Objektorientierte Programmierung 4

OOP

Mehrfachvererbung

Ableitung von mehr als einer Klasse

```
1 class A:
2     pass
3
4 class B:
5     pass
6
7 class C (A, B):
8     pass
```

OOP

Was wird vererbt?

- alles,
- außer bei Problemen

```
1 class A:
2     pass
3
4 class B:
5     pass
6
7 class C (A, B):
8     pass
```

OOP

Mehrfachvererbung -- Probleme

- welcher Konstruktor?
- welche Attribute / Methoden bei Namensgleichheit?

OOP

Mehrfachvererbung -- Probleme

- welcher Konstruktor?
- welche Attribute / Methoden bei Namensgleichheit?

Lösung: von links nach rechts, unten nach oben

OOP

Mehrfachvererbung

- komplexes Thema
- Lösungen teilweise sehr ad hoc
- Empfehlung: Finger weg!

OOP

Mehrfachvererbung

- Beispiel super()
- wandert auch horizontal

OOP

Mehrfachvererbung

Aufruf von methode() in übergeordneter Klasse:

- super().methode()
- klassenname.methode(**self**)

OOP

Aufgabe

- Implementieren Sie eine Klasse Tier.
- Leiten Sie von Tier die zwei Klassen Pflanzenfresser und Fleischfresser ab.
- Implementieren Sie eine Klasse Allesfresser so, dass sie sowohl von Pflanzenfresser als auch von Fleischfresser erbt und beide Konstruktoren ausführt.

OOP

Polymorphie:

ermöglicht, dass ein Bezeichner abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps annimmt. (wikipedia)

OOP

Polymorphie anders als in anderen Sprachen

```
class A:
    def __init__(self):
        print("A konstruiert.")

    def schreib(self, eins ):
        print("eins")

    def schreib(self, eins , zwei ):
        print("zwei")
```

OOP

Polymorphie anders als in anderen Sprachen

```
class A:
    def __init__(self):
        print("A konstruiert.")

    def schreib(self, eins ):
        print("eins")

    def schreib(self, eins , zwei  ):
        print("zwei")
```

verschiedene Profile in derselben Funktionsdefinition realisiert

OOP

Polymorphie anders als in anderen Sprachen

```
1  class A:
2      def __init__(self):
3          print("A konstruiert.")
4
5      def schreib(self, eins , zwei = None):
6          if ( zwei == None ):
7              print("eins")
8          else:
9              print("zwei")
```

Regeln für optionale Parameter etc. wie bei Funktionen

OOP

Klassenvariablen

```
1  class A:
2      klassenvariable = 0
3
4      def __init__(self):
5          A.klassenvariable  += 1
6
7      def kv_aendern(self, k):
8          A.klassenvariable = k
```

- ein einziger Wert
- gleich in allen Instanzen
- definiert außerhalb der Methoden
- Zugriff über Klasse und Punkt

OOP

Klassenvariablen

- zugänglich auch über erbende Klassen

```
1  class Oben:
2      klassvar = 0
3
4  class Mitte(Oben):
5      pass
6
7  class Unten(Mitte):
8      pass
9
10 print( Oben.klassvar, Mitte.klassvar,
Unten.klassvar)
11
12 Oben.klassvar +=4
13 print( Oben.klassvar, Mitte.klassvar,
Unten.klassvar)

0 0 0
4 4 4
```

OOP

Klassenvariablen

- bei Schreibzugriff über erbende Klasse wird neue Klassenvariable angelegt

```
1  class Oben:
2      klassvar = 0
3
4  class Mitte(Oben):
5      pass
6
7  class Unten(Mitte):
8      pass
9
10 print( Oben.klassvar, Mitte.klassvar,
Unten.klassvar)
11
12 Mitte.klassvar +=4
13 print( Oben.klassvar, Mitte.klassvar,
Unten.klassvar)

0 0 0
0 4 4
```

OOP

Klassenvariablen

- Lesezugriff auch über Instanzen

```
1  class Klasse:
2      klassvar = 77
3
4  k = Klasse()
5
6  print( k.klassvar, Klasse.klassvar)

77 77
```

OOP

Klassenvariablen

- bei Schreibzugriff über Instanzen wird stets Instanzattribut angelegt (ähnlich lokalen Variablen)

```
1  class Klasse:
2      klassvar = 77
3
4  k = Klasse()
5  k.klassvar = 88
6
7  print( k.klassvar, Klasse.klassvar)

88, 77
```

OOP

Erweitern Sie die Klassen in

Tiere.py

um zwei Klassenvariablen, die

- die Anzahl aller instanziierten Tiere sowie
- die Anzahl aller instanziierten Hunde zählen.

Modifizieren sie die Konstruktoren, so dass sie die Werte aktualisieren.

