Python – Grundlagen objektorientierter Programmierung

1 Eine einfache Klasse

Eine Klasse enthält den Programmcode zur allgemeinen Beschreibung von Objekten. Funktionen (Methoden) beschreiben das Verhalten der Objekte. Variablen (Attribute) repräsentieren Eigenschaften der Objekte.

Wir zeigen das Beispiel eines Fahrzeuges, mit den Eigenschaften Kfz-Kennzeichen und Kilometerstand. Das Fahrzeug kann fahren, tanken sowie das Kennzeichen anzeigen oder neu setzen.

```
class Fahrzeug():
    def __init__(self, km, kennz):
        self.km = km
        self.kennz = kennz

    def __str__(self):
        info = "Fahrzeug: "+self.kennz+" : "+str(self.km)+" km"
        return info

    def get_id(self):
        return self.kennz

    def set_id(self,kennz):
        if len(kennz)>0 and len(kennz)<10:
            self.kennz = kennz

    def fahren(self,weg):
        self.km = self.km + abs(weg)

    def tanken(self):
        print("Bitte nicht rauchen!!!")
```

Constructor: wird automatisch beim Erzeugen der Instanz aufgerufen. Belegt Instanzvariablen mit Werten

string-Methode: erzeugt einen String, der beim print der Klasseninstanz angezeigt wird.

get-Methode: liefert den Wert einer Instanzvariablen

set-Methode: setzt den Wert einer Instanzvariablen

Die Klasse Fahrzeug wird nun benutzt, um damit Dinge zu tun, Dafür werden Instanzen der Klasse gebildet und an Variablen zugewiesen. Hier im Beispiel wird die Variable car für eine Instanz der Klasse Fahrzeug verwendet.

```
car = Fahrzeug(101, 'DD A 123')
car.fahren(12)
car.set_id('DD B 222')
print(car)
```

```
Fahrzeug: DD B 222 : 113 km
```

.... Instanz erzeugen → Aufruf des Constructors
.... Aufruf der fahren-Methode
.... setzen eines neuen Kennzeichens
.... Anzeige der Instanz-Informationen
→ Aufruf __str__

2 Grundlagen der Vererbung

Vererbung ermöglicht es spezialisierte Unterklassen aus einer Oberklasse abzuleiten. Die Objekte der abgeleiteten Unterklassen verfügen damit über die gleichen Eigenschaften wie die Oberklasse. Sie können jedoch zusätzlich spezielle Attribute und Methoden einführen.

Wir zeigen das Beispiel eines Pkws als spezialisierte Unterklasse eines Fahrzeuges. Der Pkw kann alles, was ein Fahrzeug kann. Zusätzlich verfügt er über Sitze. Standardmäßig hat ein Pkw 4 Sitze. Über entsprechende get- und set-Methoden kann die Anzahl Sitze ermittelt bzw. verändert werden

```
class Pkw(Fahrzeug):
    def __init__(self, km, kennz, sitze=4):
        super().__init__(km, kennz)
        self.sitze = sitze

    def set_sitze(self, sitze):
        if sitze>0 and sitze<6:
            self.sitze = sitze

    def get_sitze(self):
        return self.sitze
```

Constructor: definiert per default 4 Sitze für Pkw-Objekte; ruft mit **super().__init__** den übergeordneten Fahrzeug-Constructor auf

```
t4 = Pkw(0, 'DD T 4')
s = t4.get_sitze()
t4.fahren(212)
print(t4)
print("Sitze:",s)
```

Aufruf einer Pkw-Methode
Aufruf einer Fahrzeug-Methode

```
Fahrzeug: DD T 4 : 212 km
Sitze: 4
```

Theoretische Grundlagen

Allgemeines Grundverständnis
Eine Klasse ist der Programmquelltext, der Eigenschaften (Attribute) und Verhalten (Methoden / Funktionen) von Objekten beschreibt.
Ein Objekt oder eine Instanz ist ein konkretes Exemplar einer Klasse, dass zum Beispiel durch eine Variable benutzbar ist (siehe 1).

