Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner
Simon Dorer, Timpe Hörig

Universität Freiburg
Institut für Informatik
Wintersemester 2025

Übungsblatt 11

Abgabe: Montag, 12.01.2026, 09:00 Uhr

Anmerkung zur Vorlesung: InitVar und private Attribute

In der Vorlesung wurde die Verwendung von `InitVar` und privaten Attributen wie folgt erklärt:

```
@dataclass
class Example:
    __attr: InitVar[int]

    def __post_init__(self, attr: int) -> None:
        self.__attr = attr
```

Dieses Verhalten ist an sich korrekt, liefert in Python jedoch folgenden Typfehler:

```
Dataclass field cannot use private name
```

Um dieses Problem zu umgehen, können wir die `InitVar`-Attribute ohne die führenden Unterstriche definieren und diese dann in der `__post_init__`-Methode den privaten Attributen zuweisen:

```
@dataclass
class Example:
    attr: InitVar[int]
    __attr: int = field(init=False)

    def __post_init__(self, attr: int) -> None:
        self.__attr = attr
```

Auf diese Weise können wir sowohl die Vorteile von `InitVar` als auch von privaten Attributen nutzen, ohne auf Typfehler zu stoßen. **Verwenden Sie dies in der folgenden Aufgabe, wenn Sie private Attribute in Kombination mit `InitVar` benötigen.**

Aufgabe 11.1 (Bankaccount; 8 Punkte; Datei: `money.py`)

(a) (4 Punkte) Implementieren Sie eine Datenklasse `BankAccount`, die einen Namen eines Kontoinhabers `__name`, eine IBAN-Adresse `__iban` und einen Kontostand `__balance` als *private* Attribute besitzt.

Dabei ist der Name ein *nicht-leerer* String, die IBAN ein String der Länge 22, der mit zwei beliebigen Großbuchstaben beginnt, gefolgt von 20 Ziffern und der Kontostand eine Gleitkommazahl, die den Kontostand in Euro angibt.

Nur der Name und die IBAN sollen beim Erstellen eines Objekts der Klasse `BankAccount` im Konstruktor übergeben werden. Da unsere Bank großzügig ist, werden neue Konten immer mit einem Startguthaben von 42.0 Euro initialisiert.

Implementieren sie zusätzlich für jedes Attribut eine Getter-Methode, die den Wert des jeweiligen Attributs zurückgibt. Nennen Sie die Methoden `get_name`, `get_iban` und `get_balance`.

Implementieren Sie außerdem eine Setter-Methode `set_name` für das Attribut `name`.

Stellen Sie sicher, dass die Validierung der Attribute sowohl im Konstruktor als auch in den Setter-Methoden durchgeführt wird. Vermeiden Sie Code-Duplikation. Falls die Validierung fehlschlägt, soll ein AssertionError ausgelöst werden.

Beispiele:

```
>>> account = BankAccount("Alice", "DE89370400440532013000")
>>> account.get_name()
'Alice'
>>> account.get_iban()
'DE89370400440532013000'
>>> account.get_balance()
42.0
>>> account.set_name("Alice Smith")
>>> account.get_name()
'Alice Smith'
```

(b) (2 Punkte) Um den Kontostand zu verändern, wollen wir die inplace-Addition (`+=`) und die inplace-Subtraktion (`-=`) überladen. Implementieren Sie hierzu die Methoden `__iadd__` und `__isub__`, die jeweils eine Gleitkommazahl `amount` als Argument erhalten. Die Funktionen sollen den Kontostand entsprechend um `amount` erhöhen oder verringern und das veränderte Objekt (`self`) zurückgeben.

Stellen Sie dabei sicher, dass `amount` immer positiv ist, dass der Kontostand bei einer Subtraktion nicht negativ wird und dass höchstens `1337` Euro auf einmal hinzugefügt werden können. Falls eine dieser Bedingungen nicht erfüllt ist, soll ein `AssertionError` ausgelöst werden.

Beispiele:

```
>>> account += 37.5
>>> account.get_balance()
79.5
>>> account -= 50.0
>>> account.get_balance()
29.5
```

(c) (1 Punkt) Implementieren Sie die Dunder-Methode `__eq__`, die zwei Bankkonten als gleich betrachtet, wenn sie dieselbe IBAN besitzen.

