manières, notamment par le biais du `method overloading` (surchargement de méthode), de l'héritage et du `duck typing`.

Types de Polymorphisme en général

- Polymorphisme de compile-time :
 - Surcharge de méthode ou de fonction (method overloading) ;
 - Surcharge d'opérateur (operator overloading).
- Polymorphisme de runtime :
 - Remplacement de méthode (method overriding) ;

Compiletime polymorphism

- Le polymorphisme au moment de la compilation se réfère à la résolution du polymorphisme pendant la compilation du programme.
- En Python, le polymorphisme au moment de la compilation n'est pas aussi commun qu'en langages statiquement typés comme C++ ou Java.
- Cependant, le polymorphisme au moment de la compilation peut être réalisé en utilisant des techniques telles que le surchargement de méthode et le surchargement d'opérateur.

Compiletime polymorphism - Sous-types

- **Surcharge de méthode utilisant des valeurs de paramètres par défaut** : La surcharge de méthode en utilisant des valeurs de paramètres par défaut permet de définir plusieurs méthodes avec la même signature mais des comportements différents en fonction des paramètres fournis lors de l'appel.
- **Surcharge de méthode utilisant des listes d'arguments de longueur variable** : Python permet de définir des méthodes avec des listes d'arguments de longueur variable à l'aide des paramètres `*args` et `**kwargs`. Cela permet de traiter un nombre variable d'arguments lors de l'appel de la méthode.

Compiletime polymorphism - Sous-types - Suite

- **Utilisation des annotations de fonction pour spécifier les types de paramètres** : Python prend en charge les annotations de fonction pour spécifier les types de paramètres et de retour d'une fonction.
- **Surcharge d'opérateur (opérateur overloading)** : La surcharge d'opérateur permet de redéfinir le comportement des opérateurs standard pour les objets d'une classe. Cela se fait en implémentant des méthodes spéciales définies par Python pour chaque opérateur.

Surcharge de méthode

- La surcharge de méthode est le fait de définir plusieurs méthodes avec le même nom mais avec des signatures de paramètres différentes.
- En Python, la surcharge de méthode n'est pas directement prise en charge car le langage ne prend pas en compte les types de paramètres lors de l'appel de méthodes.
- Cependant, on peut simuler la surcharge de méthode en utilisant des valeurs par défaut pour les paramètres ou en utilisant des arguments arbitraires.

Surcharge d'opérateur

- La surcharge d'opérateur se réfère à la redéfinition des opérateurs prédéfinis tels que +, -, *, /, etc., pour les types d'objets personnalisés.
- En Python, la surcharge d'opérateur est réalisée en définissant des méthodes spéciales dans les classes, telles que `__add__`, `__sub__`, `__mul__`, `__truediv__`, etc.
- Cela permet d'utiliser des opérateurs prédéfinis avec des objets personnalisés et de leur donner un comportement approprié.

Exemple de polymorphisme par surcharge d'opérateur

```
1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __add__(self, other):
7         return Point(self.x + other.x, self.y + other.y)
8
9     def __str__(self):
10        return f"Point({self.x}, {self.y})"
11
12 p1 = Point(2, 3)
13 p2 = Point(5, 7)
14 p3 = p1 + p2 # calls __add__
15
16 print(p3) # calls __str__
17 # Output: Point(7, 10)
```

Runtime polymorphism

- Le polymorphisme au moment de l'exécution se réfère à la résolution du polymorphisme pendant l'exécution du programme.
- En Python, le polymorphisme au moment de l'exécution est courant en raison de la nature dynamique du langage.
- Il est souvent réalisé par le biais du duck typing, où les objets sont évalués sur la base de leur comportement plutôt que de leur type.

Runtime polymorphism - Sous-types

- **Redéfinition de méthode (method overriding)** : Le polymorphisme de runtime en Python est souvent implémenté par la redéfinition de méthode, où une méthode dans une classe enfant remplace (ou « masque ») une méthode avec le même nom dans la classe parent.
- **Implémentation de méthodes abstraites en utilisant des classes de base abstraites (ABC)** : Les classes abstraites sont utilisées pour définir des méthodes abstraites qui doivent être implémentées par les sous-classes. Python fournit le module `abc` pour créer des classes abstraites.
- **Duck typing** : Le duck typing est une technique utilisée en Python où le type d'un objet est déterminé par son comportement plutôt que par son type explicite. Si un objet possède une méthode ou un attribut spécifique, il est traité comme s'il appartenait à un type particulier.