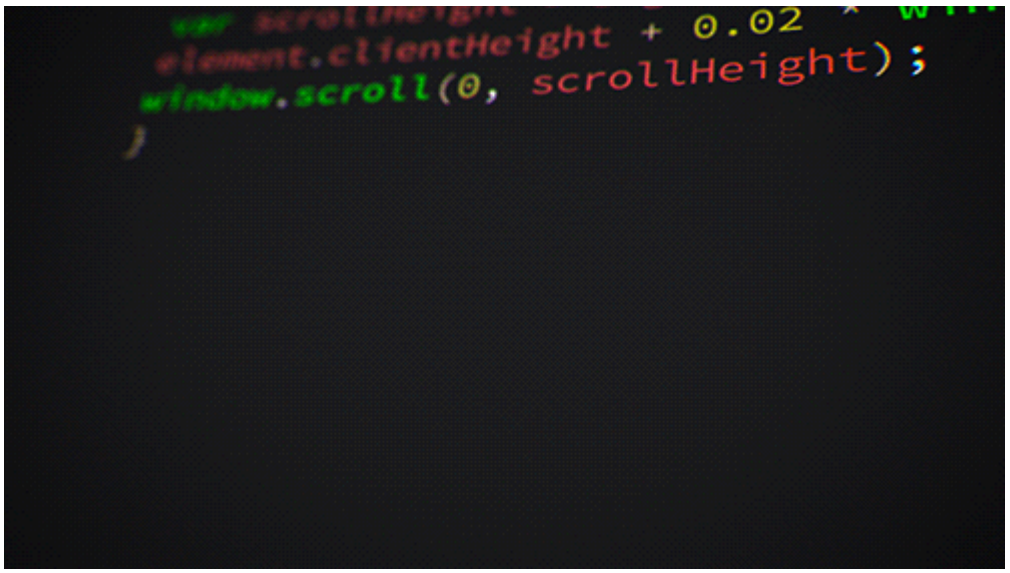
Objektorientierung in Python

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Objektorientierung in Python`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/09_PythonOOP.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Bernhard Jung



Fragen an die heutige Veranstaltung ...

- Wie lassen sich die Konzepte der OOP in Python ausdrücken?
- Welche spezifischen Einschränkungen gibt es dabei?

Objektorientierung in Python

Klassen werden verwendet, um benutzerdefinierte Datenstrukturen zu erstellen und definieren Funktionen, sogenannte Methoden, die das Verhalten und die Aktionen identifizieren, die ein aus der Klasse erstelltes Objekt mit seinen Daten ausführen kann.

Eine kurze Auffrischung Ihrer Erinnerungen zur objektorientierter Programmierung in C++ ...

Comparison.cpp

```cpp
#include <iostream>
#include <cmath>

class Rectangle {
  private:
    float width, height;
  public:
    Rectangle(float w, float h){
        this->width = abs(w);  // ensure non-negative width
        this->height = abs(h);
    }
    float area() {return width*height;}
    Rectangle operator+=(Rectangle offset) {
        float ratio = (offset.area() + this->area()) / this->area();
        this->width = ratio *  this->width;
        return *this;
    }
};

int main () {
  Rectangle rect_a(3,4);
  Rectangle rect_b(1,3);
  std::cout << "Fläche a : " << rect_a.area() << "\n";
  std::cout << "Fläche b : " << rect_b.area() << "\n";
  rect_a += rect_b;
  std::cout << "Summe    : " << rect_a.area();

  return 0;
}
```

```
Fläche a : 12
Fläche b : 3
Summe    : 15Fläche a : 12
Fläche b : 3
Summe    : 15
```

Zeile	Bedeutung
4-18	Definition der Klasse `Rectangle` (Schablone für Daten, Methoden, Operatoren)
6	Gekapselte Daten der Klasse, diese sind "von Außen" nicht sichtbar
8	Konstruktor mit Evaluation der übergebenen Parameter
12	Methode über den Daten der Klasse
13	Individueller Operator `+` mit einer spezifischen Bedeutung
21-26	Generierung von Objekten mittels Konstruktoraufruf und Parameterübergabe, Methoden- und Operatoraufrufe

Objektorientierte Programmierung (OOP) ist ein Paradigma, das über die Ideen der Prozeduralen Programmierung hinaus geht. Es definiert Objekte und deren Verhalten. Dabei baut es auf 3 zentralen Grundprinzipien auf:

1. **Kapselung** Objekte kapseln ihre Daten, Operatoren, Methoden usw. sofern diese nicht als "öffentlich" deklariert sind.

> *Was intern passiert bleibt intern!*

2. **Vererbung** Objekte können "Fähigkeiten" an andere, speziellere Objekte weitergeben.

> *Von wem hat er das denn wohl?*

3. **Polymorphismus** Objekte werden durch Kapselung und Vererbung austauschbar!

> *Was bist denn Du für einer?*

Vorteile der objektorientierten Programmierung

- höhere Wartbarkeit durch Abstraktion
- Wiederverwendbarkeit von Code (bessere Wiederverwendbarkeit je kleiner und allgemeiner die Objekte gehalten sind)
- schlanker und übersichtlicher Code durch Vererbung

Warum also nicht immer objektorientiert entwickeln?

OOP verführt ggf. dazu, das eigentliche Problem durch einen aufwändigen Entwurf unnötig zu verkomplizieren. Dabei ist die Entwicklung der Gesamtstruktur eines komplexen Softwareprojektes aus n Objekten eine Kunst und braucht viel Übung! Erst, wenn man entsprechende Regeln kennt und sinnvoll anwendet, zeigen sich die Vorteile des Paradigmas.

... und in Python?

