Test-Driven Design mit Python

Dr. Uwe Ziegenhagen

12.06.2019

Warum Softwaretests?

- Je später der Bug gefunden wird, desto teurer ist die Beseitigung!

Design and architecture	Implementation	Integration testing	Customer beta test	Postproduct release
1X*	5X	10X	15X	30X

*X is a normalized unit of cost and can be expressed in terms of person-hours, dollars, etc.
Source: National Institute of Standards and Technology (NIST)†

By catching defects as early as possible in the development cycle, you can significantly reduce your development costs.

Abbildung: Quelle: NIST via
`https://xbsoftware.com/blog/cost-bugs-software-testing/`

Arten von Softwaretests

- Unit Testing: Auf Funktionsebene
- Component Test: Auf Ebene der Bibliothek/des Moduls
- System Test: Gesamt-Sytem mit externen Interfaces
- Performance Test: Timing, Ressourcenauslastung

$\Rightarrow$ Fokus auf Unit Testing!

Unittest

- Teste individuelle Funktionen
- Je Testfall ein Test, Positiv-Test oder Negativ-Test
- Können in „Test-Suites“ gruppiert werden
- Sollten automatisiert und schnell laufen können
- Nicht in der Produktionsumgebung laufen lassen!

Test-Driven Development

- TDD = „Test-Driven Development“
- Wikipedia `https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung`
- Entwickelt durch Kent Beck als Teil von Extreme Programming
- SUnit als erstes Testframework für Smalltalk
- Zusammen mit Erich Gamma JUnit, die Basis für Frameworks in diversen Sprachen
- Quintessenz: Schreibe Testcode vor Programmcode

TDD - Vorgehen

- schreibe einen neuen Test
- teste und lass den neuen Test fehlschlagen
- schreibe Programmcode, so dass der neue Test durchläuft
- lass die ganze Testsuite laufen
- refactor den Code, falls notwendig[1]
- Wiederhole...

[1] Make it work, make it good, make it fast!

Uncle Bobs Gesetze des TDD

„Uncle Bob“: Robert Cecil Martin, Autor von „Clean Code“

(a) Schreibe keinen Produktionscode, bevor Du nicht fehlschlagenden Testcode hast.

(b) Schreib nicht mehr Testcode als für einen fehlschlagenden Test nötig ist.

(c) Schreib nicht mehr Produktionscode, als für das Bestehen des aktuell fehlschlagenden Unittests nötig ist.

Test-Diven Development

Vor- und Nachteile

Nachteile

- mehr Arbeit
- mehr Disziplin

Vorteile

- mittel- und langfristig weniger Arbeit
- mehr Denken, *was* das Programm tun soll, bevor man zum eigentlichen *wie* kommt?
- Vertrauen in eigenen Code wird größer
- Bereitschaft zum Refactoring wird höher

Zwei Module schauen wir uns näher an, **doctest** und **pytest**

Doctesting in Python

- `doctest` = Python-Standardmodul
- kombiniere Tests zusammen mit eigentlichem Code in einer Datei
- Einsatzwecke:
  - Check, ob Docstrings noch aktuell sind
  - Test ob interaktive Beispiele einer Datei wirklich so laufen wie angegeben
- `python -m doctest -v <Datei.py>`
- `-v` verbose; zeige detaillierte Ergebnisse
- ohne `-v` nur Ausgabe, falls Fehler auftauchen
- Hinweis: Auslagerung von Testcode in externe Datei ist möglich!

Doctesting in Python

- Tests kommen in die Funktionsbeschreibung
- Quelle: interaktiven Aufrufe der Funktionen
- Aufruf dann mittels `python -m doctest -v` *`Dateiname.py`*

```
1  def faculty(n):
2      """
3      Calculates n! for integers >= 0
4      >>> faculty(10)
5      3628800
6      >>> faculty(0)
7      1
8      >>> faculty(1)
9      1
10     >>> faculty(3)
11     6
12     """
13     if n == 0:
14         return 1
15     else:
16         return n * faculty(n-1)
```

Doctesting in Python - Ergebnisse 1

```
C:\Users\Codes>python -m doctest -v SomeMath.py
Trying:
    faculty(10)
Expecting:
    3628800
ok
Trying:
    faculty(0)
Expecting:
    1
ok
Trying:
    faculty(1)
Expecting:
    1
ok
```

Doctesting in Python - Ergebnisse 2

```
1  Trying:
2      faculty(3)
3  Expecting:
4      6
5  ok
6  1 items had no tests:
7      SomeMath
8  1 items passed all tests:
9     4 tests in SomeMath.faculty
10 4 tests in 2 items.
11 4 passed and 0 failed.
12 Test passed.
```

Aufgabe: Ausprobieren!

