Programmieren in Python
12. Unit-Testing

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

Wozu Unit-Testing?

Autoritäre Antwort: Weil wir es im Praktikum von euch verlangen.

Etwas bessere Antwort: Weil es die Entwicklung vereinfacht.

- Mehr Vertrauen in Korrektheit des Codes.
- Leichteres Refactoring: Laufen die Tests danach noch durch?
- Garantie, dass einmal gefundene Bugs nicht wieder auftauchen.
- Spezifikation und Dokumentation.

Eigenschaften von Unit-Test

- Isolierte Tests der einzelnen Module
- Test auch von Fehlverhalten (etwa bei unerwarteten Eingaben)
- Regelmäßige automatische Ausführung
- Test des Vertrages, nicht der Algorithmen

Unit-Testing-Frameworks für Python

Für Python existieren mehrere Unit-Testing-Frameworks:

- `py.test`
- `unittest`
- `nose`

Wir verwenden `py.test`, ein Kommandozeilenwerkzeug zum Sammeln, Ausführen und Auswerten automatisierter Tests:

- Evtl. erst `easy_install3` installieren:
  `sudo apt-get install python3-setuptools`.
- Installation über „`easy_install3 -U py`“: Version 1.3.4.

Eigenschaften von py.test

- Keine API – Alles passiert automagisch
- Automatische Erkennung von Tests (Modulen/Klassen/Methoden/Funktionen)
- Parametrisierbare Tests
- Zusicherungen mit der normalen assert-Anweisung
- Zusicherungen von erwarteten Ausnahmen
- Markieren von Tests zum Überspringen oder Fehlschlagen

Beispiel für Test-Modul

square.py

```
def square(x):
    return x*x
```

test_square.py

```
from square import square
def test_square():
    assert square(2) == 4
```

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square PASS
================ 1 passed in 0.01 seconds ================
```


Fehlschlagende Tests (1/2)

test_square.py

```
from square import square

def test_square():
    assert square(2) == 4

def test_square_huh():
    assert square(3) == 8
```


Fehlschlagende Tests (2/2)

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square PASS
test_square.py:6: test_square_huh FAIL

====================== FAILURES ==========================
___________________ test_square_huh _____________________
    def test_square_huh():
>       assert square(3) == 8
E       assert 9 == 8
E        +  where 9 = square(3)

test_square.py:7: AssertionError
========== 1 failed, 1 passed in 0.04 seconds ==========
```

Debug-Informationen (1/2)

Die Standardausgabe wird pro Funktionsaufruf abgefangen und nur bei fehlgeschlagenen Tests angezeigt.

test_square.py

```
from square import square

def test_square():
    print("Debug output of test that passes.")
    assert square(2) == 4

def test_square_huh():
    print("Debug output of test that fails.")
    assert square(3) == 8
```


Debug-Informationen (2/2)

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square PASS
test_square.py:7: test_square_huh FAIL
====================== FAILURES ========================
___________________ test_square_huh ____________________
    def test_square_huh():
>       assert square(3) == 8
E       assert 9 == 8
E        +  where 9 = square(3)

test_square.py:9: AssertionError
------------------- Captured stdout ---------------------
Debug output of test that fails.
=========== 1 failed, 1 passed in 0.04 seconds ==========
```


Mehrere Tests in einer Funktion: Generative Tests (1/6)

Unelegante Lösung:

test_square.py

```
from square import square

def test_square():
    assert square(1) == 1
    assert square(2) == 4
    assert square(3) == 8
    assert square(4) == 16
```


Mehrere Tests in einer Funktion: Generative Tests (2/6)

Nur ein fehlgeschlagener Test für mehrere Assertions.

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square FAIL
====================== FAILURES ========================
___________________ test_square ________________________
    def test_square():
        assert square(1) == 1
        assert square(2) == 4
>       assert square(3) == 8
E       assert 9 == 8
E        +  where 9 = square(3)

test_square.py:6: AssertionError
================ 1 failed in 0.04 seconds ===============
```

Mehrere Tests in einer Funktion: Generative Tests (3/6)

Elegantere Lösung:

test_square.py

```
from square import square

def test_square():
    expected = [(1,1), (2,4), (3,8), (4,16)]
    for x, e in expected:
        assert square(x) == e
```


Mehrere Tests in einer Funktion: Generative Tests (4/6)

Immer noch nur ein fehlgeschlagener Test für mehrere Assertions.

