

# Unittests für Einsteiger

Johannes Hubertz

PyCon.de – Köln, 14. Oktober 2013

Vorstellungsrunde  
Weshalb Tests?  
Doctests am Beispiel  
TDD: Test driven development  
Ein wenig Theorie  
Unittests Grundlagen  
Beispiele  
Änderungen, Unittests zeigen Wirkung  
Prakt. Beispiel von den Teilnehmern  
nosetests, tox  
Fragen und Kritik

## Netzwerk IPv4 und IPv6

Linux für Firewalls und VPN-Server (IPsec, StrongSwan, OpenVPN)

Linux als Rendezvous-Server mit OpenSSH

Router: statisch, dynamisch, OSPF, OSPFv3, radvd, ...

Switch: bridge-utils, OpenVSwitch, Cisco™ ...

## Virtualisierung

Server: Xen, KVM, Clusterbau, Heartbeat, drbd, iSCSI ...

Firewalls: iptables ⇒ sspe, ip6tables ⇒ adm6

Netzwerk: pox, OpenVSwitch, Hardwareswitches, OpenStack ...

Shell, C, Perl und Python haben bisher stets zur Lösung geführt

## pythonic work under GNU General Public License

<https://github.com/s10/{findcnt,adm6,conv,opti}>

# Warum wollen wir testen?

## Wer redet von Softwarekrise?

- Komplexität steigt, Fehlerrate ist bestenfalls konstant
- Seit den 1980'ern ist die Rede von der Softwarekrise
- Fehlerfreiheit ist nicht herstellbar

## Software-Engineering ist die Lösung (seit den 1990'ern?)

- Wasserfallmodell, V-Modell
- Anforderungsprofile, Pflichtenheft, Lastenheft
- Modularisierung, Ada, 4th-GL, ...
- Qualität wird besser, aber nicht gut

## Und heute?

- Continuous Integration, SCRUM, bedarfsorientierte Entwicklung ...
- Agile Methoden: Untested Software is broken by design ...

## doctest: Einfachste Methode, python code zu testen

Der Python-Interpreter wird einfach nachempfunden, Aufruf:

```
$ python -m doctest [-v] Textdatei
```

Der **Inhalt** der **Textdatei** wird interpretiert:

```
# Kommentar wird ignoriert
```

```
>>> Python-Statment wird ausgeführt
```

```
... eingrücktes Folge-Statment wird angehängt
```

Leerzeile initiiert Ausführung vorheriger Zeilen

Diese und nicht eingerückte Inhalte werden mit Ergebnis verglichen!

Aufruf mit **[-v]** zeigt den Ablauf

## Beispiel: README.txt mit doctest für ein dictionary

```
1 # README.txt as doctest example #
2 #
3 # first we create a variable of type dictionary
4 # then let us have one entry
5 >>> a = {}
6 >>> a['color'] = 'blue'
7
8 # What is the content?
9 >>> a
10 {'color': 'blue'}
```

### Ausführung:

```
1 hans@jha:~/pycon-tests$ python -m doctest README.txt
2 hans@jha:~/pycon-tests$ python -m doctest -v README.txt
3 Trying:
4     a = {}
5 Expecting nothing
6 ok
7 Trying:
8     a['color'] = 'blue'
9 Expecting nothing
10 ok
11 Trying:
12     a
13 Expecting:
14     {'color': 'blue'}
```

15 ok

```
16 1 items passed all tests:
17   3 tests in README.txt
18 3 tests in 1 items.
19 3 passed and 0 failed.
20 Test passed.
21 hans@jha:~/pycon-tests$
```

doctest kann auch in docstrings geschrieben sein und einiges mehr...

Mehr zum Thema:

<http://docs.python.org/2/library/doctest.html>



# Mathematik macht glücklich

*Definition:*

Sei  $M$  eine Menge.

Seien  $N_j \neq \emptyset$  ( $j, k \in J$ ) Untermengen von  $M$ .