OOP

Zugriff auf Attribute

- direkt möglich
- kann Wert, Typ ändern

```
1  class A:
2      def __init__(self):
3          self.x = 0
4
5  a = A()
6  print(type(a.x))
7  a.x = "Null"
8  print(type(a.x))

<class 'int'>
<class 'str'>
```

- über Methoden besser zu regeln

OOP

Zugriff nur über Methoden kann nicht erzwungen werden

- können besondere Attribute schaffen
- property([fget, fset, fdel, doc])

OOP

property ist sog. decorator

- bestimmt Methoden, ruft diese automatisch auf
- keine Funktion sondern Klasse

OOP

mit property Zugriff automatisch über Methoden

- überall in Definition

  mit "_" vor Bezeichner
- außer bei x = property

```
1  class B:
2      def __init__(self):
3          self._x = 0
4
5      def setze_x(self, k):
6          print("setze_x")
7          if isinstance(k, int):
8              self._x = k
9
10     def lese_x(self):
11         print("lese_x")
12         return self._x
13
14     x = property(lese_x, setze_x)
```

OOP

property -- alternative Syntax

- als Decorator
- mit @, vor getter
- alle Methoden haben Namen des Attributs
- werden nicht überschrieben

```
class B:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        print("lese_y")
        return self._x

    @x.setter
    def x(self, k):
        print("setze_x")
        if isinstance(k,int):
            self._x = k
        else:
            raise TypeError("Kein Int")
```

OOP

Aufgabe

Implementieren Sie eine Klasse *einwert*, die

- ein Feld *wert* enthält, dessen Zugriff über property() geregelt wird.
- Die setter-Methode soll dafür sorgen, dass ausschließlich Listen an wert zugewiesen werden können und sonst ein TypeError ausgelöst wird.

OOP

hatten Klassenvariablen / statische Variablen

jetzt: **statische Methoden**

- ohne self
- nicht an Instanz gebunden
- Aufruf über Klasse

```
1  class A:
2      def m():
3          print("Hallo!")
4
5      m = staticmethod(m)
6
7  print( A.m() )

Hallo!
```

OOP

statische Methoden

- ebenfalls in Decorator-Syntax möglich

```
1 class A:
2
3     @staticmethod
4     def m():
5         print("Hallo!")
6
```

OOP

statische Methoden

- gehören inhaltlich zur Klasse
- greifen weder auf Klassenvariablen noch Instanzfelder zu
- nur im Klassen- / Instanznamensraum nötig
- könnten auch in übergeordnetem Namensraum stehen

OOP

hatten Klassenvariablen / statische Variablen

jetzt: **Klassenmethoden**

- Klasse als Parameter
- Aufruf über Instanz oder Klasse

```
1  class A:
2   def m(cls):
3       print("Ich bin", cls)
4   m = classmethod(m)
5
6  class B(A):
7   pass
8  A.m()
9  a = A()
10 b = B()
11 a.m()
12 b.m()

Ich bin <class '__main__.A'>
Ich bin <class '__main__.A'>
Ich bin <class '__main__.B'>
```

OOP

Klassenmethoden

- ebenfalls als Decorator

```
1
2  class A:
3
4      @classmethod
5      def m(cls):
6          print("Ich bin", cls)
```

OOP

Klassenmethoden

- über cls Zugang zu Klassenvariablen, anderen Klassenmethoden
- könnten nicht in übergeordnetem Namensraum stehen

OOP

Aufgabe

Fügen Sie der Klasse Hund eine Klassenmethode hinzu, die die Anzahl der erzeugten Tierinstanzen und die Anzahl der erzeugten Hundeinstanzen in einem Tupel zurückgibt.

OOP

Magic Methods und Magic Attributes

- beginnen und enden mit doppeltem __
- "magisch" weil meist benutzt ohne expliziten Aufruf
- Beispiel __init__

OOP

Magic Methods und Magic Attributes

__str__

- bestimmt was

  Konvertierung in String

- via str(obj) liefert

```
1  class A:
2   def m(cls):
3       print("Ich bin", cls)
4   m = classmethod(m)
5
6  class B(A):
7   pass
8  A.m()
9  a = A()
10 b = B()
11 a.m()
12 b.m()

Ich bin <class '__main__.A'>
Ich bin <class '__main__.A'>
Ich bin <class '__main__.B'>
```

OOP

Magic Methods und Magic Attributes

- probieren Sie aus, was str( ) für ein Hund-Objekt liefert
- implementieren sie __str__-Methoden für Hund und Katze mit geeigneter Ausgabe
- probieren Sie aus, was str( ) danach für ein Rassehund-Objekt liefert

OOP

Magic Methods und Magic Attributes

__dict__ konvertiert zu

Dictionary wie folgt

```
print(Hund.__dict__)

{'__module__': '__main__',

'__init__': <function
Hund.__init__ at 0x7bf1a4184fe0>,

'spricht': <function Hund.spricht
at 0x7bf1a4185080>,

'frisst': <function Hund.frisst at
0x7bf1a4185120>,

'__doc__': None}
```

