Informatik I: Einführung in die Programmierung

11. Programmentwicklung: Testen und Debuggen

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

29. Dezember 2022

1 Programmentwicklung

- Fehlertypen
- Syntaktische Fehler
- Laufzeitfehler
- Logische Fehler

Programm-
entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-
fassung

Wie kommen Fehler ins Programm?

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, . . .
- “Rund 50% des Programmieraufwands wird für die Identifikation und Beseitigung von Fehlern aufgewendet.”
- “The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” — Tom Cargill, Bell Labs
- Auch “fertige” Software hat noch 1–18 Fehler pro 1000 Zeilen Code!
- Wichtig: Werkzeuge für die Fehlersuche und für die Qualitätskontrolle durch automatisches Testen

Programm-
entwicklung
Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-
fassung

Beispiel

- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Ausdrucks über den ganzen Zahlen errechnet. Der Ausdruck wird durch einen Ausdrucksbaum repräsentiert.
- Beispiel: `Node ('*', Node ('+', leaf(2), leaf(5)), leaf(6))` $\mapsto$ 42
- Methode: Traversierung des Ausdrucksbaums.
- Annahme: der Baum ist nicht leer

Das Programm

Evaluating an Expression Tree

```
def expeval(tree)
    match tree:
        case Node('+', left, right):
            return expeval(left)+expval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(rigt)
        case Node('/', left, right):
            return expeval(left)/expeval(right))
```

Arten von Fehlern

UNI FREIBURG

Syntaxfehler

Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung. Meist einfach zu finden und zu reparieren.

Laufzeitfehler

Während der Ausführung passiert nichts (das Programm hängt) oder es gibt eine Fehlermeldung (Exception).

Logische Fehler

Alles „läuft“, aber die Ausgaben und Aktionen des Programms sind anders als erwartet. Das sind die gefährlichsten Fehler. Beispiel: *Mars-Climate-Orbiter*.

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Syntaktische Fehler

- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
  - Schlüsselwort als Variablennamen benutzt
  - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
  - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
  - Unbalancierte Klammern
  - = statt == in Booleschen Ausdrücken
  - Die Einrückung!
- Oft helfen Editoren mit Syntaxunterstützung.
- Wenn nichts anderes mehr hilft: Sukzessives Auskommentieren

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Das Beispielprogramm

- Dieses Programm enthält 2 Syntaxfehler.
- Das syntaktisch korrekte Programm:

```
Evaluating an Expression tree

def expeval(tree):
    match tree:
        case Node('+', left, right):
            return expeval(left)+expval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(rigt)
        case Node('/', left, right):
            return expeval(left)/expeval(right)
```

Programm-
entwicklung
Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).
- Es befindet sich in einer Endlosschleife.
  - **Beispiel**: in einer `while`-Schleife wird die Schleifenvariable nicht geändert!

→ Abbrechen mit Ctrl-C.

- Dann Fehler einkreisen und identifizieren (siehe Debugging)

Laufzeitfehler: Exceptions

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Wert hat einen anderen Typ als erwartet.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
  - KeyError: Ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch kennen).
  - AttributeError: Versuch ein nicht existentes Attribut anzusprechen.
    - **Beispiel**: Zugriff auf Attribut `rigt`
- Es gibt einen Stack-Backtrace und eine genaue Angabe der Stelle.

Checkliste Ursachenforschung I

- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche kleinere Eingabe, bei der Fehler ebenfalls auftritt.
  - **Beispiel**: Die Eingabe ist ein Baum
  - Tritt der Fehler bereits bei einem Teilbaum auf?
  - Liegt es an der Markierung der Wurzel?
  - Schneide Teilbäume (auf sinnvolle Art und Weise) ab um kleinere Eingaben zu erhalten.
- Erstelle einen Testfall aus der kleinen Eingabe und der erwarteten Ausgabe (s.u.).

Checkliste Ursachenforschung II

- Führe den so erstellten Testfall aus.
- Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.
- Gehe von der Fehlerstelle schrittweise rückwärts bis alles (Inhalte von Variablen und Attributen) wieder richtig erscheint.

⇒ Der Fehler wurde durch die letzte Anweisung manifestiert.
  - Enthält die Anweisung selbst einen Fehler?
  - Falls nicht: Warum wurde sie ausgeführt? Das kann an umschließenden (fehlerhaften) bedingten Anweisungen liegen!

