Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Michael Uhl
Wintersemester 2022

Universität Freiburg
Institut für Informatik

Übungsblatt 8

Abgabe: Montag, 12.12.2022, 9:00 Uhr morgens

Hinweis

Hier ist eine Übersicht der bisherigen Typannotationen:

```
from typing import Optional
x: int             = 42
x: float           = 42.0
x: complex         = 42.0 + 23.0 * 1j
x: bool            = True
x: str             = 'foo'
x: None            = None
x: list[int]       = [1, 2, 3]
x: list[list[int]] = [[1, 2, 3], [4, 5]]
x: MyClass         = MyClass(...)  # Angenommen MyClass wurde definiert
x: int | bool      = 42
x: int | bool      = True
x: Optional[int]   = None          # Kurzform für int | None
```

Für komplexere `Union`s empfiehlt es sich zur Lesbarkeit Typ-Aliase zu definieren:

```
MyType = int | bool | list[int] | MyClass | None
def combine(x: MyType, y: MyType) -> MyType:
    # ...
```

Aufgabe 8.1 (Vererbung; Datei: `fleet.py`; Punkte: 16 = 6+4+6)

Für einen Fuhrpark bestehend aus PKWs, LKWs, Bussen und Fahrrädern soll eine Klassenhierarchie entworfen werden.

Dabei sollen die unterschiedlichen Fahrzeuge sowohl gemeinsame als auch unterschiedliche Attribute besitzen, die jeweils durch einen `int` beschrieben werden.

- `Fahrzeug`
- `Kraftfahrzeug`
- `Bus` – Zustand(%), Neupreis(€), Leergewicht (kg), Baujahr, Leistung (kW), Anzahl Sitzplätze, Anzahl Stehplätze.
- `Fahrrad` – Zustand(%), Neupreis(€), Leergewicht (kg), Baujahr, Rahmengröße (cm).
- `PKW` – Zustand(%), Neupreis(€), Leergewicht (kg), Baujahr, Leistung (kW), Anzahl Sitzplätze.
- `LKW` – Zustand(%), Neupreis(€), Leergewicht (kg), Baujahr, Leistung (kW), Anzahl Sitzplätze, Zuladung (kg)

Vermeiden Sie, soweit möglich, Wiederholungen in den folgenden Aufgabenteilen, indem sie Vererbung sinnvoll nutzen.

(a) Implementieren Sie die Klassenhierarchie wie oben angegeben. Machen Sie dabei Gebrauch von Vererbung, sodass Argumente soweit möglich in den Oberklassen `Fahrzeug` und `Kraftfahrzeug` definiert werden. Das bedeutet, Attribute sollen nicht doppelt definiert werden.

Verwenden Sie Datenklassen wie in der Vorlesung beschrieben. Geben Sie die Attribute dabei in der oben angegebenen Reihenfolge an.

Überprüfen Sie die Klassen, indem Sie die folgende Ausgabe vergleichen. Die Zeilenumbrüche sind hier nur eingefügt, da die Ausgabe sonst nicht in das Dokument passt.

```
>>> print(Bus(25, 100000, 2500, 1995, 200, 80, 40))
Bus(zustand=25, neupreis=100000, leergewicht=2500, baujahr=1995,
leistung=200, sitzplaetze=80, stehplaetze=40)
>>> print(Fahrrad(100, 500, 10, 2019, 55))
Fahrrad(zustand=100, neupreis=500, leergewicht=10, baujahr=2019,
rahmengroesse=55)
>>> print(PKW(95, 20000, 1200, 2016, 90, 5))
PKW(zustand=95, neupreis=20000, leergewicht=1200, baujahr=2016,
leistung=90, sitzplaetze=5)
>>> print(LKW(65, 150000, 3200, 1991, 280, 3, 4600))
LKW(zustand=65, neupreis=150000, leergewicht=3200, baujahr=1991,
leistung=280, sitzplaetze=3, zuladung=4600)
```

(b) Im folgenden soll überprüft werden, ob die eingegeben Werte für die Attribute korrekt sind. Implementieren Sie die Methode `__post_init__`, diese wird bei Datenklassen automatisch nach der Instantiierung aufgerufen.

