LU11c - Pytest

Pytest ist ein Testframework um einzelne Funktionen eines Programms automatisiert zu testen.

Grundlagen

Das fortlaufende Testen deiner Programmfunktionen ist ein wichtiger Schritt, um die korrekte Verarbeitung sicherzustellen. Jeder neu geschriebene oder veränderte Code kann Fehler enthalten. Anstatt jedesmal manuell alle Tests durchzuführen, wollen wir die Tests automatisieren. Dies bedeutet zwar einen zusätzlichen Aufwand, um die Testfunktionen zu schreiben. Diese Zeit sparen wir aber wieder ein, wenn wir diese Tests immer und immer wieder nutzen können.

Für die Programmieraufgaben in den Programmiermodulen sind die Testfunktionen bereits vorgegeben. Daher konzentrieren wir uns darauf, diese Tests und deren Resultate zu verstehen.

Aufbau einer Testfunktion

Die einzelnen Unit Tests werden als Funktionen in Python programmiert. Zum Beispiel:

```
def test_normal():
    result = factorial(7)
    assert result == 5040
```

- Der Funktionsname für den Test muss mit `test` beginnen. Andernfalls erkennt `pytest` die Funktion nicht.
- Wir rufen die zu testende Funktion (im Beispiel `factorial`) mit den Testdaten auf.
- Der Befehl `assert` vergleicht das tatsächliche Resultat mit dem erwarteten Resultat.

Assert

Der Befehl `assert` ist eine spezielle Bedingung, die wir zum Testen und Debuggen von Code verwenden. Falls die Bedingung erfüllt ist, wird `true` zurück gegeben. Sonst wird eine `AssertionError`-Exception geworfen. Wir könnten das gleiche Resultat auch mit `if`/`else` erreichen:

assert	**if / else**

`assert result == 5040`	`if result == 5040:`
	`  return true`
	`else:`
	`  raise AssertionError`

Tests durchführen

Einzelne Tests

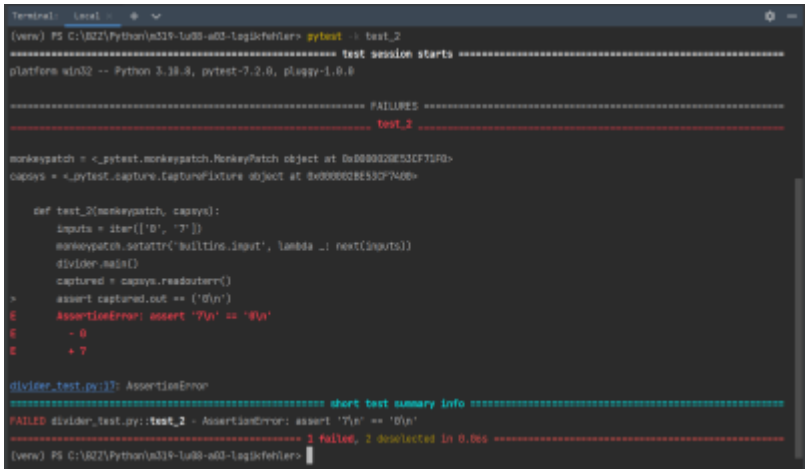
Während der Entwicklung führe ich jede Testfunktion einzeln aus. Dadurch werde ich nicht von einer grossen Anzahl an Fehlermeldungen auf einmal überwältigt.

Um eine einzelne Testfunktion auszuführen, kann ich ...

- a) ... auf das grüne Dreieck-Symbol vor dem Funktionsnamen klicken.
- b) ... im Terminal `pytest -k TESTFUNKTION` eingeben.
  Ersetze *TESTFUNKTION* mit dem Namen der Funktion.

Vollständiger Test

Bevor ich einen Push ins GitHub-Repository mache, führe ich jeweils alle Testfunktionen aus. Dadurch stelle ich sicher, dass meine Änderungen keine unerwarteten Auswirkungen auf andere Programmteile haben.



Testresultate auswerten

Falls ein Test nicht erfolgreich war, musst du die Ausgaben analysieren.

Beispiel 1

```
E       AssertionError: assert '7\n' == '0\n'
E         - 0
E         + 7
```

Hier wurde der Test ausgeführt, aber das Resultat entspricht nicht den Erwartungen.

- Erwartetes Resultat: '0'
- Tatsächliches Resultat: '7'

Übrigens: Das '\n' steht für einen Zeilenumbruch.

Beispiel 2

```
divider_test.py::test_2 FAILED
[100%]
divider_test.py:11 (test_2)
monkeypatch = <_pytest.monkeypatch.MonkeyPatch object at 0x000001B8EC7B08E0>
capsys = <_pytest.capture.CaptureFixture object at 0x000001B8EC7B0AF0>

    def test_2(monkeypatch, capsys):
        inputs = iter(['0', '7'])
        monkeypatch.setattr('builtins.input', lambda _: next(inputs))
>       divider.main()

divider_test.py:15:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _

    def main():
        """
        Ermittelt den grössten gemeinsamen Teiler von zwei Ganzzahlen
        :return: None
        """
        first_number = int(input("Gib die erste Ganzzahl ein > "))
        second_number = int(input("Gib die zweite Ganzzahl ein > "))
        while second_number != 0:
>           foo = second_number / first_number
E           ZeroDivisionError: division by zero

divider.py:9: ZeroDivisionError
```

Bei diesem Beispiel ist das Programm abgestürzt. Eine wichtige Information steckt in der letzten Zeile: `divider.py:9: ZeroDivisionError`:

- Das Modul `divider.py` hat den Absturz verursacht.
- Der Absturz erfolgte auf Zeile 9 des Moduls
- Ein `ZeroDivisionError` wurde ausgelöst.

In einem solchen Fall muss du die Programmlogik genau studieren und evtl. den Test mit Hilfe des Debuggers schrittweise durchführen.

M431-LU11