Dann sind die  $N_j$  eine **Partition** von  $M$ , wenn

$$M = \bigcup_{j \in J} N_j \text{ und } N_j \cap N_k = \emptyset$$

Noch Fragen?

## Definition:

Eine **Äquivalenzrelation** (Bezeichnung:  $\sim$ ) ist eine zweistellige Relation, für die folgende Regeln gelten:

1:  $x \sim x$  (reflexiv)

Jedes Element steht zu sich selbst in Relation.

2:  $x \sim y \Rightarrow y \sim x$  (symmetrisch)

Wenn  $x$  zu  $y$ , dann steht auch  $y$  zu  $x$  in Relation.

3:  $x \sim y \wedge y \sim z \Rightarrow x \sim z$  (transitiv)

Wenn  $x$  zu  $y$  und  $y$  zu  $z$ , dann steht auch  $x$  zu  $z$  in Relation.

Satz: Sei  $\mathbb{M}$  eine Menge

Sei  $\sim$  eine Äquivalenzrelation auf  $\mathbb{M}$ . Für  $m \in \mathbb{M}$  setzen wir:

$$[m] = \{m' \mid m' \in \mathbb{M}, m' \sim m\}$$

und nennen  $[m]$  die Äquivalenzklasse von  $m$  (bezüglich  $\sim$ ). Damit gilt:

$$\mathbb{M} = \bigcup_{m \in \mathbb{M}} [m]$$

Ferner ist

$$[m] \cap [m'] = \begin{cases} \emptyset & \text{für } m \not\sim m' \\ [m] = [m'] & \text{für } m \sim m' \end{cases}$$

Sind  $[m_j]$  mit  $j \in \mathbb{J}$  die verschiedenen Äquivalenzklassen, so ist

$$\mathbb{M} = \bigcup_{j \in \mathbb{J}} [m_j]$$

eine Partition von  $\mathbb{M}$ .

Mathematik ist zu kompliziert?

## Ein Beispiel:

Sei  $\mathbb{M}$  eine Menge roter, gelber und grüner Kugeln.

Äquivalenzrelation:  $x \sim y \iff$  Kugel  $x$  hat gleiche Farbe wie  $y$ .

1 Reflexivität:  $f(x) = f(x)$

2 Symmetrie:  $f(x) = f(y) \iff f(y) = f(x)$

3 Transitivität:  $f(x) = f(y) \wedge f(y) = f(z) \implies f(x) = f(z)$

Äquivalenzklasse:  $\mathbb{M} \supset \mathbb{G} = \{x | x \in \mathbb{M} \cap f(x) = \text{gelb}\} \iff$  alle gelben Kugeln.

Eine Partition besteht aus der Vereinigung **aller** so gebildeten Teilmengen aus je den roten, gelben und grünen Kugeln.

Alles klar?

Früher sprachen wir auch von Fallunterscheidung

Nur exakte Betrachtung ermöglicht exakte Lösung

Mathematik *kann* helfen, Formalisierung unterstützt

Die Aufgabenstellung: Implementieren Sie

**Def.:** Funktion  $\text{signum}(x)$ , hier als  $\text{sign}(x)$  für  $x \in \mathbb{R}$ :

$$\text{sign}(x) = \begin{cases} 1 & \text{für } x > 0 \\ 0 & \text{für } x = 0 \\ -1 & \text{für } x < 0 \end{cases}$$

**Wie** kann das implementiert werden?

**Wieviele** Fehlermöglichkeiten sind dabei?

Nur mit den richtigen Tests wird **Zuverlässigkeit** erreicht

**Was** sind die richtigen Tests?

Anders gefragt, wie geht das?

Die Antwort auf alle Fragen: **Test Driven Development**

## TDD: Wie geht das?

### Vorgehensweise in mehreren Schritten

- 1 Umgebung definieren: def, Klasse oder Modul?
- 2 Äquivalenzklassen bestimmen
- 3 Grenzwerte feststellen
- 4 Prototyp schreiben ohne jeden Inhalt: **pass**
- 5a mind. 1 Test schreiben pro Äquivalenzklasse
- 5b Code zur Erfüllung der Tests in den Prototyp eintragen
- 5c Tests laufen lassen bis zur Fehlerfreiheit
- 6 Ergänzende Tests mit fehlerhaften Eingabewerten schreiben

## Merke:

Zuerst den Test, danach erst den Code schreiben.