Verwenden Sie dabei Vererbung, sodass die Attribute in der selben Klasse geprüft werden, in der sie auch definiert wurden. Die Subklassen müssen dann nur noch die Attribute prüfen, welche neu hinzu gekommen sind.

Verwenden Sie den Befehl `assert BEDINGUNG, AUSGABESTRING` zur Prüfung der Werte. Verwenden Sie insbesondere **kein** `print`.

- *Zustand* muss mindestens 0 und maximal 100 sein.
- *Neupreis* muss mindestens 0 sein.
- *Leergewicht* muss größer als 0 sein.
- *Baujahr* muss größer als 1900 sein.
- *Leistung* muss größer als 0 sein.
- *Anzahl Sitzplätze* muss größer als 0 sein.
- *Anzahl Stehplätze* muss kleiner oder gleich der `Anzahl Sitzplätze` sein.
- *Rahmengröße* muss größer als 0 sein.
- *Zuladung* muss größer als 0 sein und darf das doppelte Leergewicht nicht überschreiten.

Die letzte Zeile der Fehlerausgabe (nach dem Traceback) soll dabei wie folgt aussehen:

```
>>> Bus(-7, 100000, 2500, 1995, 200, 80, 40)
AssertionError: Zustand -7% muss zwischen 0% und 100% liegen.
>>> Bus(25, -12, 2500, 1995, 200, 80, 40)
AssertionError: Neupreis -12€ muss mindestens 0€ sein.
>>> Bus(25, 100000, -3, 1995, 200, 80, 40)
AssertionError: Leergewicht -3kg muss größer als 0kg sein.
>>> Bus(25, 100000, 2500, 1800, 200, 80, 40)
AssertionError: Baujahr 1800 muss größer als 1900 sein.
>>> Bus(25, 100000, 2500, 1995, -42, 80, 40)
AssertionError: Leistung -42kW muss größer als 0kW sein.
>>> Bus(25, 100000, 2500, 1995, 200, -80, 40)
AssertionError: Sitzplätze -80 muss größer als 0 sein.
>>> Bus(25, 100000, 2500, 1995, 200, 40, 80)
AssertionError: Stehplätze 80 muss kleiner oder gleich Sitzplätze 40 sein.
>>> Fahrrad(100, 500, 10, 2019, -55)
AssertionError: Rahmengröße -55cm muss größer als 0cm sein.
>>> LKW(65, 150000, 3200, 1991, 280, 3, 46000)
AssertionError: Zuladung 46000kg muss größer als 0kg und maximal 6400kg sein.
```

(c) Jetzt sollen einige Methoden definiert werden, um die Klassen sinnvoll zu nutzen. Nutzen sie Vererbung: Definieren Sie die Methode in der Superklasse, falls nötig überschreiben Sie die Methode in der Subklasse und rufen dann die Methode der Superklasse mit `super` auf.

Methode `gewicht` der Klasse `Fahrzeug` soll das *Gesamtgewicht* des Fahrzeugs als `int` zurückgeben.

Methode `plaetze` der Klasse `Kraftfahrzeug` soll die maximal mögliche Anzahl an Passagieren als `int` zurückgeben.

Methode `maut` der Klasse `Fahrzeug` soll einen Preis (in Cent) als `int` zurückgeben. Der Preis berechnet sich wie folgt: Ein Fünftel des Gesamtgewichtes des Fahrzeugs plus das 25-fache der maximal möglichen Anzahl an Passagieren. Bei der Division soll abgerundet werden. Doppelte Maut für LKWs. Fahrräder dürfen umsonst die Brücke überqueren.

Hinweis: In der Klasse `Fahrzeug` selbst kann die Methode `maut` noch nicht implementiert werden. Es handelt sich um eine sogenannte *abstrakte Methode*. Implementieren Sie sie daher wie folgt:

```
def maut(self) -> int:
    raise NotImplementedError
```

Anschließend überschreiben Sie die Methode in den Subklassen und rufen sie **nicht** mit `super` auf, da sonst der `NotImplementedError` entsteht.

