

Test Driven Development.

JOIN² Workshop

Alexander Wagner

JOIN² Workshop Aachen

Aachen, 10.06.2015



> Ausgangslage



- > Ausgangslage
- > Fehler



- > Ausgangslage
- > Fehler
- > Beispiel: addition



- > Ausgangslage
- > Fehler
- > Beispiel: addition
- > Beispiel: addition via Tests

Überblick



- > Ausgangslage
- > Fehler
- > Beispiel: addition
- > Beispiel: addition via Tests
- > `unittests`



- > Ausgangslage
- > Fehler
- > Beispiel: addition
- > Beispiel: addition via Tests
- > unittests
- > Beispiel: Erweiterung

Überblick



- > Ausgangslage
- > Fehler
- > Beispiel: addition
- > Beispiel: addition via Tests
- > `unittests`
- > Beispiel: Erweiterung
- > Do it yourself!

Ausgangslage

Gemeinsamer Code, aber:

- > kleine Gruppe von Entwicklern
- > jeder hat *seine* Module
- > manuelle Tests (viel Handarbeit, schwer reproduzierbar)

Gemeinsamer Code, aber:

- > kleine Gruppe von Entwicklern
- > jeder hat *seine* Module
- > manuelle Tests (viel Handarbeit, schwer reproduzierbar)

Wir werden mehr:

- > neue Partner, die nicht von Anfang an dabei waren

Gemeinsamer Code, aber:

- > kleine Gruppe von Entwicklern
- > jeder hat *seine* Module
- > manuelle Tests (viel Handarbeit, schwer reproduzierbar)

Wir werden mehr:

- > neue Partner, die nicht von Anfang an dabei waren
- > “Fremde” müssen Fehler finden und reparieren
- > Erweiterungen müssen aufgeteilt werden

Gemeinsamer Code, aber:

- > kleine Gruppe von Entwicklern
- > jeder hat *seine* Module
- > manuelle Tests (viel Handarbeit, schwer reproduzierbar)

Wir werden mehr:

- > neue Partner, die nicht von Anfang an dabei waren
- > “Fremde” müssen Fehler finden und reparieren
- > Erweiterungen müssen aufgeteilt werden
- > Jülich: ISO-9000

Definition

DIN EN ISO 9000:2005

“Qualitätsmanagement – Grundlagen und Begriffe”

Definition

DIN EN ISO 9000:2005

“Qualitätsmanagement – Grundlagen und Begriffe”

- > Merkmalswert, der die **vorgegebenen Forderungen** nicht erfüllt
- > **Nichterfüllung** einer **Anforderung**
- > **Anforderung**: Erfordernis oder Erwartung, das oder die festgelegt, üblicherweise vorausgesetzt oder verpflichtend ist.

Definition

DIN EN ISO 9000:2005

“Qualitätsmanagement – Grundlagen und Begriffe”

- > Merkmalswert, der die **vorgegebenen Forderungen** nicht erfüllt
- > **Nichterfüllung** einer **Anforderung**
- > **Anforderung**: Erfordernis oder Erwartung, das oder die festgelegt, üblicherweise vorausgesetzt oder verpflichtend ist.

Eine Routine ist fehlerhaft, wenn sie nicht tut, was **gefordert** wird.

Definitionen

Programmfehler: *Abweichung* des *IST* (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom *SOLL* (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze überschreitet. ([Wikipedia](#))

Definitionen

Programmfehler: *Abweichung* des *IST* (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom *SOLL* (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze überschreitet. ([Wikipedia](#))

Fehlersuche:

Definitionen

Programmfehler: *Abweichung* des *IST* (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom *SOLL* (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze überschreitet. ([Wikipedia](#))

Fehlersuche:

> Definiere *SOLL*

Definitionen

Programmfehler: *Abweichung* des *IST* (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom *SOLL* (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze überschreitet. ([Wikipedia](#))

Fehlersuche:

- > Definiere *SOLL*
- > Bestimme *IST*

Definitionen

Programmfehler: *Abweichung* des **IST** (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom **SOLL** (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze überschreitet. ([Wikipedia](#))

Fehlersuche:

- > Definiere **SOLL**
- > Bestimme **IST**
- > **IST** $\hat{=}$ **SOLL**

Testcases

Kennt die Maschine alle SOLL-Zustände, kann sie diese automatisch mit dem IST-Zustand vergleichen.

Testcases

Kennt die Maschine alle SOLL-Zustände, kann sie diese automatisch mit dem IST-Zustand vergleichen.

Voraussetzungen:

Testcases

Kennt die Maschine alle SOLL-Zustände, kann sie diese automatisch mit dem IST-Zustand vergleichen.

