Modul: EPI:
Einführung in die Programmierung EPR

VE06 Glassboxtests und Test in Python

Prof. Dr. Franziska Matthäus / Prof. Dr. Matthias Kaschube / Dr. Karsten Tolle
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Eine Selbstverständlichkeit:

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

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

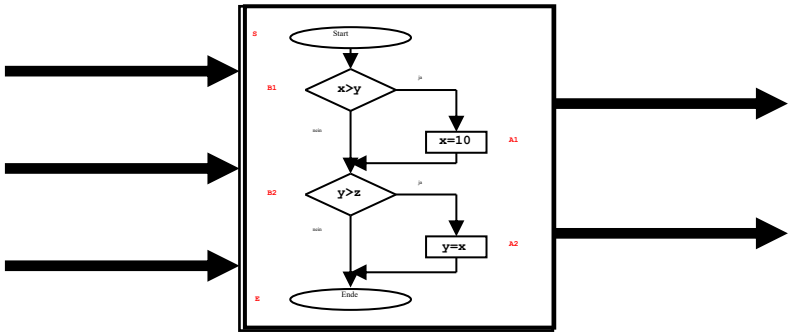
Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Inhalt

- Einführung ins Software-Testen
- Testfallentwurfsverfahren
  - Spezifikationsbasierter Test (Blackbox-Test)
  - Glassbox-Test
- **Implementierungen von Tests in Python**
  - unter Benutzung von __name__ ...
  - als doctest (ein Python Spezifikum!)
  - als unittest. ... mit Methoden der Klasse TestCase
  - Interne Selbsttests (internal self-checks) mit Assertions (Zusicherungen)
  - Testgetriebene Entwicklung oder "Im Anfang war der Test"

Bisher Blackbox Test – nun: Glass-Box (*structural testing*)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle



- 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.

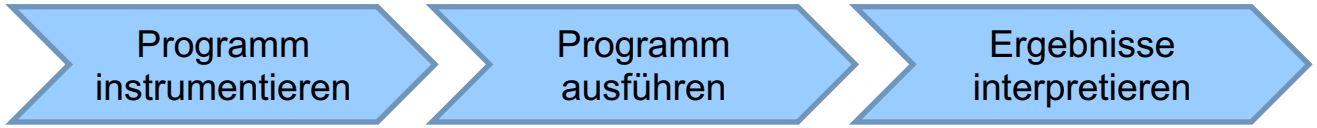
Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die Teilschritte des Glassboxtests

1. Beim Glassbox-Test wird das Programm speziell „präpariert“ um die Ausführung protokollieren zu können (➔ instrumentieren). Hierzu sind **spezielle Test-Werkzeuge** erforderlich.

2. Die Ausführung erfolgt genau gleich wie beim Blackbox-Test.
   Protokolierung der Ergebnisse **und des Laufs!**

3. Das Resultat des Glassbox-Tests ist die Information
   (a) Welcher Programmcode ausgeführt wurde und welcher nicht.
   (b) Wo treten ggf. welche Fehler auf?



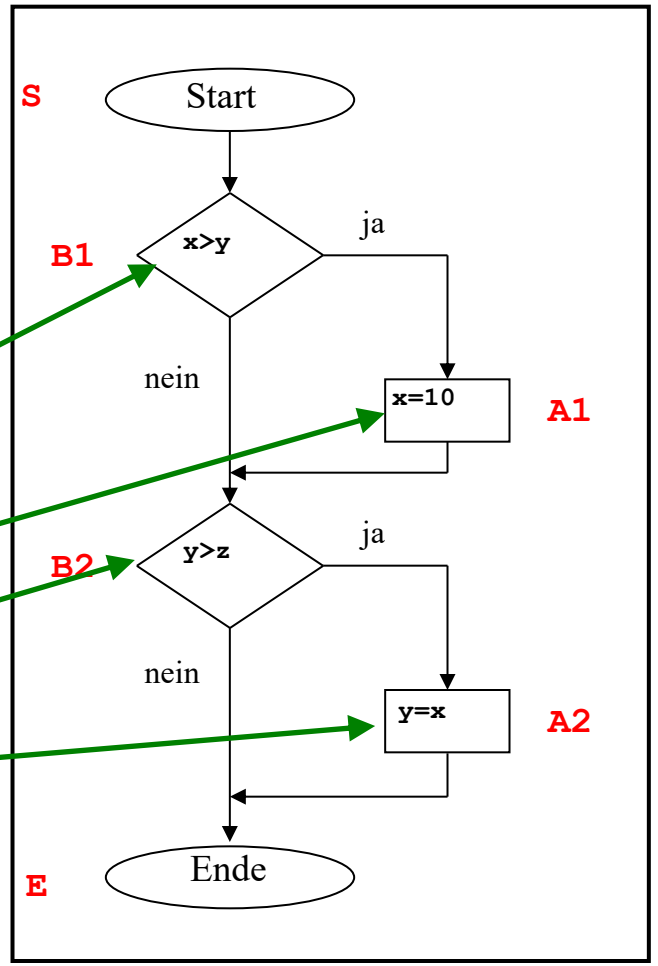
Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Instrumentierung Schritt 1

