

Pythonskripte testen: wie und warum?

Gert-Ludwig Ingold

```
*****  
File "example", line 1, in example  
Failed example:  
  1+1  
Expected:  
  1  
Got:  
  2  
*****  
1 items had failures:  
  1 of  1 in example  
***Test Failed*** 1 failures.
```

 `git clone https://github.com/gertingold/lit2016.git`

Wer testet seine Programme?

- 😊 Programme werden praktisch immer bei der Programmentwicklung durch Vergleich mit dem erwarteten Verhalten getestet.
- 😞 Das passiert oft sehr informell und kaum reproduzierbar.

Einige Überlegungen zum Testen

- ▶ Tests dokumentieren überprüfte Funktionalität von Code

Beispiel: https://hg.python.org/cpython/file/tip/Lib/test/test_math.py

```
...
def testSqrt(self):
    self.assertRaises(TypeError, math.sqrt)
    self.fctest('sqrt(0)', math.sqrt(0), 0)
    self.fctest('sqrt(1)', math.sqrt(1), 1)
    self.fctest('sqrt(4)', math.sqrt(4), 2)
    self.assertEqual(math.sqrt(INF), INF)
    self.assertRaises(ValueError, math.sqrt, NINF)
    self.assertTrue(math.isnan(math.sqrt(NAN)))
...
```

Einige Überlegungen zum Testen

- ▶ Tests dokumentieren überprüfte Funktionalität von Code
- ▶ Tests sollten jederzeit erlauben, rasch die Funktionalität des Codes zu überprüfen
 - ▶ vor dem Einchecken ins Versionskontrollsystem
 - ▶ beim Refactoring
 - ▶ ...
- ▶ Testsuites sind (praktisch) nie vollständig
 - ▶ zu jedem gefundenen Fehler einen Test schreiben, damit der Fehler nie wieder auftreten kann
 - ▶ zu jedem neuen Feature Tests schreiben
 - ▶ ggf. immer wieder Tests ergänzen
- ▶ Tests sollten den Code (nahezu) vollständig ausführen
siehe `coverage.py` (coverage.readthedocs.org)
- ▶ gut getesteter Code, z.B. aus der Python Standard Library, muss nicht noch einmal getestet werden

Unittests

- ▶ **Unittest:** Test von elementaren Einheiten eines Programms, z.B. Funktionen oder Methoden
- ▶ Unittests erleichtern die Lokalisierung von Fehlern
- ▶ Unittests unterstützen eine modularisierte Programmierweise und verbessern damit den Code
- ▶ das Zusammenspiel von Komponenten muss separat getestet werden (Integrationstests, Systemtests)
- ▶ das unittest-Modul von Python kann auch für Integrationstests verwendet werden

Zwei Module aus der Python Standard Library:

- ▶ `doctest`
- ▶ `unittest`

Zwei Module aus der Python Standard Library:

- ▶ doctest
- ▶ unittest

Was sind Docstrings?

```
def welcome(name):  
    """  
    be nice and greet somebody  
    name: name of the person  
  
    """  
    return 'Hallo {}!'.format(name)
```

in der Python-Shell:

```
>>> help(welcome)
```

```
Help on function welcome in module __main__:
```

```
welcome(name)  
    be nice and greet somebody  
    name: name of the person
```

Ein Gruß an Guido van Rossum

```
def welcome(name):  
    """  
    be nice and greet somebody  
    name: name of the person  
  
    >>> welcome('Guido')  
    'Hallo Guido!'  
  
    """  
    return 'Hallo {}!'.format(name)
```

-
- ▶ die Benutzung der Funktion wird dokumentiert
 - ▶ dabei wird die Syntax der Python-Shell verwendet
 - >>> Eingabe-Prompt
 - ... Fortsetzungszeile
 - ▶ nach der Ausgabe folgt eine Leerzeile oder der nächste Prompt
 - ▶ es lässt sich aber auch die korrekte Funktionsweise testen

Ein erster Test

Die Funktion sei in der Datei `example1_v2.py` definiert:

```
$ python -m doctest example1_v2.py  
$
```

