

Informatik 2 für Regenerative Energien

Klausur vom 11. Juli 2025: Lösungen

Jörn Loviscach

Versionsstand: 11. Juli 2025, 19:00



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Lösungen von Gemini 2.5 Pro auf Basis des PDFs der Klausur, in dem der Programmcode ausschließlich in Form dreier seitenfüllender Screenshots vorliegt. In der Lösung zu Aufgabe 1 einige Unterstriche escaped, damit TeX durchlief. Die Tabelle in jener Lösung zwecks Seitenumbruch zu longtable gemacht. Handschriftliche Anmerkungen in Rot von Jörn Loviscach.

Aufgabe 1: Fehlerkorrektur

Die 15 Fehler im Programmlisting und ihre Korrekturen sind:

Zeile	Korrektur
10	def messen(self): pass
11	(Eine mit @abstractmethod markierte Methode benötigt einen Rumpf, z.B. pass.)
30	self.zustand = wert
	(Die Variable wert ist der Parameter, nicht self.wert.)
35	wert = self.zustand + änderung
	(Die Zuweisungssyntax war falsch.)
39	wert = 100
	(Das Zuweisungszeichen = fehlte.)
40	self.setze_zustand(wert)
	(Der Aufruf einer eigenen Methode erfordert self.)
53	def __init__(self, name):
	(Der Parameter name, der in Zeile 54 verwendet wird, fehlte in der Signatur.)
55	self.größen = []
	(Das Attribut muss eine Liste sein, damit .append in Zeile 61 funktioniert, nicht None.)
77	continue

78

2. SZ
(diskutierbar)

(*break* würde die Schleife komplett abbrechen, *continue* überspringt nur den aktuellen Durchlauf.)

80 regel = raum.modus[größe]
(Der Zugriff auf einen Dictionary-Wert erfolgt mit eckigen Klammern [], nicht mit runden.)

82 aktueller_wert = sensor.messen()
(Der Aufruf einer Methode erfordert Klammern ().)

90 inkr = -1 if invers else 1
(Es fehlte das Zuweisungszeichen = und die Logik für invers war falsch; es sollte -1 sein.)

94 aktor.ändere_zustand(inkr)
95 (Logikfehler: Um den Wert zu senken (z.B. Jalousie öffnen), muss die Änderung negativ sein; bei invers ist inkr=-1, also muss inkr übergeben werden.)

98 mein_hörsaal = Raum('D2')
(Der Name des Raumes muss als String-Literal übergeben werden, nicht als undefinierte Variable.)

113 automation = Automation()
(Um eine Instanz einer Klasse zu erzeugen, muss der Konstruktor mit () aufgerufen werden.)

114 automation.setze_modus(mein_hörsaal, klausurmodus)
(Die Reihenfolge der Argumente war vertauscht im Vergleich zur Methodendefinition.)

z. 87

steht genau so in Lösung

Aufgabe 2: Programmausgabe

Die ersten drei Zeilen der Ausgabe des korrigierten Programms lauten:

```
LichtD2: 27.0
JalousieD2: 24.0
LichtD2: 31.0
```

Herleitung: Die Endlosschleife in Zeile 116 ruft wiederholt `automation.regle_raum(mein_hoersaal)` auf. Diese Methode durchläuft die zwei konfigurierten Größen ('Helligkeit' und 'Blendstärke').

1. Schleifendurchlauf:

- **Regelung 'Helligkeit':**

- Initialer Zustand von 'LichtD2': 23.
- Der 'Helligkeitssensor' misst konstant 30.
- Die Regel ist>('proportional', 70)'.
- 'diff = 70 - 30 = 40'.
- 'invers' ist 'False'.
- Aufruf: 'aktor.ändere_zustand(0.1 * 40)', also 'ändere_zustand(4)'.
- Neuer Zustand: '23 + 4 = 27'.
- 'setze_zustand' wird aufgerufen und druckt die **erste Zeile: LichtD2: 27.**

- **Regelung 'Blendstärke':**

- Initialer Zustand von 'JalousieD2': 23.
- Der 'Blendsensor' misst konstant 90.
- Die Regel ist('tolerant', 10, 30)'.
- 'aktueller_wert' (90) ist größer als der obere Schwellenwert (30).
- 'invers' ist 'True', also wird 'inkr' zu -1.
- Aufruf: 'aktor.ändere_zustand(-inkr)', also 'ändere_zustand(-(-1))', was 'ändere_zustand(1)' ist.
- Neuer Zustand: '23 + 1 = 24'.
- 'setze_zustand' druckt die **zweite Zeile: JalousieD2: 24.**

2. Schleifendurchlauf:

- **Regelung 'Helligkeit':**

- Der Zustand von 'LichtD2' ist jetzt 27.

