



www.uni-frankfurt.de



Modul: Programmierung B-PRG

Grundlagen der Programmierung 1

V11 Software-Tests zur Qualitätssicherung 2

Umsetzungen in Python

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)



Rückblick

V11 Software-Tests zur Qualitätssicherung 1



Eine Selbstverständlichkeit:

(Nicht überheblich werden ;-))

Testen kann die Anwesenheit von Fehlern aufzeigen,
aber nie einen Nachweis von Fehlerfreiheit liefern!

Edsger W. Dijkstra, Notes on structured programming, Academic Press, 1972

2 Vorlesung PRG 1 Softwaretests Prof. Dr. Detlef Krömker

Inhalt

- **Einführung ins Software-Testen**
 - Übersicht zu Prüfverfahren
 - Motivation und Begriffe
 - Fundamentaler Testprozess
 - Testziele
- **Testfallentwurfsverfahren**
 - Spezifikationsbasierter Test (Blackbox-Test)
 - Glassbox-Test
- Implementierungen von Tests in Python

3
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker




Spezifikationsbasierter Test - Übersicht

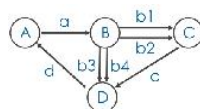
- Das SUT wird von außen, an den Systemschnittstellen, betrachtet (der sogenannte Blackbox-Test).
- Grundlage zur Herleitung der Testfälle bildet die Spezifikation.
- Methoden zur Herleitung der Testfälle :
 - Anforderungsbasiertes Testen
 - Äquivalenzklassen-Methode
 - Grenzwertanalyse
 - Klassifikationsbaummethode
 - Entscheidungstabellentest
 - **Zustandsbasierter Test**
 - Anwendungsfallbasierter Test
 - Zufallstest
 - Modellbasierter Test



4
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker

Nachtrag: Zustandsbezogener Test

- Ausgangsbasis ist die Spezifikation des Programms **als Zustandsgraph** (endlicher Automat, *state chart*)
- Zustände und Zustandsübergänge sind folgendermaßen beschrieben.
Beispiel:



- Ein **Testfall** wird aus
 - dem Ausgangszustand,
 - dem Ereignis (→ Eingabedaten) und
 - dem Soll-Folge-Zustand (→ Soll-Resultat) gebildet.

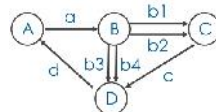
Testumfang für einen betrachteten Zustand

- Alle Ereignisse, die zu einem Zustandswechsel führen.
- Alle Ereignisse, die auftreten können, aber ignoriert werden
- Alle Ereignisse, die auftreten können und eine Fehlerbehandlung erfordern.

5

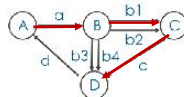
Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömer



Zustands-überdeckung

Jeder Zustand muss mindestens einmal erreicht werden.

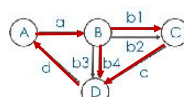


Testfälle:

Aa
Bb1
Cc:

Zustandspaar-überdeckung

Von jedem Zustand muss in jeden möglichen Folgezustand gewechselt werden.

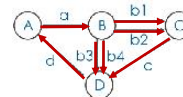


Testfälle:

Aa
Bb1
Bb4
Cc:
Dd

Transitions-überdeckung

Alle Zustandsübergänge müssen mindestens einmal wirksam werden.





Testfälle:

Aa
Bb1
Bb2
Bb3
Bb4
Cc:
Dd

6

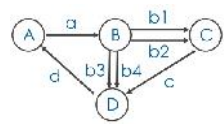
Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömer

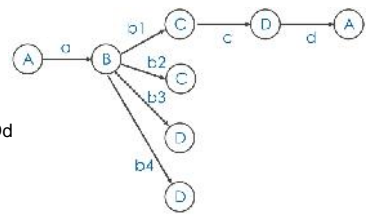



Roundtrip-Folgen

- Beginnend im Startzustand
- Endend in einem Endzustand oder in einem Zustand, der bereits in dieser oder einer anderen Roundtrip-Folge enthalten war
- Zustandsübergangsbaum:



Testfälle:
Aa Bb1 Cc Dd
Aa Bb2
Aa Bb3
Aa Bb4



7
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer




Spezifikationsbasierter Test - Übersicht

- Das SUT wird von außen, an den Systemschnittstellen, betrachtet (der sogenannte Blackbox-Test).
- Grundlage zur Herleitung der Testfälle bildet die Spezifikation.
- Methoden zur Herleitung der Testfälle :
 - Anforderungsbasiertes Testen
 - Äquivalenzklassen-Methode
 - Grenzwertanalyse
 - Klassifikationsbaummethode
 - Entscheidungstabellentest
 - Zustandsbasierter Test
 - Anwendungsfallbasierter Test
 - Zufallstest
 - Modellbasierter Test



Erledigt


8
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer


Weitere Testverfahren (Terminologie)

- **Smoke-Test:** „Vorabtest“, der prüft, ob der Prüfling betriebs- und testbereit ist - wird z.B. von den Entwicklern vor einem **ersten** Release durchgeführt. Im Allgemeinen besteht ein **smoke test** aus einer Sammlung von **Tests**, die neue oder reparierte Software durchlaufen muss.
- **Syntaxtest:** Bei Vorliegen einer formal definierten Interaktionssprache.
- **Regressionstest** : Test auf (unbeabsichtigte Neben-) Effekte bei Korrektur, Anpassung oder Erweiterung.
- „Verhält sich das Programm noch so, wie es sich vor der Modifikation verhalten hatte?“
- Verwendung in der Regel in der Wartung
- Eingabedaten können aus den Betriebsdaten gewonnen werden, Sollresultate werden aus der Version vor der Änderung gewonnen.

Inhalt

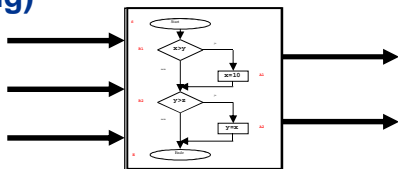
- **Einführung ins Software-Testen**
 - Übersicht zu Prüfverfahren
 - Motivation und Begriffe
 - Fundamentaler Testprozess
 - Testziele
- **Testfallentwurfsverfahren**
 - Spezifikationsbasierter Test
 - **Glassbox-Test**
 - Erfahrungsbasierte Verfahren







Glass-Box (structural testing)

- Alternative Bezeichnungen: Coverage-Test, Whitebox-Test, Strukturtest.
- Der Glassbox-Test (GBT) beurteilt die Vollständigkeit eines Tests anhand der vollständigen Ausführung einzelner „Bausteine“ des Programms → Glassbox-Test-Entität (GBT-Entität).
- Die Implementierung ist im Gegensatz zu Black-Box-Tests bekannt und wird zur Bestimmung von Testfällen benutzt.
- Die Theorie ist sehr weit erforscht (z.B. mit der Graphentheorie)
- Erlaubt den Einsatz von Testmetriken, z.B. Überdeckung, Coverage.



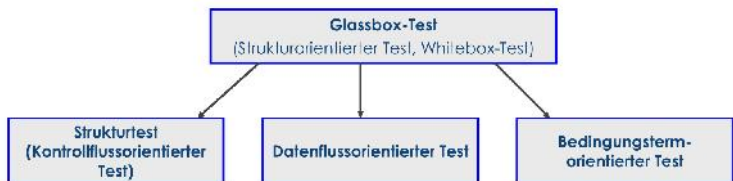
Grundlagen der Programmierung 1
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker





Glassbox-Test

In der Praxis wird der Glassbox-Test in der Regel in drei Gruppen aufgeteilt:





```

graph TD
    A["Glassbox-Test  
(Strukturorientierter Test, Whitebox-Test)"] --> B["Strukturtest  
(Kontrollflussorientierter Test)"]
    A --> C["Datenflussorientierter Test"]
    A --> D["Bedingungsorientierter Test"]


```

12
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker






Teilschritte beim Glassbox-Test

- Beim Glassbox-Test wird das Programm speziell „präpariert“ um die Ausführung protokollieren zu können (→ instrumentieren). Hierzu sind **spezielle Test-Werkzeuge** erforderlich.
- Die Ausführung erfolgt genau gleich wie beim Blackbox-Test
- Das Resultat des Glassbox-Tests ist die Information **welcher Programmcode ausgeführt wurde und welcher nicht**



13
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer

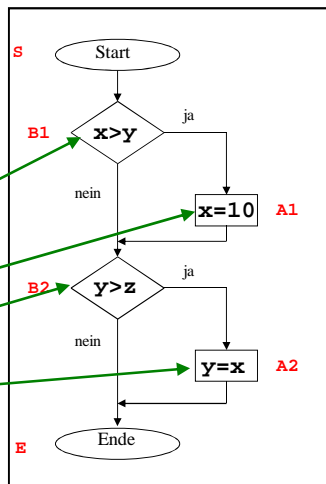



Instrumentierung Schritt 1


Code wird in **Bedingungen** und **Anweisungen** unterteilt und in ein Diagramm übernommen.