- ▶ »keine Neuigkeiten sind gute Neuigkeiten«
oder:
erfolgt keine Ausgabe, so wurden alle vorhandenen Tests fehlerfrei ausgeführt, sofern sie nicht unterdrückt wurden
- ▶ es wird das Modul `doctest` aus der Python-Standardbibliothek verwendet
ausführliche Dokumentation unter:
<http://docs.python.org/library/doctest.html>
- ▶ mit der Option `-v` wird `doctest` gesprächiger

Und noch einmal mit mehr Details

```
$ python -m doctest -v example1_v2.py
Trying:
    welcome('Guido')
Expecting:
    'Hallo Guido!'
ok
1 items had no tests:
    example1_v2
1 items passed all tests:
   1 tests in example1_v2.welcome
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Ein erster Fehler ...

```
def welcome(name):  
    """  
    be nice and greet somebody  
    name: name of the person  
  
    >>> welcome('Guido')  
    'Hello Guido!'  
  
    """  
    return 'Hallo {}!'.format(name)
```

... und das Ergebnis

```
$ python -m doctest example1_v3.py
*****
File "example1_v3.py", line 6, in example1_v3.welcome
Failed example:
    welcome('Guido')
Expected:
    'Hello Guido!'
Got:
    'Hallo Guido!'
*****
1 items had failures:
  1 of  1 in example1_v3.welcome
***Test Failed*** 1 failures.
```

- ▶ bei Fehlern werden Details auch ohne die Option -v ausgegeben

Erst die Tests, dann das Programmieren

Test-driven development (TDD):

Formuliere erst die Tests und entwickle dann das Programm bis alle Tests fehlerfrei ausgeführt werden.

Wunschliste als Tests

```
def welcome(name):  
    """  
    be nice and greet somebody  
    name: name of the person  
  
    >>> welcome()  
    'Hello!'  
  
    >>> welcome(lang='de')  
    'Hallo!'  
  
    >>> welcome('Guido')  
    'Hello Guido!'  
  
    """  
    return 'Hallo {}!'.format(name)
```

Fehler, die es zu beseitigen gilt

```
*****
File "example1_v4.py", line 6, in example1_v4.welcome
Failed example:
  welcome()
Exception raised:
  Traceback (most recent call last):
    File "python3.5/doctest.py", line 1320, in __run
      compileflags, 1), test.globs)
    File "<doctest example1_v4.welcome[0]>", line 1, in <module>
      welcome()
  TypeError: welcome() missing 1 required positional argument: 'name'
*****
File "example1_v4.py", line 9, in example1_v4.welcome
Failed example:
  welcome(lang='de')
Exception raised:
  Traceback (most recent call last):
    File "python3.5/doctest.py", line 1320, in __run
      compileflags, 1), test.globs)
    File "<doctest example1_v4.welcome[1]>", line 1, in <module>
      welcome(lang='de')
  TypeError: welcome() got an unexpected keyword argument 'lang'
*****
File "example1_v4.py", line 12, in example1_v4.welcome
Failed example:
  welcome('Guido')
Expected:
  'Hello Guido!'
Got:
  'Hallo Guido!'
*****
1 items had failures:
  3 of  3 in example1_v4.welcome
***Test Failed*** 3 failures.
```

Ausnahmen sind manchmal gewollt

```
def welcome(name='', lang='en'):
    """
    be nice and greet somebody
    name: name of the person, may be empty
    lang: two character language code

    >>> welcome()
    'Hello!'

    >>> welcome(lang='de')
    'Hallo!'

    >>> welcome('Guido')
    'Hello Guido!'