- Die Messung (30) und die Regel (70) sind unverändert. 'diff' ist wieder 40.
 - Aufruf: 'ändere_zustand(4)'.
 - Neuer Zustand: '27 + 4 = 31'.
 - 'setze_zustand' druckt die **dritte Zeile: LichtD2: 31**.
- Die Regelung für 'Blendstärke' würde als nächstes 'JalousieD2: 25' ausgeben, usw.

Aufgabe 3: Exception bei fehlendem Modus

Um eine Exception zu werfen, wenn für einen Raum noch kein Modus festgelegt wurde, muss die Methode `regle_raum` in der Klasse `Automation` angepasst werden. Am Anfang der Methode wird geprüft, ob das Attribut `modus` des übergebenen Raums noch `None` ist.

Die Änderung in der Klasse `Automation`:

```
def regle_raum(self, raum):
    if raum.modus is None:
        raise ValueError(f"Fuer Raum '{raum.name}' wurde kein Modus fes

    for element in raum.größen:
        # ... (Rest der Methode bleibt gleich)
```

Aufgabe 4: Messergebnisse merken

Damit sich jeder Sensor alle bisherigen Messergebnisse merkt, sind folgende Änderungen nötig:

1. **Im Konstruktor der Basisklasse Sensor** wird eine leere Liste initialisiert, um die Messwerte zu speichern.
2. **In den messen-Methoden der abgeleiteten Klassen** (Helligkeitssensor, Blendsensor) wird jeder neue Messwert vor der Rückgabe an diese Liste angehängt.

Änderung in Sensor:

```
class Sensor(ABC):
    def __init__(self, name):
        self.name = name
        self.messwerte = [] # Hinzugefügte Zeile

    @abstractmethod
```

```
def messen(self):  
    pass
```

Änderung in Helligkeitssensor:

```
class Helligkeitssensor(Sensor):  
    def messen(self):  
        wert = 30 # Simuliert einen Helligkeitwert  
        self.messwerte.append(wert) # Hinzugefügte Zeile  
        return wert
```

Änderung in Blendsensor:

```
class Blendsensor(Sensor):  
    def messen(self):  
        wert = 90 # Simuliert einen Blendwert  
        self.messwerte.append(wert) # Hinzugefügte Zeile  
        return wert
```

Aufgabe 5: Klasse Aktorgruppe

Die Klasse `Aktorgruppe` kann von `Aktor` abgeleitet werden. Ihr Konstruktor nimmt eine Liste von `Aktor`-Instanzen entgegen. Die Methoden zum Setzen und Ändern des Zustands werden so überschrieben, dass sie den Befehl an alle Aktoren in der Gruppe weiterleiten.

```
class Aktorgruppe(Aktor):
    """
    Eine Gruppe von Aktoren, die immer gemeinsam gesteuert werden.
    """
    def __init__(self, name, aktoren_liste):
        """
        Initialisiert die Gruppe.
        :param name: Name der Aktorgruppe, z.B. 'Jalousien_Suedseite'.
        :param aktoren_liste: Eine Liste von Aktor-Instanzen.
        """
        super().__init__(name)
        self.aktoren = aktoren_liste
        # Der Zustand der Gruppe selbst ist nicht definiert,
        # nur die Zustände der Mitglieder.
        self.zustand = None

    def setze_zustand(self, wert):
        """
        Setzt den Zustand aller Aktoren in der Gruppe auf einen festen
        """
        # Der Print-Befehl aus der Basisklasse wird hier bewusst nicht
        # aufgerufen, da jede untergeordnete Instanz dies selbst tut.
        for aktor in self.aktoren:
            aktor.setze_zustand(wert)

    def ändere_zustand(self, änderung):
        """
        Ändert den Zustand aller Aktoren in der Gruppe um einen relativ
        """
        for aktor in self.aktoren:
            aktor.ändere_zustand(änderung)
```