Das schafft Klarheit im Denken. . .



**Def.:** Funktion  $\text{signum}(x)$ , hier als  $\text{sign}(x)$  für  $x \in \mathbb{R}$ :

$$\text{sign}(x) = \begin{cases} 1 & \text{für } x > 0 \\ 0 & \text{für } x = 0 \\ -1 & \text{für } x < 0 \end{cases}$$

**Schritt 1:** Umgebung definieren

Der Einfachheit halber reicht hier ein **def sign(x):**

**Schritt 2:** Äquivalenzklassen bestimmen

Die Definition macht es uns einfach:  $> 0, = 0, < 0$

**Schritt 3:** Grenzwerte bestimmen

Die Null kennt zwei Annäherungen:  $-0, +0$

## Schritt 4: Prototyp schreiben (Stichwort: pass)

```
1 import unittest
2
3 def sign(input_value):
4     """ implementation of a math function: sign(x) """
5     pass
```

## Schritt 5: Ersten Test schreiben

```
8 import unittest
9
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     pass
13
14
15 class Signtest(unittest.TestCase):
16     """ tests for sign(x) """
17
18     def test_sign_basic(self):
19         """ check basic return values >0 """
20         value = sign(1)
21         expect = 1
22         self.assertEqual(expect, value, msg = "think positive")
23
24
25 if __name__ == "__main__":
26     unittest.main()
```

## Schritt 5: Ersten Test ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1
2 F
3 -----
4 FAIL: test_sign_basic (__main__.Signtest)
5 check basic return values
6 -----
7 Traceback (most recent call last):
8   File "./sign_1", line 22, in test_sign_basic
9     self.assertEqual(expect, value, msg = "think positive")
10 AssertionError: think positive
11
12 -----
13 Ran 1 test in 0.001s
14
15 FAILED (failures=1)
16 hans@jha:~/pycon-tests$
```

## Schritt 5: zu testende Funktion ändern, korrigieren

```
8 import unittest
9
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     return 1
13
14
15 class Signtest(unittest.TestCase):
16     """ tests for sign(x) """
17
18     def test_sign_basic(self):
19         """ check basic return values >0 """
20         value = sign(1)
21         expect = 1
22         self.assertEqual(expect, value, msg = "think positive")
23
24
25 if __name__ == "__main__":
26     unittest.main()
```

## Schritt 5: Ersten Test nochmals ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1-ok -v
2 test_sign_basic (__main__.Signtest)
3 check basic return values ... ok
4
5 -----
6 Ran 1 test in 0.000s
7
8 OK
9 hans@jha:~/pycon-tests$
```

## Schritt 5: Zweiten Test schreiben

```
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     return 1
13
14
15 class Signtest(unittest.TestCase):
16     """ tests for sign(x) """
17
18     def test_sign_basic(self):
19         """ check basic return values >0 """
20         value = sign(1)
21         expect = 1
22         self.assertEqual(expect, value, msg = "think positive")
23
24     def test_sign_negative(self):
25         """ check basic return values <0 """
26         value = sign(-1)
27         expect = -1
28         self.assertEqual(expect, value, msg = "think negative")
29
30
31 if __name__ == "__main__":
32     unittest.main()
```

## Schritt 5: Zweiten Test ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1_-1
2 F.
3 -----
4 FAIL: test_sign_negative (__main__.Signtest)
5 check basic return values: <0
6 -----
7 Traceback (most recent call last):
8   File "./sign_1_-1", line 28, in test_sign_negative
9     self.assertEqual(expect, value, msg = "think negative")
10    AssertionError: think negative
11
12 -----
13 Ran 2 tests in 0.001s
14
15 FAILED (failures=1)
```

## Schritt 5: zu testende Funktion ändern, korrigieren

