8.4.2 Statische Methoden

Statische Methoden einer Klasse können aufgerufen werden, ohne vorher eine Instanz zu erzeugen. Dazu müssen in der Klassendefinition statische Methoden durch den Aufruf der `staticmethod()`-Funktion erzeugt werden.

Diese Art von Methoden wird oft verwendet, um eine Sammlung thematisch zusammengehöriger Operationen zu programmieren (*Toolbox*).

In [1]:
```
# Beispiel einer Toolbox aus dem Bereich der Statistik
class Statistik:
    def mittelwert(s):                              #1
        if s:
            return float(sum(s)) / len(s)

    def spannweite(s):
        # größte minus kleinste Zahl der Zahlenliste s
        if s:
            return max(s) - min(s)

    def median(s):
        if s:
            s1 = sorted(s)
            if len(s) % 2 == 0:  # Länge ist gerade
                return (s1[len(s)//2 - 1] + s1[len(s)//2]) / 2.0
            else:
                return s1[(len(s)-1)//2]

    mittelwert = staticmethod(mittelwert)           #2
    spannweite = staticmethod(spannweite)
    median = staticmethod(median)


s = [1, 4, 9, 11, 5]
print(Statistik.mittelwert(s))
print(Statistik.median(s))
print(Statistik.spannweite(s))
```

```
6.0
5
10
```

Die Klassendefinition weist einige Besonderheiten auf:

- Es gibt keine `__init__()`-Methode, denn es gibt keinerlei Attribut. Die erzeugten Objekte der Toolbox-Klasse bleiben immer im gleichen Zustand
- In der Parameterliste der statischen Methoden gibt es kein Argument `self` (**#1**)
- Die durch `def` definierten Methoden werden durch den Aufruf von `staticmethod()` zu statischen Methoden der Klasse (**#2**)

8.5 Drei Grundprinzipien der OOP

Bei der Entwicklung eines objektorientierten Modells (wie z.B. der `class Geld`) wurden drei wichtige Prinzipien der objektorientierten Denkweise verwendet:

1. Abstraktion

Eine Klasse kann man als *Abstraktion* einer Menge von Objekten aus der Wirklichkeit betrachten. *Abstrahieren* heißt damit, die wesentlichen Aspekte zu ermitteln und alles Unwichtige wegzulassen.

2. Verkapselung

Zu Beginn der Veranstaltung wurde das Prinzip der *schrittweisen Verfeinerung* angewandt und für Teilaufgabe z.B. Funktionen definiert. Das Augenmerk wurde dabei allein auf Operationen gelegt. Das Besondere der OOP ist, dass Operationen und logisch zusammenhängende Attribute zu einer Einheit (Klasse) verschmolzen werden. Dieses nennt man *Verkapselung* (*encapsulation*)

3. Geheimnisprinzip

Das Geheimnisprinzip (*information hiding*) besagt, dass der Zustand eines Objektes nach außen nicht sichtbar sind. Bei konsequenter Einhaltung sind alle Attribute stark privat. Änderungen des Zustands können von außen nur über Methodenaufrufe angestoßen werden.

8.6 Vererbung

Der Begriff *Vererbung* (engl. *inheritance*) beschreibt die Beziehung zwischen einer allgemeinen Klasse (Basisklasse, Oberklasse) und einer spezialisierten Klasse (Unterklasse, Subklasse, abgeleitete Klasse). Die Unterklasse besitzt sämtliche Attribute und Methoden der Oberklasse. Man sagt, die Oberklasse *vererbt* ihre Merkmale an ihre Unterklassen.

Unterklassen besitzen darüber hinaus i.d.R. noch zusätzliche Attribute und Methoden. Diese sind damit spezieller, und daher weniger abstrakt.

8.6.1 Spezialisierungen

Der Syntax zur Implementierung einer Unterklasse lautet:

```
class Unterklasse(Oberklasse):
    neuesAttribut = Standardwert
    ...
    def neueMethode(self, ...):
        ...
```

Mit einer Unterklasse kann man vorhandene Methoden (oder Attribute) der Oberklasse verfeinern oder durch Überschreiben (*overriding*) neu definieren. Man sagt: Die Methodendefinition der Oberklasse wird überschrieben.