Beispiel:

```
>>> bus = Bus(25, 100000, 2500, 1995, 200, 80, 40)
>>> print("Maut:", bus.maut())
Maut: 3500
```

Methode `alter` der Klasse `Fahrzeug` soll die Differenz vom Jahr 2022 und dem Baujahr in Jahren zurückgeben. Falls das Baujahr in der Zukunft liegt, soll die Funktion 0 zurückgeben.

Methode `marktwert` der Klasse `Fahrzeug` soll den Marktwert wie folgt berechnen: Der Prozentwert berechnet sich als Zustand minus 5 mal das Alter des Fahrzeugs. Marktwert ist dann der Prozentwert mal der Neupreis. Rabatt von 50% auf Fahrräder.

Der Marktwert soll 0 nicht unterschreiten.

Beispiel:

```
>>> pkw = PKW(95, 20000, 1200, 2016, 90, 5)
>>> print("Alter:", pkw.alter(), "Marktwert:", pkw.marktwert())
Alter: 5 Marktwert: 14000
```

Aufgabe 8.2 (Testing; Datei: `test_fleet.py`; Punkte: 2)

Jetzt erstellen Sie unittests, um die Funktionalität der `fleet.py` zu prüfen. Die Datei `test_fleet.py` muss sich im selben Ordner wie `fleet.py` befinden, sie müssen auch python in diesem Ordner aufrufen, damit der `import` funktioniert. Importieren Sie die Bus-Klasse mit `from fleet import Bus`.

Installieren Sie das Paket `pytest` mit `python3.10 -m pip install pytest`[1] und importieren Sie das Paket z.B. mit `import pytest`.

(a) Erstellen Sie die Funktion `test_maut`. In dieser Funktion erstellen Sie einen Bus: `bus = Bus(25, 100000, 2500, 1995, 200, 80, 40)` und prüfen mit `assert`, ob die Maut des Busses 3500 ist.

(b) Nun sollen Sie sicherstellen, dass die Attribute korrekt geprüft werden. Erstellen Sie die Funktion `test_assert` und fügen Sie folgenden Code ein:

```
with pytest.raises(AssertionError):
    Bus(25, 100000, -2500, 1995, 200, 80, 40)
```

Der Bus hat ein negatives Gewicht und dadurch entsteht ein `AssertionError`. `pytest` wird dann den Fehler abfangen und den Test bestätigen. Falls *kein* Fehler auftritt, schlägt der Test fehl mit `did not raise AssertionError`.

Starten Sie die Testsuite mit dem Befehl `pytest`.

Hinweis: Sie haben mit diesen Tests nur einen kleinen Teil der Funktionalität getestet, diese Testsuite ist also bei weitem nicht vollständig und hat nur eine geringe Abdeckung.

Aufgabe 8.3 (Erfahrungen; 2 Punkte; Datei: `NOTES.md`)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplatform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `3.5 h` steht dabei für 3 Stunden 30 Minuten.

Bei Gruppenaufgaben müssen alle Gruppenmitglieder die Abgabe hochladen. Ferner müssen Sie die RZ-Accounts aller Gruppenmitglieder angeben, sodass wir Sie bei der Korrektur zuordnen können und unsere Tutoren die Aufgabe nicht mehrfach korrigieren müssen. Fügen Sie hierzu in der `NOTES.md`-Datei unter der Zeile für den Zeitbedarf eine weitere Zeile hinzu die exakt das folgende Format hat:

```
Gruppe: xy123, xy234, xy345
```

Hierbei stehen `xy123`, `xy234`, und `xy345` für die RZ-Accounts der Gruppenmitglieder. Der Buildserver überprüft ebenfalls, ob Sie das Format korrekt angegeben haben.

[1] Windowsanwender: anstelle von `python3.10` versuchen Sie `python`, `py3` oder `py`.

Prüfen Sie, ob der Buildserver mit Ihrer Abgabe zufrieden ist, so wie es im Video zur Lehrplatform gezeigt wurde.