```
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     if input_value < 0:
13         return -1
14     return 1
15
16
17 class Signtest(unittest.TestCase):
18     """ tests for sign(x) """
19
20     def test_sign_basic(self):
21         """ check basic return values >0 """
22         value = sign(1)
23         expect = 1
24         self.assertEqual(expect, value, msg = "think positive")
25
26     def test_sign_negative(self):
27         """ check basic return values <0 """
28         value = sign(-1)
29         expect = -1
30         self.assertEqual(expect, value, msg = "think negative")
31
32
33 if __name__ == "__main__":
34     unittest.main()
```

## Schritt 5: Zweiten Test nochmals ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1_-1-ok -v
2 test_sign_negative (__main__.Signtest)
3 check basic return values: <0 ... ok
4 test_sign_positive (__main__.Signtest)
5 check basic return values: >0 ... ok
6
7 -----
8 Ran 2 tests in 0.001s
9
10 OK
11 hans@jha:~/pycon-tests$
```

## Schritt 5: Dritten Test schreiben

```
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     if input_value < 0:
13         return -1
14     return 1
15
16
17 class Signtest(unittest.TestCase):
18     """ tests for sign(x) """
19
20     def test_sign_basic(self):
21         """ check basic return values >0 """
22         value = sign(1)
23         expect = 1
24         self.assertEqual(expect, value, msg = "think positive")
25
26     def test_sign_zero(self):
27         """ check basic return values =0 """
28         value = sign(0)
29         expect = 0
30         self.assertEqual(expect, value, msg = "think of zero")
31
32     def test_sign_negative(self):
33         """ check basic return values <0 """
```

## Schritt 5: Dritten Test ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1_0_-1
2 ..F
3 -----
4 FAIL: test_sign_zero (__main__.Signtest)
5 check basic return values: =0
6 -----
7 Traceback (most recent call last):
8   File "./sign_1_0_-1", line 30, in test_sign_zero
9     self.assertEqual(expect, value, msg = "think_of_zero")
10    AssertionError: think of zero
11
12 -----
13 Ran 3 tests in 0.005s
14
```

## Schritt 5: Dritten Test schreiben

```
10 def sign(input_value):
11     """ implementation of a math function: sign(x) """
12     if input_value < 0:
13         return -1
14     if input_value == 0:
15         return 0
16     return 1
17
18
19 class Signtest(unittest.TestCase):
20     """ tests for sign(x) """
21
22     def test_sign_basic(self):
23         """ check basic return values >0 """
24         value = sign(1)
25         expect = 1
26         self.assertEqual(expect, value, msg = "think positive")
27
28     def test_sign_zero(self):
29         """ check basic return values =0 """
30         value = sign(0)
31         expect = 0
32         self.assertEqual(expect, value, msg = "think of zero")
33
34     def test_sign_negative(self):
35         """ check basic return values <0 """
```

## Schritt 5: Dritten Test ausführen

```
1 hans@jha:~/pycon-tests$ ./sign_1_0_-1
2 ...
3 -----
4 Ran 3 tests in 0.001s
5
6 OK
7 hans@jha:~/pycon-tests$
```



# Rückblick, Verbesserungspotential?

```
17 class Signtest(unittest.TestCase):
18     """tests for sign(x)"""
19
20     def test_sign_basic(self):
21         """check basic return values >0"""
22         value = sign(1)
23         expect = 1
24         self.assertEqual(expect, value, msg = "think positive")
25
26     def test_sign_zero(self):
27         """check basic return values =0"""
28         value = sign(0)
29         expect = 0
30         self.assertEqual(expect, value, msg = "think of zero")
31
32     def test_sign_negative(self):
33         """check basic return values <0"""
34         value = sign(-1)
35         expect = -1
36         self.assertEqual(expect, value, msg = "think negative")
37
38
39 if __name__ == "__main__":
40     unittest.main()
```

# Verbesserungspotential: Überflüssige Variablen und überflüssige Methoden

