Vererbung

Vererbung in Python

Definition

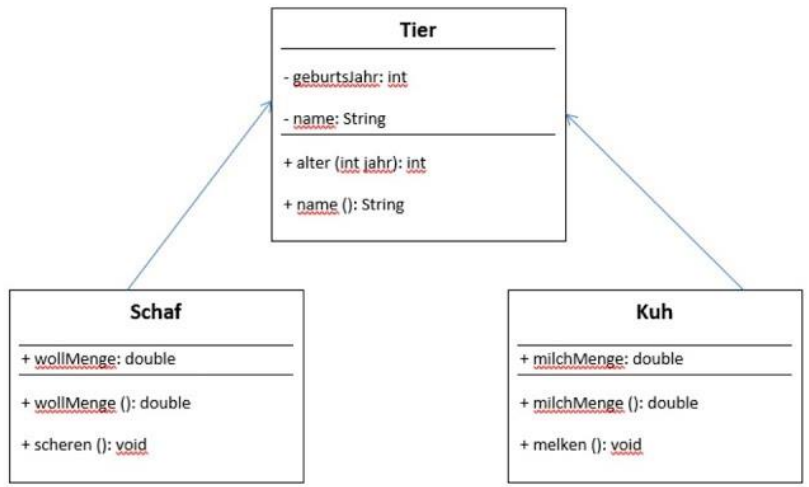
Vererbung beschreibt die Vorgehensweise, eine neue Klasse als Erweiterung einer bereits bestehenden Klasse zu entwickeln.

Die neue Klasse wird auch Sub-, Kind- oder Unterklasse genannt.

Die bestehende Klasse wird Basis-, Eltern- oder Superklasse genannt.

WAS IST VERERBUNG?

(OBJEKTORIENTIERTE PROGRAMMIERUNG)



Übernehmen, ergänzen und überschreiben

Beim Vererben übernimmt die Subklasse die Attribute und Methoden der Basisklasse.

Eine übernommene Methode kann dabei überschrieben (d. h. neu definiert) werden.

Die Subklasse kann dann noch zusätzliche Attribute und Methoden ergänzen.

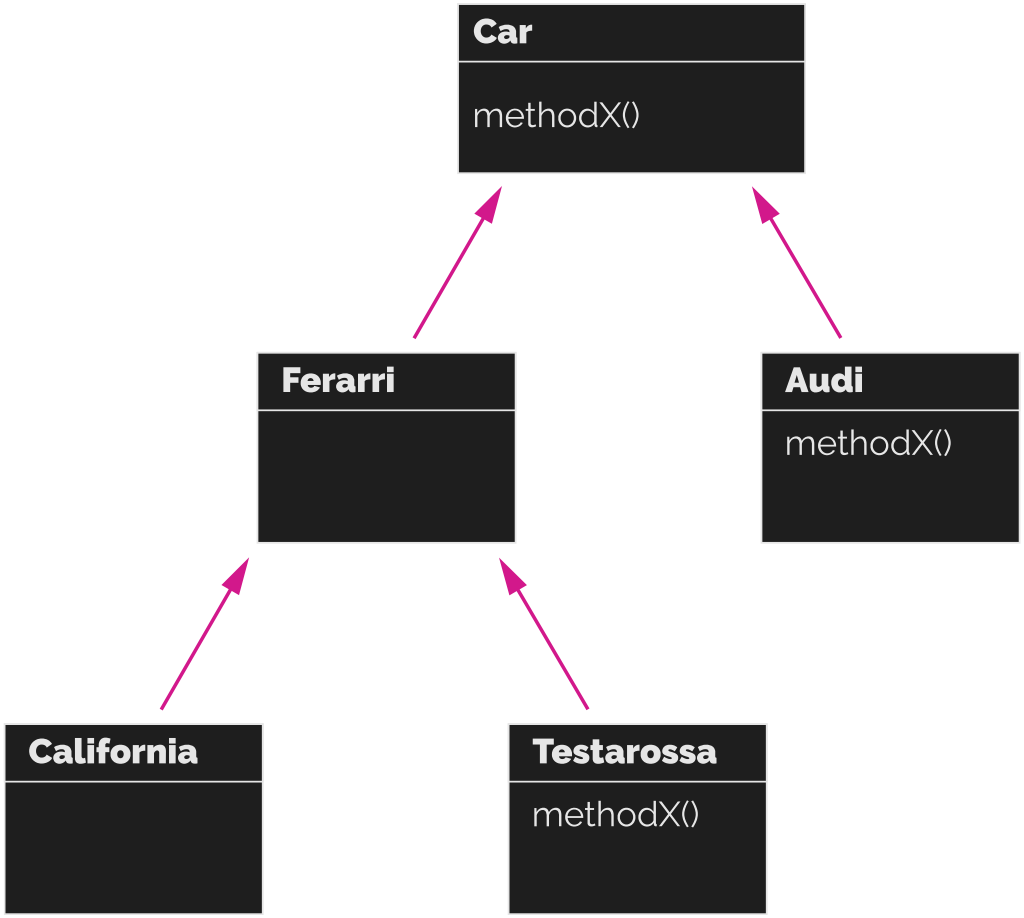
Die Klasse Schaf aus der Seite davor übernimmt die Methoden alter() und name() sowie die Attribute geburtsjahr und name aus der Elternklasse Tier.

