Informatik I: Einführung in die Programmierung

12. Programmentwicklung: Testen und Debuggen

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

18. November 2016

UNI FREIBURG

Programmentwicklung

Wie kommen Fehler ins Programm?

UNI FREIBURG

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.

Wie kommen Fehler ins Programm?

UNI FREIBURG

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Tatsächlich ist ja oft nicht einmal klar, was das „Richtige“ ist.

Wie kommen Fehler ins Programm?

UNI FREIBURG

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Tatsächlich ist ja oft nicht einmal klar, was das „Richtige“ ist.
- Selbst für die klaren Fälle: Schreibfehler, zu kurz gedacht, falsche Annahmen

Wie kommen Fehler ins Programm?

UNI FREIBURG

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Tatsächlich ist ja oft nicht einmal klar, was das „Richtige“ ist.
- Selbst für die klaren Fälle: Schreibfehler, zu kurz gedacht, falsche Annahmen
- Man schätzt, dass rund 50% des Programmieraufwands für die Identifikation und Beseitigung von Fehlern aufgewendet wird.

Wie kommen Fehler ins Programm?

UNI FREIBURG

- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Tatsächlich ist ja oft nicht einmal klar, was das „Richtige“ ist.
- Selbst für die klaren Fälle: Schreibfehler, zu kurz gedacht, falsche Annahmen
- Man schätzt, dass rund 50% des Programmieraufwands für die Identifikation und Beseitigung von Fehlern aufgewendet wird.
- Wichtig: Tools für die Fehlersuche und für die Qualitätskontrolle durch automatisches Testen

Beispiel

UNI FREIBURG

- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Integer-Ausdrucks, der durch ein Ausdrucksbaum beschrieben wird, errechnet.

Beispiel

UNI FREIBURG

- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Integer-Ausdrucks, der durch ein Ausdrucksbaum beschrieben wird, errechnet.
- Zum Beispiel: `['*', ['+', [2, None, None], [5, None, None]], [6, None, None]]` $\mapsto$ 42

Beispiel

UNI FREIBURG

- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Integer-Ausdrucks, der durch ein Ausdrucksbaum beschrieben wird, errechnet.
- Zum Beispiel: `['*', ['+', [2, None, None], [5, None, None]], [6, None, None]]` $\mapsto$ 42
- Methode: Rekursive Traversierung des Ausdrucksbaums.

Beispiel

UNI FREIBURG

- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Integer-Ausdrucks, der durch ein Ausdrucksbaum beschrieben wird, errechnet.
- Zum Beispiel: `['*', ['+', [2, None, None], [5, None, None]], [6, None, None]]` $\mapsto$ 42
- Methode: Rekursive Traversierung des Ausdrucksbaums.

Evaluating an expression tree

```
def expreval(tree)
    if tree[0] == '+':
        return expreval(tree[1])+exprval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[3])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2]))
```

Arten von möglichen Fehlern

UNI FREIBURG

Syntaktische Fehler: Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung und sie sind meist einfach zu finden und zu reparieren.

Arten von möglichen Fehlern

UNI FREIBURG

Syntaktische Fehler: Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung und sie sind meist einfach zu finden und zu reparieren.

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

Arten von möglichen Fehlern

UNI FREIBURG

Syntaktische Fehler: Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung und sie sind meist einfach zu finden und zu reparieren.

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

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

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)

Syntaktische Fehler

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:

Syntaktische Fehler

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-Schlüsselwort als Variablennamen benutzt

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-Schlüsselwort als Variablennamen benutzt
  - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-Schlüsselwort als Variablennamen benutzt
  - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
  - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-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

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-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

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-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!

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-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 Python-Syntaxunterstützung.

Syntaktische Fehler

UNI FREIBURG

- Der Interpreter gibt Zeile und Punkt an, an dem der Fehler fest gestellt wurde (in IDLE wird die Zeile markiert)
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische mögliche Fehler:
  - Python-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 Python-Syntaxunterstützung.
- Im schlechtesten Fall: Sukzessives Löschen und Probieren

Das Beispielprogramm

UNI FREIBURG

- Unser Programm enthält 2 syntaktische Fehler.