```
17 class Signtest(unittest.TestCase):
18     """tests for sign(x)"""
19
20     def test_sign_basic(self):
21         """check basic return values >0, =0, <0"""
22         self.assertEqual(1, sign(1), msg = "think positive")
23         self.assertEqual(0, sign(0), msg = "think of zero")
24         self.assertEqual(-1, sign(-1), msg = "think negative")
25
26
27 if __name__ == "__main__":
28     unittest.main()
```

## class TestCase hat Methoden :

Method	checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a,b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>a is None</code>
<code>assertIsNotNone(x)</code>	<code>a is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

## class TestCase hat Methoden :

Method

checks that

---

```
assertRaises(exc, fun, *args, **kwds)
```

```
    fun(*args, **kwds) raises exc
```

```
assertRaisesRegexp(exc, r, fun, *args, **kwds)
```

```
    fun(*args, **kwds) raises exc and the message matches regex r
```

Die Aufgabenstellung: Implementieren Sie

**Def.:** Funktion Complex signum( $z$ ), hier als  $c\text{sign}(z)$  für  $z \in \mathbb{C}$ :

$$c\text{sign}(x) = \begin{cases} 1 & \text{für } \Re(z) > 0 \\ -1 & \text{für } \Re(z) < 0 \\ \text{sign}(\Im(z)) & \text{für } \Re(z) = 0 \end{cases}$$

## Merke:

Zuerst den Test, danach erst den Code schreiben.

Das schafft Klarheit im Denken...

Have fun!

## Schritt 5: Prototyp mit pass und Test schreiben

```
12 def csign(input_value):
13     """
14     implementation of a math function: csign(z)
15     z must be a complex number
16     """
17     pass
18
19 class Csigntest(unittest.TestCase):
20     """
21     tests for csign(x)
22     """
23     def test_01_sign_complex(self):
24         """
25         check correct return values of csign
26         """
27         self.assertEqual(1, csign(complex(1, 2)), msg = "think positive")
28         self.assertEqual(1, csign(complex(0, 2)), msg = "think zero positive")
29         self.assertEqual(0, csign(complex(0, 0)), msg = "think of zero")
30         self.assertEqual(-1, csign(complex(0, -2)), msg = "think zero negative")
31         self.assertEqual(-1, csign(complex(-1, 1)), msg = "think negative")
32
33
34 if __name__ == "__main__":
35     unittest.main()
```

## Schritt 5: Ersten Test ausführen

```
1 hans@jha:~/pycon-tests$ python csign_1.py -f
2 F
3 -----
4 FAIL: test_01_sign_complex (__main__.Csigntest)
5 -----
6 Traceback (most recent call last):
7   File "csign_1.py", line 27, in test_01_sign_complex
8     self.assertEqual(1, csign(complex(1,2)), msg = "think_␣positive")
9 AssertionError: think positive
10
11 -----
12 Ran 1 test in 0.006s
13
14 FAILED (failures=1)
15 hans@iha:~/pvcon-tests$
16 Johannes Hubertz
```

## Schritt 5: Funktion testerfüllend schreiben

```
12 def csign(input_value):
13     """
14     implementation of a math function: csign(z)
15     z must be a complex number
16     """
17     if type(input_value) == complex:
18         re = input_value.real
19         im = input_value.imag
20         if re < 0:
21             return sign(re)
22         else:
23             return sign(im)
24     else:
25         msg = "csign(z) only accepts type complex"
26         raise ValueError, msg
27
28
29 class Csigntest(unittest.TestCase):
30     """
31     tests for csign(x)
32     """
33     def test_01_sign_complex(self):
34         """
35         check correct return values of csign
36         """
37         self.assertEqual(1, csign(complex(1, 2)), msg = "think positive")
38         self.assertEqual(1, csign(complex(0, 2)), msg = "think zero positive")
39         self.assertEqual(0, csign(complex(0, 0)), msg = "think of zero")
40         self.assertEqual(-1, csign(complex(0, -2)), msg = "think zero negative")
41         self.assertEqual(-1, csign(complex(-1, 1)), msg = "think negative")
```

## Schritt 5: Funktion testerfüllend schreiben