Spezialisierung und Generalisierung

Vererbung kann dann ins Spiel kommen, wenn eine Klasse als Spezialisierung einer anderen Klasse konzipiert wird.

```
class Car:
    def __init__(name):
        self.name = name

class Bmw(Car):
    def __init__(name):
        self.name = name
```


Car
methodX()
Ferarri
Audi
methodX()
California
Testarossa
methodX()

Elternkonstruktor initialisieren mit super()

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

Um das Elternobjekt mit den nötigen Werten zu initialisieren, müssen wir die Funktion super nutzen, um den Elternkonstruktor __init__() aufzurufen.

Super

super() ist eine eingebaute Funktion, die verwendet wird, um auf die Methoden und Attribute der Elternklasse (der sogenannten Superklasse oder Basisklasse) in einer abgeleiteten Klasse (Unterklasse oder abgeleitete Klasse) zuzugreifen.

Die Verwendung von super() ist häufig in der Vererbungshierarchie nützlich, wenn man in der abgeleiteten Klasse eine Methode überschreiben möchten, die in der Elternklasse bereits definiert ist.

class object

Die Klasse object bildet die Wurzel der Klassenhierarchie und dient als grundlegendes Bausteinelement für alle benutzerdefinierten und integrierten Klassen in Python. Sie ist die Grundklasse, von der alle anderen Klassen erben. Sie stellt einige grundlegende Methoden und Verhaltensweisen bereit, wie z. B. __str__, __repr__

Seit Python 3.x muss nicht mehr spezifiert werden, dass von object geerbt wird. Jede Klasse erbt implizit von object.

```
class Dog(object):
    ...
```

oder

```
class Dog:
    ...
```

Method Resolution Order

Method Resolution Order (MRO) ist ein Konzept in Python, das die Reihenfolge beschreibt, in der Python Klassenmethoden sucht und aufruft. Dies ist wichtig, wenn es in einer Vererbungshierarchie mehrere Klassen gibt und die Methode in verschiedenen Klassen mit dem gleichen Namen vorhanden ist.

Die Klasse, die am weitesten links in der MRO steht, liefert die Methode, die beim Aufruf gewählt wird.

Dog.mro()

oder

Dog.__mro__()

Beispiel

```
class MyClass1:
    def say_hello(self):
            print("Hello")

class MyClass2:
    def say_hello(self):
            print("Hello")

class MyMultiDerivedClass(MyClass1, MyClass2):
    pass

print(MyMultiDerivedClass.mro())
```

Polymorphie

Polymorphie ist ein wichtiges Konzept in der objektorientierten Programmierung, das es einem Objekt ermöglicht, sich auf unterschiedliche Weisen zu verhalten, abhängig von seinem konkreten Typ oder seiner Klasse.

Die Polymorphie in der Programmierung ermöglicht es, dass Objekte unterschiedlicher Klassen eine gemeinsame Schnittstelle teilen und sich in verschiedenen Formen verhalten können.

Eine gängige Methode zur Implementierung von Polymorphie in Python besteht darin, Methoden in der Basisklasse zu definieren und diese Methoden in den abgeleiteten Klassen zu überschreiben. Dies ermöglicht es den abgeleiteten Klassen, die Methode auf ihre eigene Weise zu implementieren.

Methodenüberschreibung Beispiel

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

Duck Typing

Duck-Typing ist ein ähnliches Konzept wie Polymorphismus, das besagt, dass die Eignung eines Objekts für eine bestimmte Operation nicht auf seinen Datentyp, sondern auf sein Verhalten oder seine Methoden basiert.

Mit anderen Worten, "Wenn es wie eine Ente aussieht und sich wie eine Ente verhält, dann ist es eine Ente."

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck (James Whitcomb Riley).

Dies ermöglicht eine lose Bindung zwischen Objekten und deren Verwendung in Funktionen oder Methoden.

Duck Typing

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
     def name(self):
        print("ITS A DUCK NO NAME")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def name(self):
        print("John Smith")

def in_the_forest(duck):
    duck.quack()
    duck.feathers()
    duck.name()

for element in [Duck(), Person()]:
      in_the_forest(element)
```