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Fehlersuche

Start mit dem Beispielausdruck

```
>>> e = Node('*', Node('+', leaf(2), leaf(5)), leaf(6))
>>> print(expeval(e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

Verkleinern! Probiere linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

- Hoppla, ein anderer Fehler!
- Offenbar wird der Fall, dass der Baum ein Blatt ist, nicht korrekt behandelt!
- Abhilfe: Einfügen von `return tree.mark` am Ende.

Nach der Korrektur

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
2
```

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

- Kleinstes Beispiel, das den Fehler verursacht.
- `e.left` beginnt mit `'+'`, also muss dort der Fehler sein.
- Korrigiere dort `expval` nach `expeval`.

Nach der Korrektur

```
>>> print(e.left)
Node('+', Node(2, None, None), Node(5, None, None))
>>> print(expeval(e.left))
7
```

Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'rigt'
```

- Keiner der Teilbäume liefert noch einen Fehler.
- Problem muss an der Wurzel beim Operator `'*'` liegen.
- Korrigiere dort `rigt` nach `right`.

Nach der Korrektur

```
>>> print (expeval (e))
42
```

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Das korrigierte Programm

- Unser Programm enthält 3 Fehler, die zu Exceptions führen.
- Das korrekte Programm:

Evaluating an Expression Tree

```
def expeval(tree):
    match tree:
        case Node('+', left, right):
            return expeval(left)+expeval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(right)
        case Node('/', left, right):
            return expeval(left)/expeval(right)
        case Node (mark, _, _):
            return mark
```

Programm-
entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-
fassung

Logische Fehler: Unerfüllte Erwartungen

- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der Erwartung abweicht, die der Programmier hat.
  - **Beispiele:** Statt Addition wird eine Multiplikation durchgeführt, metrische und imperiale Messwerte werden ohne Konversion verglichen.
- Tatsächlich liegt hier erst dann ein Fehler vor, wenn das erwartete Verhalten (formal) spezifiziert war. Auch informelle Vorgaben können verletzt werden.
- Best practice: Beschreibe das erwartete Verhalten durch Beispiele / Testfälle.
- Unvollständig, aber besser als nichts!

Programm-entwicklung
Fehlertypen
Syntaktische Fehler
Laufzeitfehler
Logische Fehler
Debuggen
Tests
Ausblick
Zusammen-fassung

Logische Fehler in unserem Programm

- Gibt es logische Fehler in unserem Programm?
- Forderung: ganzzahlige Arithmetik, aber Operator „/“ liefert Gleitkommazahl!

Evaluating an Expression Tree