Evaluating an expression tree

```
def expreval(tree)
    if tree[0] == '+':
        return expreval(tree[1])+exprval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[3])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2]))
```

Das Beispielprogramm

UNI FREIBURG

- Unser Programm enthält 2 syntaktische Fehler.
- Das syntaktisch korrekte Programm:

Evaluating an expression tree

```
def expreval(tree):
    if tree[0] == '+':
        return expreval(tree[1])+exprval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[3])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2]))
```

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggfs. Timeout vorsehen).

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggfs. Timeout vorsehen).
- Es befindet sich in einer Endlosschleife oder Endlosrekursion (d.h. kommt nie zum Basisfall, in Python wird bei Rekursion schnell abgebrochen!)

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggfs. Timeout vorsehen).
- Es befindet sich in einer Endlosschleife oder Endlosrekursion (d.h. kommt nie zum Basisfall, in Python wird bei Rekursion schnell abgebrochen!)
  - **Beispiel**: in einer `while`-Schleife wird die Schleifenvariable nicht geändert!

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggfs. Timeout vorsehen).
- Es befindet sich in einer Endlosschleife oder Endlosrekursion (d.h. kommt nie zum Basisfall, in Python wird bei Rekursion schnell abgebrochen!)
  - **Beispiel**: in einer `while`-Schleife wird die Schleifenvariable nicht geändert!

$\rightarrow$ Abbrechen mit Ctrl-C oder *Restart Shell* in IDLE.

Laufzeitfehler: Das Programm „hängt“

UNI FREIBURG

- Das Programm wartet auf eine Eingabe ($\rightarrow$ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggfs. Timeout vorsehen).
- Es befindet sich in einer Endlosschleife oder Endlosrekursion (d.h. kommt nie zum Basisfall, in Python wird bei Rekursion schnell abgebrochen!)
  - **Beispiel**: in einer `while`-Schleife wird die Schleifenvariable nicht geändert!

$\rightarrow$ Abbrechen mit Ctrl-C oder *Restart Shell* in IDLE.

- Dann Fehler einkreisen und identifizieren (siehe Debugging)

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3

Laufzeitfehler: Exceptions

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3
  - KeyError: Ist ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch).

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3
  - KeyError: Ist ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch).
  - AttributeError: Ein nicht existentes Attribut wurde versucht anzusprechen (lernen wir noch).

Laufzeitfehler: Exceptions

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3
  - KeyError: Ist ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch).
  - AttributeError: Ein nicht existentes Attribut wurde versucht anzusprechen (lernen wir noch).
- Es gibt einen Stack-Backtrace und eine genaue Angabe der Stelle.

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3
  - KeyError: Ist ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch).
  - AttributeError: Ein nicht existentes Attribut wurde versucht anzusprechen (lernen wir noch).
- Es gibt einen Stack-Backtrace und eine genaue Angabe der Stelle.

→ Nachdenken oder Fehler durch Ausgabe von Variablenwerten versuchen zu verstehen

Laufzeitfehler: Exceptions

UNI FREIBURG

- Typische Fehler:
  - NameError: Benutzung einer nicht initialisierten Variablen.
  - TypeError: Anderer Typ erwartet als dann tatsächlich benutzt wird.
  - IndexError: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
    - **Beispiel**: Zugriff auf Teilbaum mit Indexwert 3
  - KeyError: Ist ähnlich wie IndexError, aber für *Dictionaries* (lernen wir noch).
  - AttributeError: Ein nicht existentes Attribut wurde versucht anzusprechen (lernen wir noch).
- Es gibt einen Stack-Backtrace und eine genaue Angabe der Stelle.

→ Nachdenken oder Fehler durch Ausgabe von Variablenwerten versuchen zu verstehen

- Dann Fehler einkreisen und identifizieren (siehe Debugging).