> **In Python ist alles ein Objekt!**

... z.B. auch Integer und Floats, die in C++ und vielen anderen Programmiersprachen keine Objekte sind

```
1  import inspect
2
3  i=5
4
5  for name, data in inspect.getmembers(i):
6      if name == '__builtins__':
7          continue
8      print(f'{name} - {repr(data)}')
```

```
__abs__ - <method-wrapper '__abs__' of int object at 0x78d4c92c4170>
__add__ - <method-wrapper '__add__' of int object at 0x78d4c92c4170>
__and__ - <method-wrapper '__and__' of int object at 0x78d4c92c4170>
__bool__ - <method-wrapper '__bool__' of int object at 0x78d4c92c4170>
__ceil__ - <built-in method __ceil__ of int object at 0x78d4c92c4170>
__class__ - <class 'int'>
__delattr__ - <method-wrapper '__delattr__' of int object at
0x78d4c92c4170>
__dir__ - <built-in method __dir__ of int object at 0x78d4c92c4170>
__divmod__ - <method-wrapper '__divmod__' of int object at
0x78d4c92c4170>
__doc__ - "int([x]) -> integer\nint(x, base=10) -> integer\n\nConvert a
number or string to an integer, or return 0 if no arguments\nare given.
If x is a number, return x.__int__().  For floating point\nnumbers,
this truncates towards zero.\n\nIf x is not a number or if base is
given, then x must be a string,\nbytes, or bytearray instance
representing an integer literal in the\ngiven base.  The literal can be
preceded by '+' or '-' and be surrounded\nby whitespace.  The base
defaults to 10.  Valid bases are 0 and 2-36.\nBase 0 means to interpret
the base from the string as an integer literal.\n>>> int('0b100',
base=0)\n4"
__eq__ - <method-wrapper '__eq__' of int object at 0x78d4c92c4170>
__float__ - <method-wrapper '__float__' of int object at
0x78d4c92c4170>
__floor__ - <built-in method __floor__ of int object at 0x78d4c92c4170>
__floordiv__ - <method-wrapper '__floordiv__' of int object at
0x78d4c92c4170>
__format__ - <built-in method __format__ of int object at
0x78d4c92c4170>
__ge__ - <method-wrapper '__ge__' of int object at 0x78d4c92c4170>
__getattribute__ - <method-wrapper '__getattribute__' of int object at
0x78d4c92c4170>
__getnewargs__ - <built-in method __getnewargs__ of int object at
0x78d4c92c4170>
__gt__ - <method-wrapper '__gt__' of int object at 0x78d4c92c4170>
__hash__ - <method-wrapper '__hash__' of int object at 0x78d4c92c4170>
__index__ - <method-wrapper '__index__' of int object at
0x78d4c92c4170>
__init__ - <method-wrapper '__init__' of int object at 0x78d4c92c4170>
__init_subclass__ - <built-in method __init_subclass__ of type object
at 0x6169ce4d6320>
__int__ - <method-wrapper '__int__' of int object at 0x78d4c92c4170>
__invert__ - <method-wrapper '__invert__' of int object at
```

```
0x78d4c92c4170>
__le__ - <method-wrapper '__le__' of int object at 0x78d4c92c4170>
__lshift__ - <method-wrapper '__lshift__' of int object at
0x78d4c92c4170>
__lt__ - <method-wrapper '__lt__' of int object at 0x78d4c92c4170>
__mod__ - <method-wrapper '__mod__' of int object at 0x78d4c92c4170>
__mul__ - <method-wrapper '__mul__' of int object at 0x78d4c92c4170>
__ne__ - <method-wrapper '__ne__' of int object at 0x78d4c92c4170>
__neg__ - <method-wrapper '__neg__' of int object at 0x78d4c92c4170>
__new__ - <built-in method __new__ of type object at 0x6169ce4d6320>
__or__ - <method-wrapper '__or__' of int object at 0x78d4c92c4170>
__pos__ - <method-wrapper '__pos__' of int object at 0x78d4c92c4170>
__pow__ - <method-wrapper '__pow__' of int object at 0x78d4c92c4170>
__radd__ - <method-wrapper '__radd__' of int object at 0x78d4c92c4170>
__rand__ - <method-wrapper '__rand__' of int object at 0x78d4c92c4170>
__rdivmod__ - <method-wrapper '__rdivmod__' of int object at
0x78d4c92c4170>
__reduce__ - <built-in method __reduce__ of int object at
0x78d4c92c4170>
__reduce_ex__ - <built-in method __reduce_ex__ of int object at
0x78d4c92c4170>
__repr__ - <method-wrapper '__repr__' of int object at 0x78d4c92c4170>
__rfloordiv__ - <method-wrapper '__rfloordiv__' of int object at
0x78d4c92c4170>
__rlshift__ - <method-wrapper '__rlshift__' of int object at
0x78d4c92c4170>
__rmod__ - <method-wrapper '__rmod__' of int object at 0x78d4c92c4170>
__rmul__ - <method-wrapper '__rmul__' of int object at 0x78d4c92c4170>
__ror__ - <method-wrapper '__ror__' of int object at 0x78d4c92c4170>
__round__ - <built-in method __round__ of int object at 0x78d4c92c4170>
__rpow__ - <method-wrapper '__rpow__' of int object at 0x78d4c92c4170>
__rrshift__ - <method-wrapper '__rrshift__' of int object at
0x78d4c92c4170>
__rshift__ - <method-wrapper '__rshift__' of int object at
0x78d4c92c4170>
__rsub__ - <method-wrapper '__rsub__' of int object at 0x78d4c92c4170>
__rtruediv__ - <method-wrapper '__rtruediv__' of int object at
0x78d4c92c4170>
__rxor__ - <method-wrapper '__rxor__' of int object at 0x78d4c92c4170>
__setattr__ - <method-wrapper '__setattr__' of int object at
0x78d4c92c4170>
__sizeof__ - <built-in method __sizeof__ of int object at
0x78d4c92c4170>
```

```
__str__ - <method-wrapper '__str__' of int object at 0x78d4c92c4170>
__sub__ - <method-wrapper '__sub__' of int object at 0x78d4c92c4170>
__subclasshook__ - <built-in method __subclasshook__ of type object at
0x6169ce4d6320>
__truediv__ - <method-wrapper '__truediv__' of int object at
0x78d4c92c4170>
__trunc__ - <built-in method __trunc__ of int object at 0x78d4c92c4170>
__xor__ - <method-wrapper '__xor__' of int object at 0x78d4c92c4170>
as_integer_ratio - <built-in method as_integer_ratio of int object at
0x78d4c92c4170>
bit_count - <built-in method bit_count of int object at 0x78d4c92c4170>
bit_length - <built-in method bit_length of int object at
0x78d4c92c4170>
conjugate - <built-in method conjugate of int object at 0x78d4c92c4170>
denominator - 1