Voraussetzungen:

> Dokumentation des SOLL_(Issue Tracker)

Testcases

Kennt die Maschine alle SOLL-Zustände, kann sie diese automatisch mit dem IST-Zustand vergleichen.

Voraussetzungen:

- > Dokumentation des SOLL_(Issue Tracker)
- > Modularisierung des Codes _(schafft Kontrollpunkte)

Testcases

Kennt die Maschine alle SOLL-Zustände, kann sie diese automatisch mit dem IST-Zustand vergleichen.

Voraussetzungen:

- > Dokumentation des SOLL_(Issue Tracker)
- > Modularisierung des Codes _(schafft Kontrollpunkte)
- > Formalisierung der SOLL-Zustände als Code
- > Vergleichsprozeden

Beispiel 1: Aufgabenstellung

Entwickle eine Routine die zwei ganze Zahlen addiert und die Summe zurückliefert.

Beispiel 1: Aufgabenstellung

Entwickle eine Routine die zwei ganze Zahlen addiert und die Summe zurückliefert.

- > Eingabe sind ganze Zahlen
- > Ausgabe ist eine Zahl

Beispiel 1: Aufgabenstellung

Entwickle eine Routine die zwei ganze Zahlen addiert und die Summe zurückliefert.

- > Eingabe sind ganze Zahlen
- > Ausgabe ist eine Zahl
- > Funktion: `addition`
- > Parameter: zwei ganze Zahlen (`a`, `b`)
- > Ergebnis: Summe von `a` und `b`

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

```
> addition(1,1)
```

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

> addition(1,1): IST == 2, SOLL == 2 ☺

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

> addition(1,1): IST == 2, SOLL == 2 ☺

> addition(2,3)

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

> addition(1,1): IST == 2, SOLL == 2 ☺

> addition(2,3): IST == 5, SOLL == 5 ☺

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3)

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹
- > addition(2.5, 3.5)

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹
- > addition(2.5, 3.5): IST == 6.0, SOLL == 5 oder 6? ☹

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹
- > addition(2.5, 3.5): IST == 6.0, SOLL == 5 oder 6? ☹
- > addition('1', '1')

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹
- > addition(2.5, 3.5): IST == 6.0, SOLL == 5 oder 6? ☹
- > addition('1', '1'): IST == '11', SOLL == 2? ☹

Beispiel 1: Implementation

Python:

```
def addition(a, b):  
    return a+b
```

Mögliche Tests:

- > addition(1,1): IST == 2, SOLL == 2 ☺
- > addition(2,3): IST == 5, SOLL == 5 ☺
- > addition(2.3, 3.3): IST == 5.6, SOLL == 5 ☹
- > addition(2.5, 3.5): IST == 6.0, SOLL == 5 oder 6? ☹
- > addition('1', '1'): IST == '11', SOLL == 2? ☹
- > addition('a', 'b'): IST == 'ab', SOLL == ? ☹

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))
- 6 Teile als Stubs implementieren ([Top Down](#), [git commit...](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))
- 6 Teile als Stubs implementieren ([Top Down](#), [git commit...](#))
- 7 Füttere Funktion mit Testfällen

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))
- 6 Teile als Stubs implementieren ([Top Down](#), [git commit...](#))
- 7 Füttere Funktion mit Testfällen
- 8 Implementiere, bis alle Tests validieren ([git commit...](#))

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))
- 6 Teile als Stubs implementieren ([Top Down](#), [git commit...](#))
- 7 Füttere Funktion mit Testfällen
- 8 Implementiere, bis alle Tests validieren ([git commit...](#))
- 9 Merge zu master

Test Driven Approach

- 1 Ticket öffnen Anforderungen beschreiben ([github](#), [join2/join2](#))
- 2 Ticket zuweisen / sich selbst nehmen ([github](#))
- 3 Branch erzeugen ([lokales git](#))
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout -b 123_myfeature`
- 4 Modularisieren des Problems ([je kleiner die Teile, desto einfacher zu testen](#))
- 5 Tests definieren, d. h. **SOLL** und **IST** ([git commit...](#))
- 6 Teile als Stubs implementieren ([Top Down](#), [git commit...](#))
- 7 Füttere Funktion mit Testfällen
- 8 Implementiere, bis alle Tests validieren ([git commit...](#))
- 9 Merge zu master
 - `$ git checkout master`
 - `$ git pull`
 - `$ git checkout 123_myfeature`
 - `$ git rebase master`
 - `$ git checkout master`
 - `$ git merge 123_myfeature`

