Automatisiertes Testen mit dem Doctest-Modul

Chemnitzer Linuxtage 2009

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Chemnitz, Germany, 2009-03-15

Überblick

- Unittests
- Automatisiertes Testen
- Testgetriebene Entwicklung
- Tests in Docstrings
- Tests in Textdateien
- Tipps und Tricks
- Test-Infrastruktur
- Zusammenfassung

Unittests

- Unittests testen einen Teil (= Unit) des Codes, zum Beispiel eine Funktion, eine Methode, eine Klasse oder ein Modul
- Unittests sind keine Funktionstests (Funktionstests testen die Funktionalität eines Programms aus Anwendersicht)

Automatisiertes Testen

Warum automatisiert testen?

- Wiederholbarkeit
- Schnelle Ausführung vieler Tests
- Hilfe bei der Refaktorierung
  (= Umstellung von Code ohne Änderung der Funktionalität)
- Dokumentation

Testgetriebene Entwicklung

- Erst den Testcode schreiben, dann den zu testenden Code
  Vorteil: Schnittstelle sauberer, das heißt, weniger von einer bestimmten Implementierung abhängig
- Entsprechend: bei Erweiterungen des bestehenden Codes zuerst Tests anpassen
- Bei Programmfehlern:
  - erst einen Test schreiben, der den Fehler nachweist,
  - dann testen,
  - dann das Programm verbessern,
  - dann wieder testen

  Vorteil: es ist besser erkennbar, dass der Test den Fehler nachweist bzw. dass er nach der Code-Änderung behoben ist

Testgetriebene Entwicklung ist zuerst ungewohnt, macht aber Spaß, wenn man sich erstmal dran gewöhnt hat :-)

Beispielcode für einen Unittest

hallo.py

```
def gruess(name):
    """Gib einen Gruss an ‘name‘ zurueck."""
    return u"Hallo %s" % name
```

Test mit dem unittest-Modul

Code

unittest_hallo.py

```
import unittest
import hallo

class TestHallo(unittest.TestCase):
    def test_gruess(self):
        ergebnis = hallo.gruess(u"Tim")
        erwartet = u"Hallo Tim"
        self.assertEqual(ergebnis, erwartet)

unittest.main()
```

Test mit unittest

Ausführung

```
~/f/projekte/clt2009$ python unittest_hallo.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
~/f/projekte/clt2009$
```

Test mit doctest

Code

hallo.py

```
import doctest

def gruess(name):
    """Gib einen Gruss an ‘name‘ zurueck.

    >>> gruess(u"Tim")
    u’Hallo Tim’
    """
    return u"Hallo %s" % name

doctest.testmod()
```

Test mit doctest

Ausführung

```
~/f/projekte/clt2009$ python hallo.py
~/f/projekte/clt2009$
```

Test mit doctest

Ausführung mit -v

```
~/f/projekte/clt2009$ python hallo.py -v
Trying:
    gruess(u"Tim")
Expecting:
    u'Hallo Tim'
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.gruess
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
~/f/projekte/clt2009$
```

Doctest bei fehlerhaftem Code

Code

hallo.py

```
import doctest

def gruess(name):
    """Gib einen Gruss an ‘name‘ zurueck.

    >>> gruess(u"Tim")
    u’Hallo Tim’
    """
    return u"Hallo, %s" % name

doctest.testmod()
```

Doctest bei fehlerhaftem Code

Ausführung

```
~/f/projekte/clt2009$ python hallo.py
**********************************************************************
File "hallo.py", line 6, in __main__.gruess
Failed example:
    gruess(u"Tim")
Expected:
    u'Hallo Tim'
Got:
    u'Hallo, Tim'
**********************************************************************
1 items had failures:
   1 of   1 in __main__.gruess
***Test Failed*** 1 failures.
~/f/projekte/clt2009$
```

Zeichenketten

Zeichenketten in der Ausgabe müssen genau so wie im Interpreter dargestellt werden:

```
>>> "Test"
'Test'

>>> u"Test" + ' mit Doctest'
u'Test mit Doctest'

>>> """Stefan's falsches Apostroph"""
"Stefan's falsches Apostroph"
```

Test auf Exceptions

- Eingerückte Punkte ersetzen den Aufrufstack
- Die Punkte haben nichts mit der ELLIPSIS-Option (siehe unten) zu tun
- Sowohl der Name als auch der Text dahinter müssen mit denen der tatsächlich ausgelösten Exception übereinstimmen

```
>>> vortragsinfo(u'Pflege von Felsenpinguinen')
Traceback (most recent call last):
    ...
CLTError: "Pflege von Felsenpinguinen" fällt aus
```

Tests in Textdateien

- Abwechselnd erklärender Text und Tests
- „Literate Testing“
- Beliebiges Markup des erklärenden Textes (zum Beispiel reStructuredText)
- Zu testende Codeblöcke einrücken
- Ausführung der Tests mit `doctest.testfile(dateiname)`

Tests in einer Textdatei

test_hallo.txt

```
Das Modul ‘hallo.py‘
====================

Das Modul ‘hallo‘ enthält die Methode ‘gruess‘, die
einem Menschen einen freundlichen Gruß entbietet. Der
Name der Person wird als Zeichenketten-Argument übergeben:

    >>> import hallo
    >>> hallo.gruess("Tim")
    u’Hallo Tim’

Das Argument kann auch ein Unicode-String sein:

    >>> hallo.gruess(u"Mike")
    u’Hallo Mike’
```

