OOP
Konventionen & Konzepte

Vererbung: Wie prüfen?

```python
class PosInt(int):
    """Positive Integer"""
    def __init__(self, val: int):
        if val < 0:
            raise ValueError("negative value")
        else:
            self.val = val

nr = PosInt(3)
print(type(nr)) # <class '__main__.PosInt'>
print(type(nr) == PosInt)      # True
print(type(nr) == int)         # False
print(isinstance(nr, PosInt)) # True
print(isinstance(nr, int))     # True
```

- type() prüft den exakten Typen
- isinstance() prüft **auch** Vererbungen

Vererbung: Polymorphie

- Prinzip: Elternklasse hat Methoden, welche je Kindklasse anders aussehen
- Für UML-Vererbung gilt:
  - Methodenname pro Kindklasse eintragen!

```python
wohnzimmer_lichtschalter = LichtSchalter()
wohnzimmer_lichtschalter.drücken() # Licht geht an!
wohnzimmer_lichtschalter.drücken() # Licht geht aus!

auto_startknopf = MotorStartKnopf()
auto_startknopf.drücken() # Motor startet!
auto_startknopf.drücken() # Motor stoppt!
auto_startknopf.drücken() # Motor startet!

sinnloser_schalter = KaputterSchalter()
print(sinnloser_schalter.ist_an) # False
sinnloser_schalter.drücken() # Nichts passiert!
print(sinnloser_schalter.ist_an) # False
sinnloser_schalter.drücken() # Nichts passiert!
```

```python
class Schalter:

    def __init__(self):
        self.ist_an = False

    def drücken(self):
        self.ist_an = not self.ist_an


class LichtSchalter(Schalter):

    def drücken(self):
        super().drücken()
        if self.ist_an:
            print("Licht geht an!")
        else:
            print("Licht geht aus!")


class MotorStartKnopf(Schalter):

    def drücken(self):
        super().drücken()
        if self.ist_an:
            print("Motor startet!")
        else:
            print("Motor stoppt!")


class KaputterSchalter(Schalter):

    def drücken(self):
        print("Nichts passiert!")
```

Abstrakte Klassen

- Bekannt: Klassen sind Baupläne für Objekte
- Neu: Klassen können Baupläne für Klassen sein
  - Für UML: Siehe 10_UML_2.pdf Seite 14
- Idee: Klassen haben „leere“ Methoden, die von den Kindklassen definiert werden (müssen!)
- Vorgehen:
  - Aus Modul `abc` folgendes importieren:
    Klasse ABC
    Decorator `abstractmethod`
  - Die abstrakte Klasse von ABC erben lassen
  - Für alle Methoden, welche die Kindklassen definieren müssen: @abstractmethod
- Achtung: Die abstrakte Klasse kann nicht mehr für die Instanziierung von Objekten genutzt werden!

```python
from abc import ABC, abstractmethod

class GeometrischeForm(ABC):

    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def berechne_flächeninhalt(self):
        pass


class Dreieck(GeometrischeForm):

    def __init__(self, grundseite: float, höhe: float):
        self.g = grundseite
        self.h = höhe

    def berechne_flächeninhalt(self) -> float:
        return 0.5 * self.g * self.h


class Kreis(GeometrischeForm):

    def __init__(self, radius: float):
        self.r = radius

    def berechne_flächeninhalt(self) -> float:
        return 3.14 * self.r ** 2

```

Datenkapselung

- Prinzip: Auf nicht öffentliche Attribute soll nur über Methoden zugegriffen werden können
- Mehr Details -> *09_UML.pdf* Seite 5-7
- Python weicht hier sehr stark von der Konvention ab
  - Dennoch macht es Sinn insbesondere private Attribute zu nutzen
- Alternativ: @property Decorator nutzen
- Beispiel siehe 11_Bsp_Datenkapselung.py

Funktions- und Operatorüberladung

- Idee: Klassen sind Datentypen und sollten daher mit Komponenten von Python (Funktionen, Operatoren) kompatibel sein
- Beispiel: Der Operator +
  - int, float: Addition (3 + 4 = 7)
  - str: Konkatenation ("3" + "4" = "34")
  - list: Erweitern ([3] + [4] = [3, 4])
  - Eigene Klasse: TypeError!
- Lässt sich ändern:
  - Eigene Klassen lassen sich in die Python-Welt einbetten

Funktions- und Operatorüberladung: print

- Objekte printen? Sagt wenig über Inhalt aus:
  - <__main__.Person object at 0x0000027A….>
- Die Darstellung der Klasse für u.A. die print() Funktion lässt sich steuern:
  - Die __repr__() Methode gibt an, wie das Objekt repräsentiert (=dargestellt) wird
  - Sie **muss** string zurückgeben!

```python
class Person:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    def __repr__(self) -> str:
        return "Person(" + "name='" + self.name + "', age=" + str(self.age) + ")"

p0 = Person("Klaus", 39)
print(p0) # Person(name='Klaus', age=39)
```

Exkurs: Formatted strings

- Problem: Strings zusammenbauen ist schwierig:

```
x = 1
y = 2.34
regular_str = "x = " + str(x) + ", y = " + str(y)
```

- Hohes Fehlerpotential: Konkatenation nur zwischen str und str erlaubt
- Lösung: Formatted strings:

```
x = 1
y = 2.34
regular_str = "x = " + str(x) + ", y = " + str(y)
formatted_str = f"x = {x}, y = {y}"
print(regular_str == formatted_str) # True
```

