Einführung in Python - Teil 4

Objektorientierte Programmierung: Teil 2

Vererbung

Wie modellieren wir die Klassen, wenn wir sehr viele Objekte mit ähnlichen oder gleichen Eigenschaften haben? Beispiel: **Tiere im Zoo**

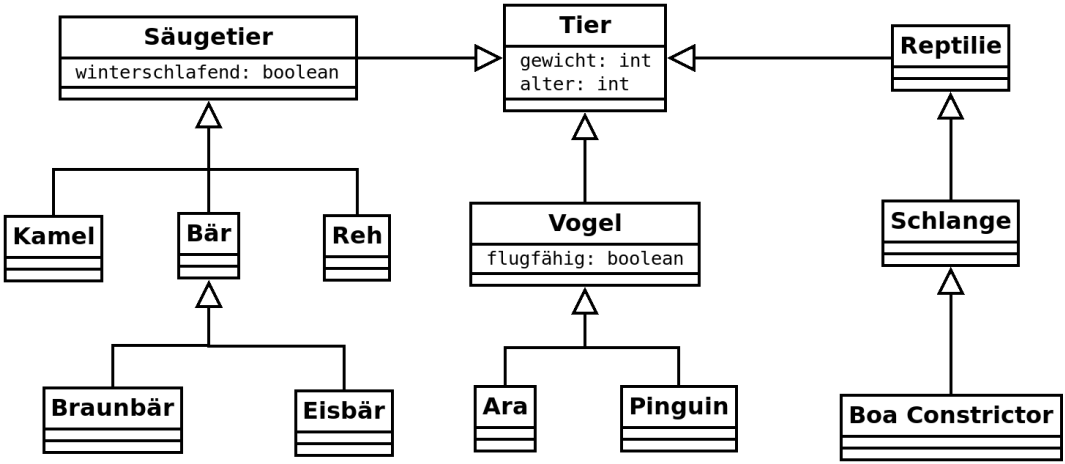
- Einteilung in Säugetiere, Vögel, Reptilien, dabei wiederum andere Unterteilungen bis zur Einteilung in konkrete Tierart
- Alle Tiere haben ähnliche Eigenschaften wie Gewicht, Alter, ...
- Spezielle Tiere haben zusätzliche Eigenschaften wie „Winterschlaf haltend“, warm- oder kaltblütig, ...

Lösungsansätze:

- Modellieren wir alles in einer ganz großen Klasse Tier?
- Modellieren wir jedes Tier für sich selbst mit den entsprechenden Attributen (Achtung: Fehlerquelle!)?

⇒ **Lösung**: Verwendung von Vererbung

Beispiel: Vererbung Zoo (UML, Ausschnitt)



Vererbung in Python

Eine Subklasse („Kindklasse“) erbt von einer Superklasse („Elternklasse“), indem der Name der Superklasse bei der Klassendefinition der Subklasse in Klammern angegeben wird:

```
class Subklasse(Superklasse):
```

Ein Aufruf des Konstruktors der Superklasse ist mit `super().__init__()` möglich

(Eine Mehrfachvererbung ist durch die Angabe mehrerer Elternklassen möglich, aber meistens nicht empfehlenswert [Andere Sprachen, zum Beispiel Java, verbieten Mehrfachvererbung ganz])

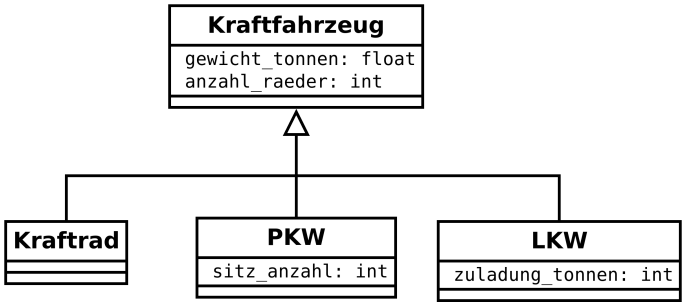
Beispiel: Vererbung Zoo (Python, Ausschnitt)

```python
class Tier:
    def __init__(self, alter: int, gewicht: int):
        self.alter: int = alter
        self.gewicht: int = gewicht

class Saeugetier(Tier):
    def __init__(self, alter: int, gewicht: int, winterschlafend: bool):
        super().__init__(alter, gewicht)
        self.winterschlafend: bool = winterschlafend

s1 = Saeugetier(10, 125, True)
assert (s1.gewicht == 125)
assert (s1.alter == 10)
assert (s1.winterschlafend)
```