`pytest`-Integration in Spyder

IDEs bieten oft Unterstützung für Unittests, Spyder bietet ein Plugin

- `https://github.com/spyder-ide/spyder-unittest`
- `conda install -c spyder-ide spyder-unittest`
- Unter „Run“ im Menü neuer Menüpunkt „Run unit tests“
- Über `configure` Auswahl von Ordner und Test-Framework
- Entwicklungsgeschwindigkeit weicht von condas E. ab (Downgrade bei Installation!)

`pytest` versus `unittest`

- `unittest` ist Teil der Python-Standarddistribution, `pytest` nicht
- Anaconda enthält beide Pakete
- `pytest` ist leichter zu benutzen, daher beliebter als `unittest`
- `unittest` ist ebenfalls sehr reif, bietet viele Möglichkeiten

`pytest` Beispiel

- eigene Funktion, um Text in Großbuchstaben zu verwandeln
- Ja, wir könnten & sollten auch `str.upper()` nutzen...[2]
- Folgen wir der TDD-Methodik!
- Ersten Testfall schreiben, als `test_irgendwas.py` speichern
- Wechsel dieses Verzeichnis, `pytest` ausführen!

```
1 import pytest
2
3 def test_my_uppercase():
4     o = my_uppercase('a')
5     assert o == 'A'
```

Listing 1: test_myupper01.py ¶

[2] Don't reinvent the wheel!

Ausgabe

```
E:\mypyt>pytest
===== test session starts ========
platform win32 -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: E:\mypyt, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0,
    arraydiff-0.3
collected 1 item

test_myupper01.py F                [100%]
===== FAILURES ============
__________ test_my_uppercase _____

   def test_my_uppercase():
>      o = my_uppercase('a')
E      NameError: name 'my_uppercase' is not defined

test_myupper01.py:4: NameError
======= 1 failed in 0.05 seconds =========
```

- `pytest` erkennt Tests, wenn sie in `test_*` Dateien stecken!

pytest: Schreiben des Produktionscodes

```
1  import pytest
2
3  def my_uppercase(string):
4      return chr(ord(string)-32)
5
6  def test_my_uppercase():
7      o = my_uppercase('a')
8      assert o == 'A'
```

Listing 2: test_myupper02.py ¶

Ausgabe

```
1  E:\mypyt>pytest test_myupper02.py
2  === test session starts =========
3  platform win32 -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
4  rootdir: E:\mypyt, inifile:
5  plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0,
       arraydiff-0.3
6  collected 1 item
7
8  test_myupper02.py .                    [100%]
9
10 === 1 passed in 0.02 seconds =====
```

pytest: Schreiben des nächsten Testfalls

```
1  import pytest
2
3  def my_uppercase(string):
4      return chr(ord(string)-32)
5
6  def test_my_uppercase01():
7      o = my_uppercase('a')
8      assert o == 'A'
9
10 def test_my_uppercase02():
11     o = my_uppercase('A')
12     assert o == 'A'
```

Listing 3: test_myupper03.py ¶

Ausgabe

```
===FAILURES ============================
___test_my_uppercase02 _____________________________
   def test_my_uppercase02():
      o = my_uppercase('A')
>     assert o == 'A'
E     AssertionError: assert '!' == 'A'
E       - !
E       + A
test_myupper03.py:12: AssertionError
===1 failed, 1 passed in 0.03 seconds ============
```

pytest: Schreiben des Produktionscodes

```
1  import pytest
2
3  def my_uppercase(string):
4      if ord(string)>=65 and ord(string)<=90:
5          return string
6      else:
7          return chr(ord(string)-32)
8
9  def test_my_uppercase01():
10     o = my_uppercase('a')
11     assert o == 'A'
12
13 def test_my_uppercase02():
14     o = my_uppercase('A')
15     assert o == 'A'
```

Listing 4: test_myupper04.py ¶

Testen. . .

```
E:\mypyt>pytest test_myupper04.py
=test session starts =
platform win32 -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0
rootdir: E:\mypyt, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0,
    arraydiff-0.3
collected 2 items

test_myupper04.py .. [100%]

=2 passed in 0.02 seconds =
```

Beispiel Spaltenindex ⇒ Spaltenlabel

- openpyxl = Python-Package, um Excel-Dateien zu bearbeiten
- erfordert u. a. Angabe des Spaltenlabels („A“, „AA“) als String
- Schreiben wir eine kleine Helferfunktion[3]
- Fachliche Anforderung:

  Spalten 1 – 26 „A“ – „Z“
  Spalten 27 – 52 „AA“ – „AZ“
  Spalten 53 – . . . „BA“ – . . .
  Spalte 16384 „XFD“
- Ansatz via Test-Driven Design

[3] openpyxl hat diese Funktion aber auch `get_column_letter` in `utils.cell`

Test 00

- Test, das die 1. Spalte das Label „A“ erhält
- Test muss fehlschlagen, da Funktion nicht definiert ist
- Aufruf mittels

```
1 import pytest
2
3 def test_column2label01():
4     label = column2label(1)
5     assert label == 'A'
```