Code wird in **Bedingungen, Schleifen** und **Zuweisungsketten** unterteilt und z.B. in ein Diagramm übernommen.

- **Beispiel**:

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



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Überdeckungsarten

Wie bestimmt man eine Testmenge?

Anweisungsüberdeckung

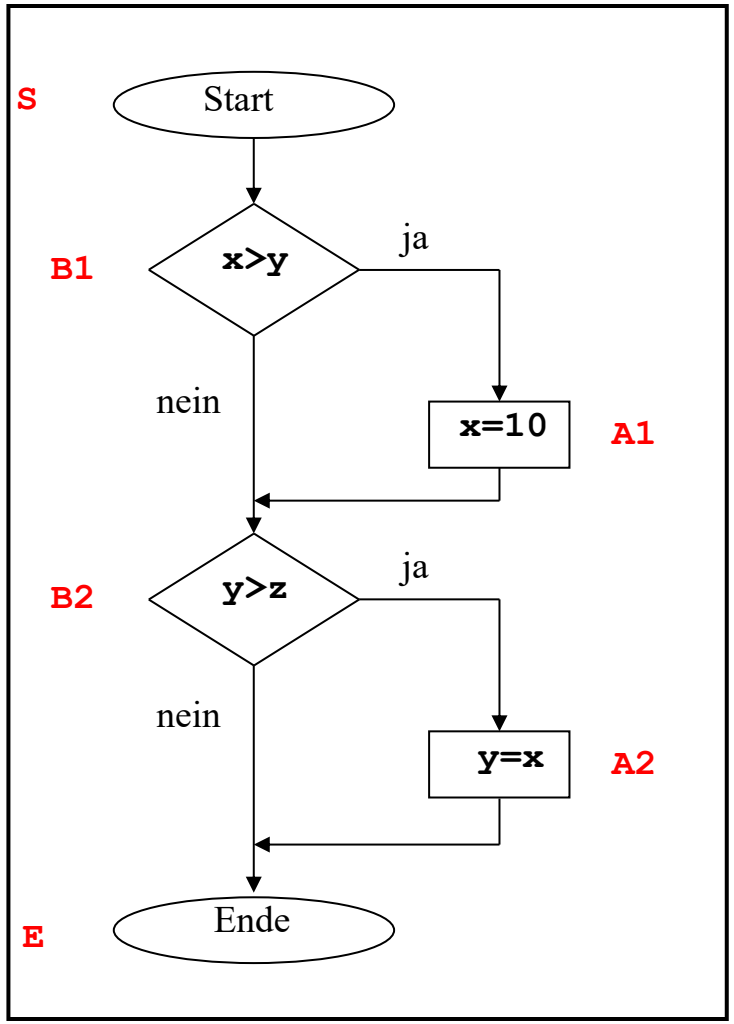
Statement coverage, **c0-Test**

Eine Testmenge T, bei **der jede** **Anweisung** A mindestens einmal durchlaufen wird. z.B.:

z.B. $T=\{(x,y,z)\} = \{(2,1,0)\}$

Ergebnis: Lauf und Funktionswert

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



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Überdeckungsarten

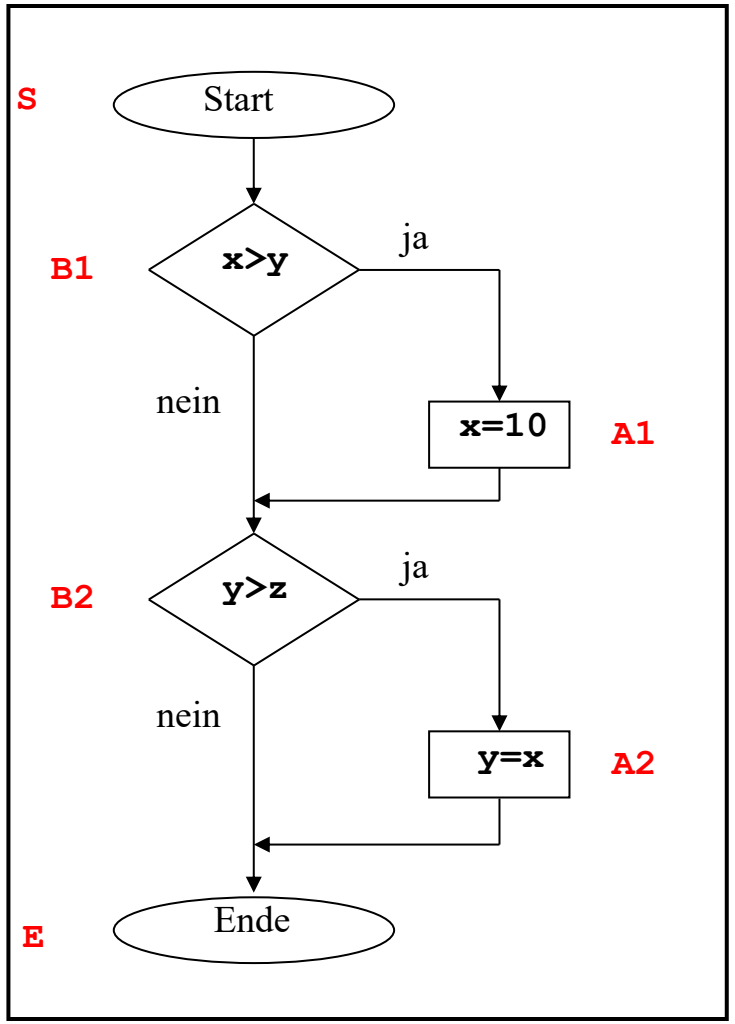
Wie bestimmt man eine Testmenge?

