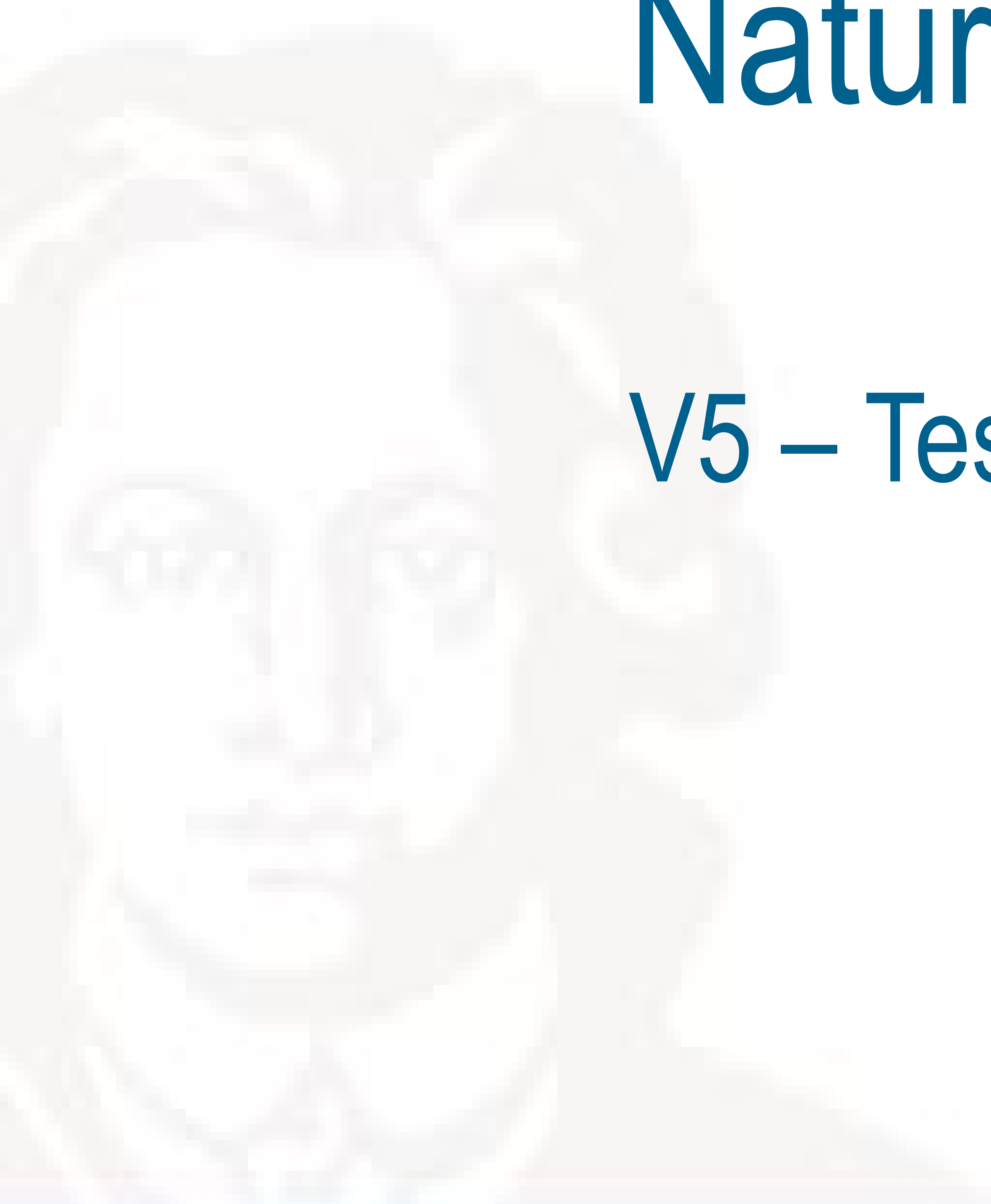


Lukas Müller

Programmieren für Studierende der Naturwissenschaften

V5 – Testen und Fehlermeldungen: Hilfe zur Selbsthilfe

Inhalte

- V1: Grundlagen der Programmierung
  P1: Hilfe beim Einrichten von Python an eigenen Rechnern, erste Programme ausführen
- V2: Elementare Datentypen und Kontrollstrukturen
  P2: Übungen
- V3: Aggregierte Datentypen
  P3: Übungen
- V4: Aggregierte Datentypen und Funktionen
  P4: Übungen
- V5: Testen, Fehlermeldungen und Selbsthilfe
  P5: Übungen