Aufgabe 6: Zählen der Einstellungen > 90

Um zu zählen, wie oft eine Beleuchtung auf über 90 eingestellt wird, erweitern wir die Klasse um eine **Klassenvariable** als Zähler. Die Methode `setze_zustand` wird überschrieben, um diesen Zähler bei Bedarf zu erhöhen. Eine **Klassenmethode** erlaubt den Abruf des Zählerstands.

```
class Beleuchtung(Aktor):
```

```
# Klassenattribut, das für alle Instanzen von Beleuchtung geteilt w
ueber_90_zaehler = 0
```

```
def setze_zustand(self, wert):
    # Prüfen, ob der neue Wert die Bedingung erfüllt.
    if wert > 90:
        Beleuchtung.ueber_90_zaehler += 1

    # Wichtig: Die Funktionalität der Basisklasse aufrufen.
    super().setze_zustand(wert)
```

nein,
aber
nicht
nötig

```
@classmethod
def get_gesamtzahl_ueber_90(cls):
    """
    Gibt die Gesamtzahl der Einstellungen auf > 90 für alle
    Beleuchtungs-Instanzen zurück.
    """
    return cls.ueber_90_zaehler
```

Aufgabe 7: Typ der Variable akademie

Der Typ der Variable akademie kann mit Pythons typing-Modul präzise beschrieben werden.

- akademie ist ein Dictionary, das Strings auf weitere Dictionaries abbildet.
- Das innere Dictionary bildet Strings auf Werte ab, die entweder eine Liste von Tupeln (String, Integer) oder ein Set von Strings sind.

Formal ausgedrückt mit Typ-Annotationen:

```
from typing import Dict, List, Set, Tuple, Union

# Typ des inneren Dictionaries, das die Daten einer Person enthaelt
PersonenDaten = Dict[str, Union[List[Tuple[str, int]], Set[str]]]

# Vollstaendiger Typ der Variable 'akademie'
AkademieTyp = Dict[str, PersonenDaten]

# Beispielzuweisung
akademie: AkademieTyp = { ... }
```

In einer einzelnen Zeile lautet der Typ:

```
Dict[str, Dict[str, Union[List[Tuple[str, int]],
Set[str]]]]
```

Die alte Art, das zu schreiben!

Aufgabe 8: Variablenwerte nach Codeausführung

Die Ausführung des Codefragments führt zu folgenden Werten für x, y und z:

- **x = 'Prag'**
- **y = 8**
- **z = False**

*Typografie:
Garde
Ausführungszichen
wären schöner!*

Gedankengang und Zwischenschritte:

1. Initialisierung:

- `städte = ['Berlin', 'Paris']`
- `tage = [3, 4]`
- `plan_a['Route']` **verweist auf dasselbe Listenobjekt wie städte.**
- `plan_a['Dauer']` **und** `plan_b['Dauer']` **verweisen beide auf dasselbe Listenobjekt wie tage.**

2. `städte.append('Prag')`

- Die Liste `städte` wird zu `['Berlin', 'Paris', 'Prag']`.
- Da `plan_a['Route']` auf dieselbe Liste zeigt, ändert es sich mit.
- Zustand von `plan_a['Route']`: `['Berlin', 'Paris', 'Prag']`

3. `plan_a['Dauer'][1] += 2`

- `plan_a['Dauer']` ist die Liste `tage`. Der Wert an Index 1 (4) wird um 2 erhöht.
- Die Liste `tage` wird zu `[3, 6]`.
- Da `plan_b['Dauer']` auf dieselbe Liste zeigt, ändert es sich mit.

4. `plan_b['Dauer'][0] -= 1`

- `plan_b['Dauer']` ist die Liste `tage`. Der Wert an Index 0 (3) wird um 1 verringert.
- Die Liste `tage` wird zu `[2, 6]`.

5. Berechnung von x, y, z:

- `x = übersicht[0]['Route'][-1]`: `übersicht[0]` ist `plan_a`. Dessen 'Route' ist `['Berlin', 'Paris', 'Prag']`. Das letzte Element (`[-1]`) ist **'Prag'**.
- `y = sum(übersicht[1]['Dauer'])`: `übersicht[1]` ist `plan_b`. Dessen 'Dauer' ist die modifizierte Liste `tage`, also `[2, 6]`. Die Summe ist `2 + 6 = 8`.

- `z = 'Prag' in plan_b['Route']`: `plan_b['Route']` ist die unveränderte Liste `['Rom', 'Madrid']`. **'Prag' ist in dieser Liste nicht enthalten. Das Ergebnis ist **False**.**