Anwendung und Probleme der Vererbung sollen nun an folgendem Beispiel veranschaulicht werden:

 spezialisierung

8.6.2 Spezialisierung der Klasse `Geld`

Im Folgenden soll die Unterklasse `Konto` implementiert werden:

In [2]:
```
# TODO Beispiel zur Klasse Konto
import time
from geld import Geld
class Konto(Geld):
    """ Spezialisierung der Klasse Geld zur Verwaltung eines Kontos
        Öffentliche Attribute:
           geerbt: waehrung, betrag, wechselkurs

        Öffentliche Methoden und Überladungen:
           geerbt: __add__(), __lt__(),
                   __le__(), __eq__(), getEuro()
           überschrieben: __str__()
           Erweiterungen:
             einzahlen(), auszahlen(), druckeKontoauszug()
    """
    def __init__(self, waehrung, inhaber):
        # TODO
        super().__init__(waehrung, 0)
        self.__inhaber = inhaber
        self.__kontoauszug = [str(self)]


    def einzahlen(self,waehrung, betrag):
        einzahlung = Geld(waehrung,betrag)
        # TODO betrag
        self.betrag = (self+einzahlung).betrag

        eintrag = time.asctime()+' ' +str(einzahlung) +\
                ' neuer Kontostand: '+ self.waehrung + \
        ' ' + str(round(self.betrag, 2))
        self.__kontoauszug += [eintrag]

    def auszahlen(self, waehrung, betrag):
        self.einzahlen(waehrung, -betrag)

    def druckeKontoauszug(self):
        for i in self.__kontoauszug:
            print(i)
        self.__kontoauszug = [str(self)]

    def __str__(self):
        return 'Konto von ' + self.__inhaber + \
               ':\nKontostand am ' + time.asctime() + \
               ': ' + self.waehrung + ' ' +\
                str(round(self.betrag, 2))


# Ende der Klassendefinition
```

```
# Klasse testen
konto = Konto('EUR', 'Tim Wegner')
konto.einzahlen('EUR', 1200)
time.sleep(2)
konto.auszahlen('USD', 50)
konto.einzahlen('GBP', 30.30)
print(konto)
konto.druckeKontoauszug()
```

```
Konto von Tim Wegner:
Kontostand am Wed May  8 17:57:15 2024: EUR 1199.76
Konto von Tim Wegner:
Kontostand am Wed May  8 17:57:13 2024: EUR 0.0
Wed May  8 17:57:13 2024 EUR 1200.0 neuer Kontostand: EUR 1200.0
Wed May  8 17:57:15 2024 USD -50.0 neuer Kontostand: EUR 1157.5
Wed May  8 17:57:15 2024 GBP 30.3 neuer Kontostand: EUR 1199.76
```

Anmerkung zu den Begriffen *Überladen* und *Überschreiben*:

Überladen	**Überschreiben**
Methoden und Funktionen haben **gleichen Namen**, aber **unterschiedliche Signatur**	Methoden und Funktionen haben **gleichen Namen** und **gleiche Signatur**
Beim Überladen spricht man von *compile time polymorphism*	Beim Überschreiben spricht man von *run time polymorphism*
Überladen *kann **auch*** bei Vererbung vorkommen	Überschreibung *kann **nur*** bei Vererbung vorkommen

Für Python gilt damit:

- es gibt nach dem klassischen Ansatz **kein Überladen** in Python, da *alles ein Objekt ist*. Das ist auch gar nicht notwendig, da es in Python die *dynamischen Datentypen* gibt. (In statisch getypten Sprachen wie Java und C++ wird es wiederrum benötigt, um die gleiche Funktion für verschiedene Typen zu definieren)

Nachtrag: Python wird auch als *duck typed language* bezeichnet. Zitat von James Whitcomb Riley:

> 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.

In [5]:
```
# TODO Versuch von klassischem Überladen
def f(x):
    return x + 3

f(4)

def f(x,y):
    return x + y

f(1,2)
f(4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 11
      8     return x + y
     10 f(1,2)
---> 11 f(4)

TypeError: f() missing 1 required positional argument: 'y'
```

In [9]:
```python
# TODO Veranschaulichung von dynamischer Typendeklaration
def multi(irgendwas, faktor):
    return faktor * irgendwas

print(multi(3,2))
print(multi(3.4, 2))
print(multi(".", 10))
print(multi(["spam"], 5))
#print(multi(".", ","))
```

```
6
6.8
..........
['spam', 'spam', 'spam', 'spam', 'spam']
```