Inhalte

- V6: Externe Packages, Einführung NumPy und SciPy
  P6: Übungen
- V7: Externe Packages 2
  P7: Übungen
- V8: Umgang mit externen Daten und Visualisierung
  P8: Übungen
- V9: Entwurf von Algorithmen ODER Aufarbeitung besprochener Themen
  P9: Übungen, selbstständige Arbeit in Kleingruppen
- V10: Betriebssysteme (Windows, Linux, macOS) ohne Übung

Statische und dynamische Programmanalyse

- Prämisse: Je später ein Fehler gefunden wird, desto schwieriger die Korrektur
- **Fehlerquellen vorab zu finden spart Zeit beim Testen:**
- Statische Programmanalyse
  - Code Review: Struktur, Semantik, Syntax und Logik
  - Regeln und Spezifikationen
- Dynamische Programmanalyse einer lauffähigen Implementierung
  - **Verifikation** – Nachweis der Korrektheit gegenüber der Spezifikation
    - Entwickeln wir richtig?
  - **Validation** – Erfüllung der Erwartungen
    - Entwickeln wir das Richtige?

Fehler oder Mangel

Text …

Zeitaufwand

- Bis zu 50% der Entwicklungszeit! Nicht selten sogar noch aufwändiger!

  - Ökonomisch sinnvoll testen können die Entwickler
    - Psychologisch bedenklich

  - Formale Testprozesse können unterschiedlich komplex ausfallen:
    - Planung der Abfolge
    - Abschnitte, die keine Daten produzieren
    - Abschnitte, die Daten produzieren
    - Abschnitte, die Daten benötigen

- Alle diese Schritte kosten Zeit!

Planung

- Testvorbereitung
  - Testmengen und Sollergebnisse
  - Gegebenenfalls Testumgebung(-en) und Testobjekt(-e)

- (Endlich!) Testdurchführung
  - Schreib- und Syntaxfehler
  - Programmlogik wird getestet und manuell protokolliert
  - Testmengen testen und protokollieren

- Testauswertung
  - Lokalisierung und Beseitigung der Fehlerursachen
  - Platzhalter und Testobjekte in echten Code rückführen
  - Alles erneut testen

Auswahl der Testfälle

- Effizienter arbeiten durch geeignete Minimierung der Testfälle
  - Nicht doppelt testen
  - Grenzfälle testen

Und / Oder

- Zufällige Testfälle generieren

„Program testing can be used to show the presence of bugs, but never show their absence!“
[Edsger Wybe Dijkstra (1930-2002): The Humble Programmer, ACM Turing Lecture 1972]

Praxis

- Fehlermeldungen
  - Typ
  - Bedeutung
  - Stelle im Code

- Ablauf falsch ohne eine Fehlermeldung?
  - Fehlerursache im Quellcode identifizieren
  - Eigene Fehlermeldungen definieren (das machen wir nicht)

- Benutzereingaben als häufige Fehlerquelle (Beispiel: Quersumme)

  Testen!
  Alle in Python „eingebaute“ Fehlermeldungen (Built-in Exceptions) lassen sich in der Dokumentation nachlesen: https://docs.python.org/3/library/exceptions.html

Fehlermeldungen in Python

NameError

- NameError
  - Zum Zeitpunkt der Ausführung ist der verwendete Name nicht bekannt

- Häufige Ursachen:
  - Variablenname falsch geschrieben
  - Variable wurde noch nicht definiert
  - Ein Modul, welches genutzt werden soll, wurde nicht importiert
  - Eine Funktion wird aufgerufen, bevor sie definiert wurde

Fehlermeldungen in Python

Syntax und EOLErrors

- ParseError deutet auf Syntaxfehler hin
  - Häufige Ursachen:
    - Fehlende Klammern
    - Fehlende Anführungszeichen
    - Fehlende Kommata
    - Fehlender Doppelpunkt