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Python Framework: unittest

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Prinzip: Eigenes Programm zum Testen

Python Framework: unittest

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Prinzip: Eigenes Programm zum Testen

- > Unterverzeichnis für Testcases: `t/`

Python Framework: unittest

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Prinzip: Eigenes Programm zum Testen

- > Unterverzeichnis für Testcases: `t/`
- > `import unittests`

Python Framework: unittest

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Prinzip: Eigenes Programm zum Testen

- > Unterverzeichnis für Testcases: `t/`
- > `import unittests`
- > Definiere Testklasse `Test_`

Python Framework: unittest

Fragen:

- > Welche Funktionen gibt es?
- > Wovon hängen die Funktionen ab?
- > Was sind die gewünschten Antworten für bestimmte Parameter?
- > Was wären pathologische Parameter?

Prinzip: Eigenes Programm zum Testen

- > Unterverzeichnis für Testcases: `t/`
- > `import unittests`
- > Definiere Testklasse `Test_`
- > Definiere Testmethoden pro Funktion `test_funktionsname`

Beispiel 2: Testcases

```
> addition(1, 1) == 2
> addition(2, 3) == 5
> addition(2.3, 3.3) == 5 (Vereinbarung)
> addition(2.5, 3.5) == 5 (Vereinbarung)
> addition('1', '1') == 2 (Vereinbarung)
> addition('a', 'b') == None (Vereinbarung)
```

Beispiel 2: Implementation mit unittest

```
gvim addition_test.py (chmod +x addition_test.py)
```

`assertEqual(IST, SOLL)` prüft auf “==”

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import unittest
import addition as a

class Test_addition(unittest.TestCase):

    def test_addition(self):
        self.assertEqual(a.addition(1, 1), 2)
        self.assertEqual(a.addition(2, 3), 5)
        self.assertEqual(a.addition(2.3, 3.3), 5)
        self.assertEqual(a.addition(2.5, 3.5), 5)
        self.assertEqual(a.addition('1', '1'), 2)
        self.assertEqual(a.addition('a', 'b'), None)
        return

if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(Test_addition)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

Beispiel 2: addition — Stub

```
gvim addition_test.py
```

- > Diese Funktion tut (noch) nichts
- > Funktionsbeschreibung als `docstring`
- > Parameter als `docstring`

Beispiel 2: addition — Stub

```
gvim addition_test.py
```

- > Diese Funktion tut (noch) nichts
- > Funktionsbeschreibung als `docstring`
- > Parameter als `docstring`

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertible to integer,  
    return None.  
  
    @param a: integer, cast to int()  
    @param b: integer, cast to int()  
    """  
    return
```

Beispiel 2: Erster Test

```
(2134) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 36, in test_addition  
    self.assertEqual(a.addition(1, 1), 2)
```

```
AssertionError: None != 2
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Beispiel 2: Erster Test

```
(2134) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 36, in test_addition
```

```
    self.assertEqual(a.addition(1, 1), 2)
```

```
AssertionError: None != 2
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Funktion verhält sich nicht wie gefordert

Beispiel 2: Erster Test

```
(2134) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 36, in test_addition
```

```
    self.assertEqual(a.addition(1, 1), 2)
```

```
AssertionError: None != 2
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Funktion verhält sich nicht wie gefordert

AssertionError: None != 2

Beispiel 2: Erster Test

```
(2134) ~/src/python> ./addition_test.py
test_addition (__main__.Test_addition) ... FAIL

=====
FAIL: test_addition (__main__.Test_addition)
-----
Traceback (most recent call last):
  File "./addition_test.py", line 36, in test_addition
    self.assertEqual(a.addition(1, 1), 2)
AssertionError: None != 2

-----

Ran 1 test in 0.000s

FAILED (failures=1)
```

Funktion verhält sich nicht wie gefordert

`AssertionError: None != 2`

> IST == None

> SOLL == 2

Beispiel 2: addition — Funktionalität ergänzen

Summenbildung einfügen

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertable to integer,  
    return -1.  
  