In C++ wäre für dieses Beispiel folgende Implementierung notwendig:

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int multi(int irgendwas, int faktor){
    return irgendwas * faktor;
}

double multi(double irgendwas, int faktor){
    return irgendwas * faktor;
}

int main(){
    cout << multi(10,3) << endl;
    cout << multi(10.3, 4) << endl;
    return 0;
}
```

8.6.3 Klassenmethoden

Die in 8.4.2 behandelten statischen Methoden dürfen nicht mit Klassenmethoden verwechselt werden. Klassenmethoden sind zwar auch nicht an Instanzen gebunden, aber - anders als statische Methoden - an eine Klasse gebunden. Das erste Argument einer Klassenmethode ist eine Referenz auf das Klassenobjekt (gebräuchlich ist die Bezeichnung `cls`). Aufgerufen werden kann sie sowohl über den Klassennamen als auch über eine Instanz:

In [10]:
```python
# Beispiel mit statischer Methode
class Pet:
    name = "Haustiere"
```

```
    def about():
        print("In dieser Klasse geht es um", Pet.name)

    about = staticmethod(about)

class Dog(Pet):
    name = "Hunde"

class Cat(Pet):
    name = "Katzen"

p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

```
In dieser Klasse geht es um Haustiere
In dieser Klasse geht es um Haustiere
In dieser Klasse geht es um Haustiere
```

In [11]:
```
# TODO Beispiel abändern mit Klassenmethode
class Pet:
    name = "Haustiere"

    def about(cls):
        print("In dieser Klasse geht es um", cls.name)

    about = classmethod(about)

class Dog(Pet):
    name = "Hunde"

class Cat(Pet):
    name = "Katzen"

p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

```
In dieser Klasse geht es um Haustiere
In dieser Klasse geht es um Hunde
In dieser Klasse geht es um Katzen
```

8.6.4 Standardklassen als Basisklassen

Von Standardklassen wie `int, float, str, bool, dict, list` können ebenso neue Klassen abgeleitet werden. In folgendem Beispiel soll eine Liste mit einem voreingestellten Wert (Default) für nicht existierende Listenelemente definiert werden. Bei einem Zugriff auf die Liste mit einem zu großen Index soll es keinen Laufzeitfehler geben (IndexError), sondern stattdessen soll ein für diesen Fall vorgesehenes Objekt zurückgegeben werden:

In [17]:
```python
# TODO Abgeleitete list-Klasse
class DefaultList(list):
    def __init__(self, s=[], default = 0):
        super().__init__(s)
        self.default = default

    def __getitem__(self, index):
        try:
            return super().__getitem__(index)
        except IndexError:
            return self.default

# Hauptprogramm
p = ["Merkur", "Venus", "Erde"]
planeten = DefaultList(p, "unbekannter Planet")
print(planeten)
print(planeten[1])

planeten.append("Mars")
print(planeten)

mehrPlaneten = DefaultList(["Jupiter", "Saturn"],
                           "unbekannter Planet")
planeten = planeten + mehrPlaneten
print(planeten)
#planeten[10]
type(planeten)
```

```
['Merkur', 'Venus', 'Erde']
Venus
['Merkur', 'Venus', 'Erde', 'Mars']
['Merkur', 'Venus', 'Erde', 'Mars', 'Jupiter', 'Saturn']
```

Out[17]:
```
list
```

8.7 Mehrfachvererbung

Erbt eine abgeleitete Klasse direkt von mehr als einer Basisklasse, spricht man in der OOP von *Mehrfachvererbung*. Ein sequentielles, mehrstufiges Erben wird dagegen nicht als Mehrfachvererbung bezeichnet:

Syntaktisch sieht die Mehrfachvererbung in Python wie folgt aus:

```python
class UnterKlasse(Basis1, Basis2, Basis3, ...):
    pass
```

Mann muss bei der Implementierung darauf achten, dass es nicht zu Namenskollisionen kommt, z.B. kann die Basisklasse `A` und die Basisklasse `B` eine Methode `M` besitzen. Dann wird die abgleitete Klasse `class X(A, B)` allerdings nur die Methode `M` der Klasse `A` übernehmen.

Ausnahme: die Methode `__init__()` wird von allen Basisklassen übernommen, z.B. ruft `super().__init__()` die Konstruktoren *aller* Basisklassen in der Reihenfolge auf, in der sie bei der Definition genannt wurden.

8.7.1 Beispiel zur Mehrfachvererbung

In [18]:
```
# Implementierung des Tier-Beispiels
class Tier:
    def __init__(self, Tier):
        print(Tier, "ist ein Tier")

class Saeugetier(Tier):
    def __init__(self, Saeugetier):
        print(Saeugetier, "ist ein Warmblüter")
        super().__init__(Saeugetier)

class Fluegellos(Saeugetier):
    def __init__(self, Fluegellos):
        print(Fluegellos, "kann nicht fliegen")
        super().__init__(Fluegellos)

class NichtMeer(Saeugetier):
    def __init__(self, NichtMeer):
        print(NichtMeer, "kann nicht schwimmen")
        super().__init__(NichtMeer)
# Mehrfachvererbung
class Hund(Fluegellos, NichtMeer):
    def __init__(self):
        print("Hund hat 4 Beine.")
        super().__init__("Hund")

h = Hund()
print("")
fledermaus = NichtMeer("Fledermaus")
```

```
Hund hat 4 Beine.
Hund kann nicht fliegen
Hund kann nicht schwimmen
Hund ist ein Warmblüter
Hund ist ein Tier

Fledermaus kann nicht schwimmen
Fledermaus ist ein Warmblüter
Fledermaus ist ein Tier
```

8.7.2 Das Diamond-Problem

Bei dem Diamond-Problem (engl. *deadly diamond of death*) handelt es sich um ein Mehrdeutigkeitsproblem. Dieses Problem kann auftreten wenn beispielsweise eine Klasse `D` auf zwei verschiedenen Vererbungspfaden (über zwei Klassen `B` und `C`) von der gleichen Basisklasse `A` abstammt. Dabei gibt es eine Methode `M`, für die gilt:

- `M` wird in `A` definiert
- `M` wird entweder in `B` oder `C` oder in beiden überschrieben
- `M` wird **nicht** in `D` überschrieben

Frage: *Von welcher Klasse wir die Methode `M` nun vererbt?*

In [23]:
```
# Beispiel von oben
class A:
    def m(self):
        print("m of A called")

class B(A):
    pass
    #def m(self):
        #print("m of B called")

class C(A):
    #pass
    def m(self):
        print("m of C called")

# Parameterliste tauschen
class D(B, C):
    pass

x = D()
x.m()
```

```
m of C called
```

→ Die Vererbungsreihenfolge spielt eine entscheidende Rolle

→ strikte und saubere Nomenklatur einhalten, wenn möglich Diamond-Architektur vermeiden!

8.7.3 `super()` und MRO

In 8.7.2 wurde gezeigt, dass es im Falle des Diamond-Problem in Python auf die Reihenfolge, mit welcher die Basisklassen durchsucht werden, geachtet werden muss. Diese Reihenfolge wird durch die *Method Resolution Order* (kurz *MRO*) festgelegt.

Im folgenden ist das Beispiel von oben zu sehen. Hierbei sind jetzt alle Klassen um die Methode `m()` mit einer `print`-Ausgabe erweitert, um den Methodenaufruf und die MRO nachzuvollziehen.

In [24]:
```
# Beispiel zur MRO
class A:
    def m(self):
        print("m of A called")
```

```
class B(A):
    def m(self):
        print("m of B called")
        A.m(self)
class C(A):
    def m(self):
        print("m of C called")
        A.m(self)
class D(B, C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)

d = D()
d.m()
```

```
m of D called
m of B called
m of A called
m of C called
m of A called
```

Problem: Die Methode der Klasse `A` wird zweimal aufgerufen.

Lösung: Verwendung von `super()`

In [25]:
```
# Lösung des Problems
class A:
    def m(self):
        print("m of A called")
class B(A):
    def m(self):
        print("m of B called")
        super().m()
class C(A):
    def m(self):
        print("m of C called")
        super().m()
class D(B, C):
    def m(self):
        print("m of D called")
        super().m()