Listing 5: test_col2label-00.py ¶

Ausgabe

```
E:\Repos\FOM.git\trunk\Skriptsprachenprogrammierung\Folien\Codes\col2label>python -m pytest test_col2label-00.py

test session starts platform win32 -- Python 3.7.3, pytest-4.5.0, py-1.8.0, pluggy-0.11.0
collected 1 item

test_col2label-00.py F

    [100%]

FAILURES
_____________________________________________ test_column2label01
   def test_column2label01():
>      label = column2label(1)
E      NameError: name 'column2label' is not defined

test_col2label-00.py:5: NameError
========================================== 1 failed in 0.07 seconds
```

Test 01

- Implementierung der Funktion

```
1  import pytest
2
3  def column2label(index):
4      return chr(64 + index)
5
6  def test_column2label01():
7      label = column2label(1)
8      assert label == 'A'
```

Listing 6: test_col2label-01.py ¶

Ausgabe

```
>python -m pytest test_col2label-01.py
====== test session starts ============
platform win32 -- Python 3.7.3, pytest-4.5.0, py-1.8.0, pluggy-0.11.0
rootdir: E:\Repos\col2label
plugins: remotedata-0.3.1, openfiles-0.3.2, doctestplus-0.3.0,
    arraydiff-0.3
collected 1 item

test_col2label-01.py .

    [100%]

===== 1 passed in 0.02 seconds ==========
```

Test 02

- Klappt es auch für 2?

```
1  # -*- coding: utf-8 -*-
2  def column2label(index):
3      return chr(64 + index)
4
5  def test_column2label01():
6      label = column2label(1)
7      assert label == 'A'
8
9  def test_column2label02():
10     label = column2label(2)
11     assert label == 'B'
```

Listing 7: test_col2label-02.py ¶

Test 03

- Für „0“ muss es einen Fehler geben, Test schlägt fehl

```
1  import pytest
2
3  def column2label(index):
4      return chr(64 + index)
5
6  def test_column2label01():
7      label = column2label(1)
8      assert label == 'A'
9
10 def test_column2label03():
11     with pytest.raises(ValueError):
12         label = column2label(0)
```

Listing 8: test_col2label-03.py ¶

Test 04

- Implementierung des Exception-Code

```
1  import pytest
2
3  def column2label(index):
4      if index > 0:
5          return chr(64 + index)
6      else:
7          raise ValueError('index must be greater or equal 1')
8
9  def test_column2label01():
10     label = column2label(1)
11     assert label == 'A'
12
13 def test_column2label03():
14     with pytest.raises(ValueError):
15         label = column2label(0)
```

Listing 9: test_col2label-04.py ¶

Test 05

- Fehler auch bei negativen Integers kleiner 0

```
1  import pytest
2
3  def column2label(index):
4      if index > 0:
5          return chr(64 + index)
6      else:
7          raise ValueError('index must be greater or equal 1')
8
9  def test_column2label03():
10     with pytest.raises(ValueError):
11         label = column2label(0)
12
13 def test_column2label04():
14     with pytest.raises(ValueError):
15         label = column2label(-2)
```

Listing 10: test_col2label-05.py ¶

Test 06

- Parameter muss Zahl (integer) sein

```
1  import pytest
2
3  def column2label(index):
4      try:
5          index = int(index)
6      except:
7          raise ValueError('index must be an integer')
8
9      if index > 0:
10         return chr(64 + index)
11     else:
12         raise ValueError('index must be greater or equal 1')
13
14 def test_column2label05():
15     with pytest.raises(ValueError):
16         label = column2label("abcd")
```

Listing 11: test_col2label-06.py ¶

Test 07

- Muss auch mit Zahlen größer 26 funktionieren ⇒ Fehler!

```
1  import pytest
2
3  def column2label(index):
4      try:
5          index = int(index)
6      except:
7          raise ValueError('index must be an integer')
8      if index > 0:
9          return chr(64 + index)
10     else:
11         raise ValueError('index must be greater or equal 1')
12
13 def test_column2label07():
14     label = column2label(27)
15     assert label == 'AA'
```

Listing 12: test_col2label-07.py ¶

Test 08

- Refactoring der Funktion

```
1  import pytest
2
3  def column2label(index):
4      try:
5          index = int(index)
6      except:
7          raise ValueError('index must be an integer')
8      if index > 0:
9          if index < 27:
10             return chr(64+index)
11         else:
12             return column2label(index // 26) + column2label(index % 26)
13     else:
14         raise ValueError('index must be greater or equal 1')
15
16 def test_column2label07():
17     label = column2label(27)
18     assert label == 'AA'
```

Listing 13: test_col2label-08.py ¶