from_bytes - <built-in method from_bytes of type object at
0x6169ce4d6320>
imag - 0
numerator - 5
real - 5
to_bytes - <built-in method to_bytes of int object at 0x78d4c92c4170>
__abs__ - <method-wrapper '__abs__' of int object at 0x7af3da194170>
__add__ - <method-wrapper '__add__' of int object at 0x7af3da194170>
__and__ - <method-wrapper '__and__' of int object at 0x7af3da194170>
__bool__ - <method-wrapper '__bool__' of int object at 0x7af3da194170>
__ceil__ - <built-in method __ceil__ of int object at 0x7af3da194170>
__class__ - <class 'int'>
__delattr__ - <method-wrapper '__delattr__' of int object at
0x7af3da194170>
__dir__ - <built-in method __dir__ of int object at 0x7af3da194170>
__divmod__ - <method-wrapper '__divmod__' of int object at
0x7af3da194170>
__doc__ - "int([x]) -> integer\nint(x, base=10) -> integer\n\nConvert a
number or string to an integer, or return 0 if no arguments\nare given.
If x is a number, return x.__int__().  For floating point\nnumbers,
this truncates towards zero.\n\nIf x is not a number or if base is
given, then x must be a string,\nbytes, or bytearray instance
representing an integer literal in the\ngiven base.  The literal can be
preceded by '+' or '-' and be surrounded\nby whitespace.  The base
defaults to 10.  Valid bases are 0 and 2-36.\nBase 0 means to interpret
the base from the string as an integer literal.\n>>> int('0b100',
base=0)\n4"
__eq__ - <method-wrapper '__eq__' of int object at 0x7af3da194170>
```

```
__float__ - <method-wrapper '__float__' of int object at
0x7af3da194170>
__floor__ - <built-in method __floor__ of int object at 0x7af3da194170>
__floordiv__ - <method-wrapper '__floordiv__' of int object at
0x7af3da194170>
__format__ - <built-in method __format__ of int object at
0x7af3da194170>
__ge__ - <method-wrapper '__ge__' of int object at 0x7af3da194170>
__getattribute__ - <method-wrapper '__getattribute__' of int object at
0x7af3da194170>
__getnewargs__ - <built-in method __getnewargs__ of int object at
0x7af3da194170>
__gt__ - <method-wrapper '__gt__' of int object at 0x7af3da194170>
__hash__ - <method-wrapper '__hash__' of int object at 0x7af3da194170>
__index__ - <method-wrapper '__index__' of int object at
0x7af3da194170>
__init__ - <method-wrapper '__init__' of int object at 0x7af3da194170>
__init_subclass__ - <built-in method __init_subclass__ of type object
at 0x59f8d3538320>
__int__ - <method-wrapper '__int__' of int object at 0x7af3da194170>
__invert__ - <method-wrapper '__invert__' of int object at
0x7af3da194170>
__le__ - <method-wrapper '__le__' of int object at 0x7af3da194170>
__lshift__ - <method-wrapper '__lshift__' of int object at
0x7af3da194170>
__lt__ - <method-wrapper '__lt__' of int object at 0x7af3da194170>
__mod__ - <method-wrapper '__mod__' of int object at 0x7af3da194170>
__mul__ - <method-wrapper '__mul__' of int object at 0x7af3da194170>
__ne__ - <method-wrapper '__ne__' of int object at 0x7af3da194170>
__neg__ - <method-wrapper '__neg__' of int object at 0x7af3da194170>
__new__ - <built-in method __new__ of type object at 0x59f8d3538320>
__or__ - <method-wrapper '__or__' of int object at 0x7af3da194170>
__pos__ - <method-wrapper '__pos__' of int object at 0x7af3da194170>
__pow__ - <method-wrapper '__pow__' of int object at 0x7af3da194170>
__radd__ - <method-wrapper '__radd__' of int object at 0x7af3da194170>
__rand__ - <method-wrapper '__rand__' of int object at 0x7af3da194170>
__rdivmod__ - <method-wrapper '__rdivmod__' of int object at
0x7af3da194170>
__reduce__ - <built-in method __reduce__ of int object at
0x7af3da194170>
__reduce_ex__ - <built-in method __reduce_ex__ of int object at
0x7af3da194170>
__repr__ - <method-wrapper '__repr__' of int object at 0x7af3da194170>
```

```
__rfloordiv__ - <method-wrapper '__rfloordiv__' of int object at
0x7af3da194170>
__rlshift__ - <method-wrapper '__rlshift__' of int object at
0x7af3da194170>
__rmod__ - <method-wrapper '__rmod__' of int object at 0x7af3da194170>
__rmul__ - <method-wrapper '__rmul__' of int object at 0x7af3da194170>
__ror__ - <method-wrapper '__ror__' of int object at 0x7af3da194170>
__round__ - <built-in method __round__ of int object at 0x7af3da194170>
__rpow__ - <method-wrapper '__rpow__' of int object at 0x7af3da194170>
__rrshift__ - <method-wrapper '__rrshift__' of int object at
0x7af3da194170>
__rshift__ - <method-wrapper '__rshift__' of int object at
0x7af3da194170>
__rsub__ - <method-wrapper '__rsub__' of int object at 0x7af3da194170>
__rtruediv__ - <method-wrapper '__rtruediv__' of int object at
0x7af3da194170>
__rxor__ - <method-wrapper '__rxor__' of int object at 0x7af3da194170>
__setattr__ - <method-wrapper '__setattr__' of int object at
0x7af3da194170>
__sizeof__ - <built-in method __sizeof__ of int object at
0x7af3da194170>
__str__ - <method-wrapper '__str__' of int object at 0x7af3da194170>
__sub__ - <method-wrapper '__sub__' of int object at 0x7af3da194170>
__subclasshook__ - <built-in method __subclasshook__ of type object at
0x59f8d3538320>
__truediv__ - <method-wrapper '__truediv__' of int object at
0x7af3da194170>
__trunc__ - <built-in method __trunc__ of int object at 0x7af3da194170>
__xor__ - <method-wrapper '__xor__' of int object at 0x7af3da194170>
as_integer_ratio - <built-in method as_integer_ratio of int object at
0x7af3da194170>
bit_count - <built-in method bit_count of int object at 0x7af3da194170>
bit_length - <built-in method bit_length of int object at
0x7af3da194170>
conjugate - <built-in method conjugate of int object at 0x7af3da194170>
denominator - 1
from_bytes - <built-in method from_bytes of type object at
0x59f8d3538320>
imag - 0
numerator - 5
real - 5
to_bytes - <built-in method to_bytes of int object at 0x7af3da194170>
```