```
def expeval(tree):
    match tree:
        case Node('+', left, right):
            return expeval(left)+expeval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(right)
        case Node('/', left, right):
            return expeval(left)//expeval(right)
        case Node (mark, _, _):
            return mark
```

Rekapitulation

Diese Beispiele hätten alle Fehler identifiziert!

- Ein Beispiel für einen konstanten Ausdruck.
- Je ein Beispiel pro Operator.
- Für die Division ein Beispiel, wo ganzzahlig dividiert werden muss.
  Z.B. `Node ('/', leaf(5), leaf(3))`

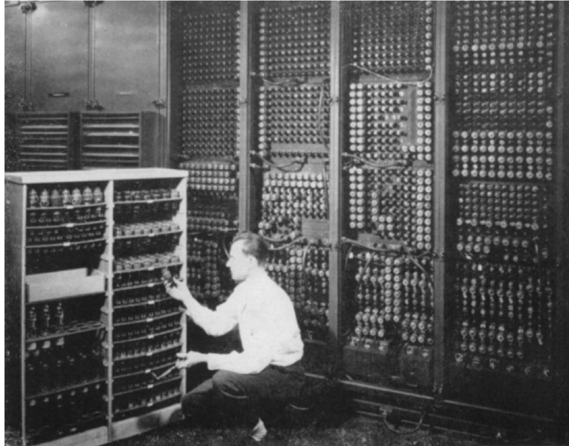
Welche Test sind erforderlich?

- Für jeden booleschen Ausdruck im Programm je ein Test, der den Ausdruck wahr bzw. falsch macht.
- Für jede Anweisung im Programm je ein Test, der zu ihrer Ausführung führt.
- (Kann eine solche Menge von Tests automatisch bestimmt werden?)

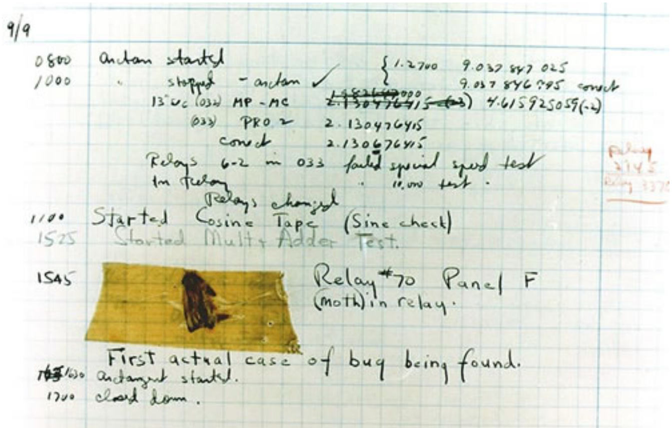
2 Debuggen

- Print-Anweisungen
- Debugger
- Debugging-Techniken

Programm-entwicklung
Debuggen
Print-Anweisungen
Debugger
Debugging-Techniken
Tests
Ausblick
Zusammen-fassung



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.



https://thenextweb.com/shareables/2013/09/18/the-very-first-computer-bug/

Debuggen = Käfer jagen und töten

Programm-entwicklung
Debuggen
Print-Anweisungen
Debugger
Debugging-Techniken
Tests
Ausblick
Zusammen-fassung

- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.
- Diese Käfer (oder andere Ursachen für Fehlfunktionen) zu finden heißt *debuggen*, im Deutschen manchmal *entwanzen*.
- Hat viel von Detektivarbeit (wer ist der Schuldige?)
- Aber nicht mystifizieren; vieles ist heute systematisiert und automatisierbar.
- Die Verbesserungen heißen Bugfixes – und sollten das Problem dann lösen!

Debugging-Techniken

Programm-
entwicklung
Debuggen
Print-Anweisungen
Debugger
Debugging-
Techniken
Tests
Ausblick
Zusammen-
fassung

Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Nachvollziehen der Berechung bis zum Fehler

1. Kleine Beispiele von Hand oder mit `pythontutor`
2. Modifikation des Programms zur Ausgabe von bestimmten Variablenwerten an bestimmten Stellen (Einfügen von `print`-Anweisungen)
3. Einsatz von Debugging-Werkzeugen:
   Post-Mortem-Analyse-Tools und Debugger

Debuggen mit Print-Anweisungen

UNI FREIBURG

Programm-entwicklung
Debuggen
Print-Anweisungen
Debugger
Debugging-Techniken
Tests
Ausblick
Zusammen-fassung

- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).
- In vielen Sprachen/Systemen können `print`-Anweisungen eingefügt werden.
- Einfachste Möglichkeit das Verhalten eines Programmes zu beobachten.
  - **Achtung**: Oft nicht angebracht, weil zusätzliche Ausgaben das Verhalten (speziell das Zeitverhalten) signifikant ändern können!
- Eine generalisierte Form ist das *Logging*, bei dem `prints` generell im Code integriert sind und mit Schaltern an- und abgestellt werden können.

Debugger – generell

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal in der Kellertabelle)
2. *Interaktive Debugger*
   - Setzen von Breakpoints (u.U. konditional)
   - Inspektion des Programmzustands (Variablenbelegung)
   - Ändern des Zustands
   - Einzelschrittausführung (Stepping / Tracing):

     Step in: Mache einen Schritt, ggfs. in eine Funktion hinein
     Step over: Mache einen Schritt, führe dabei ggfs. eine Funktion aus
     Step out: Beende den aktuellen Funktionsaufruf
     Go/Continue: Starte Ausführung bzw. setze fort
     Quit: Beendet alles.

Debugging-Techniken

1. Formuliere eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
2. Instrumentiere die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
3. Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.
4. Formuliere einen Bugfix erst dann, wenn das Problem verstanden ist. Einfache Lösungen sind oft nicht hilfreich.
5. Teste nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.
6. Weitere Tests laufen lassen (s.u.).
7. Wenn es nicht weiter geht: frische Luft und eine Tasse Kaffee hilft!

Programm-entwicklung
Debuggen
Print-Anweisungen
Debugger
Debugging-Techniken
Tests
Ausblick
Zusammen-fassung

3 Automatische Tests

- Testgetriebene Entwicklung
- Unittests
- pytest

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung

Testfälle erzeugen

- Testen eines Programms heißt, fehlerhaftes Verhalten zu provozieren.
- Ein Testfall besteht aus einer Eingabe und dem erwarteteten Ergebnis. Die Testfälle bilden eine Testsuite.
- Schon vor dem Programmieren systematisch Testfälle erstellen:
  - Basisfälle, Grenzfälle (z.B. erstes bzw. letztes Element einer Datenstruktur).
  - Jede Anweisung soll durch einen Testfall abgedeckt (d.h. ausgeführt) werden.
  - Versuche Eingaben zu finden, die die Bedingungen im Programm unabhängig voneinander wahr bzw. falsch machen (soweit möglich).
- Beim Programmieren: Tests, die zur Entdeckung eines Fehlers geführt haben, müssen aufbewahrt werden!

Testfälle aus Typen

Viele Testfälle ergeben sich schon aus der Typsignatur!

- `list[T]`: ein Test mit leerer Liste, mehrere Tests mit nicht-leeren Listen verschiedener Länge mit Elementen variiert gemäß `T`
- `tuple[T1,T2,...]`: Kombiniere Testfälle gemäß `T1`, `T2`, dots
- `Optional[T]`: ein Test mit `None`, Tests mit Werten variiert gemäß `T`
- `T1|T2|...`: Tests mit Werten gemäß `T1`, Tests mit Werten gemäß `T2`, ...
- `Any`: Tests zumindest mit allen Grundtypen (`int`, `float`, `str`, `bool`, `...`)
- Datenklasse: Generiere Tests gemäß der Typen der Attribute und kombiniere
- `Literal[V1,V2,...]`: Die Werte dieses Typs sind genau `V1`, `V2` usw. Also je ein Testfall mit `V1`, `V2`, ...

Testen während der Entwicklung

Testgetriebene Entwicklung

- Formuliere zu Beginn mit dem Auftraggeber Testfälle, die nach und nach während der Entwicklung erfüllt werden.
- Der Fortschritt der Entwicklung des Systems kann anhand der Anzahl der bestandenen Tests gemessen werden.

Regressionstest

Wiederholung von Tests um sicher zu stellen, dass nach Änderungen der Software keine neuen (oder alten) Fehler eingeschleppt wurden.

UNI FREIBURG

Programm-entwicklung
Debuggen
Tests
Testgetriebene Entwicklung
Unittests
pytest
Ausblick
Zusammen-fassung

Unittest

- Testfälle für Teile eines Systems (Modul, Funktion, usw.).
- Überprüfen die Funktion der Einzelteile.
- Automatisch ausführen nach Änderung / beim Einchecken $\rightarrow$ Regressionstests!

Automatisierung von Tests in Python

- Mehrere Werkzeuge zur Automatisierung von Tests.
- `pytest` — Beispiel für ein umfassendes Framework
- Installation durch “`pip3 install pytest`”.

Verwendung von `pytest` (1)

- Operiert auf dem Modullevel.
- Für jede FUT (function under test) werden Testfunktionen geschrieben.
- Der Name einer Testfunktion beginnt mit "`test_...`".
- Typischerweise in Dateien namens "`test_`*xyz*`.py`".
- Die Testfunktion enthält Aufrufe der FUT, wobei die erwarteten Rückgabewerte als Assertions formuliert sind.
- Syntax der `assert`-Anweisung:
  `assert` *Bedingung* `[`, *String*`]`
- `assert` überprüft die *Bedingung*.
  - Wenn sie erfüllt ist, wird die Testfunktion weiter ausgeführt.
  - Anderenfalls Abbruch mit Exception und Ausgabe von *String*.

Programm-entwicklung
Debuggen
Tests
Testgetriebene Entwicklung
Unittests
pytest
Ausblick
Zusammen-fassung

Verwendung von pytest (2)

Beispiel für einen Testfall