Das korrigierte Programm

UNI FREIBURG

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

Evaluating an expression tree

```
def expreval(tree):
    if tree[0] == '+':
        return expreval(tree[1])+exprval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[3])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2])
```

Das korrigierte Programm

UNI FREIBURG

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

Evaluating an expression tree

```
def expreval(tree):
    if tree[0] == '+':
        return expreval(tree[1])+expreval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[2])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2])
    else:
        return tree[0]
```

Semantische Fehler: Unerfüllte Erwartungen

UNI FREIBURG

- Ein semantischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der Erwartung abweicht, die der Programmier hat.

Semantische Fehler: Unerfüllte Erwartungen

UNI FREIBURG

- Ein semantischer 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 Werte werden ohne Konversion verglichen.

Semantische Fehler: Unerfüllte Erwartungen

UNI FREIBURG

- Ein semantischer 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 Werte werden ohne Konversion verglichen.
- Tatsächlich kann man hier eigentlich erst von einem Fehler sprechen, wenn man das erwartete Verhalten formal spezifiziert hatte. Aber auch informelle Vorgaben können natürlich verletzt werden.

Semantische Fehler: Unerfüllte Erwartungen

UNI FREIBURG

- Ein semantischer 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 Werte werden ohne Konversion verglichen.
- Tatsächlich kann man hier eigentlich erst von einem Fehler sprechen, wenn man das erwartete Verhalten formal spezifiziert hatte. Aber auch informelle Vorgaben können natürlich verletzt werden.
- Auf jeden Fall kann man das erwartete Verhalten (partiell) durch Beispiele einfach beschreiben.

Semantische Fehler: Unerfüllte Erwartungen

UNI FREIBURG

- Ein semantischer 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 Werte werden ohne Konversion verglichen.
- Tatsächlich kann man hier eigentlich erst von einem Fehler sprechen, wenn man das erwartete Verhalten formal spezifiziert hatte. Aber auch informelle Vorgaben können natürlich verletzt werden.
- Auf jeden Fall kann man das erwartete Verhalten (partiell) durch Beispiele einfach beschreiben.

→ Durch Nachdenken versuchen, den relevanten Programmteil zu identifizieren, dann einkreisen (siehe Debugging).

Semantische Fehler in unserem Programm

UNI FREIBURG

- Gibt es semantische Fehler in unserem Programm?
- Wir hatten Integer-Arithmetik gefordert, aber „/“ liefert eine Gleitkommazahl!

Evaluating an expression tree

```
def expreval(tree):
    if tree[0] == '+':
        return expreval(tree[1])+expreval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[2])
    elif tree[0] == '/':
        return expreval(tree[1])/expreval(tree[2])
    else:
        return tree[0]
```

Semantische Fehler in unserem Programm

UNI FREIBURG

- Gibt es semantische Fehler in unserem Programm?
- Wir hatten Integer-Arithmetik gefordert, aber „/“ liefert eine Gleitkommazahl!

Evaluating an expression tree

```
def expreval(tree):
    if tree[0] == '+':
        return expreval(tree[1])+expreval(tree[2])
    elif tree[0] == '-':
        return expreval(tree[1])-expreval(tree[2])
    elif tree[0] == '*':
        return expreval(tree[1])*expreval(tree[2])
    elif tree[0] == '/':
        return expreval(tree[1])//expreval(tree[2])
    else:
        return tree[0]
```

UNI FREIBURG

Debuggen

Debuggen = Käfer jagen und töten

UNI FREIBURG

- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.

Debuggen = Käfer jagen und töten

UNI FREIBURG

- 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*.

Debuggen = Käfer jagen und töten

UNI FREIBURG

- 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?)

Debuggen = Käfer jagen und töten

UNI FREIBURG

- 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?)
- Die Verbesserungen heißen Bugfixes – und sollten das Problem dann lösen!

Debuggen a la Mr.Robot

UNI FREIBURG

Most coders think debugging software is about fixing a mistake, but that is bullshit.