Klassen in Python

Alle Klassendefinitionen beginnen mit dem Schlüsselwort `class`, gefolgt vom Namen der Klasse und einem Doppelpunkt. Jeder Code, der unterhalb der Klassendefinition eingerückt ist, wird als Teil des Klassenhauptteils betrachtet.

Analog zu C++ nutzt Python für die Interaktion mit den Klassenelementen eine *dot notation*.

OOPclass.py

```python
import inspect

class Dog:    # Schlüsselwort "class"
    family = "Canidae"
    name = "Bello"
    age = 5

d = Dog()
print(d.family)
d.name = "Russel"
print(d.name)

for name, data in inspect.getmembers(d):
    if name == '__builtins__':
        continue
    print(f'{name} - {repr(data)}')
```

```
Canidae
Russel
__class__ - <class '__main__.Dog'>
__delattr__ - <method-wrapper '__delattr__' of Dog object at
0x707623cbfc10>
__dict__ - {'name': 'Russel'}
__dir__ - <built-in method __dir__ of Dog object at 0x707623cbfc10>
__doc__ - None
__eq__ - <method-wrapper '__eq__' of Dog object at 0x707623cbfc10>
__format__ - <built-in method __format__ of Dog object at
0x707623cbfc10>
__ge__ - <method-wrapper '__ge__' of Dog object at 0x707623cbfc10>
__getattribute__ - <method-wrapper '__getattribute__' of Dog object at
0x707623cbfc10>
__gt__ - <method-wrapper '__gt__' of Dog object at 0x707623cbfc10>
__hash__ - <method-wrapper '__hash__' of Dog object at 0x707623cbfc10>
__init__ - <method-wrapper '__init__' of Dog object at 0x707623cbfc10>
__init_subclass__ - <built-in method __init_subclass__ of type object
at 0x5c0efdf4de60>
__le__ - <method-wrapper '__le__' of Dog object at 0x707623cbfc10>
__lt__ - <method-wrapper '__lt__' of Dog object at 0x707623cbfc10>
__module__ - '__main__'
__ne__ - <method-wrapper '__ne__' of Dog object at 0x707623cbfc10>
__new__ - <built-in method __new__ of type object at 0x5c0ee5f9a800>
__reduce__ - <built-in method __reduce__ of Dog object at
0x707623cbfc10>
__reduce_ex__ - <built-in method __reduce_ex__ of Dog object at
0x707623cbfc10>
__repr__ - <method-wrapper '__repr__' of Dog object at 0x707623cbfc10>
__setattr__ - <method-wrapper '__setattr__' of Dog object at
0x707623cbfc10>
__sizeof__ - <built-in method __sizeof__ of Dog object at
0x707623cbfc10>
__str__ - <method-wrapper '__str__' of Dog object at 0x707623cbfc10>
__subclasshook__ - <built-in method __subclasshook__ of type object at
0x5c0efdf4de60>
__weakref__ - None
age - 5
family - 'Canidae'
name - 'Russel'
Canidae
Russel
__class__ - <class '__main__.Dog'>
```

```
__delattr__ - <method-wrapper '__delattr__' of Dog object at
0x763b58863c10>
__dict__ - {'name': 'Russel'}
__dir__ - <built-in method __dir__ of Dog object at 0x763b58863c10>
__doc__ - None
__eq__ - <method-wrapper '__eq__' of Dog object at 0x763b58863c10>
__format__ - <built-in method __format__ of Dog object at
0x763b58863c10>
__ge__ - <method-wrapper '__ge__' of Dog object at 0x763b58863c10>
__getattribute__ - <method-wrapper '__getattribute__' of Dog object at
0x763b58863c10>
__gt__ - <method-wrapper '__gt__' of Dog object at 0x763b58863c10>
__hash__ - <method-wrapper '__hash__' of Dog object at 0x763b58863c10>
__init__ - <method-wrapper '__init__' of Dog object at 0x763b58863c10>
__init_subclass__ - <built-in method __init_subclass__ of type object
at 0x56897fe63e60>
__le__ - <method-wrapper '__le__' of Dog object at 0x763b58863c10>
__lt__ - <method-wrapper '__lt__' of Dog object at 0x763b58863c10>
__module__ - '__main__'
__ne__ - <method-wrapper '__ne__' of Dog object at 0x763b58863c10>
__new__ - <built-in method __new__ of type object at 0x568948353800>
__reduce__ - <built-in method __reduce__ of Dog object at
0x763b58863c10>
__reduce_ex__ - <built-in method __reduce_ex__ of Dog object at
0x763b58863c10>
__repr__ - <method-wrapper '__repr__' of Dog object at 0x763b58863c10>
__setattr__ - <method-wrapper '__setattr__' of Dog object at
0x763b58863c10>
__sizeof__ - <built-in method __sizeof__ of Dog object at
0x763b58863c10>
__str__ - <method-wrapper '__str__' of Dog object at 0x763b58863c10>
__subclasshook__ - <built-in method __subclasshook__ of type object at
0x56897fe63e60>
__weakref__ - None
age - 5
family - 'Canidae'
name - 'Russel'
```