- EOL: end-of-line
  - Häufige Ursachen:
    - Fehlende Klammern
    - Fehlende Anführungszeichen
    - Fehlende Kommata
    - Fehlender Doppelpunkt

Fehlermeldungen in Python

TypError

- Deutet auf falsche Datentypen in einer Operation hin

- Häufige Ursachen:
  - Falsche Datentypen bei einfacher Operation
  - Ein falscher Datentyp wurde in eine Funktion eingesetzt und führt dort zu falschen Berechnungen
  - Ein return wurde vergessen und daher wird das Funktionsergebnis zu „None“ ausgewertet und weiterverwendet

Weitere Fehler

- **IndentationError:**
  - Falsche Einrückung
- **IndexError:**
  - Einsatz ungültiger Indizes, z.B.
    - ```
      a = [1,2]
      print (a[2])
      ```
- **KeyError**
  - Es wird versucht auf einen Schlüssel im Dictionary zuzugreifen, der nicht vorhanden ist
- **IOError**:
  - Operationen auf nicht vorhandenen Dateien
      ```

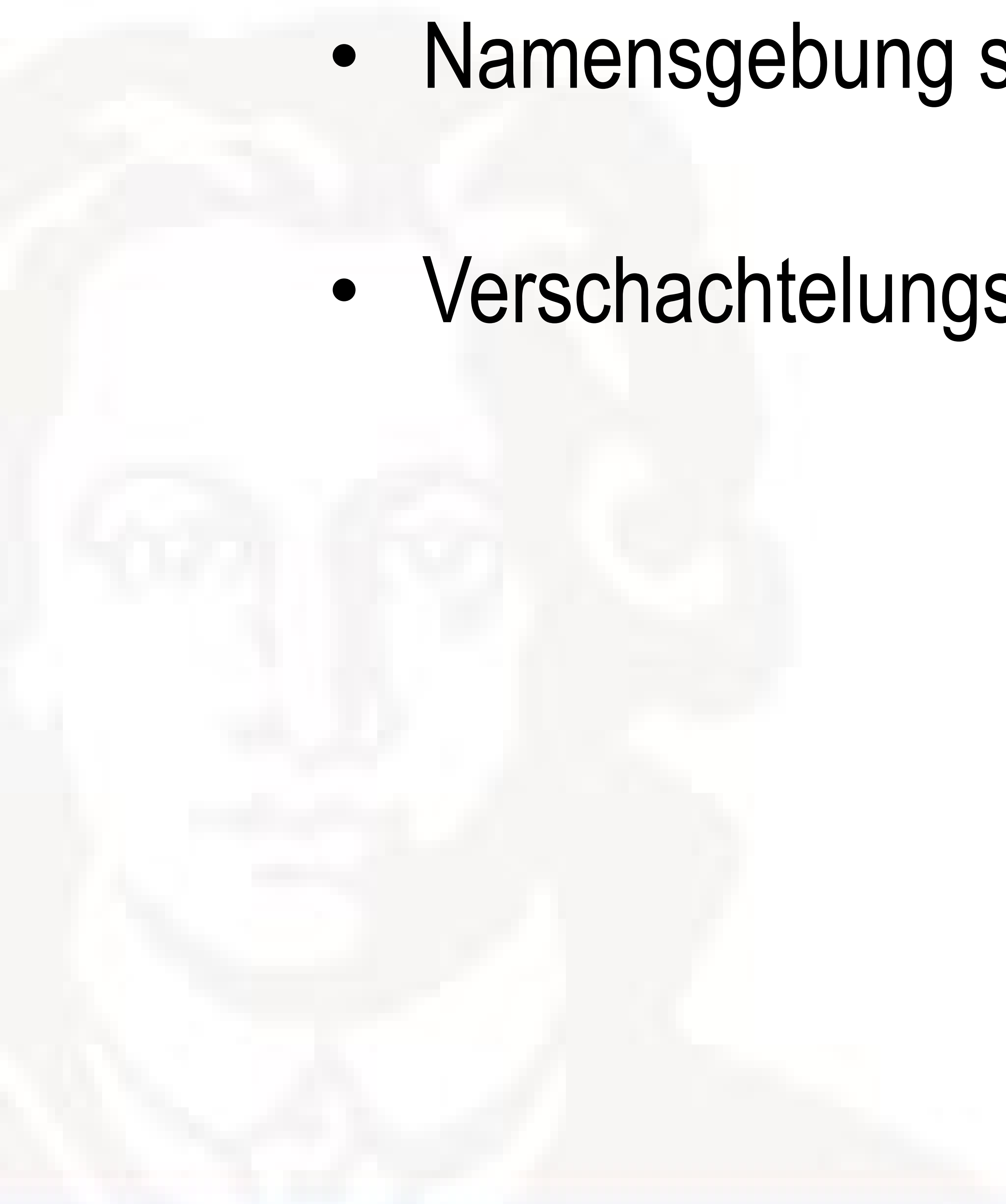
Fehler ohne Fehlermeldungen?