Debuggen a la Mr.Robot

UNI FREIBURG

Most coders think debugging software is about fixing a mistake, but that is bullshit. Debugging is actually all about finding the bug, about understanding why the bug was there to begin with, about knowing that its existence was no accident. It came to you to deliver a message, like an unconscious bubble floating to the surface, popping with a revelation you've secretly known all along.

Debug-Techniken

UNI FREIBURG

Für das Debugging gibt es verschiedene Methoden:

1. Nachdenken (inklusive mentaler Simulation der Programmausführung oder `pythontutor`)

Debug-Techniken

UNI FREIBURG

Für das Debugging gibt es verschiedene Methoden:

1. Nachdenken (inklusive mentaler Simulation der Programmausführung oder `pythontutor`)
2. Modifikation des Programms zur Ausgabe von bestimmten Variablenwerten an bestimmten Stellen (Einfügen von `print`-Anweisungen)

Debug-Techniken

UNI FREIBURG

Für das Debugging gibt es verschiedene Methoden:

1. Nachdenken (inklusive mentaler Simulation der Programmausführung oder `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-Statements

UNI FREIBURG

- Wenn ein System ein abweichendes Verhalten zeigt, versucht man interne Werte zu messen (z.B. bei Hardware mit einem Oszilloskop)

Debuggen mit Print-Statements

UNI FREIBURG

- Wenn ein System ein abweichendes Verhalten zeigt, versucht man interne Werte zu messen (z.B. bei Hardware mit einem Oszilloskop)
- In Python (und vielen anderen Sprachen/Systemen) kann man einfach `print`-Anweisungen einfügen und das Programm dann laufen lassen.

Debuggen mit Print-Statements

UNI FREIBURG

- Wenn ein System ein abweichendes Verhalten zeigt, versucht man interne Werte zu messen (z.B. bei Hardware mit einem Oszilloskop)
- In Python (und vielen anderen Sprachen/Systemen) kann man einfach `print`-Anweisungen einfügen und das Programm dann laufen lassen.
- Ist die einfachste Möglichkeit, Verhalten eines Programmes zu beobachten, speziell wenn man bereits einen Verdacht hat.

Debuggen mit Print-Statements

UNI FREIBURG

- Wenn ein System ein abweichendes Verhalten zeigt, versucht man interne Werte zu messen (z.B. bei Hardware mit einem Oszilloskop)
- In Python (und vielen anderen Sprachen/Systemen) kann man einfach `print`-Anweisungen einfügen und das Programm dann laufen lassen.
- Ist die einfachste Möglichkeit, Verhalten eines Programmes zu beobachten, speziell wenn man bereits einen Verdacht hat.
  - **Achtung**: Solche zusätzlichen Ausgaben können natürlich das Verhalten (speziell das Zeitverhalten) signifikant ändern!

Debuggen mit Print-Statements

UNI FREIBURG

- Wenn ein System ein abweichendes Verhalten zeigt, versucht man interne Werte zu messen (z.B. bei Hardware mit einem Oszilloskop)
- In Python (und vielen anderen Sprachen/Systemen) kann man einfach `print`-Anweisungen einfügen und das Programm dann laufen lassen.
- Ist die einfachste Möglichkeit, Verhalten eines Programmes zu beobachten, speziell wenn man bereits einen Verdacht hat.
  - **Achtung**: Solche zusätzlichen Ausgaben können natürlich das Verhalten (speziell das Zeitverhalten) signifikant ändern!
- Eine generalisierte Form ist das *Logging*, bei dem man `prints` generell in seinen Code integriert und dann Schalter hat, um das Loggen an- und abzustellen.