    @param a: integer, if string try to cast to int()  
    @param b: integer, if string try to cast to int()  
    """  
    sumab = a + b  
    return sumab
```

Beispiel 2: Zweiter Test

```
(2144) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 38, in test_addition
```

```
    self.assertEqual(a.addition(2.3, 3.3), 5)
```

```
AssertionError: 5.5999999999999996 != 5
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Beispiel 2: Zweiter Test

```
(2144) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 38, in test_addition
```

```
    self.assertEqual(a.addition(2.3, 3.3), 5)
```

```
AssertionError: 5.5999999999999996 != 5
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Return ist ein **float**: int-Konversion fehlt

- > Testcase: `assertEqual(a.addition(2.3, 3.3), 5)`
- > IST == 5.5999999999999996
- > SOLL == 5

Beispiel 2: addition — int-Konversion I

Konversion des Ergebnisses

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertible to integer,  
    return -1.  
  
    @param a: integer, if string try to cast to int()  
    @param b: integer, if string try to cast to int()  
    """  
    sumab = int(a + b)  
    return sumab
```

Beispiel 2: Dritter Test

```
(2138) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 38, in test_addition
```

```
    self.assertEqual(a.addition(2.5, 3.5), 5)
```

```
AssertionError: 6 != 5
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Beispiel 2: Dritter Test

```
(2138) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
```

```
FAIL: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 38, in test_addition
```

```
    self.assertEqual(a.addition(2.5, 3.5), 5)
```

```
AssertionError: 6 != 5
```

```
-----
```

```
Ran 1 test in 0.000s
```

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

Logischer Fehler: Returnwert ist falsch

- > Testcase: `assertEqual(a.addition(2.5, 3.5), 5)`
- > IST == 6
- > SOLL == 5

Beispiel 2: addition — int-Konversion II

Funktionalität ergänzen: int-Konversion

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertible to integer,  
    return -1.  
  
    @param a: integer, if string try to cast to int()  
    @param b: integer, if string try to cast to int()  
    """  
    inta = int(a)  
    intb = int(b)  
    sumab = inta + intb  
    return sumab
```

Beispiel 2: Vierter Test

```
(2148) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... ERROR
```

```
=====
```

```
ERROR: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
  File "./addition_test.py", line 41, in test_addition
```

```
    self.assertEqual(a.addition('a', 'b'), None)
```

```
  File "~/src/addition.py", line 38, in addition
```

```
    inta = int(a)
```

```
ValueError: invalid literal for int(): a
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
FAILED (errors=1)
```

Beispiel 2: Vierter Test

```
(2148) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... ERROR
```

```
=====
```

```
ERROR: test_addition (__main__.Test_addition)
```

```
-----
```

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

```
File "./addition_test.py", line 41, in test_addition
```

```
    self.assertEqual(a.addition('a', 'b'), None)
```

```
File "~/src/addition.py", line 38, in addition
```

```
    inta = int(a)
```

```
ValueError: invalid literal for int(): a
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
FAILED (errors=1)
```

Value Error

- > Testcase: `assertEqual(a.addition('a', 'b')), None`
- > Versuch, ein char nach int zu casten

Beispiel 2: addition — int-Konversion II

Catch Exceptions:

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertible to integer,  
    return -1.  
  
    @param a: integer, if string try to cast to int()  
    @param b: integer, if string try to cast to int()  
    """  
    inta = 0  
    intb = 0  
    try:  
        inta = int(a)  
    except ValueError:  
        return None  
    try:  
        intb = int(b)  
    except ValueError:  
        return None  
    sumab = inta + intb  
    return sumab
```

Beispiel 2: Fünfter Test

```
(2140) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... ok
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```

Alle Tests validieren. Alles Ok?

Funktionalität erweitern

Übergabe von Zahlen als String liefert neue Testcases:

- > `self.assertEqual(a.addition('2.3', '3.3'), 5)`
- > `self.assertEqual(a.addition('2.5', '3.5'), 5)`

Funktionalität erweitern

Übergabe von Zahlen als String liefert neue Testcases:

- > self.assertEqual(a.addition('2.3', '3.3'), 5)
- > self.assertEqual(a.addition('2.5', '3.5'), 5)

```
(2148) ~/src/python> ./addition_test.py
test_addition (__main__.Test_addition) ... FAIL
```

```
=====
FAIL: test_addition (__main__.Test_addition)
-----
```

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

```
  File "./addition_test.py", line 42, in test_addition
    self.assertEqual(a.addition('2.3', '3.3'), 5)
```

```
AssertionError: None != 5
```

```
-----
Ran 1 test in 0.000s
```

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

`int('2.3') => ValueError!`

Beispiel 3: Funktionserweiterung

```
def addition(a, b):  
    """  
    Given two integers a, b add them together and return the sum  
    as integer. If the input is not convertible to integer,  
    return -1.  
  
    @param a: integer, if string try to cast to int()  
    @param b: integer, if string try to cast to int()  
    """  
    inta = 0  
    intb = 0  
    try:  
        inta = int(float(a))  
    except ValueError:  
        return None  
    try:  
        intb = int(float(b))  
    except ValueError:  
        return None  
    sumab = inta + intb  
    return sumab
```