Tests in einer Textdatei

Ausführung

```
~/f/projekte/clt2009$ python \
    -c "import doctest; \\
        doctest.testfile('test_hallo.txt')"
~/f/projekte/clt2009$
```

Für mehrere Dateien (gegebenenfalls im Makefile):

```
~/f/projekte/clt2009$ find testverzeichnis \
    -name 'test_*.txt' \
    -exec python -c "import doctest; \\
                     doctest.testfile('{}')" \;
```

Tipps und Tricks
ELLIPSIS

Problem: Ein Teil der Ausgabe ändert sich bei jeder Ausführung

```
>>> class C(object): pass
...
>>> C()
<__main__.C object at 0xb7ce61ec>
```

Lösung: ELLIPSIS-Option

```
>>> C()  #doctest: +ELLIPSIS
<__main__.C object at 0x...>
```

Tipps und Tricks
<BLANKLINE>

Problem: Eine Leerzeile signalisiert das Ende der Ausgabe
und verhindert dadurch den Vergleich von Ausgaben
mit enthaltenen Leerzeilen

```
>>> mit_leerzeile()  # funktioniert so nicht
eins

drei
```

Lösung: <BLANKLINE> statt Leerzeile

```
>>> mit_leerzeile()
eins
<BLANKLINE>
drei
```

Tipps und Tricks
IGNORE_EXCEPTION_DETAIL

Problem: Verschiedene Python-Versionen verwenden unterschiedliche Texte nach dem Namen einer Exception

```
>>> list(3)  # Python 2.5
Traceback (most recent call last):
    ...
TypeError: 'int' object is not iterable
```

(bei Python 2.4: „iteration over non-sequence“)

Lösung: Option IGNORE_EXCEPTION_DETAIL

```
>>> list(3)  #doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
    ...
TypeError: 'int' object is not iterable
```

Tipps und Tricks
NORMALIZE_WHITESPACE

Problem: Formatierte Ausgaben sind übersichtlicher, lassen aber den Test fehlschlagen

```
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Lösung: Option NORMALIZE_WHITESPACE

```
>>> range(20)  #doctest: +NORMALIZE_WHITESPACE
[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Tipps und Tricks

Dictionaries

Problem: Die Reihenfolge von Dictionary-Elementen kann wechseln

```
>>> vorwahlen
{'Hannover': '0511', 'Chemnitz': '0371'}
```

Lösung: Dictionaries vergleichen oder Bestandteil prüfen

```
>>> vorwahlen == dict(Chemnitz='0371', Hannover='0511')
True
```

```
>>> vorwahlen['Hannover']
'0511'
```

```
>>> 'Hannover' in vorwahlen
True
```

Tipps und Tricks

Zahlen vergleichen

Problem: Die Zahlendarstellung hängt ab von Betriebssystem, C-Bibliothek, Python-Version etc.

```
>>> 1./13
0.076923076923076927
```

Lösung: Zahlen in Zeichenketten konvertieren

```
>>> "%.8f" % (1./13)
'0.07692308'
```

```
>>> print round(1./13, 8)
0.07692308
```

```
>>> round(1./13, 8)  # funktioniert nicht
0.0769230800000000005
```

Test-Infrastruktur für Projekte

- Alle Tests leicht ausführbar machen (Makefile, Python-Skript etc.)
- Alle Tests häufig ausführen, zumindest vor einem Commit – auf jeden Fall aber vor einem Release (Empfehlung: automatisch am Anfang von `make release`)
- In Docstrings nur einfache Verwendungsbeispiele schreiben, die man darin auch ohne Testabsicht aufnehmen würde
- Speziellere Tests nur in Textdateien
- Beim Schreiben der Dateien Zielgruppe im Auge behalten – Entwickler oder Anwender des Codes?
- Verzeichnisstruktur
  - Testdateien im gleichen Verzeichnis wie der zu testende Code, mit Präfix (beispielweise „_test_“); alternativ:
  - Testdateien in eigenem Verzeichnis, normalerweise ebenfalls mit Präfix

Zusammenfassung

- Automatisierte Tests empfehlen sich zum wiederholten gründlichen Testen, als Hilfe beim Refaktorieren und zur testgetriebenen Entwicklung
- Einfache Doctests sind nichts anderes als Codebeispiele in Docstrings; sie werden mit `doctest.testmod` ausgeführt
- Doctests können auch in Textdateien geschrieben werden, was die Python-Dateien übersichtlich hält; `doctest.testfile` arbeitet solche Tests ab
- Erwartete Ausgaben müssen genau wie im Interpreter dargestellt werden
- Diverse Doctest-Optionen und andere Hilfen unterstützen das Schreiben von Doctests
- Jedes Projekt sollte eine durchdachte Test-Infrastruktur haben. Die Tests sollten möglichst oft ausgeführt werden, vor allem unmittelbar vor einem Release

Danke für Ihre Aufmerksamkeit! :-)

Fragen?

Anmerkungen?

Diskussion?