Aufgabe: Erläutern Sie die Ausgabe folgenden Codes. Wie müssen wir das Ergebnis interpretieren?

OOPclass.py

```
1  import inspect
2
3  class Dog:
4      family = "Canidae"
5      name = "Bello"
6      age = 5
7
8  d1 = Dog()
9  print("dog 1:", d1.name, d1.age)
10 print(d1)
11 d2 = Dog()
12 print("dog 2:", d2.name, d2.age)
13 print(d2)
14
15 print(d1 == d2)
```

```
dog 1: Bello 5
<__main__.Dog object at 0x76b8dde83c10>
dog 2: Bello 5
<__main__.Dog object at 0x76b8ddd7fb50>
False
dog 1: Bello 5
<__main__.Dog object at 0x7f791ca27c10>
dog 2: Bello 5
<__main__.Dog object at 0x7f791c927b50>
False
```

Antworten:

- d1 und d2 sind Objekte vom Typ "Hund" mit gleichen Attributen (Name, Alter, Familie), aber es handelt sich trotzdem um verschiedene Objekte (deren Daten an verschiedenen Stellen im Speicher liegen)
- Gleichheit von zwei Objekten der Klasse Hund wird offenbar nicht als Gleichheit aller Attribute berechnet.
    - Man könnte Gleichheit von Hunden auch anders definieren (durch Definition einer speziellen Methode `__eq__`)

Zudem fällt auf: Name und Alter sind für individuelle Hunde üblicherweise verschieden, die Familie "Canidae" bezieht sich aber auf alle Hunde. Um dies besser zu modellieren, sollte zwischem Instanzvariablen und Klassenvariablen unterschieden werden ...

OOP Grundelemente in Python

Frage: Für welche Aufgaben ist der Konstruktor in einer Klasse verantwortlich?

OOPclass.py

```
class Dog:
    family = "Canidae"      # Klassenvariable
    def __init__(self, name, age):
        self.name = name    # Instanzvariable
        self.age = age

i = Dog("Rex", 5)
print(i.name, i.family, i.age)
```

```
Rex Canidae 5
```

Instanzmethoden sind Funktionen, die innerhalb einer Klasse definiert sind und nur von einer Instanz dieser Klasse aufgerufen werden können. Genau wie bei `__init__()` ist der erste Parameter einer Instanzmethode immer self.

OOPclass.py

```
class Dog:
    family = "Canidae"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def makeSound(self):      # : nicht vergessen!
        print(f"{self.name} says Wuff")

i = Dog("Rex", 5)
i.makeSound()
```

```
Rex says Wuff
Rex says Wuff
```

Aufgabe: Schreiben Sie eine Methode, so dass eine Instanz von Dog in Abhängigkeit von ihrem Alter schläft. Recherchieren Sie dazu unter `python delay` die notwendigen Methoden der `time` Klasse.

Wie Sie bereits bei der Inspektion der `list`, `int` aber auch der `Dog` Klasse gesehen haben, existiert eine Zahl von vordefinierten Funktionen - die sogenannten *dunder Methods*. Das Wort *dunder* leitet sich von *double underscore* ab.

Methode	Typ	implementiert
`__init__()`	Konstruktor	
`__str__()`	Methode	Generiert einen String aus den Objektdaten
...		
`__add__()`	Operator Obj `+` Obj	Arithmetische Operation
...		
`__eq__()`	Operator Obj `==` Obj	Logische Operation
`__lt__()`	Operator Obj `<=` Obj	
...		

Eine gute Einführung und detailierte Erklärung liefert Link

Kapselung

Python nutzt zwei führende Unterstriche, um Methoden und Variablen als *private* zu markieren.

private.py

```
1  class A:
2    def method_public(self):
3      print("This is a public method")
4
5    def __method_private(self):
6      print("This is a private method")
7
8  obj = A()
9  obj.method_public()
```

```
This is a public method
This is a public method
```

Auf private Methoden einer Klasse kann weder außerhalb der Klasse noch von irgendeiner Basisklasse aus zugegriffen werden kann.

> Wie können wir die private Methode überhaupt aufrufen?

Vererbung

> Was stört Sie an folgendem Codebeispiel?