Logische Fehler

- Durchaus möglich
  - Werte und Variablentypen mit `print` zur Kontrolle ausgeben lassen
  - Verzweigungen mit `print` ausgeben lassen
- Zur Not verdächtige Bereiche auskommentieren und untersuchen (Klammerbeispiel)
- Fehler in Schleifen
  - Index statt Eintrag oder umgekehrt
  - range(n) beachten: Start bei 0 und Ende bei n-1
  - Abbruchbedingungen checken

Fehler vermeiden

- Kommentare nutzen
- Docstrings einsetzen
- Namensgebung sinnvoll gestalten
- Verschachtelungstiefe gering halten



Docstrings (Dokumentationsstrings)

- Konventionen einhalten
  - Erster Satz kurz
  - Danach Leerzeile
  - Weitere Erklärungen
  - Leerzeile

- Sprache: Englisch

```python
def fact (n):
    """Computes the factorial of n."""
    if (n <= 1):
        return 1
    else:
        return n*fact(n-1)
```

```python
def fact (n):
    """Ein kurzer Satz, der die Funktionsweise erklärt.

    Hier könnten zusätzliche Infos stehen

    """
    if (n <= 1):
        return 1
    else:
        return n*fact(n-1)
```

Docstrings, wenn man keine gesonderte Dokumentation schreiben möchte?

- Der Docstring eines Moduls m wird in m.__doc__ gespeichert und kann so auch ausgelesen werden

```
>>> import testfile
>>> testfile.fact.__doc__
'Ein kurzer Satz, der die Funktionsweise erklärt.\n\n  Hier könnten zusätzliche
Infos stehen\n\n  '
>>> help(testfile.fact)
Help on function fact in module testfile:

fact(n)
    Ein kurzer Satz, der die Funktionsweise erklärt.

    Hier könnten zusätzliche Infos stehen
```

- Eine on-the-fly Pflege der Docstrings kann die Dokumentationsarbeit deutlich erleichtern und sorgt für stets aktuelle Angaben zu ihrem Code

- Insbesondere für das kollaborative Arbeiten empfehlenswert

Wenn ein Fehler nicht zu vermeiden ist

- In manchen Situationen können Fehler nicht vermieden werden
- Oder gehören sogar zur Konzeption dazu

- Lösung: Exceptions
  - Ein mächtiges Werkzeug
  - Mit Bedacht einzusetzen
  - Fehlermeldungen ignorieren
  - …oder darauf gezielt reagieren lassen:

```
try:
    <Anweisungen>
except <ArtDerFehlermeldung>: #Oder alle Fehler durch except:
    <AlternativeAnweisungen>
```

