LU03e - Klassen und Fehlerfälle testen

Auf einen Blick
Gute Tests prüfen nicht nur den Normalfall. Sie prüfen auch Grenzwerte, Spezialfälle und ungültige Eingaben. Bei ungültigen Eingaben soll das Programm meistens eine **Exception** auslösen. Mit `pytest.raises` testen Sie, ob das passiert. Klassen testen Sie, indem Sie ein Objekt erzeugen, eine Methode aufrufen und danach den Zustand des Objekts prüfen.

Lernziele

Sie können

- für eine Funktion Normalfälle, Grenzwerte, Spezialfälle und ungültige Eingaben bestimmen,
- mit `pytest.raises` prüfen, ob eine bestimmte Exception ausgelöst wird,
- Methoden einer Klasse nach dem Muster Arrange – Act – Assert testen.

Welche Testfälle braucht es?

Für jede Funktion überlegen Sie sich Testfälle in diesen vier Gruppen. Beispiel: `minmax(numbers)` liefert die kleinste und die grösste Zahl einer Liste.

Gruppe	**Frage**	**Beispiel `minmax`**
Normalfall (positiver Test)	Funktioniert der typische Fall?	`[3, 1, 2]` → `(1, 3)`
Grenzwerte	Was passiert am Rand des gültigen Bereichs?	Liste mit genau einem Element `[5]` → `(5, 5)`
Spezialfälle	Gibt es Werte, die anders behandelt werden könnten?	nur negative Zahlen, gleiche Zahlen, Fliesskommazahlen
Ungültige Eingaben (negativer Test)	Reagiert die Funktion richtig auf Fehler?	leere Liste `[]` → `ValueError`

Bei ungültigen Eingaben unterscheiden wir wie in LU01a:

- **formal falsch:** falscher Datentyp oder falsches Format, zum Beispiel `'abc'` statt einer Zahl,
- **formal korrekt, aber logisch ungültig:** richtiger Datentyp, aber unzulässiger Wert, zum Beispiel ein negativer Betrag.

Systematische Verfahren, um solche Testfälle zu finden (Äquivalenzklassen, Grenzwertanalyse), vertiefen Sie in LU05 und LU06.

Exceptions testen mit pytest.raises

Löst eine Funktion bei ungültigen Eingaben eine Exception aus, ist das ihr **korrektes** Verhalten. Der

Test muss also bestehen, wenn die Exception kommt, und fehlschlagen, wenn sie ausbleibt.

```
import pytest
from minmax import minmax


def test_minmax_empty():
    with pytest.raises(ValueError):
        minmax([])
```

- Der Code im `with`-Block muss die angegebene Exception auslösen. Sonst schlägt der Test fehl mit `DID NOT RAISE`.
- Eine **andere** Exception, zum Beispiel ein `TypeError`, lässt den Test ebenfalls fehlschlagen.
- Schreiben Sie nur die eine Zeile in den `with`-Block, welche die Exception auslösen soll.

Mit `match` prüfen Sie zusätzlich den Text der Fehlermeldung:

```
def test_withdraw_too_much():
    account = BankAccount('Anna', 100.0)
    with pytest.raises(ValueError, match='ungenügend'):
        account.withdraw(100.01)
```

Klassen testen

bank_account.py

```
class BankAccount:
    def __init__(self, owner, balance=0.0):
        self.owner = owner
        self._balance = balance

    @property
    def balance(self):
        return self._balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError('Betrag muss grösser als 0 sein')
        self._balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError('Betrag muss grösser als 0 sein')
        if amount > self._balance:
            raise ValueError('Kontostand ungenügend')
        self._balance -= amount
```

test_bank_account.py

Jeder Test erzeugt ein **eigenes** Objekt. So beeinflussen sich die Tests nicht gegenseitig (Prinzip *Independent*, siehe LU03a).

```
import pytest
from bank_account import BankAccount


def test_new_account_is_empty():
    account = BankAccount('Anna')
    assert account.balance == 0.0


def test_deposit():
    # Arrange
    account = BankAccount('Anna', 100.0)
    # Act
    account.deposit(50.0)
    # Assert
    assert account.balance == 150.0


def test_withdraw_whole_balance():
    account = BankAccount('Anna', 100.0)
    account.withdraw(100.0)
    assert account.balance == 0.0


def test_deposit_negative_amount():
    account = BankAccount('Anna', 100.0)
    with pytest.raises(ValueError):
        account.deposit(-20.0)
    assert account.balance == 100.0


def test_withdraw_too_much():
    account = BankAccount('Anna', 100.0)
    with pytest.raises(ValueError, match='ungenügend'):
        account.withdraw(100.01)
    assert account.balance == 100.0
```

- `test_withdraw_whole_balance` prüft den **Grenzwert**: Den ganzen Kontostand abzuheben ist erlaubt.
- `test_withdraw_too_much` prüft den Wert **knapp daneben**: Ein Rappen mehr ist nicht erlaubt.
- Nach einer Exception prüfen die Tests zusätzlich, dass der Kontostand **unverändert** ist. Eine fehlgeschlagene Buchung darf das Konto nicht verändern.

Wiederholt sich das Arrange in vielen Tests, können Sie es in eine Fixture

auslagern. Das lernen Sie in LU04a.

Kontrollfragen

1. Nennen Sie für die Funktion `average(numbers)` je einen Testfall für Normalfall, Grenzwert, Spezialfall und ungültige Eingabe.
2. Wann schlägt ein Test mit `pytest.raises(ValueError)` fehl?
3. Warum prüft `test_deposit_negative_amount` nach der Exception noch den Kontostand?
4. Weshalb erzeugt jeder Test ein eigenes `BankAccount`-Objekt, statt eines für alle Tests zu verwenden?

M450-LU03, M450-C1I

Marcel Suter

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m450/learningunits/lu03/exceptions

Last update: **2026/11/17 07:58**