RedundantCode.py

```
1  class Student:
2    def __init__(self, fname, lname):
3      self.firstname = fname
4      self.lastname = lname
5
6    def printname(self):
7      print("Student -", self.firstname, self.lastname)
8
9
10 class StaffMember:
11   def __init__(self, fname, lname):
12     self.firstname = fname
13     self.lastname = lname
14
15   def printname(self):
16     print("Staff -", self.firstname, self.lastname)
17
18 humboldt = Student("Alexander", "von Humboldt")
19 cotta = StaffMember("Bernhard", "von Cotta")
20
21 humboldt.printname()
22 cotta.printname()
```

```
Student - Alexander von Humboldt
Staff - Bernhard von Cotta
Student - Alexander von Humboldt
Staff - Bernhard von Cotta
```

Vererbung überträgt das Verhalten einer Basisklasse auf eine abgeleitete Klasse. Dadurch wird redundanter Code gespart.

Inheritance.py

```
1  class Person:
2    def __init__(self, fname, lname):
3      self.firstname = fname
4      self.lastname = lname
5
6    def printname(self):
7      print(self.firstname, self.lastname)
8
9  class Student(Person):
10   pass
11
12 class StaffMember(Person):
13   pass
14
15 humboldt = Student("Alexander", "von Humboldt")
16 cotta = StaffMember("Bernhard", "von Cotta")
17
18 humboldt.printname()
19 cotta.printname()
```

```
Alexander von Humboldt
Bernhard von Cotta
Alexander von Humboldt
Bernhard von Cotta
```

Unterklassen erweitern ihre Oberklasse typischerweise um zusätzliche Attribute und Methoden. Methoden der Unterklasse können Methoden der Oberklasse überschreiben (*Method Overriding*). Im folgenden Beispiel wird dies anhand von zwei *dunder Methoden* gezeigt:

- in der `__init__`-Methode (Konstruktor zur Erzeugung von Instanzen) ist der Unterklassen wird jeweils ein weiteres Attribut angelegt. Die Konstruktoren der Unterklassen müssen auch den Konstruktor der Oberklasse aufrufen.
- die `__str()__`-Methode liefert eine String-Repräsention des Objekts, die inbesonder auch von `print` genutzt wird. Die Implementierung in den Unterklassen rufen hier auch die `__str__`-Methode der Oberklasse auf.

Inheritance2.py

```
"Unterklassen mit zusätzlichen Attributen"

class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student(Person):
    def __init__(self, fname, lname, id=10000):
        super().__init__(fname, lname)
        self.student_id = id

    def __str__(self):
        return "Student: " + super().__str__() + f", ID: {self.studen

class StaffMember(Person):
    def __init__(self, fname, lname, id=2000):
        super().__init__(fname, lname)
        self.staff_id = id

    def __str__(self):
        return "Staff Member: " + super().__str__() + f", ID: {self
            .staff_id}"

humboldt = Student("Alexander", "Humboldt")
cotta = StaffMember("Bernhard", "von-Cotta", id=2001)

print(humboldt)
print(cotta)
```

```
Student: Alexander Humboldt, ID: 10000
Staff Member: Bernhard von-Cotta, ID: 2001
Student: Alexander Humboldt, ID: 10000
Staff Member: Bernhard von-Cotta, ID: 2001
```

Instanzvariablen und Klassenvariablen

Die Unterscheidung zwischen Instanzvariablen und Klassenvariablen wurde schon oben bei der Klasse Dog angesprochen.

Instanzvariablen (oder *Member Variables*) sind typischerweise für jedes Objekt unterschiedliche belegt. Beispiele sind etwa der Name von Personen oder Tieren (Alexander, Mary, Fido, ...) oder die Matrikelnummer von Studierenden.

Klassenvariablen beziehen dagegen sich auf die Klasse selbst.

- Der Zugriff auf Klassenvariablen sollte nach dem Schema **Klassenname.Klassenvariable** erfolgen!

Im folgenden Beispiel wird in der Klasse Student eine Klassenvariable *next_available_id* zur automatischen Generierung eindeutiger Martikelnummern für neue Studierende genutzt.

ClassVariable.py

```
1  class Person:
2      def __init__(self, fname, lname):
3          self.firstname = fname
4          self.lastname = lname
5
6      def __str__(self):
7          return f"{self.firstname} {self.lastname}"
8
9  class Student(Person):
10
11     next_available_id = 10000
12
13     def __init__(self, fname, lname):
14         super().__init__(fname, lname)
15         self.student_id = Student.next_available_id
16         Student.next_available_id += 1
17
18     def __str__(self):
19         return "Student: " + super().__str__() + f", ID: {self.studen
20
21
22 humboldt = Student("Alexander", "Humboldt")
23 hegeler = Student("Mary", "Hegeler")
24
25 print(humboldt)
26 print(hegeler)
27 print("Next available student ID:", Student.next_available_id)
```

```
Student: Alexander Humboldt, ID: 10000
Student: Mary Hegeler, ID: 10001
Next available student ID: 10002
Student: Alexander Humboldt, ID: 10000
Student: Mary Hegeler, ID: 10001
Next available student ID: 10002
```

Python und C++ mit Blick auf OOP Konzepte

- Das Konzept der (Methoden-)Überladung wird in Python nicht nativ unterstützt!
  - in C++ kann es in einer Klasse mehrere Methoden gleichen Namens geben, sofern sich die Typen der Parameter unterscheiden
  - in Python kann es dagegen nur eine Methode mit demselben Namen geben
  - als Konsequenz werden in Python oft Funktionen mit relativ vielen Parametern definiert
    - viele oder oft auch alle Parameter haben Default-Werte, oft `None`
    - zusätzliche Typannotationen der Parameter verbessern die Lesbarkeit
    - der Aufruf der Methoden erfolgt dann flexibel über Schlüsselwort-Argumente (d.h. durch explizite Angabe des Parameternames)