  - Änderung: f vor Anführungszeichen, Variablen in geschweiften Klammern
  - Vorteile: Kürzer, lesbarer, kein str() notwendig
- Mehr Infos: https://docs.python.org/3/tutorial/inputoutput.html#fancier-output-formatting

Funktions- und Operatorüberladung: len

- Die Länge eines Objekts herausfinden (wo es sinnvoll ist)
- Mit der __len__ Methode!
- Rückgabe **muss** int sein!

```python
class Warenkorb:

    def __init__(self, inhalt: list):
        self.inhalt = inhalt

    def __len__(self) -> int:
        return len(self.inhalt)


wk = Warenkorb(["Monitor", "Maus", "Kabel"])
print(len(wk))

# ohne __len__:
# TypeError: object of type 'Warenkorb' has no len()

# mit __len__:
# 3
```

Funktions- und Operatorüberladung: bool

- Objekten einen boolschen Wahrheitswert verpassen
- Mit der __bool__ Methode
- Rückgabe **muss** bool sein!

```python
class Warenkorb:

    def __init__(self, inhalt: list):
        self.inhalt = inhalt

    def __bool__(self) -> bool:
        return len(self.inhalt) > 0


wk = Warenkorb(inhalt=[])
print(bool(wk)) # False
wk.inhalt.append("Monitor")
print(bool(wk)) # True
```

Funktions- und Operatorüberladung: getitem

- Mehrere Funktionen:
  - Elemente eines Objekts adressieren können (wie bei Listen)
  - Objekte iterable machen
- Mit der __getitem__ Methode
- Freiheit beim Adressieren:
  - Über Indices (-> Listen)
  - Über Keys (-> Dictionary)
  - **Wie** letztlich adressiert wird, entscheidet ihr!
    (z.b. key = "M" -> hole alle Artikel mit M)

```python
class Warenkorb:

    def __init__(self, inhalt: list):
        self.inhalt = inhalt

    def __getitem__(self, idx: int):
        return self.inhalt[idx]


wk = Warenkorb(inhalt=["Monitor", "Maus", "Tastatur"])
print(wk[0])   # Monitor
print(wk[-1])  # Tastatur
print(wk[1:2]) # ["Maus"]

for elem in wk:
    print(elem) # Monitor
                # Maus
                # Tastatur
```

Funktions- und Operatorüberladung: add

- Objekte sollen mit dem Operator + zurecht kommen
- Mit zwei Methoden:
  - __add__
    Addieren und zuweisen
  - __iadd__
    Direkt addieren (+=)
- Rückgabe:
  - __add__
    Neues Objekt der selben Klasse
  - __iadd__
    Verändertes Objekt

```python
class Warenkorb:

    def __init__(self, inhalt: list):
        self.inhalt = inhalt

    def __add__(self, other):
        neuer_inhalt = self.inhalt.copy()
        neuer_inhalt.append(other)
        return Warenkorb(neuer_inhalt)

    def __iadd__(self, other):
        self.inhalt.append(other)
        return self


wk = Warenkorb(inhalt=["Monitor"])
print(wk.inhalt)  # ['Monitor']
wk = wk + "Maus"  # Hier greift __add__()
wk += "Tastatur"  # Hier greift __iadd__()
print(wk.inhalt)  # ['Monitor', 'Maus', 'Tastatur']
```

Funktions- und Operatorüberladung

- Fazit: Es lässt sich sehr viel anpassen!
- Vollständige Liste: https://docs.python.org/3/reference/datamodel.html
- Vieles ist sehr speziell
- Hier ist eine gute Dokumentation **sehr wichtig**:
  - Geht nicht davon aus, dass andere Leute wissen warum z.b. Addition mit Objekten eurer Klasse so arbeitet wie sie arbeitet!

Nützliche Decorator: @dataclass

- Idee: Vereinfachung der Klassendefinition
- Besonders nützlich für *datenorientierte Klassen*
  - Klassen, die keine bzw. sehr wenige Methoden haben
- Kann allerdings auch generell genutzt werden
  - Vorteil: Spart (sehr) viele Zeilen und macht alles übersichtlicher
  - Definiert __init__ & __repr__ automatisch (unter Anderem!)

```python
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    is_alive: bool = True

p0 = Person("Klaus", 39)
print(p0) # Person(name='Klaus', age=39, is_alive=True)
print(p0.__dict__) # {'name': 'Klaus', 'age': 39, 'is_alive': True}
```

Nützliche Decorator: @classmethod

- Problem: Objekte der Klasse sollen auf unterschiedliche Arten erstellt werden können:
  - Z.b. aus einer .csv oder .json Datei oder aus Datenbanken oder ganz anders
- Vermeintliche Lösung: In der __init__ prüfen, um was für eine Quelle es sich handelt
  - Erzeugt eine lange, unübersichtliche __init__
- Idee: Methode soll nicht im Kontext des Objekts sondern der Klasse genutzt werden
  - Geht grundsätzlich ohne Decorator aber gefährlich!

```python
from datetime import date

class Person:

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    @classmethod
    def from_birth_year(cls, name: str, birth_year: int):
        return cls(name=name, age=date.today().year - birth_year)


p0 = Person.from_birth_year("Alex", 1999)
print(p0.__dict__) # {'name': 'Alex', 'age': 23}
```

- Regeln:

1. @classmethod Decorator über der Definition
2. Erster Parameter: cls (referenziert die Klasse)
3. Rückgabe: Objekt der Klasse cls

Hausaufgabe

- Siehe 11_Hausi.py