```
12 def csign(input_value):
13     """
14     implementation of a math function: csign(z)
15     z must be a complex number
16     """
17     if type(input_value) == complex:
18         re = input_value.real
19         im = input_value.imag
20         if re <= 0:
21             return sign(re)
22         else:
23             return sign(im)
24     else:
25         msg = "csign(z) only accepts type complex"
26         raise ValueError, msg
27
28
29 class Csigntest(unittest.TestCase):
30     """
31     tests for csign(x)
32     """
33     def test_01_sign_complex(self):
34         """
35         check correct return values of csign
36         """
37         self.assertEqual(1, csign(complex(1, 2)), msg = "think positive")
38         self.assertEqual(1, csign(complex(0, 2)), msg = "think zero positive")
39         self.assertEqual(0, csign(complex(0, 0)), msg = "think of zero")
40         self.assertEqual(-1, csign(complex(0, -2)), msg = "think zero negative")
41         self.assertEqual(-1, csign(complex(-1, 1)), msg = "think negative")
```

## Schritt 5: Ersten Test ausführen

```
1 hans@jha:~/pycon-tests$ python csign_1_1.py -v
2 test_01_sign_complex (__main__.Csigntest) ... ok
3
4 -----
5 Ran 1 test in 0.004s
6
7 OK
8 hans@iha:~/pycon-tests$
9 Johannes Hubertz
```



## Schritt 5: Tests komplettieren

```
12 def csign(input_value):
13     """
14     implementation of a math function: csign(z)
15     z must be a complex number
16     """
17     if type(input_value) == complex:
18         re = input_value.real
19         im = input_value.imag
20         if re < 0:
21             return sign(re)
22         else:
23             return sign(im)
24     else:
25         msg = "csign(z) only accepts type complex"
26         raise ValueError, msg
27
28
29 class Csigntest(unittest.TestCase):
30     ...
31     def test_02_sign_complex_exception(self):
32         """
33         check exceptions of csign
34         """
35         expect = "csign(z) only accepts type complex"
36         value = 27.5
37         try:
38             csign(value)
39             self.assertFail()
40         except Exception as inst:
41             self.assertEqual(inst.message, expect)
```

## Schritt 5: Tests komplettieren

```
12 def csign(input_value):
13     """
14     implementation of a math function: csign(z)
15     z must be a complex number
16     """
17     if type(input_value) == complex:
18         re = input_value.real
19         im = input_value.imag
20         if re < 0:
21             return sign(re)
22         else:
23             return sign(im)
24     else:
25         msg = "csign(z) only accepts type complex"
26         raise ValueError, msg
27
28
29 class Csigntest(unittest.TestCase):
30     ...
31     def test_03_sign_complex_other_types(self):
32         """
33         check exceptions of csign for list, dict, etc
34         """
35         expect = "csign(z) only accepts type complex"
36         values = [[1, 2, ], {'a': 4711}, "string_type", ]
37         for value in values:
38             try:
39                 csign(value)
40                 self.assertFail()
41             except Exception as inst:
42                 self.assertEqual(inst.message, expect)
```

## Schritt 5: Komplettierten Test ausführen

```
1 hans@jha:~/pycon-tests$ ./csign_1_ok.py -f
2 ...
3 -----
4 Ran 3 tests in 0.001s
5
6 OK
7 hans@jha:~/pycon-tests$ ./csign_1_ok.py -f -v
8 test_01_sign_complex (__main__.Csigntest) ... ok
9 test_02_sign_complex_exeception (__main__.Csigntest) ... ok
10 test_03_sign_complex_other_types (__main__.Csigntest) ... ok
11
12 -----
13 Ran 3 tests in 0.005s
14
15 OK
16 hans@jha:~/pycon-tests$
```

## Schritt 5: Komplettierten Test mit nosetests ausführen