▸ **Beispiel:**

```
def func(x,y,z):
    if x > y:
        x = 10
    if y > z:
        y = x
    return z
```



Grundlagen der Programmierung
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer





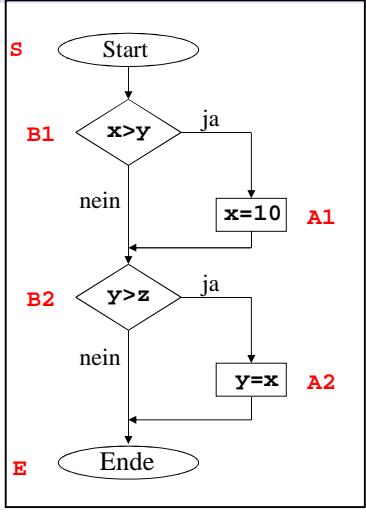
Überdeckungsarten

Wie bestimmt man eine Testmenge?


Anweisungsüberdeckung
Statement coverage, **c0-Test**


Eine Testmenge T, bei der jede **Anweisung A** mindestens einmal durchlaufen wird. z.B.: $T = \{(2, 1, 0)\}$

```
def func(x, y, z):
    if x > y:
        x = 10
    if y > z:
        y = x
    return z
```



Grundlagen der Programmierung
Prof. Dr. Detlef Krömker



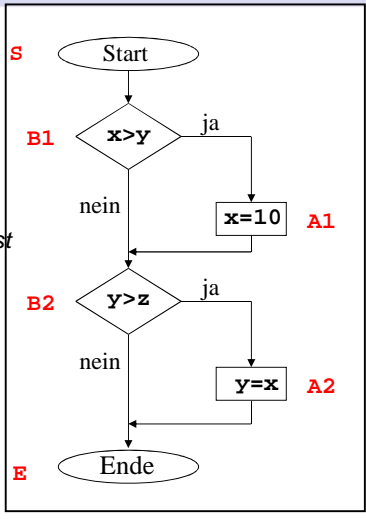


Überdeckungsarten



Wie bestimmt man eine Testmenge?

Ablaufzweigüberdeckung
Zweigüberdeckung, Branch coverage, **c1-Test**

Eine Testmenge T, bei der jeder Ablaufzweig überdeckt wird, d.h. bei jeder Bedingung B wird jeder Zweig mindestens einmal durchlaufen.
z.B.: $T = \{(0, 0, 0), (2, 1, 0)\}$



Grundlagen der Programmierung
Prof. Dr. Detlef Krömker

Überdeckungsarten



Wie bestimmt man eine Testmenge?

Ablaufpfadüberdeckung, c2a-Test
 Eine Testmenge T, bei der sämtliche mögliche Ablaufpfade im Programm durchlaufen werden. Die Anzahl kann sehr groß werden.
 z.B.: $T = \{(0,0,0), (1,0,0), (0,1,0), (2,1,0)\}$

```

graph TD
    S([Start]) --> B1{x > y}
    B1 -- ja --> A1[x = 10]
    B1 -- nein --> B2{y > z}
    B2 -- ja --> A2[y = x]
    B2 -- nein --> E([Ende])
  
```

Grundlagen der Programmierung und Prüfung Softwaretests
Prof. Dr. Detlef Krömer

Weitere Überdeckungen

- ▶ **Schleifenüberdeckung**
 - ▶ Anteil der Schleifen, die nicht, einmal oder mehrfach durchlaufen werden
 - ▶ Auch als Boundary-Interior-Test oder als c2b-Test bezeichnet
- ▶ **Funktionsüberdeckung**
 - ▶ Anteil der ausgeführten Funktionen
- ▶ **Datenflusstest** (hat kaum praktische Bedeutung)
- ▶ **Bedingungstest:** Testen zusammengesetzter Bedingungsdrücke, z. B.
`if((A and B) or (C and D))`
 - Verbreitete Metriken:**
 - ▶ Einfache Bedingungsüberdeckung (simple condition coverage)
 - ▶ Bedingungs-/Entscheidungsüberdeckung (condition/decision coverage)
 - ▶ Mehrfach-Bedingungsüberdeckung (multiple condition coverage)
 - ▶ Modifizierte Bedingungs-/Entscheidungsüberdeckung (modified condition/decision coverage, MC/DC)