d = D()
d.m()
```

```
m of D called
m of B called
m of C called
m of A called
```

Der Python-Interpreter baut seine MRO nach dem *C3 superclass linearization*-Algorithmus auf. Man spricht deshalb von Linearisierung, weil aus der Baumstruktur eine lineare Reihenfolge gebildet wird. Diese geordnete Liste kann man sich mit der `mro()`-Methode anzeigen lassen

In [26]:
```
# TODO mro()-Methode
D.mro()
```

Out[26]:
```
[__main__.D, __main__.B, __main__.C, __main__.A, object]
```

Ein umfangreicheres Beispiel:

In [27]:
```
# Code des umfangreichen Beispiels
class A():
    pass
class B1(A):
    pass
class B2(A):
    pass
class B3(A):
    pass
class B4(A):
    pass
class B5(A):
    pass
class C1(B1, B2, B3):
    pass
class C2(B4, B2, B5):
    pass
class C3(B4, B1):
    pass
class D(C1, C2, C3):
    pass
d = D()
D.mro()
```

Out[27]:
```
[__main__.D,
 __main__.C1,
 __main__.C2,
 __main__.C3,
 __main__.B4,
 __main__.B1,
 __main__.B2,
 __main__.B3,
 __main__.B5,
 __main__.A,
 object]
```

8.8 Typische Fehler

8.8.1 Versehentliches Erzeugen neuer Attribute

Wie wir bereits gesehen hatten, kann es beispielsweise durch einen einfachen Schreibfehler dazuführen, dass zu einer bestehenden Klasse dynamisch ein Attribut hinzugefügt wird, ohne dass es zu einer Fehlermeldung führt:

In [43]:
```
# einfaches Beispiel
class Behaelter:

    # TODO
    __slots__ = ["volumen"]

    def __init__(self, volumen):
```

```
        self.volumen = volumen

# Hauptprogramm
tasse = Behaelter(250)
glas = Behaelter(500)
print("In der Tasse sind", tasse.volumen, "ml. ")
ex = input("Wie viel wollen Sie ausschütten? ")
Behaelter.volumen = tasse.volumen - float(ex)
print("Neuer Inhalt:", tasse.volumen, "ml. ")
print(glas.volumen)
glas.volumen = 1234
print(tasse.volumen, glas.volumen)
```

```
In der Tasse sind 250 ml.
Wie viel wollen Sie ausschütten? 100
Neuer Inhalt: 150.0 ml.
150.0
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[43], line 18
     16 print("Neuer Inhalt:", tasse.volumen, "ml. ")
     17 print(glas.volumen)
---> 18 glas.volumen = 1234
     19 print(tasse.volumen, glas.volumen)

AttributeError: 'Behaelter' object attribute 'volumen' is read-only
```

Um das Problem zu verhindern, kommen die `__slots__` ins Spiel. Anstatt eines dynamischen Dictionarys stellen Slots eine statische Struktur zur Verfügung, die ein weiteres Hinzufügen von Attributen verhindert, sobald eine Instanz erstellt worden ist.

Slots werden genutzt, indem man in der Klassendefinition eine Liste mit allen Attributen definiert. Diese Liste hat den Namen `__slots__`.

8.8.2 Verwechseln von Methoden und Attributen

Wenn man beim Aufruf einer Methode die Klammern weglässt, kommt es nicht zwangsläufig zu einem Laufzeitfehler. Das System liefert in diesem Fall einen String, der das Methoden-Objekt beschreibt. So kann es zu nur schwer auffindbaren logischen Fehlern kommen.

In [47]:
```
# Beispiel
class Rechteck:
    def __init__(self, laenge, breite):
        self.laenge = laenge
        self.breite = breite
    # folgende Methode entgegen der Konventionen definiert
    def flaeche(self):
        return self.laenge*self.breite

a = Rechteck(2,1)
b = Rechteck(1,2)
# was liefert folgender Ausdruck?
a.flaeche == b.flaeche
#richtig:
a.flaeche() == b.flaeche()
```