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.: $T=\{(0,0,0), (2,1,0)\}$



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

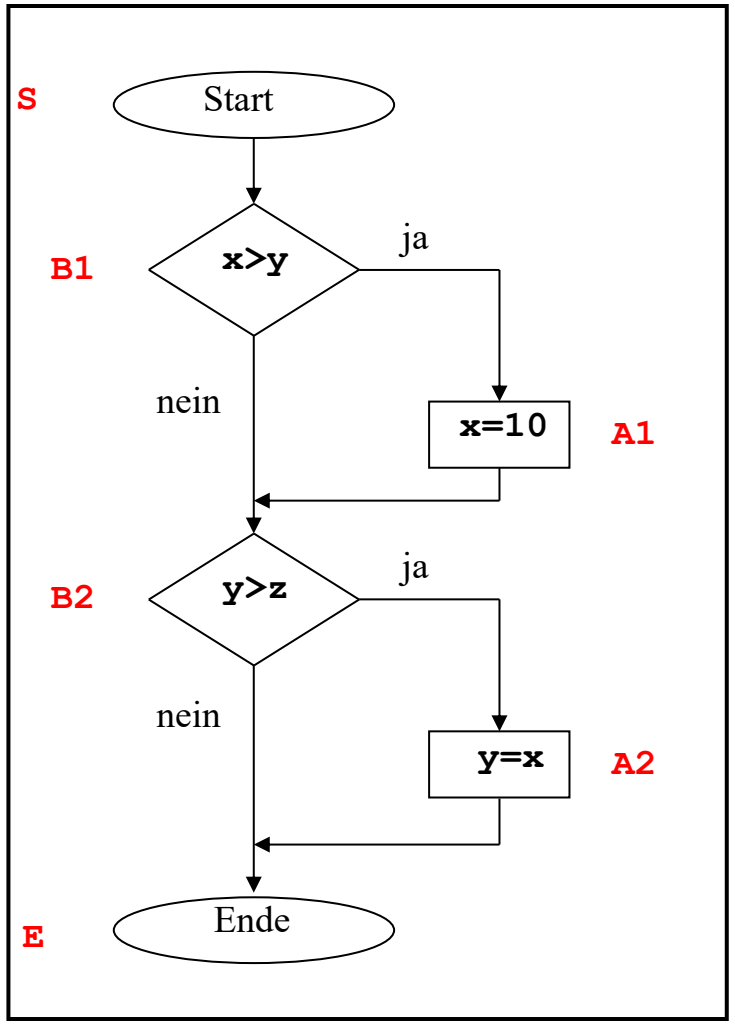
Überdeckungsarten

Wie bestimmt man eine Testmenge?

Ablaufpfadüberdeckung, c2a-Test

Eine Testmenge T, bei der sämtliche mögliche Ablaufpfade im Programm durchlaufen werden. Achtung: Die Anzahl kann sehr groß werden.

z.B.: $T=\{(0,0,0), (1,0,0), (0,1,0), (2,1,0)\}$



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Weitere Überdeckungsarten

- **Schleifenüberdeckung**
  - Anteil der Schleifen, die **nicht**, einmal oder mehrfach durchlaufen werden
  - Auch als Boundery-Interiour-Test oder als **c2b-Test** bezeichnet
- **Funktionsüberdeckung**
  - Anteil der ausgeführten Funktionen
- **Datenflusstest** (hat kaum praktische Bedeutung)
- **Bedingungstest:** Testen zusammengesetzter Bedingungsausdrücke, 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 conditon coverage)
  - Modifizierte Bedingungs-/Entscheidungsüberdeckung (modified condition/decision coverage, MC/DC)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Zusammenfassung:Überdeckungseigenschaften beim Glassbox-Test

Überdeckungsmaße (*coverage*) = Anteile in %:

1. **Anweisungs**überdeckung
2. **Ablaufzweig**überdeckung
3. **Ablaufpfad**überdeckung

Ablaufbezogene 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. manche Tippfehler, inkonsistente Schnittstellen, Abweichung von der Spezifikation.

Zusammenfassung: Nutzen des Glassbox-Tests

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

1. **Codeüberdeckungsmaße als Metrik zur Testgüte**
   Objektives Vollständigkeitskriterium, das beispielsweise als Test-Endekriterium 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.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Grey-Box-Test

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

- White-Box-Test: Er wird oft von den gleichen Entwicklern wie das zu testende System geschrieben.

- Black-Box-Test: Es besteht Unkenntnis über die Interna des zu testenden Systems, weil der Grey-Box-Test **vor dem zu testenden System geschrieben wird:**

  testgesteuerte Programmierung (*test first development* oder *test-driven development*, *TDD*) ist eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen eingesetzt wird.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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.
- **Test-Ideen (RUP)**
  - Grundlage bildet ein „Brainstorming-Prozess“
  - Test Ideen werden in einer Liste gesammelt und sukzessive verfeinert.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Inhalt

- Einführung ins Software-Testen
- Testfallentwurfsverfahren
  - Spezifikationsbasierter Test (Blackbox-Test)
  - Glassbox-Test
- **Implementierungen von Tests in Python**
  1. unter Benutzung von __name__ ...
  2. als doctest (ein Python Spezifikum!)
  3. als unittest. ... mit Methoden der Klasse TestCase
  4. Interne Selbsttests (internal self-checks) mit Assertions (Zusicherungen)
  5. Testgetriebene Entwicklung oder "Im Anfang war der Test"

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

1. Tests unter Benutzung von __name__ ...

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

Nehmen wir an, wir haben ein Modul mit dem Namen "abc": Dies ist 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 (oder aus der IDLE) aufgerufen, also z.B: mittels

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

dann hat diese Variable den Wert '__main__'.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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.")
```

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

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Vorgehen (2)

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

```
if __name__ == "__main__":
    # Implement Tests here.
```

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 eine einfache Methode und weit verbreitet für Modultests.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

2. Tests mit dem 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 dem >>>) und den zugehörigen Ausgaben (in der folgenden Zeile). (Genau so, wie sie im interaktiven Python-Interpreter aussehen würden.)
- Die Testblöcke sind im Docstring vom umgebenden Text durch Leerzeilen abzugrenzen.
- Aufgerufen wird die Ausführung des Tests durch `doctest.testmod()`.

Beispiel (1):

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

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Das folgende Programm ist als Modul unter fibonacci.py gespeichert.