```
1 hans@jha:~/pycon-tests$ nosetests csign_1_ok.py -v
2 check correct return values of csign ... ok
3 check exceptions of csign ... ok
4 check exceptions of csign for list, dict, etc ... ok
5
6 -----
7 Ran 3 tests in 0.002s
8
9 OK
10 hans@jha:~/pycon-tests$
```

## Schritt 5: nosetests mit Testabdeckung und html-Ausgabe

```
1 hans@jha:~/pycon-tests$ nosetests -v sign_5.py csign_1_ok.py --with-coverage --cover-html
2 check return values for integer numbers ... ok
3 check return values for floating numbers ... ok
4 check exception for list input ... ok
5 check exception for complex numbers ... ok
6 check correct return values of csign ... ok
7 check exceptions of csign ... ok
8 check exceptions of csign for list, dict, etc ... ok
9
10 Name          Stmts  Miss  Cover   Missing
11 -----
12 csign_1_ok     39     3   92%    51, 64, 70
13 sign_5        41     3   93%    55, 68, 74
14 -----
15 TOTAL         80     6   93%
16 -----
17 Ran 7 tests in 0.007s
18
19 OK
20 hans@jha:~/pycon-tests$
```

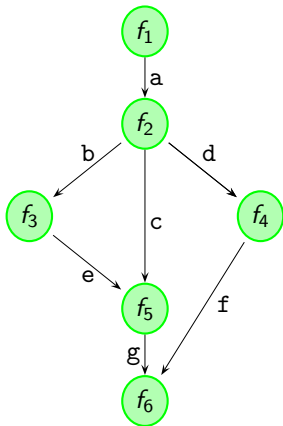
## Coverage for `csign_1_ok` : 92%

39 statements 36 run 3 missing 0 excluded

```
1 #!/usr/bin/env python
2 # -*- mode: python -*-
3 # -*- coding: utf-8 -*-
4
5 """
6     Example for a math function prototype
7 """
8 import unittest
9 import cmath
10 from sign_5 import sign
11
12 def csign(input_value):
13     """
14     implementation of a math function: sign(z)
15     x must be a complex number
16     """
17     if type(input_value) == complex:
18         re = input_value.real
19         im = input_value.imag
20         if re <= 0:
21             return sign(re)
22         else:
23             return sign(im)
24     else:
25         msg = "csign(z) only accepts type complex"
26         raise ValueError, msg
27
28
29 class Csigntest(unittest.TestCase):
30     """
31     tests for csign(x)
32     """
33     def test_01_sign_complex(self):
34         """
35         check correct return values of csign
36         """
37         self.assertEqual(1, csign(complex(1,2)), msg = "think positive" )
38         self.assertEqual(1, csign(complex(0,2)), msg = "think zero positive" )
39         self.assertEqual(0, csign(complex(0,0)), msg = "think of zero" )
40         self.assertEqual(-1, csign(complex(0,-2)), msg = "think zero negative" )
41         self.assertEqual(-1, csign(complex(-1,1)), msg = "think negative" )
42
```

# Verschiedene Wege ...

*Konfuzius: Auch der längste Weg beginnt mit dem ersten Schritt*



1. Unittests für alle Funktionen erfüllen

2. Alle Übergänge getestet?

3. Mögliche Pfade finden  $\implies$  Äquivalenzklassen bilden!  
[[a, c, g, ], [a, b, e, g, ], [a, d, f, ], ]

Mögliche Formen:

(chain, branch, junction, loop)

4. Modultest / Integrationstest entwerfen

5. Design / Anforderungen erfüllt?

## tox: Noch ein interessantes Werkzeug



## **tox:** Universelles Werkzeug nicht nur für Tests ...

Beliebige Programme zum Test

Beliebige Python Versionen

Virtualenv pro Test

As you like it ...

# tox: nosetests für beide Dateien ausführen mit Testabdeckung