Aufgabe: Vererbung

Gegeben Sei folgende Vererbungshierarchie:

- Kraftfahrzeug
  - Kraftrad
  - Personenkraftwagen (PKW)
  - Lastkraftwagen (LKW)



Definieren Sie die Einteilung als Klassen mit Vererbung und achten Sie auf die Vererbung der Attribute!

Lösung: Vererbung (1)

```python
class Kraftfahrzeug:
    def __init__(self, gewicht_tonnen: float, anzahl_raeder: int):
        self.gewicht_tonnen: float = gewicht_tonnen
        self.anzahl_raeder: int = anzahl_raeder

class Kraftrad(Kraftfahrzeug):
    def __init__(self, gewicht_tonnen: float):
        super().__init__(gewicht_tonnen, 2)

krad = Kraftrad(0.7)
print(krad.gewicht_tonnen)  # 0.7
print(krad.anzahl_raeder)  # 2
```

Lösung: Vererbung (2)

```python
class PKW(Kraftfahrzeug):
    def __init__(self, gewicht_tonnen: float, sitz_anzahl: int):
        super().__init__(gewicht_tonnen, 4)
        self.sitz_anzahl: int = sitz_anzahl

pkw = PKW(1.6, 5)
print(pkw.anzahl_raeder)  # 4

class LKW(Kraftfahrzeug):
    def __init__(self, gewicht_tonnen: float, zuladung_tonnen: float):
        super().__init__(gewicht_tonnen, 16)
        self.zuladung_tonnen: float = zuladung_tonnen

lkw = LKW(5.8, 42.0)
print(lkw.anzahl_raeder) # 16
```

Motivation: Enums

In verschiedenen Situationen werden beim Programmieren Datentypen mit einer festen Anzahl von Werten (Instanzen) benötigt, zum Beispiel:

- Die drei Grundfarben im RGB-System (Rot, Grün, Blau)
- Schwierigkeitsgrade einer Aufgabe (Leicht, Mittel, Schwer)
- ...

Solche Datentypen lassen sich prinzipiell auch mit Strings oder mit Zahlen modellieren, dies kann jedoch leicht zu Fehlern führen:

```
rgb_color: Dict[str, int] = {"red": 255, "blue": 81, "green": 125}


print(rgb_color["yellow"])
# KeyError: 'yellow'
```

Enums (1)

Im Paket `enum` gibt es die Basisklasse `Enum`, um solche Aufzählungstypen zu erstellen:

```
from enum import Enum


class BaseColors(Enum):
    Red = "red"
    Blue = "blue"
    Green = "green"
```

Hiermit wird ein Aufzählungstyp `Color` erstellt, der die drei Werte `Red`, `Blue` und `Green` annehmen kann.

Statt der Zuordnung zu Strings kann (je nach Situation) auch andere Datentypen (zum Beispiel Ganzzahlen) verwendet werden.

Enums (2)

```
# Green -- green
print(BaseColors.Green.name, '--', BaseColors.Green.value)
print(BaseColors('blue'))  # BaseColors.Blue
rgb_color = {
    BaseColors.Red: 255,
    BaseColors.Blue: 81,
    BaseColors.Green: 125
}
print(rgb_color[BaseColors.Green])  # 125

print(rgb_color[BaseColors.Blu])  # AttributeError: Blu
print(rgb_color["red"])  # KeyError: 'red'
# Warnung bei Verwendung einer IDE (z. B. PyCharm)
```

Test Driven Development

TDD - Diesmal richtig

Unser bisheriges Testkonzept mit `assert` hat noch ein paar kleine „Macken“:

- Die Fehlermeldung ist nicht wirklich aussagekräftig
- Die Tests lassen sich nur umständlich der getesteten Methode zuordnen
- ...