```
""" A Fibonacci Module."""

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
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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
>>>
```

Die Überprüfung der Ergebnisse ist wichtg.

Man kann die Tests aber auch vor der Entwicklung des Codes auf der Basis der Spezifikation schreiben.

Dazu muss man das Verhalten des Interpreters aber sehr genau kennen.

Blackbox Test / **test driven dev**

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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:

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Beispiel (4):

```
""" A Fibonacci Module."""

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

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    """
```

Instrumentieren!

Beispiel (4-Fortsetzung):

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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

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

Es ist o.k., erst hier zu importieren, damit doctest nicht immer importiert werden muss.

Instrumentieren!

```
>>>
 RESTART: C:/Users/kroemker/AppData/Local/Programs/Python/Python37/
Scripts/fibonacci.py
>>>
```

Beispiel (5) jetzt mit "kleinem" Fehler

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

...

```
    a, b = 1, 1  # A little mistake here.

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

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

Da wird die Ausgabe schon umfangreicher.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die Ausgabe:

```
>>>
 RESTART: C:\Users\kroemker\AppData\Local\Programs\Python\Python37\Scripts\fibonacci.py
**********************************************************************
File "C:\Users\kroemker\AppData\Local\Programs\Python\Python37\Scripts\fibonacci.py", line 6,
in __main__.fib
Failed example:
    fib(0)
Expected:
    0
Got:
    1
**********************************************************************
File "C:\Users\kroemker\AppData\Local\Programs\Python\Python37\Scripts\fibonacci.py", line
10, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    89
```

Ausgabe (Fortsetzung)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
**********************************************************************
1 items had failures:
   2 of   3 in __main__.fib
***Test Failed*** 2 failures.
>>>
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Anmerkungen zu doctest.

- Jede Zeile '>>>' wird genauso ausgeführt wie in einer Python-shell und zählt als testcase.
- Die nächste Zeile enthält das erwartete Ergebnis.
- Wenn irgendwas nicht exakt gleich ist (einschließlich nachfolgender blanks im Ergebnis), wird ein Fehler gemeldet.
- Tests, die nicht oder vorläufig nicht durchgeführt werden sollen, kann man mit der +SKIP-**Direktive** wie folgt markieren:

```
>>> fib(10) # doctest: +SKIP
```

Weitere Direktiven findet man in der Dokumentation des doctest-Moduls im Kapitel https://docs.python.org/3/library/doctest.html#doctest-directives

Zusammenfassung *doctest*

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Es werden alle (und nur diese) 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.

Achtung: doctest testet auf (exakte) Gleichheit! (Ein blank hinter dem Ergebnis liefert ein falsches Ergebnis, was man schwer sieht!

Man darf diese Methode mögen oder auch nicht.

Sicherlich ist sie einfach zu nutzen. Gut für einfache Testfälle

Aber blöd ist, dass die **vielen (?)** Testfälle alle direkt im Source Code stehen.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Man kann den *doctest* in eine Datei auslagern:

```
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)
```

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

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
""" 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()
```

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

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
""" 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!

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

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)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Zusammenfassung

- *Doctests* sind für einfachere Testsituationen sehr nützlich, da sie leicht zu schreiben sind und gleichzeitig die Dokumentation von Code unterstützen.
- Allerdings sind sie für komplexere Testszenarien, insbesondere im numerischen Bereich, weniger gut geeignet.
- Dann greift man eher auf *unit tests* zurück.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Noch ganz kurz unit-tests

- Bitte schauen Sie sich diese in der Dokumentation einmal an, siehe:
- https://docs.python.org/3/library/unittest.html

Bei unit tests hat man sehr viel mehr Möglichkeiten über die sogenannten asserts.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Zusammenfassung

- Jetzt haben wir praktikable Methoden um Tests zu automatisieren
- ➔ Regressionstests
- Das Thema Tests ist damit aber nicht ausgeschöpft … da fehlt noch viel – aber nicht in EPI.