- Vereinfachung: Versuche das und wenn ein Fehler auftritt, führe jenes aus

Try and Except

Funktionsweise

try:
<Anweisungen>

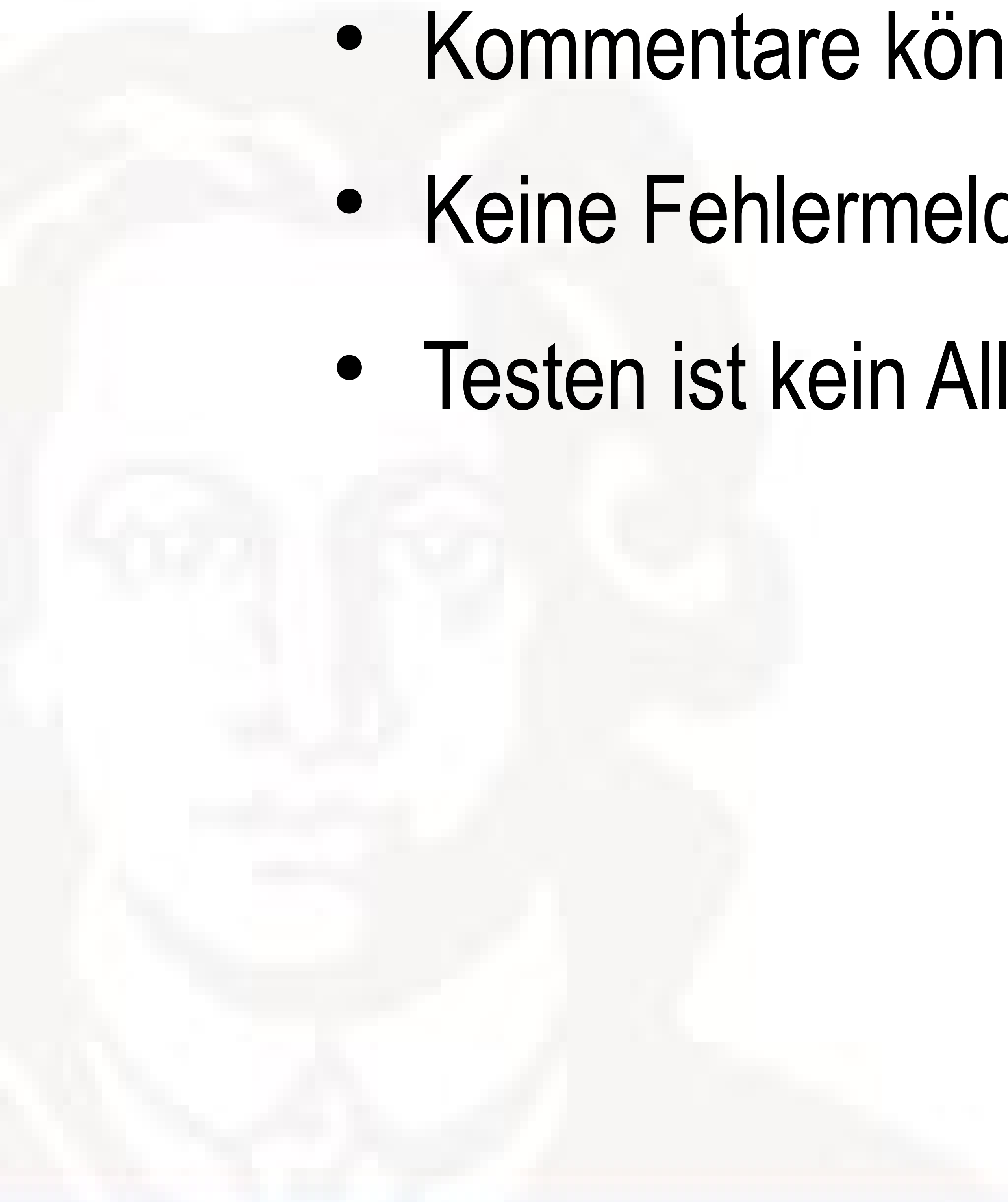
except <ArtDerFehlermeldung>:
<AlternativeAnweisungen>

- Führe try-Block aus
  - Kein Fehler: except-Block wird übersprungen
  - Fehler: Führe except-Block aus
  - Ein try kann von mehreren excepts gefolgt werden, welche unterschiedliche Fehler entsprechend behandeln

Beispiel

```
while True:
    try:
        n = input("Bitte eine Ganzzahl (integer) eingeben: ")
        n = int(n)
        break
    except ValueError:
        print("Keine Integer! Bitte nochmals versuchen ...")
print('Super! Das wars!')
```

Fazit



- Testen ist zeitaufwändig und gehört zur Softwareentwicklung dazu\
- Fehlermeldungen können umgangen werden
  - Sie zu ignorieren kann jedoch fatale Auswirkungen haben
- Kommentare können das Testen massiv erleichtern
- Keine Fehlermeldungen zu sehen bedeutet noch lange keine fehlerfreie Implementierung
- Testen ist kein Allheilmittel! Allerdings kommt man als Entwickler ohne nicht aus!