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
2. *Interaktive Debugger*

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
2. *Interaktive Debugger*
   - Setzen von Breakpoints (u.U. konditional)

Debugger – generell

UNI FREIBURG

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

Debugger – generell

UNI FREIBURG

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

Debugger – generell

UNI FREIBURG

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

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
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

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
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

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 im Stapeldiagramm)
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

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
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 Programmausführung bzw. setze fort

Debugger – generell

UNI FREIBURG

1. *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler
   - Stack Backtrace wie in Python
   - Früher: Speicherbelegung (Hex-Dump)
   - Heute: Variablenbelegung (global und lokal im Stapeldiagramm)
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 Programmausführung bzw. setze fort

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.
   - *Stack Viewer*: Erlaubt eine Post-Mortem-Analyse des letzten durch eine Exception beendeten Programmlaufs.

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.
   - *Stack Viewer*: Erlaubt eine Post-Mortem-Analyse des letzten durch eine Exception beendeten Programmlaufs.
   - *Debugger*: Startet den Debug-Modus:

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.
   - *Stack Viewer*: Erlaubt eine Post-Mortem-Analyse des letzten durch eine Exception beendeten Programmlaufs.
   - *Debugger*: Startet den Debug-Modus:
     - Es erscheint ein Fenster, in dem der Aufruf-Stapel, globale und lokale Variablen angezeigt werden. Ggfs. wird auch der aktuelle Quellcode angezeigt.

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.
   - *Stack Viewer*: Erlaubt eine Post-Mortem-Analyse des letzten durch eine Exception beendeten Programmlaufs.
   - *Debugger*: Startet den Debug-Modus:
     - Es erscheint ein Fenster, in dem der Aufruf-Stapel, globale und lokale Variablen angezeigt werden. Ggfs. wird auch der aktuelle Quellcode angezeigt.
     - Man kann Breakpoints setzen, indem man im Quellcode eine Zeile rechts-klickt (Mac: Ctrl-Klick).

Debugger – in Python

UNI FREIBURG

1. `pdb` ist ein Konsolen-orientierter Debugger, der auch Post-Mortem-Analyse anbietet (siehe `http://docs.python.org/3.3/library/pdb.html`).
2. IDLE enthält einen weniger mächtigen, aber einfach zu bedienenden GUI-Debugger. Im Debug-Menü:
   - *Goto File/Line*: Wenn der Cursor in einer Traceback-Zeile steht, springt der Editor zur angegebenen Stelle.
   - *Stack Viewer*: Erlaubt eine Post-Mortem-Analyse des letzten durch eine Exception beendeten Programmlaufs.
   - *Debugger*: Startet den Debug-Modus:
     - Es erscheint ein Fenster, in dem der Aufruf-Stapel, globale und lokale Variablen angezeigt werden. Ggfs. wird auch der aktuelle Quellcode angezeigt.
     - Man kann Breakpoints setzen, indem man im Quellcode eine Zeile rechts-klickt (Mac: Ctrl-Klick).
     - Stepping mit den Go/Step usw. Knöpfen.

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)
3. Versuchen Sie zu verstehen, wie es zu dem Fehler kommt: Was ist die tiefere Ursache?

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)
3. Versuchen Sie zu verstehen, wie es zu dem Fehler kommt: Was ist die tiefere Ursache?
4. Formulieren Sie einen Bugfix erst dann, wenn Sie glauben, das Problem verstanden zu haben. Einfache Lösungen sind oft nicht hilfreich.

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)
3. Versuchen Sie zu verstehen, wie es zu dem Fehler kommt: Was ist die tiefere Ursache?
4. Formulieren Sie einen Bugfix erst dann, wenn Sie glauben, das Problem verstanden zu haben. Einfache Lösungen sind oft nicht hilfreich.
5. Testen Sie nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)
3. Versuchen Sie zu verstehen, wie es zu dem Fehler kommt: Was ist die tiefere Ursache?
4. Formulieren Sie einen Bugfix erst dann, wenn Sie glauben, das Problem verstanden zu haben. Einfache Lösungen sind oft nicht hilfreich.
5. Testen Sie nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.
6. Lassen Sie weitere Tests laufen (s.u.).