Dog.py

```python
class Dog:
    def __init__(self,
                 name:str | None = None,
                 age:int | None = None,
                 breed:str | None = None):
        self.name = name
        self.age = age
        self.breed = breed

    def __str__(self):
        return f"Dog(Name: {self.name}, Age: {self.age}, Breed: {self
          .breed})"

# Creating Dog instances with different combinations of arguments
d1 = Dog(name = "Buddy", age=3, breed = "Golden Retriever")
d2 = Dog(age = 5, name = "Max")
d3 = Dog(name = "Bella", breed = "Beagle")
d4 = Dog()

print(d1, d2, d3, d4, sep="\n")
```

```
Dog(Name: Buddy, Age: 3, Breed: Golden Retriever)
Dog(Name: Max, Age: 5, Breed: None)
Dog(Name: Bella, Age: None, Breed: Beagle)
Dog(Name: None, Age: None, Breed: None)
```

- Private ist in Python nicht wirklich private
  - direkter Aufruf privater Methoden über ihren eigentlichen Methodennamen resultiert in einem Fehler
  - private Methoden *können* jedoch mittels "Name Mangling" aufgerufen werden
  - private Methoden *sollte* man trotzdem nicht von außerhalb der Klasse aufrufen (schlechter Stil)

NameMangling.py

```
1  class A:
2    def fun(self):
3      print("This is a public method.",
4            "It may call a private method but this is none of your busi
               )
5
6    def __fun(self):
7      print("This is a private method.")
8
9  obj = A()
10 obj.fun()
11 # obj.__fun()   # <- AttributeError
12 obj._A__fun()   # <- Name Mangling  "_classname__function"
```

```
This is a public method. It may call a private method but this is none
of your business.
This is a private method.
This is a public method. It may call a private method but this is none
of your business.
This is a private method.
```

OOP Beispiele

Nehmen wir an, dass wir eine Liste von Vornamen erzeugen wollen. Dabei soll sichergestellt werden, dass diese unabhängig von den Eingaben der Bediener vergleichbar sind. Zudem sollen fehlerhafte Eingaben, die zum Beispiel Zahlen enthalten erkannt und gefiltert werden.

NameList.py

```
1  class NameList(list):
2    def __init__(self):
3      super().__init__()
4
5    def append(self, item):
6      if isinstance(item, str) and item.isalpha():
7        super().append(item.lower())
8      else:
9        print(f"Cannot add {item} to name list!",
10             "Expected a string with alphabetic characters only.")
11
12   def uniques(self):
13     # return set(self)  # no duplicates, but unordered
14     return sorted(set(self))  # no duplicates, ordered
15
16 friends = NameList()
17 friends.append("Jannes")
18 friends.append("linda")
19 friends.append("Moritz")
20 friends.append("MORITZ")
21 friends.append("Linda2")  # name with digit is not allowed
22 friends.append(42)   # wrong data type for name lists
23
24 print(friends)
25 print(friends.uniques())
```

```
Cannot add Linda2 to name list! Expected a string with alphabetic
characters only.
Cannot add 42 to name list! Expected a string with alphabetic
characters only.
['jannes', 'linda', 'moritz', 'moritz']
['jannes', 'linda', 'moritz']
Cannot add Linda2 to name list! Expected a string with alphabetic
characters only.
Cannot add 42 to name list! Expected a string with alphabetic
characters only.
['jannes', 'linda', 'moritz', 'moritz']
['jannes', 'linda', 'moritz']
```

Dafür schreiben wir eine abgeleitet Listenklasse mit einer eigenen Implementierung von `append()`.

Aufgabe Erweitern Sie die Implementierung auf die `extend()` Methode der Listen.

Zum Vergleich mit dem einführenden C++-Beispiel hier noch eine Python-Implementierung der Rectangle-Klasse in C++. Der `+=`-Operator wird hier mittels der dunder-Methode `__iadd__()` (in-place addition) implementiert.

Rectangle.py

```python
class Rectangle:
    def __init__(self, width: float, height: float):
        self.width = abs(width)  # ensure non-negative width
        self.height = abs(height)

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

    def __iadd__(self, offset):
        ratio = (offset.area() + self.area()) / self.area()
        self.width = ratio * self.width
        return self

if __name__ == "__main__":
    rect_a = Rectangle(3, 4)
    rect_b = Rectangle(1, 3)
    print(f"Fläche a : {rect_a.area()}")
    print(f"Fläche b : {rect_b.area()}")
    rect_a += rect_b
    print(f"Summe    : {rect_a.area()}")
```

```
Fläche a : 12
Fläche b : 3
Summe    : 15.0
```

Dataclasses

Dataclasses, die es seit Python 3.7 gibt, ermöglichen eine einfache und komfortable Definition von Klassen, die hauptsächlich der Datenhaltung dienen, aber deren Verhalten (durch Definition von Methoden) weniger wichtig ist. Sie sind in etwa vergleichbar zu `struct`s in C++.

Für Dataclasses wird viel "Boilerplate-Code" automatisch generiert, z.B. die Methoden `__init__()`, `__eq__()` und `__repr__()`.

Die automatische Code-Generierung erfolgt durch einen *Dekorator* `@dataclass`.