18 Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer

Überdeckungseigenschaften beim Glassbox-Test

Überdeckungsmaße (coverage) = Anteile in %:

1. **Anweisungsüberdeckung**
2. **Ablaufzweigüberdeckung**
3. **Ablaufpfadüberdeckung**

Ablaufbezogenes Tests (Glassbox) weisen folgende Eigenschaften auf:

- Es lässt sich **nur eine bestimmte Klasse** von Fehlern auffinden, z. B. grobe Abbruchfehler, unerreichbare Zweige, Irrpfade, endlose Schleifen.
- **Nicht erkannt** werden können z. B. Tippfehler, inkonsistente Schnittstellen, Abweichung von der Spezifikation.

Nutzen des Glassbox-Tests

1. Codeüberdeckungsmaße als Metrik zur Testgüte

Objektives Vollständigkeitskriterium, das beispielsweise als Testendeckriterium verwendet werden kann

2. Testsuite-Erweiterung

Der Glassbox-Test zeigt Programmcode, der für eine Testsuite nicht ausgeführt wird und damit ungetestet bleibt.

3. Grundlage für selektiven Regressionstest


Anstelle der „rerun-all“-Strategie sollen nur einzelne, ausgewählte Testfälle ausgeführt werden.

4. Testsuite-Reduktion

Für den Regressionstest soll die Testsuite verkleinert werden, ohne dabei aber (wesentlich) an Testgüte einzubüßen

5. Unterstützung beim Programmcodeverständnis

Der Glassbox-Test zeigt, welcher Programmcode von welchem Testfall ausgeführt wird.




Grey-Box-Test

... ist eine Kombination von Black-Box-Test und White-Box-Test.

- Vom White-Box-Test: er wird oft von den gleichen Entwicklern wie das zu testende System geschrieben.
- Vom Black-Box-Test: Anfänglich die Unkenntnis über die Interna des zu testenden Systems, weil der Grey-Box-Test **vor dem zu testenden System geschrieben wird (Test-First-Programmierung)**.

21 Vorlesung PRG 1 Softwaretests Prof. Dr. Detlef Krömker



Erfahrungsbasierte Verfahren

- **"Error guessing"**
 - Einsatz in höheren Testebenen, um systematische Tests zu ergänzen.
 - Kann von systematisch erstellten Testfällen „inspiriert“ werden.
 - Error Guessing kann äußerst unterschiedliche Grade von Effizienz erreichen, abhängig von der Erfahrung des Testers.
- **Exploratives Testen**
 - „Ad-hoc“-Testfallentwurf, Testdurchführung, Testprotokollierung
 - Die Testgüte hängt in hohem Maße von der Kompetenz der Tester ab.
 - Der Ansatz, erscheint dann als sinnvoll, wenn es nur wenig oder ungeeignete Spezifikationen gibt
- **Test-Ideen (RUP)**
 - Grundlage bildet ein „Brainstorming-Prozess“
 - Test Ideen werden in einer Liste gesammelt und sukzessive verfeinert.

22 Vorlesung PRG 1 Softwaretests Prof. Dr. Detlef Krömker

Noch ein Wort zur Testorganisation

Typ1: Testen liegt ausschließlich in der Verantwortung des einzelnen Entwicklers. Jeder Entwickler testet seine eigenen Programme.

Typ2: Testen liegt in der Verantwortung des Entwicklungsteams. Die Entwickler testen ihre Programme gegenseitig.



Typ3: Mindestens ein Mitglied des Entwicklerteams ist für Testarbeiten abgestellt. Es erledigt alle Testarbeiten des Teams.

Typ4: Es gibt ein oder mehrere dedizierte Testteams innerhalb des Projekts (die nicht an der Entwicklung beteiligt sind).

Typ5: Eine separate Organisation (Testabteilung, externer Testdienstleister, Testlabor) übernimmt das Testen.