```
1 hans@jha:~/pycon-tests$ cat tox.ini
2 [tox]
3 envlist = py26, py27
4 #envlist = py26, py27, pep8
5
6 [testenv]
7 commands =
8     /usr/local/bin/nosetests
9     /usr/bin/nosetests3
10
11 [testenv:py26]
12 basepython =
13     python2.6
14 commands =
15     python setup.py clean
16     python setup.py build
17     /usr/local/bin/nosetests -v sign_5.py csign_1_ok.py
18
19 [testenv:py27]
20 basepython =
21     python2.7
22 commands =
23     python setup.py clean
24     python setup.py build
25     /usr/local/bin/nosetests -v sign_5.py csign_1_ok.py
26
27 [testenv:py32]
28 basepython =
29     python3.2
30 commands =
31     /usr/bin/2to3 --add-suffix='3' -n -w ./sign_5.py
32     /usr/bin/2to3 --add-suffix='3' -n -w ./csign_1_ok.py
33     /usr/bin/python3.2 setup.py clean
34     /usr/bin/python3.2 setup.py build
35     /usr/bin/nosetests3 -v sign_5.py3 csign_1_ok.py3
36
37 [testenv:pep8]
38 commands =
39     /usr/local/bin/pep8 --show-pep8 --show-source sign_5.py csign_1_ok.py
40 hans@jha:~/pycon-tests$
```

## tox: nosetests für beide Dateien ausführen mit Testabdeckung

```
1 hans@jha:~/pycon-tests$ tox
2 GLOB sdist-make: /home/hans/Arch/2012/04/15/pycon/examples/setup.py
3 py26 inst-nodeps: /home/hans/Arch/2012/04/15/pycon/examples/.tox/dist/pycon-examples-0.0.1.zip
4 py26 runtests: commands[0]
5 running clean
6 py26 runtests: commands[1]
7 running build
8 running build_py
9 py26 runtests: commands[2]
10 check return values for integer numbers ... ok
11 check return values for floating numbers ... ok
12 check exception for list input ... ok
13 check exception for complex numbers ... ok
14 check correct return values of csign ... ok
15 check exceptions of csign ... ok
16 check exceptions of csign for list, dict, etc ... ok
17
18 -----
19 Ran 7 tests in 0.004s
20
21 OK
22 py27 inst-nodeps: /home/hans/Arch/2012/04/15/pycon/examples/.tox/dist/pycon-examples-0.0.1.zip
23 py27 runtests: commands[0]
24 running clean
25 py27 runtests: commands[1]
26 running build
27 running build_py
28 py27 runtests: commands[2]
29 check return values for integer numbers ... ok
30 check return values for floating numbers ... ok
31 check exception for list input ... ok
32 check exception for complex numbers ... ok
33 check correct return values of csign ... ok
34 check exceptions of csign ... ok
35 check exceptions of csign for list, dict, etc ... ok
36
37 -----
38 Ran 7 tests in 0.004s
39
40 OK
41 ----- summary -----
42   py26: commands succeeded
43   py27: commands succeeded
44   congratulations :)
45 hans@jha:~/pycon-tests$
```

## Quellen und Hinweise

Python Testing Cookbook, Pact Publishing, 2011

Python Testing Beginners Guide, Pact Publishing, 2010

<https://github.com/rbreu/python-course>

<http://wiki.python-forum.de/pycologne/Protokoll20130410>

[http://www.python-course.eu/python3\\_tests.php](http://www.python-course.eu/python3_tests.php)

<http://pythontesting.net/start-here/>

[https://github.com/gregmalcolm/python\\_koans/wiki](https://github.com/gregmalcolm/python_koans/wiki)

**Untested Software is broken by design**

Ich bedanke mich für Ihre Aufmerksamkeit  
hubertz-it-consulting GmbH jederzeit zu Ihren Diensten:  
verlässliche Netzwerke für vertrauliche Kommunikation

**Ihre Sicherheit ist uns wichtig!**

**Frohes Schaffen**

Johannes Hubertz

it-consulting \_at\_ hubertz dot de



powered by  python™ and  L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>