```
print(a.flaeche)
print(b.flaeche)
```

```
<bound method Rechteck.flaeche of <__main__.Rechteck object at 0x00000156E8C52ED0
>>
<bound method Rechteck.flaeche of <__main__.Rechteck object at 0x00000156E7813990
>>
```

8.9 Hinweise zum Programmierstil

8.9.1 Bezeichner

- Als Namen für *Klassen* verwendet man Substantive im Singular mit großem Anfangsbuchstabe, z.B. `File, Exception, Pickler, Geld`
- Namen für *Attribute* sind Substantive, die (evtl nach einem oder zwei Unterstrichen) mit einem kleinen Buchstaben beginnen, z.B. `__name__, betrag`
- *Methoden* werden - wie Funktionen - meist durch ein Verb benannt, dass mit einem kleinen Buchstaben beginnt, z.B. `load(9, __add__(), berechneSumme()`
- Methoden zum Lesen und Schreiben von Attributwerten benennt man nach dem Muster `getAttribut` und `setAttribut`
- *Modulnamen* bestehen aus Kleinbuchstaben. Der Dateiname für ein Modul setzt sich aus dem Modulnamen und der Extension `.py` zusammen, z.B. `geld.py, time.py, fibonacci.py`

8.9.2 Sichtbarkeit

Generell gilt: eine Klasse soll so wenig wie möglich über ihren internen Aufbau verraten (→ Geheimnisprinzip). Sofern ein direkter Zugriff von außen nicht notwendig ist, sollten Attribute privat sein. Bei öffentlichen Attributen besteht die Gefahr, dass das Objekt in einen inkonsistenten Zustand gerät. Klassen mit öffentlichen Attributen sind daher anfälliger für Fehler.

Guter Programmierstil wäre daher eine ausschließliche Verwendung von privaten Attributen und den bereits genannten *Setter*- und *Getter*-Methoden.

Eine moderne Alternative sind die *Properties*, die bereits in 8.3.4 behandelt wurden.

8.9.3 Dokumentation von Klassen

Klassen sollten mit Dokumentationsstring versehen werden. Dieser besteht aus einer langen Zeichenkette zwischen dreifachen Anführungszeichen oder Hochkommata) und ist nach folgendem Muster aufgebaut:

- die erste Zeile enthält eine Kurzbeschreibung der Aufgabe der Klasse
- es folgt eine Leerzeile
- die öffentlichen Methoden und Attribute werden aufgelistet und kurz beschrieben

Wenn die Klasse von einer Basisklasse abgeleitet ist, sollte dies erwähnt werden. Stellen Sie die Unterschiede zur Basisklasse kurz da und verweisen Sie darauf hin, welche Methoden überschrieben worden sind. Beispiel in 8.6.2

8.9.4 (Un-)Pythonic?

Wie man bereits gesehen hat, gibt es für gleiche Aufgaben verschiedene, aber auch syntaktisch verschiedene Lösungen. Die einen Lösungen sind "schöner" als die anderen, aber was entspricht dem Programmierstil standardmäßig am ehesten?

Dafür gibt es abschließend noch einen Verweis auf die drei bekanntesten *Python Style Guides*:

1. PEP 8 als generelle Richtlinie der offiziellen Python Dokumentation
2. Hitchhiker's Guide als Community basierte Richtlinie/Anleitung
3. Google Python Style Guide offizielle, von Google veröffentlichte Gestaltungsrichtlinien für Python-Code

Ein *pythonischer* Programmierstil zeichnet sich beispielsweise durch die Verwendung von Dekorateuren aus (siehe dazu auch im Hitchhiker's Guide)

In [48]:
```
# TODO pythonisch machen: Beispiel zu statischen Methoden
class Statistik:
    @staticmethod
    def mittelwert(s):
        if s:
            return float(sum(s)) / len(s)

    @staticmethod
    def spannweite(s):
        # größte minus kleinste Zahl der Zahlenliste s
        if s:
            return max(s) - min(s)

    @staticmethod
    def median(s):
        if s:
            s1 = sorted(s)
            if len(s) % 2 == 0:  # Länge ist gerade
                return (s1[len(s)//2 - 1] + s1[len(s)//2]) / 2.0
            else:
                return s1[(len(s)-1)//2]

    #mittelwert = staticmethod(mittelwert)
    #spannweite = staticmethod(spannweite)
    #median = staticmethod(median)


s = [1, 4, 9, 11, 5]
print(Statistik.mittelwert(s))
print(Statistik.median(s))
print(Statistik.spannweite(s))
```

```
6.0
5
10
```

In [49]:
```python
# TODO pythonisch machen: Beispiel Klassenmethoden
class Pet:
    name = "Haustiere"

    @classmethod
    def about(cls):
        print("In dieser Klasse geht es um", cls.name)

    # about = classmethod(about)

class Dog(Pet):
    name = "Hunde"

class Cat(Pet):
    name = "Katzen"

p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

```
In dieser Klasse geht es um Haustiere
In dieser Klasse geht es um Hunde
In dieser Klasse geht es um Katzen
```