Bei der objektorientierten Programmierung ist also stets zwischen Programmcode der Klassendefinition und dem anwendenden Programmcode, der mit den entsprechenden Instanzen arbeitet, zu unterscheiden.

Durch Vererbung oder Ableitung wird eine neue Klasse als spezielle Ausprägung einer allgemeineren Basisklasse gebildet. Dabei übernimmt die Spezialklasse alle Eigenschaften und Methoden der Basisklasse. Zudem kann sie neue Eigenschaften und Methoden ergänzen (siehe 2).

Was ist self?
Der Parameter self repräsentiert die Instanz, die ein Attribut oder eine Methode aufruft. Bei der Definition einer Methode ist er stets als erster Parameter anzugeben. Beim Methodenaufruf wird er nie mit angegeben. Er steht ja quasi als Instanzvariable vor der Methode.

Was ist super?
Die Funktion super() liefert eine Referenz auf die jeweils übergeordnete Basisklasse. Damit können für die übergeordnete Basisklasse Methoden und ggf. Klassenattribute zugegriffen werden.

Was sind magische Methoden?
Magische Methoden sind solche Methoden, die allgemein übliches Verhalten der Objekte definieren. Dazu gehören u.a. das Erzeugen mit dem Konstruktor, das Löschen mit dem Destruktor, die Arithmetik-Operationen, die Vergleichsoperationen und noch ein paar andere. Magische Methoden sind am doppelten Unterstrich (__) vor und nach dem standardmäßig vordefinierten Methodennamen erkennbar.
__init__ ... Konstruktor
__del__ ... Destruktor wird beim Löschen eines Objektes mit del aufgerufen
__str__ ... String-Darstellung des Objekts – wird bei print aufgerufen
__add__ __sub__ __mul__ __div__
... grundlegende Arithmetik-Operationen
__lt__ __eq__ __ge__
... Beispiele für Vergleichsoperationen

3 Freier Zugriff auf Attribute

Standardmäßig ist der Zugriff auf Attribute einer Klasse in Python auch von außerhalb der Klasse frei möglich. Dadurch können Attributwerte unbemerkt manipuliert werden!

Wir zeigen das Beispiel der Klasse Lkw, die das Attribut "last" zur Speicherung der Beladung des Lkw benutzt. Die Methode "laden" verhindert unsinnige negative Lastwerte.
Der Code zur Anwendung der Lkw-Instanz, zeigt die Manipulation der Last.

```
class Lkw(Fahrzeug):
    def __init__(self, km, kennz):
        super().__init__(km, kennz)
        self.last = 0

    def laden(self, masse):
        if masse>0:
            self.last = self.last + masse
        return self.last
```

```
man = Lkw(0, 'DD L 288')
man.laden(1200)
print(man)
man.last = -999
print('Ladung:',man.last,'kg')
```

Veränderung des Attributwertes durch Direktzugriff über die Instanzvariable.

```
Fahrzeug: DD L 288 : 0 km
Ladung: -999 kg
```

Programmierdisziplin:
Für das Auslesen und Setzen von Klassenattributen sind entsprechende Methoden zu programmieren und dann auch zu nutzen. Diese Methoden stellen insbesondere beim Setzen von Attributwerten sicher, dass nur gültige Werte gespeichert werden, wie bei der Methode "laden" des Lkw.
Betrachte dazu auch die Methoden der Klasse Person.

4 Polymorphie – spezielles Verhalten

Die erneute Programmierung einer bereits in der übergeordneten Klasse vorhandenen Methode, ermöglicht die Realisierung spezialisierten Verhaltens einer Klasse. Dies wird als Polymorphie bezeichnet.

Wir zeigen das Beispiel. eines Elektroautos am Beispiel der Methode "tanken". Das Elektroauto ist von der Klasse Pkw abgeleitet. Seine Methode zum Tanken ist aber anders als die eines Pkw.

```
class Ekw(Pkw):
    def tanken(self):
        print("Ladekabel anschliessen.")
```

Kein eigener Constructor. Damit wird automatisch der übergeordnete Constructor von Pkw benutzt.

```
id3 = Ekw(0, 'DD E 99')
id3.fahren(10)
id3.tanken()
print(id3)
```

Spezifische Programmierung der Methode "tanken" in dieser Klasse.

```
Ladekabel anschliessen.
Fahrzeug: DD E 99 : 10 km
```

Weitere Ideen

- ein Van als Pkw mit standardmäßig 8 Sitzen
- ein Fahrtenbuch, bei dem die Fahrten einer Person mit diversen Autos dokumentiert werden

5 Die Person: Autobesitz, Operatorüberladung und Destruktor

In der objektorientierten Programmierung ist es möglich, Operatoren (Rechenzeichen) zu überladen. Dafür wird die Bedeutung eines Operators für die Klasse definiert.