```
import pytest
###
def test_expeval_b():
    """Test of expeval that fails."""
    exp = Node('*', Node('+', leaf(3), leaf(5)),
                    leaf(6))
    assert expeval(exp) == 42
```

- Ausführung mit `pytest expeval.py`

Programm-entwicklung
Debuggen
Tests
Testgetriebene Entwicklung
Unittests
pytest
Ausblick
Zusammen-fassung

Verwendung von pytest (3)

Die Ausgabe in obigem Beispiel:

```
======================== test session starts =========================

expeval.py::test_expeval_b FAILED

============================== FAILURES ==============================
__________________________ test_expeval_b ___________________________

    def test_expeval_b():
        """Test of expeval that fails."""
        exp = Node('*', Node('+', leaf(3), leaf(5)),
                         leaf(6))
>       assert expeval(exp) == 42
E       assert 48 == 42
E        +  where 48 = expeval(Node('*', Node('+', leaf(3), leaf(5)), leaf(6)))

expeval.py:50: AssertionError
================= 1 failed, 1 passed in 0.02 seconds =================
```

Verwendung von pytest (4)

Testsuite wie oben besprochen

```
import pytest
###
def test_expeval_1():
    """Testing expeval from example."""
    e = Node('*', Node('+', leaf(2), leaf(5)), leaf(6))
    assert expeval (e.left.left) == 2
    assert expeval (e.left) == 7
    assert expeval (e) == 42

def test_expeval_2():
    """Testing logical bug in expeeval"""
    assert expeval (Node ('/', leaf(2), leaf(3))) == 0
```

Verwendung von pytest (5)

Ausgabe vor Bugfix des logischen Fehlers

```
Peters-MacBook-Pro:python thiemann$ pytest trees.py
=============================== test session starts ================================
platform darwin -- Python 3.7.0, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /Users/thiemann/svn/teaching/info2018/slides/python, inifile:
collected 2 items

trees.py .F                                                                  [100%]

==================================== FAILURES ======================================
________________________________ test_expreval_2 __________________________________

    def test_expreval_2():
>       assert expreval (Node ('/', leaf(2), leaf(3))) == 0
E       AssertionError: assert 0.6666666666666666 == 0
E        +  where 0.6666666666666666 = expreval(<trees.Node object at 0x10f4ed978>)
E        +    where <trees.Node object at 0x10f4ed978> = Node('/', <trees.Node obje
ct at 0x10f4ede48>, <trees.Node object at 0x10f4ed898>)
E        +      where <trees.Node object at 0x10f4ede48> = leaf(2)
E        +      and   <trees.Node object at 0x10f4ed898> = leaf(3)

trees.py:35: AssertionError
====================== 1 failed, 1 passed in 0.07 seconds ========================
```


Programm-entwicklung

Debuggen

Tests

Testgetriebene Entwicklung

Unittests

pytest

Ausblick

Zusammen-fassung

Verwendung von pytest (6)

Programm-entwicklung
Debuggen
Tests
Testgetriebene Entwicklung
Unittests
pytest
Ausblick
Zusammen-fassung

Ausgabe nach Bugfix

```
Peters-MacBook-Pro:python thiemann$ pytest trees.py
=============================== test session starts ================================
platform darwin -- Python 3.7.0, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /Users/thiemann/svn/teaching/info2018/slides/python, inifile:
collected 2 items

trees.py ..                                                                  [100%]

============================ 2 passed in 0.02 seconds ==============================
```

4 Ausblick

Programm-
entwicklung

Debuggen

Tests

Ausblick

Zusammen-
fassung

Aublick: Fehlerfreies Programmieren?

- Können wir (von Menschen erschaffener) Software für AKWs, Flugzeuge, Autos, usw. vertrauen?
- Aktive Forschungsrichtungen innerhalb der Informatik
  - Verbesserung der Testmethoden — *keine* Garantie für Korrektheit!
  - Maschinelle Beweise (d.h. für alle Fälle gültig) der Korrektheit.
- Kein Schutz gegen Fehler in der Spezifikation gegen die geprüft wird!
- Auch das Beweissystem kann Fehler besitzen (aber Selbstanwendung).

→ In jedem Fall *reduzieren wir die Fehlerwahrscheinlichkeit*!

- Heute wird auch über *probabilistische Korrektheit* nachgedacht und geforscht.

5 Zusammenfassung

Programm-
entwicklung

Debuggen

Tests

Ausblick

Zusammen-
fassung

Zusammenfassung

- Fehlerfreie Programmentwicklung gibt es nicht.
- Wir unterscheiden zwischen syntaktischen Fehlern, Laufzeitfehlern und logischen Fehlern.
- Fehlersuche: Debuggen
- Checkliste Ursachenforschung
- Der Debuggingprozess: Eingabe minimieren, Testfall erstellen, Werte beobachten, Hypothese entwickeln
- Fehler verstehen und beseitigen: Bugfix.
- Testgetriebene Entwicklung, Erstellung sinnvoller Testfälle
- Automatische Tests erhöhen die Qualität von Software!
- `pytest` ist ein Werkzeug zur Automatisierung von Regressionstests.