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square FAIL
====================== FAILURES ========================
____________________ test_square ________________________
    def test_square():
        expected = [(1,1), (2,4), (3,8), (4,16)]
        for x, e in expected:
>           assert square(x) == e
E           assert 9 == 8
E            +  where 9 = square(3)

test_square.py:6: AssertionError
================ 1 failed in 0.04 seconds ===============
```


Mehrere Tests in einer Funktion: Generative Tests (5/6)

Noch elegantere Lösung:

test_square.py

```
from square import square

def expected_result(x, e):
    assert square(x) == e

def test_square():
    expected = [(1,1), (2,4), (3,8), (4,16)]
    for x, e in expected:
        yield expected_result, x, e
```


Mehrere Tests in einer Funktion: Generative Tests (6/6)

Shell

```
# py.test -v test_square.py

================== test session starts ==================
test path 1: test_square.py
test_square.py:3: test_square[0] PASS
test_square.py:3: test_square[1] PASS
test_square.py:3: test_square[2] FAIL
test_square.py:3: test_square[3] PASS
====================== FAILURES ========================
___________________ test_square[2] ______________________
x = 3, e = 8
    def expected_result(x, e):
>       assert square(x) == e
E       assert 9 == 8
E        +  where 9 = square(3)

test_square.py:4: AssertionError
=========== 1 failed, 3 passed in 0.05 seconds ==========
```

Assertions

Randbemerkung: Es können beliebige Aussagen mit `assert` zugesichert/getestet werden, nicht nur Werte-Gleichheit.

- `assert x == y`
- `assert x != y`
- `assert x in [y1, ..., yn]`
- `assert pred(x)`
- `...`

Erwartete Ausnahmen (1/2)

Erwartete Ausnahmen/Fehler können getestet werden.

exception.py

```
def produce_index_error(number):
    if number < 10:
        return number
    raise IndexError
```

test_exception.py

```
import py
from exception import produce_index_error

def test_for_exception():
    with py.test.raises(IndexError):
        produce_index_error(20)
```


Erwartete Ausnahmen (2/2)

Shell

```
# py.test -v test_exception.py

================== test session starts ==================
test path 1: test_exception.py
test_exception.py:4: test_for_exception PASS
================ 1 passed in 0.01 seconds ===============
```

Test würde fehlschlagen, wenn erwarteter `IndexError` nicht geworfen würde.

Erwartetes Fehlschlagen von Tests (1/2)

Erwartetes Fehlschlagen von Tests kann deklariert werden.

test_square.py

```
import py
from square import square

def test_square():
    assert square(2) == 4

def test_square_huh():
    py.test.xfail("Aware of the bug -- fix delayed.")
    assert square(3) == 8
```

Erwartetes Fehlschlagen von Tests (2/2)

Shell

```
# py.test -v test_square.py

================== test session starts ====================
test path 1: test_square.py
test_square.py:4: test_square PASS
test_square.py:7: test_square_huh xfail

=========== 1 passed, 1 xfailed in 0.04 seconds ==========
```


Überspringen von Tests (1/2)

Test können gezielt übersprungen werden.

test_square.py

```
import py
from square import square

def test_square():
    assert square(2) == 4

def test_square_huh():
    py.test.skip("Aware of the bug -- fix delayed.")
    assert square(3) == 8
```


Überspringen von Tests (2/2)

Shell

```
# py.test -v test_square.py

================== test session starts ====================
test path 1: test_square.py
test_square.py:4: test_square PASS
test_square.py:7: test_square_huh SKIP

=========== 1 passed, 1 skipped in 0.04 seconds ==========
```


Kommandozeilenoptionen (Auswahl)

```
Usage: py.test [options] [file_or_dir] [file_or_dir] [...]

Options:
  -h, --help       show this help message and exit
  -x, --exitfirst  exit instantly on first error or
                   failed test.
  --pdb            start the interactive Python
                   debugger on errors.
  -s               disable catching of stdout/stderr
                   during test run.
  -v, --verbose    increase verbosity.
```

Automatische Erkennung von Tests

- py.test läuft rekursiv durch das Dateisystem.
- Es sucht nach Dateien mit Namen test_<something>.py und <something>_test.py.
- In diesen Dateien sucht und verwendet es alle
  - Funktionen/Methoden mit Namen test_<something> und
  - Klassen mit Namen Test<Something>.

Klassen und Module (1/3)

test_with_classes.py

```
class TestSomething(object):
    def test_something(self):
        assert True
    def other_method(self):
        assert False

class OtherClass(object):
    def test_something(self):
        assert False
    def other_method(self):
        assert False
```


Klassen und Module (2/3)

Damit Methode als Test verwendet wird, muss der Klassenname Test<Something> und der Methodenname test_<something> sein.

Shell

```
# py.test -v test_with_classes.py

================== test session starts ==================
test path 1: test_with_classes.py
test_with_classes.py:2: TestSomething.test_something PASS

================ 1 passed in 0.01 seconds ===============
```


Klassen und Module (3/3)
Test-Fixture-Management

Wenn Tests unabhängig voneinander sein sollen, muss oft vor jedem individuellen Test ein bestimmter, für alle Tests gleicher, Zustand hergestellt und ggf. danach wieder aufgeräumt werden.

Entsprechende Methoden müssen für py.test folgende Namen haben:

- Pro test_<something>-Methode:
  - setup_method bzw.
  - teardown_method
- Pro Test<Something>-Klasse:
  - setup_class bzw.
  - teardown_class
- Pro Test-Modul/-Datei:
  - setup_module bzw.
  - teardown_module