Wir zeigen das Beispiel des kleiner-als-Operators (__lt__) zum Vergleich anhand der Anzahl besessener Autos der Person. Der << - Operator (__lshift__) wird als Verkauf eines Autos definiert. Zudem illustriert die Klasse Person den Aufruf eines Destruktors.

```
class Person():

    def __init__(self, name, adresse):
        self.name = name
        self.adresse = adresse
        self.autos = []

    def get_name(self):
        return self.name

    def get_adresse(self):
        return self.adresse

    def set_adresse(self, adresse):
        self.adresse = adresse

    def kaufe_auto(self, auto):
        self.autos.append(auto)

    def verkaufe_auto(self):
        car = self.autos.pop()
        return car

    def print_autos(self):
        print("Autos von:",self.get_name(),":")
        for i in self.autos:
            print(i)

    def __lshift__(self, other):
        auto = other.verkaufe_auto()
        self.kaufe_auto(auto)

    def __lt__(self,other):
        if len(self.autos) < len(other.autos):
            return True
        else:
            return False

    def __del__(self):
        print(self.name,": The End.")
```

Constructor: setzt Attribute, u.a. eine leere Liste [], um darin später Autos zu speichern

Speichert gekaufte Autos in der Liste

Entnimmt verkaufte Autos aus der Liste

Zeigt alle Autos der Person mittels print

Operatorüberladung << Autoverkauf: other verkauft ein Auto self kauft das Auto

Operatorüberladung < Der kleiner-als-Vergleich wird an der Anzahl Autos der Personen gemessen

Destructor: die letzte Aktion, bevor eine Instanz aus dem Speicher gelöscht wird

```
pia = Person('Pia','Bergweg 12, Bonn')
ian = Person('Ian','Markt 7, Suhl')
bus = Van(0,'BN VA 111')
id3 = Ekw(0,'BN E 7')
pia.kaufe_auto(bus)
pia.kaufe_auto(id3)
pia.print_autos()
ian.print_autos()
print("Vergleich:", pia < ian)
print("Pia verkauft ein Auto an Ian:")
ian << pia
pia.print_autos()
ian.print_autos()
del pia
del ian
```

.... wer hat mehr Autos?

.... ian bekommt ein Auto von pia

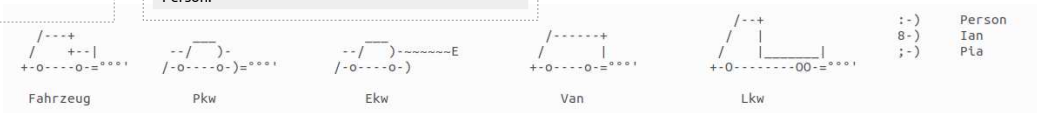
.... die Instanzen von pia und ian werden gelöscht

```
Autos von: Pia :
Fahrzeug: BN VA 111 : 0 km
Fahrzeug: BN E 7 : 0 km
Autos von: Ian :
Vergleich: False
Pia verkauft ein Auto an Ian:
Autos von: Pia :
Fahrzeug: BN VA 111 : 0 km
Autos von: Ian :
Fahrzeug: BN E 7 : 0 km
Pia : The End.
Ian : The End.
```

.... Pia hat weniger Autos als Ian? False!

....das Ergebnis des Autoverkaufs

.... del löst den Destructor aus



Informatik studieren HTW D
Hochschule für Technik und Wirtschaft Dresden
University of Applied Sciences

Fragen, Hinweise, Bestellanfragen an:
robert.ringel@htw-dresden.de

vers. 0.1