    """
    greetings = {'de': 'Hallo',
                 'en': 'Hello',
                 'fr': 'Bonjour'}

    try:
        greeting = greetings[lang]
    except KeyError:
        errmsg = 'unknown language: {}'.format(lang)
        raise ValueError(errmsg)

    if name:
        greeting = ' '.join([greeting, name])
    return greeting+'!'
```

- ▶ eine nicht implementierte Sprache soll zu einem `ValueError` führen

Behandlung von Ausnahmen

- ▶ Problem: die Ausgabe ist häufig komplex

```
Traceback (most recent call last):
```

```
File "example1_v6.py", line 25, in welcome  
    greeting = greetings[lang]
```

```
KeyError: 'nl'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
```

```
File "example1_v6.py", line 33, in <module>  
    welcome('Guido', 'nl')
```

```
File "example1_v6.py", line 28, in welcome  
    raise ValueError(errmsg)
```

```
ValueError: unknown language: nl
```

- ▶ als Doctest genügt aber:

```
>>> welcome('Guido', 'nl')
```

```
Traceback (most recent call last):
```

```
ValueError: unknown language: nl
```

Direktiven für doctest

- ▶ Platzhalter für beliebige Ausgabe

```
>>> welcome('Guido', 'nl') # doctest: +ELLIPSIS
Traceback (most recent call last):
ValueError: ...
```

- ▶ der Test soll vorläufig nicht durchgeführt werden

```
>>> welcome('Guido', 'nl') # doctest: +SKIP
'Goedendag Guido!'
```

- ▶ siehe die Dokumentation für weitere Direktiven:
<http://docs.python.org/library/doctest.html>
- ▶ interessant ist z.B. noch +NORMALIZE_WHITESPACE

Doctests in beliebigem Text

Doctests sind nicht auf die Verwendung Docstrings begrenzt, sondern können in beliebige Textdateien eingebettet und dort getestet werden.

example2.txt:

Eine einfache Verzweigung in Python:

```
>>> x = 1
>>> if x < 0:
...     print('x ist negativ')
... else:
...     print('x ist nicht negativ')
x ist nicht negativ
```

Am Ende des Tests muss sich eine Leerzeile befinden.

```
$ python -m doctest -v example2.txt
```

```
Trying:
```

```
    x = 1
```

```
Expecting nothing
```

```
ok
```

```
Trying:
```

```
    if x < 0:
```

```
        print('x ist negativ')
```

```
    else:
```

```
        print('x ist nicht negativ')
```

```
Expecting:
```

```
    x ist nicht negativ
```

```
ok
```

```
1 items passed all tests:
```

```
    2 tests in example2.txt
```

```
2 tests in 1 items.
```

```
2 passed and 0 failed.
```

```
Test passed.
```

Vor- und Nachteile von Doctests

- 😊 leicht zu schreiben
- 😊 unterstützt die Dokumentation durch Beschreibung des Benutzerinterfaces
- 😊 lässt sich – im Gegensatz zum Rest der Dokumentation – leicht auf Korrektheit überprüfen
- 😊 kann zum Testen von Code in jeder Art von Text verwendet werden
- 😞 nicht gut für aufwändigere Testsuiten geeignet, zumindest nicht in Docstrings
- 😞 nicht für alle Testszzenarien geeignet, z.B. Tests von numerischen Codes bei denen das Ergebnis typischerweise nur näherungsweise korrekt ist

Zwei Module aus der Python Standard Library:

- ▶ `doctest`
- ▶ `unittest`

Zwei Module aus der Python Standard Library:

- ▶ doctest
- ▶ unittest

Pascalsches Dreieck

```
def pascal(n):  
    x = 1  
    yield x  
    for k in range(n):  
        x = x*(n-k)//(k+1)  
        yield x  
  
if __name__ == '__main__':  
    for n in range(7):  
        line = ' '.join(map(lambda x: '{:2}'.format(x), pascal(n)))  
        print(str(n)+line.center(25))
```

```
$ python pascal_v1.py
```

```
0          1  
1         1 1  
2        1 2 1  
3       1 3 3 1  
4      1 4 6 4 1  
5     1 5 10 10 5 1  
6    1 6 15 20 15 6 1
```

Erste Unittests

test_pascal_v1.py:

```
from unittest import main, TestCase
from pascal_v1 import pascal

class TestExplicit(TestCase):
    def test_n0(self):
        self.assertEqual(list(pascal(0)), [1])

    def test_n1(self):
        self.assertEqual(list(pascal(1)), [1, 1])

    def test_n5(self):
        self.assertEqual(list(pascal(5)), [1, 5, 10, 10, 5, 1])

if __name__ == '__main__':
    main()
```

```
$ python test_pascal_v1.py
```

```
...
```

```
-----
Ran 3 tests in 0.000s
```

```
OK
```

- ▶ die Namen von Test-Methoden müssen mit test beginnen

Ein Fehler ...

test_pascal_v2.py:

```
from unittest import main, TestCase
from pascal_v1 import pascal
```

```
class TestExplicit(TestCase):
    def test_n0(self):
        self.assertEqual(list(pascal(0)), [1])

    def test_n1(self):
        self.assertEqual(list(pascal(1)), [1, 1])

    def test_n5(self):
        self.assertEqual(list(pascal(5)), [1, 4, 6, 4, 1])

if __name__ == '__main__':
    main()
```

- ▶ Der Fehler wurde hier in den Test eingebaut, er könnte aber genauso gut im Programm stecken.

... und das Ergebnis

```
$ python test_pascal_v2.py
..F
```

```
=====
FAIL: test_n5 (__main__.TestExplicit)
```

```
-----
Traceback (most recent call last):
```

```
  File "test_pascal_v2.py", line 12, in test_n5
```

```
    self.assertEqual(list(pascal(5)), [1, 4, 6, 4, 1])
```

```
AssertionError: Lists differ: [1, 5, 10, 10, 5, 1] != [1, 4, 6, 4, 1]
```

```
First differing element 1:
```

```
5
```

```
4
```

```
First list contains 1 additional elements.
```

```
First extra element 5:
```

```
1
```

```
- [1, 5, 10, 10, 5, 1]
```

```
+ [1, 4, 6, 4, 1]
```

```
-----
Ran 3 tests in 0.001s
```

```
FAILED (failures=1)
```

Ein erwarteter Fehler

Auszug aus test_pascal_v3.py:

```
from unittest import expectedFailure, main, TestCase
[...]  
class TestExplicit(TestCase):  
[...]  
    @expectedFailure  
    def test_n5(self):  
        self.assertEqual(list(pascal(5)), [1, 4, 6, 4, 1])
```

```
$ python test_pascal_v3.py
```

```
..X
```

```
-----  
Ran 3 tests in 0.001s
```

```
OK (expected failures=1)
```

Was tun bei größeren Argumenten?

bei großen Argumenten ist es nicht mehr sinnvoll, gegen das explizite Resultat zu testen → andersartige Tests sind erforderlich

binomische Formeln:

$$(a + b)^2 = 1 \cdot a^2 + 2 \cdot ab + 1 \cdot b^2$$

$$(a - b)^2 = 1 \cdot a^2 - 2 \cdot ab + 1 \cdot b^2$$

Die Koeffizienten sind die Einträge bzw. die Einträge mit alternierendem Vorzeichen aus dem pascalschen Dreieck für $n = 2$.

Für $a = b = 1$:

$$1 + 2 + 1 = 2^2 \quad \text{allgemein: } 2^n$$

$$1 - 2 + 1 = 0 \quad \text{gilt für alle } n$$

weitere Möglichkeit:

Überprüfe, ob sich aufeinanderfolgende Zeilen des pascalschen Dreiecks durch Addition von benachbarten Elementen erzeugen lassen.

Implementation der Tests (I)

[...]

```
class TestSums(TestCase):
    def test_sum(self):
        for n in (10, 100, 1000, 10000):
            self.assertEqual(sum(pascal(n)), 2**n)

    def test_alternate_sum(self):
        for n in (10, 100, 1000, 10000):
            self.assertEqual(sum(alternate(pascal(n))), 0)
```

[...]