OOP

Magic Methods und Magic Attributes

__name__ liefert den Namen der Klasse

```
print(Hund.__name__)

Hund
```

OOP

Magic Methods und Magic Attributes

__module__ gibt an, in welchem Modul definiert

```
print(Hund.__module__)

__main__
```

OOP

Magic Methods und Magic Attributes

__bases__ gibt an, von welchen Klassen abgeleitet

```
print(Rassehund.__bases__)

(<class '__main__.Hund'>,)
```

OOP

Magic Methods und Magic Attributes

__module__ auch für Instanz

__bases__ und __name__

nur für Klasse

```
print(hu.__module__)

__main__
```

OOP

Magic Methods und Magic Attributes

__class__ liefert Klasse einer Instanz

Darüber auch Aufruf für __name__

```
print(hu.__class__)

<class '__main__.Hund'>


print(hu.__class__.__name__)

Hund
```

OOP

Magic Methods und Magic Attributes

viele weitere im Buch

OOP

Name Mangling

- Bezeichner beginnend mit _: ***bitte*** nicht ändern
- mit zwei __ kann bestimmtes Verhalten erzwungen werden

OOP

Name Mangling

Vermengen von Namen

- __ erlaubt am Beginn eines Bezeichners
- in Objekt nicht?

```
1  class MyClass:
2
3      def __init__(self):
4          self.__private_var = "I am private"
5
6      def get_private_var(self):
7          return self.__private_var
8
9  my_object = MyClass()
10 print(my_object.get_private_var())
11 print(my_object.__private_var)

I am private
AttributeError: 'MyClass' object has no
attribute '__private_var'
```

OOP

Name Mangling

Vermengen von Namen

- __ ist erlaubt
- aber von außen nicht zugreifbar

```
1  class MyClass:
2
3      def __init__(self):
4          self.__private_var = "I am private"
5
6      def get_private_var(self):
7          return self.__private_var
8
9  my_object = MyClass()
10 print(my_object.get_private_var())
11 print(my_object.__private_var)

I am private
AttributeError: 'MyClass' object has no
attribute '__private_var'
```

OOP

Name Mangling

Vermengen von Namen

- Name geändert zu _MyClass__private_var

```
class MyClass:

    def __init__(self):
        self.__private_var = "I am private"

    def get_private_var(self):
        return self.__private_var

my_object = MyClass()
print(my_object.get_private_var())
print(my_object._MyClass__private_var)
```

```
I am private
AttributeError: 'MyClass' object has no
attribute '__private_var'
```

OOP

Name Mangling

Vermengen von Namen

- bei __ am Beginn des Bezeichners: mangling
- wenn am Ende nochmal __: kein mangling (magic methods)

OOP

Name Mangling

Vermengen von Namen

- fast *private*
- aber mit Hintertür

OOP

Aufgabe

Implementieren Sie eine Klasse *vermenge*, die ein Feld __*mengwert* und eine Methode zum Auslesen desselben enthält.

Erzeugen Sie eine Instanz, ändern Sie von außen __mengwert, und lesen Sie dann dessen Wert über die Methode aus.

What is the expected output?

```
1 class A:
2     def __str__(self): return "A"
3
4 class B(A): pass
5
6 print(B())
```

a) keine Ausgabe

b) <__main__.B object at 0x03100FD0>

c) A

d) TypeError: __str__() missing 1 required positional argument

What is the expected output?

```python
class A:
    def __init__(self, v):
        self.x = v + 1

a = A(1)
print(a.x)
```

a) AttributeError: 'A' object has no attribute 'x'

b) TypeError: __init__() takes 2 positional arguments but 1 were given

c) 1

d) 2

Select the line number(s) from the options which will print Spam

```
1 class Spam:
2     def v0(self):
3         print(__name__)
4 print(__name__)
5 s = Spam()
6 s.v0()
7 print(s.__class__.__name__)
8 print(Spam.__name__)
9 print(s.__name__)
```

a) Line 3

b) Line 4

c) Line 7

d) Line 8

e) Line 9

Which of the options below are valid given the following code?

```
class A(object): pass
class B(object): pass
class C(object): pass
class D(object): pass
class E(object): pass
class K1(A,B,C): pass
class K2(D,B,E): pass
class K3(D,A): pass
<<< INSERT CODE HERE >>>
```

```
a) class Foo(K1,K2,K3): pass

b) class Foo(K1,K3,K2): pass

c) class Foo(K2,K1,K3): pass

d) class Foo(K2,K3,K1): pass

e) class Foo(K3,K1,K2): pass
```

Select the choices which will return TRUE

```
class X: pass

class Y: pass

class Z(X, Y): pass

x, y, z = X(), Y(), Z()
```

a) `isinstance(X, z) and isinstance(Y, z)`

b) `isinstance(z, X) and isinstance(z, Y)`

c) `isinstance(z, (list, X, Y))`

d) `isinstance((list, X, Y), z)`

e) `isinstance(z, X, Y)`

What is the expected output?

```
class Ham:

    def __init__(self):
        print(type(self).__name__ + '.__init__()', end=' ')
        self.__update()

    def update(self):
        print(type(self).__name__ + '.update()')
    __update = update

Ham()
```

a) The script will run but will not output anything

b) Ham.__init__()

c) Ham.__init__() Ham.update()

d) AttributeError: 'Ham' object has no attribute '_Ham__update'