Neuer Test

```
(2148) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... ok
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```

```
(2148) ~/src/python> ./addition_test.py  
test_addition (__main__.Test_addition) ... ok
```

```
-----  
Ran 1 test in 0.000s
```

```
OK
```

Die Änderungen ändern das Verhalten für vorherige Tests nicht.
⇒ **Echte Erweiterung**

Do it yourself! — Trivial cases

Beispiel: bfe_conferenceinfo.py

- > trivial: bfe_fue.py: checks
- > trivial: bfe_idnumber.py: checks, pydoc
- > trivial: bfe_idnumber_src.py: checks
- > trivial: bfe_institute.py: checks
- > trivial: bfe_jcrlink.py: checks, pydoc
- > trivial: bfe_journalcover.py: checks, pydoc
- > trivial: bfe_recordurl.py: checks
- > trivial: bfe_StaffSubmission.py: checks, pydoc

Do it yourself! — Simple cases

Beispiel: bfe_conferenceinfo.py

- > simple: bfe_ApprovalState.py: [modularize](#), [pydoc](#), [checks](#)
- > simple: bfe_authlinking.py: [checks](#), [pydoc](#)
- > simple: bfe_authoridlist.py: [checks](#), [pydoc](#)
- > simple: bfe_classification.py: [checks](#), [pydoc](#)
- > simple: bfe_correctionrequest.py: [checks](#), [pydoc](#)
- > simple: bfe_csv.py: [checks](#), [pydoc](#)
- > simple: bfe_dblinkup.py: [checks](#)
- > simple: bfe_doctypes.py: [checks](#), [pydoc](#)
- > simple: bfe_endnoteheader.py: [checks](#), [pydoc](#)
- > simple: bfe_experiments.py: [checks](#), [pydoc](#)
- > simple: bfe_field_searchlinked.py: [checks](#), [pydoc](#)
- > simple: bfe_icon.py: [checks](#), [pydoc](#)
- > simple: bfe_JSVC.py: [checks](#), [pydoc](#)
- > simple: bfe_keywords.py: [modularize](#), [checks](#), [pydoc](#)
- > simple: bfe_modifylnk.py: [checks](#), [pydoc](#)
- > simple: bfe_openaccess.py: [checks](#), [pydoc](#)
- > simple: bfe_openurllink.py: [checks](#), [pydoc](#)
- > simple: bfe_publist.py: [checks](#)
- > simple: bfe_rsstoc.py: [checks](#), [pydoc](#)
- > simple: bfe_sfx.py: [checks](#), [pydoc](#)
- > simple: bfe_statkeyJS.py: [checks](#), [pydoc](#)
- > simple: bfe_topbanner.py: [checks](#), [pydoc](#)
- > simple: bfe_typelist.py: [checks](#), [pydoc](#)
- > simple: bfe_url_hgf.py: [checks](#), [pydoc](#)



Do it yourself! — medium

Beispiel: bfe_conferenceinfo.py

- > medium: bfe_experimentsJS.py: minor modularization, checks, pydoc
- > medium: bfe_fieldsort.py: introduce api, modularize, checks, pydoc
- > medium: bfe_journalreference.py: modularize, checks, pydoc
- > medium: bfe_linkedrecords.py: checks, pydoc
- > medium: bfe_openaire.py: checks
- > medium: bfe_statid.py: modularize, checks, pydoc
- > medium: bfe_whoami.py: checks, pydoc

Do it yourself! — complex

Beispiel: `bfe_conferenceinfo.py`

- > `complex: bfe_authors_hgf.py:` `modularize, checks, pydoc`
- > `complex: bfe_bibtex_hgf.py:` `modularize, checks, pydoc`
- > `complex: bfe_fulltext_mini.py:` `modularize, checks, pydoc`
- > `complex: bfe_grantsJS.py:` `modularize, checks, pydoc`
- > `complex: bfe_photos_hgf.py:` `modularize, checks, pydoc`
- > `complex: bfe_ris.py:` `modularize, checks, pydoc`

Do it yourself! — libraries

Beispiel: bfe_conferenceinfo.py

- > simple: libBibformat_hgf.py: [checks](#)
- > simple: libHelpers.py: [checks](#)
- > simple: libwebsubmit_hgf.py: [checks](#), [pydoc](#)
- > medium: libStatistics_hgf.py: [checks](#)
- > complex: libJSGetAllChildren.py: [checks](#), [pydoc](#)

Vielen Dank!



Alexander Wagner
Deutsches Elektronen-Synchrotron
Central Library

Tel.: +49-40-8998-1758
alexander.wagner@desy.de

<http://library.desy.de>