Debugging-Techniken

UNI FREIBURG

1. Formulieren Sie eine Hypothese, warum der Fehler auftritt und an welcher Stelle des Programms sich dieser Fehler manifestiert!
2. Konzentrieren Sie sich auf diese Stelle und instrumentieren Sie die Stelle (Breakpoints oder `print`-Anweisungen)
3. Versuchen Sie zu verstehen, wie es zu dem Fehler kommt: Was ist die tiefere Ursache?
4. Formulieren Sie einen Bugfix erst dann, wenn Sie glauben, das Problem verstanden zu haben. Einfache Lösungen sind oft nicht hilfreich.
5. Testen Sie nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.
6. Lassen Sie weitere Tests laufen (s.u.).
7. Wenn es nicht weiter geht, stehen Sie auf, gehen Sie an die frische Luft und trinken eine Tasse Kaffee!

UNI FREIBURG

Automatische Tests

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann
- Systematisch testen:

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann
- Systematisch testen:
  - Basisfälle und andere Grenzfälle

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann
- Systematisch testen:
  - Basisfälle und andere Grenzfälle
  - Decken Sie jeden Zweig in Ihrem Code durch einen Test ab

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann
- Systematisch testen:
  - Basisfälle und andere Grenzfälle
  - Decken Sie jeden Zweig in Ihrem Code durch einen Test ab
  - Gibt es Interaktionen zwischen verschiedenen Programmteilen, versuchen Sie auch diese abzudecken

Testfälle erzeugen

UNI FREIBURG

- Um fehlerhaftes Verhalten zu provozieren, müssen wir das Programm natürlich testen.
- Man startet das Programm auf Daten (bzw. interagiert) und wartet, bis es crasht.
- Am besten systematisch Testfälle sammeln, die man immer wieder für das Programm nutzen kann
- Systematisch testen:
  - Basisfälle und andere Grenzfälle
  - Decken Sie jeden Zweig in Ihrem Code durch einen Test ab
  - Gibt es Interaktionen zwischen verschiedenen Programmteilen, versuchen Sie auch diese abzudecken
  - Wichtig: Tests, die zur Entdeckung eines Fehlers geführt haben, sollten auf jeden Fall für spätere Wiederholungen aufbewahrt werden

Testgetriebene Entwicklung

UNI FREIBURG

- Regressionstest: Wiederholung von Tests um sicher zu stellen, dass nach Änderungen der Software keine neuen (oder alten) Fehler eingeschleppt wurden.
- Eine Möglichkeit die Entwicklung eines Systems voran zu treiben ist, als erstes Tests zu formulieren, die dann Stück für Stück erfüllt werden.
- Die Qualität des Systems kann dann mit Hilfe der Anzahl der bestandenen Tests gemessen werden.

Modultests oder Unittests

UNI FREIBURG

- Um zu garantieren, dass die Einzelteile eines System funktionieren, benutzt man sogenannte Unittests.
- Dieses sind Testfälle für Teile eines Systems (Modul, Funktion, usw.).
- Normalerweise werden diese automatisch ausgeführt.
- In Python gibt es u.a. zwei Werkzeuge/Module:
  1. `unittest` - ein komfortables (aber auch aufwändig zu bedienendes) Modul für die Formulierung und Verwaltung von Unit-Tests
  2. `doctest` - ein einfaches Modul, das Testfälle aus den docstrings extrahiert und ggfs. automatisch ausführt.

doctest-Modul

UNI FREIBURG

- Fügen Sie Testfälle aus Shellinteraktionen in ihre `docstrings` ein, z.B. so:

Testbeispiel

```
import doctest

def expreval(tree):
"""Takes an integer expression tree and evaluates it.

>>> expreval([5, None, None])
5
>>> expreval(['*', [7, None, None], [6, None, None]])
42

"""
...
```

Der __main__ -Trick

UNI FREIBURG

- Nach dem Laden des Programms kann man alle solche Tests ausführen lassen.