Felder von Datenklassen werden über Typannotationen sowie optionalen Default-Werten definiert (man verwendet den Begriff *Feld* auch, weil die Felder syntaktisch eher wie Klassenvariablen aussehen, durch den Dekorator aber zu Instanzvariablen gemacht werden; durch den anderen Begriff wird es weniger verwirrend).

dataclass_example.py

```python
from dataclasses import dataclass

@dataclass
class Dog:
    name: str | None = None  # field 'name' with default value None
    age: int | None = None
    breed: str | None = None

d1 = Dog(name = "Buddy", age=3, breed = "Golden Retriever")
d2 = Dog(age = 5, name = "Max")
d3 = Dog(name = "Bella", breed = "Beagle")
d4 = Dog(breed = "Beagle", name = "Bella")
d5 = Dog()

print(d1, d2, d3, d4, d5, sep="\n")
print("Is d3 equal to d4?", d3 == d4)  # True, as the field values ar
  same
```

```
Dog(name='Buddy', age=3, breed='Golden Retriever')
Dog(name='Max', age=5, breed=None)
Dog(name='Bella', age=None, breed='Beagle')
Dog(name='Bella', age=None, breed='Beagle')
Dog(name=None, age=None, breed=None)
Is d3 equal to d4? True
Dog(name='Buddy', age=3, breed='Golden Retriever')
Dog(name='Max', age=5, breed=None)
Dog(name='Bella', age=None, breed='Beagle')
Dog(name='Bella', age=None, breed='Beagle')
Dog(name=None, age=None, breed=None)
Is d3 equal to d4? True
```

Hinter den Kulissen erzeugt der Dekorator u.a. automatisch einen Konstruktor:

```
"Automatically generated __init__ method by @dataclass:"
    def __init__(self,
                 name: str | None = None,
                 age: int | None = None,
                 breed: str | None = None):
        self.name = name
        self.age = age
        self.breed = breed
```

Interessant an Dataclasses ist zudem, dass Instanzen entweder als veränderliche (Default) oder unveränderliche Objekte definiert werden können.

- Erweitert man den Dekorator um den Parameter `frozen=True`, dann sind die Instanzen unveränderlich
- Ein Vorteil von unveränderlichen Objekten ist, dass sie Elemente von Sets oder Schlüssel von Dictionaries sein können.

frozen_dataclass.py

```python
from dataclasses import dataclass

@dataclass(frozen=True)  # default: frozen=False
class Dog:
    name: str | None = None  # field 'name' with default value None
    age: int | None = None
    breed: str | None = None

d1 = Dog(name = "Buddy", age=3, breed = "Golden Retriever")
d2 = Dog(name = "Bella", breed = "Beagle")
d3 = Dog(breed = "Beagle", name = "Bella")

dogs = {d1, d2, d3}  # d2 and d3 are considered equal
print(dogs)
```

```
{Dog(name='Bella', age=None, breed='Beagle'), Dog(name='Buddy', age=3, breed='Golden Retriever')}
{Dog(name='Bella', age=None, breed='Beagle'), Dog(name='Buddy', age=3, breed='Golden Retriever')}
```

Aufgabe Definieren Sie ein Dictionary, in welchem Instanzen der Dataclass Dog als Schlüssel verwendet werden. Experimentieren Sie dabei mit den beiden möglichen Werten den Parameter `frozen` des Dekorators (False bzw. True).

Quiz

Objektorientierung in Python

Für welche der genannten Grundprinzipien der objektorientierten Programmierung treffen folgende Aussagen zu:

Kapselung	Vererbung	Polymorphie	
◯	◯	◯	Welche konkrete Implementierung der Methode aufgerufen wird, hängt davon ab mit welchem konkreten Objekt sie aufrufen wird.
◯	◯	◯	Objekte schützen ihre Daten und Methoden sofern diese nicht als "öffentlich" deklariert sind.
◯	◯	◯	Objekte können ihre Daten und Methoden an andere, spezielle Objekte weitergeben.

Klassen in Python

Mit welchem Schlüsselwort beginnen Klassendefinitionen in Python?

Wodurch muss `[_____]` ersetzt werden, um den Nachnamen von Student `neuer` auszugeben?

```
class Student:
    lastName = "Neuer"
    firstName = "Markus"
    age = 20

neuer = Student()
print([_____])
```

OOP Grundelemente in Python

Wie lauten die Ausgaben foldender Programme?

```
class Player:
    max_health = 100
    max_experience = 10

    def __init__(self, name, level):
        self.name = name
        self.level = level
        self.health = Player.max_health
        self.experience = Player.max_experience

p1 = Player("Peter", 2)
p2 = Player("Frank", 6)
print(p2.level, p2.experience)
```

```
class Player:
    max_health = 100
    max_experience = 10

    def __init__(self, name, level):
        self.name = name
        self.level = level
        self.health = Player.max_health
        self.experience = Player.max_experience

    def level_up(self):
        self.level += 1
        self.health = Player.max_health
        self.experience = 1

p1 = Player("Peter", 2)
p2 = Player("Frank", 6)
p2.level_up()
print(p2.level, p2.experience)
```

Welche dieser Methoden ist eine *dunder Method*?

- ◯ `__sum__`
- ◯ `sum`
- ◯ `_sum`
- ◯ `_sum_`

Kapselung

Welche der im folgenden Code aufgeführten Methoden und Variablen sind öffentlich und welche privat?

```
class Dog:
  def bark(self):
      print("woof")

  def __bark_loud(self):
    print("WOOF!")

Fifi = Dog()
Fifi.bark_loud()
```

`bark()`	`__bark_loud()`	
◯	◯	öffentlich
◯	◯	privat

Was ist die Ausgabe des oben gezeigten Codes?

- ◯ woof
- ◯ WOOF!
- ◯ Das Programm wird mit einem Error abgebrochen

Ist es in Python grundsätzlich möglich auch private Methoden auszuführen?

- ◯ Ja
- ◯ Nein

Wordurch muss `[_____]` ersetzt werden, um die Ausgabe `WOOF!` zu erzeugen?

```
class Dog:

  def bark(self):
      [_____]

  def __bark_subtle(self):
    print("woof")

  def __bark_loud(self):
    print("WOOF!")

Fifi = Dog()
Fifi.bark()
```

Vererbung

Wodurch muss `[_____]` ersetzt werden, um eine neue Klasse `Auto` zu erstellen, die das Verhalten der Klasse `Fahrzeug` erbt?

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

class [_____]:
  pass
```

```
a1 = Auto(70)
```