Daher werden wir nun die Klasse `TestCase` aus dem Paket `unittest` kennenlernen, die einige dieser Probleme behebt.

Die Basisklasse TestCase

„Echte“ Unittests werden in Python über die Klasse `TestCase` realisiert. Diese muss aus dem Paket `unittest` importiert werden.

Es muss eine Testklasse erstellt werden, die `TestCase` erweitert. Es ist Konvention, diese Klasse mit dem Namen der zu testenden Klasse und dem Suffix „Test“ zu benennen.

Für jede zu testende Methode wird eine Testmethode erstellt, die normalerweise mit dem Präfix „test_“ und dem Namen der zu testenden Methode benannt wird.

Die Klasse `TestCase` stellt verschiedene `assert`*X*-Methoden bereit, die bessere Fehlermeldungen als `assert` bieten.

Minimalbeispiel Unittests

```python
import unittest

class MyClass:
    def __init__(self, x: bool):
        self.x = x

    def my_method(self) -> int:
        if self.x:
            return 0
        else:
            return 1
```

```python
class MyClassTest(unittest.TestCase):
    def test_my_method(self):
        self.assertEqual(0,
                         MyClass(True).my_method())
        self.assertEqual(1,
                         MyClass(False).my_method())

        # AssertionError: 1 != 0
        self.assertEqual(-1,
                         MyClass(False).my_method())

if __name__ == "__main__":
    # Run unit tests
    unittest.main()
```

Assert-Methoden von TestCase

Methode	Funktion
`assertEqual(exp, act)`	Gleichheitsvergleich
`assertAlmostEqual(exp, act)`	Gleichheit von Fließkommzahlen (Rundungsfehler!)
`assertNotEqual(exp, act)`	Ungleichheit
`assertTrue(bed)`	Ausdruch ist wahr
`assertFalse(bed)`	Ausdruck ist falsch
`assertRaises(typ)`	Exception wird geworfen

und viele mehr . . .

Beispiele: Assert-Methoden

```
class MyTestCase(TestCase):
    def test_examples(self):
        self.assertEqual(1, 1)
        self.assertAlmostEqual(1.0, 2 / 2)
        self.assertNotEqual(1, 2)
        self.assertTrue(1 == 1)
        self.assertFalse(1 != 1)
        self.assertGreater(2, 1)

        with self.assertRaises(Exception):
            raise Exception("...")
```

Aufgabe: TDD

Schreiben Sie eine Testklasse für folgende Klasse `Dish`, die ein Gericht darstellt:

```
class Dish:
    def __init__(self, name: str, price_cent: int):
        if price_cent <= 0:
            raise Exception("Preis muss größer als 0 sein!")
        self.name: str = name
        self.price_cent: float = price_cent

    def __eq__(self, other) -> bool:
        return isinstance(other, Dish) and self.name == other.name and \
               self.price_cent == other.price_cent

    def __str__(self) -> str:
        return "{}: {} €".format(self.name, (self.price_cent / 100.))
```

Lösung: TDD (1)

```
class DishTest(unittest.TestCase):
    def test_init(self):
        dish = Dish('my_dish', 100)
        self.assertEqual(dish.name, 'my_dish')
        self.assertEqual(dish.price_cent, 100)

        with self.assertRaises(Exception):
            Dish('illegal_dish', -1)

    def test_str(self):
        self.assertEqual("my_dish: 1.0 €",
                         str(Dish('my_dish', 100)))
        self.assertEqual("other_dish: 3.99 €",
                         str(Dish('other_dish', 399)))
```

Lösung: TDD (2)

```
def test_eq(self):
    dish = Dish('my_dish', 100)
    # compare with same instance
    self.assertEqual(dish, dish)
    # compare with equal instance
    self.assertEqual(dish, Dish('my_dish', 100))
    # compare with different instance
    self.assertNotEqual(dish, Dish('other_dish', 200))
    # compare with instance of other class
    self.assertNotEqual(dish, ('my_dish', 100))
```