Zum Beispiel:

```
>>> account1 = BankAccount("Bob", "DE89370400440532013000")
>>> account2 = BankAccount("Charlie", "DE89370400440532013000")
>>> account3 = BankAccount("Bob", "FR7630006000011234567 8")
>>> account1 == account2
True
>>> account1 == account3
False
>>> account2 == account3
False
```

(d) (1 Punkt) Implementieren Sie Dunder-Methode `__str__`, die eine lesbare String-Repräsentation des Bankkontos in der Form `<name> - <iban>: <balance> EUR` zurückgibt.

Zum Beispiel:

```
>>> str(account)
'Alice Smith - DE89370400440532013000: 29.5 EUR'
```

Aufgabe 11.2 (Rekursion; 6 Punkte; Datei: `tail.py`)

Das Collatz-Problem, auch als (3n+1)-Vermutung bezeichnet, ist ein ungelöstes mathematisches Problem. Es besagt, dass die folgende Folge, für jede nicht negative natürliche Zahl `n`, immer die Zahl 1 erreicht:

$$c_{i+1} = \begin{cases} \frac{c_i}{2}, & \text{wenn } c_i \text{ gerade ist} \\ 3c_i + 1, & \text{wenn } c_i \text{ ungerade ist} \end{cases}$$

wobei $c_0 = n$.

(a) (2 Punkte) Schreiben Sie eine rekursive, aber *nicht* endrekursive, Funktion `collatz_steps_rec`, die eine natürliche Zahl `n` als Argument erhält und die Anzahl der Schritte zurückgibt, die benötigt werden, um von `n` zur Zahl 1 zu gelangen, indem die obige Vorschrift angewendet wird.

(b) (2 Punkte) Schreiben Sie ein endrekursive Funktion `collatz_steps_tail`, die sich wie `collatz_steps_rec` verhält, aber endrekursiv implementiert ist. Gehen Sie hierbei so vor, wie in der Vorlesung gezeigt.

(c) (2 Punkte) Schreiben Sie eine nicht-rekursive Funktion `collatz_steps_iter`, die sich wie `collatz_steps_tail` verhält, aber iterativ implementiert ist. Gehen Sie hierbei so vor, wie in der Vorlesung gezeigt.

```
>>> collatz_steps_rec(1)
0
>>> collatz_steps_rec(1) == collatz_steps_tail(1) == collatz_steps_iter(1)
```

```
True

>>> collatz_steps_rec(6)
8
>>> collatz_steps_rec(6) == collatz_steps_tail(6) == collatz_steps_iter(6)
True
```

Aufgabe 11.3 (Dictionaries; 6 Punkte; Datei: `dicts.py`)

In dieser Aufgabe definieren Sie zwei generische Funktionen über Dictionaries.

(a) (3 Punkte) Implementieren Sie eine Funktion `reverse_dict`, die ein beliebiges Dictionary als Eingabe erhält und ein neues Dictionary zurückgibt, in dem die Schlüssel und Werte vertauscht sind. Beachten Sie, dass Dictionaries nicht injektiv sind. Deshalb sollen die Werte im neuen Dictionary Sets sein, die alle ursprünglichen Schlüssel enthalten, die auf diesen Wert gezeigt haben.

```
>>> reverse_dict({'a': 1, 'b': 2, 'c': 1})
{1: {'c', 'a'}, 2: {'b'}}
>>> reverse_dict({True: "yes", False: "no"})
{'yes': {True}, 'no': {False}}
```

(b) (3 Punkte) Implementieren Sie die Funktion `filter_dict`, die ein Dictionary und zwei Listen als Eingabe erhält: eine Liste von Schlüsseln und eine Liste von Werten. Die Funktion soll ein neues Dictionary zurückgeben, der nur die Schlüssel-Wert-Paare aus dem ursprünglichen Dictionary enthält, deren Schlüssel *nicht* in der Schlüssel-Liste und deren Werte *nicht* in der Werte-Liste enthalten sind.

```
>>> filter_dict({'a': 1, 'b': 2, 'c': 3}, ['a'], [1, 3])
{'b': 2}
>>> filter_dict({True: "yes", False: "no"}, [False], [])
{True: 'yes'}
```

Aufgabe 11.4 (Erfahrungen; 0 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 Webplattform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht dabei für 7 Stunden 30 Minuten.