Python-Interpreter

```
>>> ==================== RESTART ================
>>> doctest.testmod()
TestResults(failed=0, attempted=30)
```

- Man kann dies automatisieren, indem man am Ende der Datei folgendes hinschreibt:

Testbeispiel

```
if __name__ == "__main__":
    doctest.testmod()
```

- Das `__name__`-Attribut ist gleich `"__main__"`, wenn das Modul mit dem Python-Interpreter gestartet wird oder es in IDLE geladen wird.

Weitere Details...

UNI FREIBURG

- Ruft man `doctest.testmod(verbose=True)` auf, bekommt man den Ablauf der Tests angezeigt.
- Will man eine Leerzeile in der Ausgabe der Test-Session haben, so muss man `<BLANKLINE>` eintippen, da eine Leerzeile als Ende des Testfalls interpretiert wird.
- Will oder kann man nicht die gesamte Ausgabe angeben, kann man Auslassungspunkte schreiben: `....` Dabei muss allerdings ein *Flag* angegeben werden:
  `# doctest: +ELLIPSIS`
- Mehr unter: `http://docs.python.org/3.3/library/doctest.html`

pytest-Modul (1)

UNI FREIBURG

- py.test ist ein umfassender Framework, um Tests zu schreiben
- Sie müssen py.test installieren, z.B. durch pip3 install pytest.
- Idee: Funktionen werden getestet, indem man Testfunktionen schreibt (und ausführt). Testfunktionen müssen immer den Prefix test_ besitzen.
- Für die zu testenden Funktionen werden die erwarteten Rückgabewerts als Assertions formuliert.
- assert-Anweisung: assert *Bedingung* [, *String*]
- assert sichert zu, dass die Bedingung wahr ist. Wenn das nicht der Fall ist, wird eine Exception ausgelöst, und der String ausgegeben.

pytest-Modul (2)

UNI FREIBURG

Testbeispiel

```
import pytest
...
def test_expreval_b():
    """Test of expreval that fails."""
    expr = ['*', ['+', [3, None, None],
                       [5, None, None]],
                 [6, None, None]]
    assert expreval(expr) == 42

if __name__ == "__main__":
    # -v switches verbose on
    pytest.main("-v %s" % __file__)
```

pytest-Modul (3)

UNI FREIBURG

Die Ausgabe in obigem Beispiel:

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

expreval.py::test_expreval_b FAILED

============================== FAILURES ================================
_________________________ test_expreval_b __________________________

    def test_expreval_b():
        """Test of expreval that fails."""
        expr = ['*', ['+', [3, None, None],
                           [5, None, None]],
                     [6, None, None]]
>       assert expreval(expr) == 42
E       assert 48 == 42
E        +  where 48 = expreval(['*', ['+', [3, None, None], [5, None, None]], [6,

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

UNI FREIBURG

Ausblick: Fehlerfreies Programmieren?

Fehlerfreies Programmieren?

UNI FREIBURG

- Können wir (von Menschen erschaffene) Software für AKWs, Flugzeuge, Autos, usw. vertrauen?
- Testmethoden werden immer beser – decken immer mehr Fälle ab!
- Manchmal können maschinelle Beweise (d.h. für alle Fälle gültig) die Korrektheit zeigen!
- Aktive Forschungsrichtung innerhalb der Informatik
- Natürlich kann aber auch wieder die Spezifikation (gegen die geprüft wird) falsch sein.
- Auch kann das Beweissystem einen Fehler besitzen.

→ Aber wir *reduzieren die Fehlerwahrscheinlichkeit*!

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

UNI FREIBURG

Zusammenfassung

Zusammenfassung

UNI FREIBURG

- Fehlerfreie Programmentwicklung gibt es nicht.
- Man unterscheidet zwischen syntaktischen, Laufzeit- und semantischen Fehlern.
- Fehler findet man durchs Debuggen.
- Fehler finden mit Hilfe von eingesetzten `Print`-Anweisungen oder Debuggern.
- Fehler verstehen und beseitigen: Bugfix.
- Automatische Tests erhöhen die Qualität von Software!
- Python bietet als einfachste Möglichkeit das `doctest`-Modul. Eine komfortablere Möglichkeit ist `pytest`.