```
def alternate(g):
    sign = 1
    for elem in g:
        yield sign*elem
        sign = -sign
```

[...]

Implementation der Tests (II)

```
from itertools import chain
```

```
[...]
```

```
class TestAdjacent(TestCase):
```

```
    def test_generate_next_line(self):
```

```
        for n in (10, 100, 1000, 10000):
```

```
            expected = [a+b for a, b
```

```
                        in zip(chain(zero(), pascal(n)),
```

```
                              chain(pascal(n), zero()))]
```

```
            result = list(pascal(n+1))
```

```
            self.assertEqual(result, expected)
```

```
[...]
```

```
def zero():
```

```
    yield 0
```

```
[...]
```

Testen auf Ausnahmen

Auszug aus pascal_v2.py:

```
def pascal(n):  
    if n < 0:  
        raise ValueError('n may not be negative')  
    x = 1  
    yield x
```

Auszug aus test_pascal_v5.py:

```
class TestParameters(TestCase):  
    def test_negative_int(self):  
        with self.assertRaises(ValueError):  
            next(pascal(-1))
```

- ▶ `assertRaises` kann auch außerhalb eines Kontexts benutzt werden, ist dann aber weniger übersichtlich
- ▶ der Generator beginnt erst mit der Ausführung des Codes, wenn ein Rückgabewert angefordert wurde

Erweiterung auf Floats

Unsere pascal-Methode kann leicht auf Float-Argumente angepasst werden.

Beispiel:

$$\sqrt[3]{1+x} = 1 + \frac{1}{3}x - \frac{1}{9}x^2 + \frac{5}{81}x^3 + \dots$$

pascal_v3.py:

```
def pascal(n):  
    x = 1  
    yield x  
    k = 0  
    while n-k != 0:  
        x = x*(n-k)/(k+1)  
        k = k+1  
        yield x
```

Wenn n keine nichtnegative ganze Zahl ist, gibt der Generator potentiell unendlich viele Werte zurück.

Testen mit Floats

Auszug aus test_pascal_v6.py:

```
from unittest import main, skip, TestCase
from pascal_v3 import pascal
```

[...]

```
class TestFractional(TestCase):
    def test_one_third(self):
        p = pascal(1/3)
        result = [next(p) for _ in range(4)]
        expected = [1, 1/3, -1/9, 5/81]
        self.assertEqual(result, expected)
```

[...]

```
class TestParameters(TestCase):
    @skip('only for integer version')
    def test_negative_int(self):
        with self.assertRaises(ValueError):
            next(pascal(-1))
```

- ▶ unpassende Tests können mit dem skip-Dekorator deaktiviert werden

Achtung, Rundungsfehler!

```
$ python test_pascal_v6.py
s...Fsss
=====
FAIL: test_one_third (__main__.TestFractional)
-----
Traceback (most recent call last):
  File "test_pascal_v6.py", line 20, in test_one_third
    self.assertEqual(result, expected)
AssertionError: Lists differ: [1, 0.3333333333333333, -0.1111111111111112,
0.0617283950617284] != [1, 0.3333333333333333, -0.1111111111111111,
0.06172839506172839]

First differing element 2:
-0.1111111111111112
-0.1111111111111111

- [1, 0.3333333333333333, -0.1111111111111112, 0.0617283950617284]
?                                     -                                     ^

+ [1, 0.3333333333333333, -0.1111111111111111, 0.06172839506172839]
?                                     -                                     ^^

-----
Ran 8 tests in 0.001s

FAILED (failures=1, skipped=4)
```

Toleranz bei Floats

- ▶ Verwendung von `math.isclose` (ab Python 3.5)
- ▶ `self.assertAlmostEqual`
- ▶ für Listen und NumPy-Arrays:
`numpy.testing.assert_allclose`
damit lässt sich auch die Toleranz gut einstellen

Auszug aus `test_pascal_v7.py`:

```
from numpy.testing import assert_allclose
```

```
[...]
```

```
class TestFractional(TestCase):
```

```
    def test_one_third(self):
```

```
        p = pascal(1/3)
```

```
        result = [next(p) for _ in range(4)]
```

```
        expected = [1, 1/3, -1/9, 5/81]
```

```
        assert_allclose(result, expected, rtol=1e-10)
```

Weiterführende Themen

- ▶ unittest: Methoden setUp und tearDown, um vorbereitenden und abschließenden Code für Tests zu definieren
- ▶ unittest.mock: ersetzt benötigte Codeteile indem es eine definierte Funktionalität bereitstellt
- ▶ weitere Testwerkzeuge:
wiki.python.org/moin/PythonTestingToolsTaxonomy
z.B. nose, coverage