Inhalt

- **Einführung ins Software-Testen**
 - Übersicht zu Prüfverfahren
 - Motivation und Begriffe
 - Fundamentaler Testprozess
 - Testziele
- **Testfallentwurfsverfahren**
 - Spezifikationsbasierter Test (Blackbox-Test)
 - Glassbox-Test
- **Implementierungen von Tests in Python**






Implementierungen von Tests in Python

Programmiersprachen sollen das "Testen" unterstützen. Python tut das.

- unter Benutzung von `__name__` ...
- als doctest (ein Python Spezifikum!)
- als unittest. ... mit Methoden der Klasse TestCase
- Interne Selbsttests (internal self-checks) mit assertions (Zusicherungen): in-line Dokumentation, um Annahmen des Programmierers offensichtlich zu machen. ("Explicit is better than implicit.")
- Testgetriebene Entwicklung oder "Im Anfang war der Test" .. in Python

25
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker

unter Benutzung von `__name__` ...

Jedes Python-Modul hat einen im built-in-Attribut `__name__` definierten Namen.

Nehmen wir an, wir haben ein Modul mit dem Namen "abc" unter "abc.py" gespeichert.

Wird dieses Modul mit "import abc" importiert, dann hat das built-in-Attribut `__name__` den Wert "abc".

Wird die Datei abc.py als eigenständiges Programm aufgerufen, also z.B. mittels

```
>>> python3 abc.py
```

dann hat diese Variable den Wert `'__main__'`.

26
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker

Vorgehen (1)

Wir programmieren unsere Tests als "normalen" Code und geben z.B. mit der print() Funktion aus:

```
print("Test für abc-Funktion erfolgreich.") oder
print("abc-Funktion liefert fehlerhafte Werte.")
```

Entscheidenden Nachteil. Wenn man das Modul module importiert, wird auch das Ergebnis des Tests angezeigt, z.B.,

```
>>> import module Test für abc-Funktion erfolgreich.
```

Das ist sehr störend **und auch nicht üblich**, wenn Module solche Meldungen beim import ausgeben. Module sollen sich "schweigend" laden lassen.

Vorgehen (2)

Lösung: `__name__` nutzen und den Test in den Teil

```
if __name__ == "__main__":
```

schreiben.

Wird unser Modul direkt gestartet, also **nicht** importiert, hat `__name__` den Wert `"__main__"`.

Es gibt **keine** Ausgaben, wenn das Modul importiert wird.

Diese Methode ist die einfachste Methode und weit verbreitet für Modultests.

doctest-Modul

- Der eigentliche *Test* befindet sich bei dieser Methode **im Docstring**.
- **Vorgehensweise:** Man muss das Modul "doctest" importieren.
- die Tests werden von Hand geschrieben oder z.B. aus einer interaktiven Sitzung (am Interpreter) in den Docstring des zu testenden Moduls beziehungsweise der (Klasse, Methode oder) Funktion kopiert.
- Die Tests bestehen aus Anweisungen (nach den >>>) und den zugehörigen Ausgaben (in der folgenden Zeile), so wie sie im interaktiven Python-Interpreter aussehen würden.
- Die Testblöcke grenzen Sie vom umgebenden Text durch Leerzeilen ab.
- Aufgerufen wird die Ausführung des Tests durch `doctest.testmod()`.

Beispiel (1):

entnommen aus: https://www.python-kurs.eu/python3_tests.php

```
import doctest

def fib(n):
    """ Die Fibonacci-Zahl für die n-te Generation wird
        iterativ berechnet.

        """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Beispiel (2):

Dieses Modul rufen wir nun in einer interaktiven Python-Shell auf und lassen ein paar (Test-)Werte berechnen:

```
>>> from fibonacci import fib
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fib(15)
610
>>>
```

Beispiel (3):

Diese Aufrufe mit den Ergebnissen kopieren wir aus der interaktiven Shell in den Docstring unserer Funktion.

Damit das Modul doctest aktiv wird, müssen wir die Methode testmod() starten, falls das Modul direkt aufgerufen wird.

Dies können wir wie üblich mit einem Test des Attributs `__name__` auf den Wert `"__main__"` machen.

Das vollständige Modul sieht nun wie folgt aus:

Beispiel (4):

```
import doctest

def fib(n):
    """ Die Fibonacci-Zahl für die n-te Generation wird
        iterativ berechnet.

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610

    """
```

33

Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömker

Beispiel (4-Fortsetzung):

```
a, b = 0, 1
for i in range(n):
    a, b = b, a + b
return a

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



Wir speichern dieses Modul unter dem Namen test

Importieren wir dieses Modul durch
`import test`
 erhalten wir **keine** Ausgabe, weil alles okay ist.

34

Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömker

Beispiel (5) jetzt mit "kleinen" Fehler

```
a, b = 1, 1 # hier ist der Fehler.

for i in range(n):
    a, b = b, a + b return a

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

35
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer




Die Ausgabe:

```
*****
File "C:/Users/kroemker/AppData/Local/Programs/Python/Python36/My_Python_projects/test.py", line 6, in
__main__.fib
Failed example:
    fib(0)
Expected:
    0
Got:
    1
*****
File "C:/Users/kroemker/AppData/Local/Programs/Python/Python36/My_Python_projects/test.py", line 8, in
__main__.fib
Failed example:
    fib(1)
Expected:
    1
Got:
    1
*****
File "C:/Users/kroemker/AppData/Local/Programs/Python/Python36/My_Python_projects/test.py", line 10, in
__main__.fib
Failed example:
    fib(2)
Expected:
    2
Got:
    1
*****
```

36
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömer



Doctest

Es werden alle Aufrufe angezeigt, die ein fehlerhaftes Ergebnis geliefert haben.

Wir sehen jeweils den Beispielaufruf hinter der Zeile "Failed example:". Hinter "Expected:" folgt der erwartete Wert, also der korrekte Wert, und hinter "Got:" folgt der von der Funktion produzierte Ausdruck, also der Wert, den doctest beim Aufruf von fib erhalten hat.

Man kann diese Methode mögen oder auch nicht.

Sicherlich ist sie einfach zu nutzen.

... und auch gut für die testgetriebene Entwicklung zu nutzen.

37

Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömker



Die Doctest in eine Datei auslagern: Eine Directory-Struktur könnte dann so aussehen

```
example_project
|-- arabic_to_roman_2          # der eigentliche Code
|-- doctests_arabic_to_roman.txt # die doctests sind in diesem File
'-- README.txt                # dies ist schlichte Höflichkeit, gewöhnen
                              # Sie sich das bitte an.
```

Die Doctests stehen im File

example_project/doctests_arabic_to_roman.txt



Aufgerufen wird dies durch:

```
import doctest
doctest.testfile(doctests_arabic_to_roman_2)
```

38

Vorlesung PRG 1 Softwaretests

Prof. Dr. Detlef Krömker

Herausforderung Ein kleineres Beispiel mit `input()` (1)

```


""" Input of two Characters; return a list. """

def get_two_chars():
    chars = []
    print("Bitte zwei Zeichen eingeben. Nach jedem jeweils <enter> drücken: ")
    for i in range(2):
        chars.append(input("> "))
    return chars

def main():
    print(get_two_chars())

if __name__ == "__main__":
    main()

```

 **Unser Problem!**

39
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker




Ein kleineres Beispiel mit `input()` (2) Ein erster Versuch – so einfach geht es nicht!

```

""" Input of two Characters; return a list .
>>> main()
['A', 'ß']

"""

def get_two_chars():
    chars = []
    print("Bitte geben Sie zwei Zahlen ein, danach jeweils <enter> drücken: ")
    for i in range(2):
        chars.append(input("> "))
    return chars

def main():
    print(get_two_chars())

if __name__ == "__main__":
    import docstring
    docstring.testmod()

```

 **Unser Problem!**

40
Vorlesung PRG 1 Softwaretests
Prof. Dr. Detlef Krömker

Ein kleineres Beispiel mit `input()` (3)

- Wir würden dieses Programm trotzdem gern "doctesten":
- Geht aber nicht, da das doctest nur die Zeile mit

```
>>> main()
```

 liest und dann auf den Input wartet.
- **Wir wollen aber auch die Eingabe von Werten automatisieren.**
- **Dazu benutzen wir ein sogenanntes Mock-Objekt**
 (kurz auch mock oder auch fake genannt)
 d.i. ein Platzhalter für echte Objekte: in diesem Fall für `input()`.
- Möglich wäre auch
 - ein **stub** (wie ein mock, nur erzeugt es immer denselben Wert)

Das ist fast alles zum Testen

Montag starten wir mit der OO-Programmierung

Danke für Ihre Aufmerksamkeit!