www.uni-frankfurt.de

Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V 26 Ausnahmebehandlung (Exceptions)

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

Unsere heutigen Lernziele

Exceptions und deren Nutzung in Python kennenlernen.

Programmierstile EAFP oder LBYL kennenlernen.

Übersicht

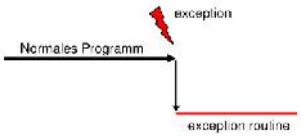
- Ausnahmen (*Exceptions*) – Was sind das?
- Ursachen für Exceptions
- Die *exception hierarchy* des C-Python Interpreters
- Ausnahmebehandlung (*exception handling)* in Python `try` … `except` …
- Ausnahmen selbst erzeugen `raise`
- **Programmierstile:** EAFP oder LBYL
- Zusammenfassung

Ausnahmen (*exceptions*) – Was sind das?

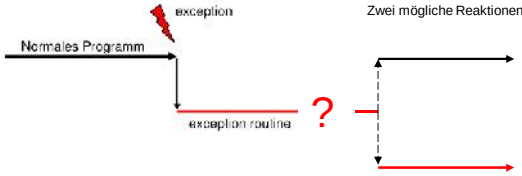
- Rein sprachlich bedeutet **Ausnahme** etwas, das von einer Regel abweicht, also ein **Sonderfall** (sagt der Duden).
- Implizit bezeichnet Ausnahme auch etwas, was **sehr selten** auftritt: Die Regel beschreibt das Häufige.

Ausnahmen (*exceptions*) konkret

- **Ausnahmen bezeichnen Situationen oder Ereignisse**, in denen der **programmierte Kontrollfluss nicht fortgesetzt** werden kann oder soll, ohne das danach ggf. schwerwiegende (und dann ggf. schwer erkennbare) Probleme entstehen.
- Auf eine Exception wird mit dem Start eines speziellen Programms (der Exceptionroutine / der Ausnahmebehandlung) reagiert.
- Das gerade laufende „normale“ Programm wird dazu unterbrochen.



Ausnahmebehandlung (*exception handling*)



Zwei mögliche Reaktionen

Das "Normale Programm" wird fortgesetzt.

z.B. Programmabbruch, das supervising Programm: (Interpreter, Betriebssystem) übernimmt die Kontrolle und beendet das "Normale Programm"

- Der Vorgang erscheint sehr ähnlich zu dem des Unterprogramms- / Methoden-Aufrufes.
  - **Unterprogrammaufruf:** die Startadresse des Unterprogramms wird vom Programmierer im "Normalen Programmcode" festlegt.
  - **Exception:** die Startadresse der Exceptionroutine wird vom überwachenden Programm (Interpreter, Betriebssystem) festgelegt.
- In beiden Fällen wird der Prozesskontext auf den Stack gerettet.

Ursachen für Exceptions

Wir unterscheiden:

- Programm-**interne Ursachen**

Traps (Fallen) = Fehler Beispiele
- Division durch Null ➔ HW
- Typfehler ➔ IP
- Überschreitung vorgegebener Speichergrenzen ➔ OS

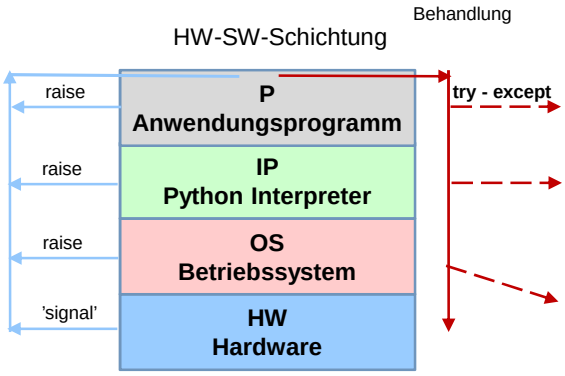
Software-Interrupt
wird vom Programm ausgelöst.

Beruhen oft auf Fehlersituationen.

- Programm-**externe Ursachen**
  = Interrupts
  - Busfehler
  - RESET-Taste wurde aktiviert
  - "Keyboard Interrupt" weil der Benutzer Control-C drückt
- HW-Interrupt ausgelöst, meist I/O
- Scheduling

Beruhen **nicht auf einer Fehlersituation**
Sie sind **" gewollte" Programm-unterbrechungen**

Ausnahmebehandlung (*exception handling*) konzeptionelle Vorstellung



Hier können einzelne **benamte Ausnahmen** abgefangen und behandelt werden, siehe unten.
nichtbehandelte Exceptions werden weitergereicht..

Alle nicht behandelten benamte exceptions werden behandelt
➔ **Abbruch von P**

Interrupts werden behandelt
IP + P laufen dann weiter
oder
Fatal error des OS
=~ BlueSreen

Behandlung durch den Interpreter

Das kennen Sie alle:

```
>>> 10/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
>>>
```

also: Angabe wo (Filename, Zeilennummer, in <modulname>)
welches Statement
welche **benamte Ausnahme** erzeugt hat
➔ Interpreter **bricht aktuellen Programmlauf ab** und geht in den Eingabemodus

Die *exception hierarchy* des Python Interpreters 3.6.2 (1)

Dies sind bis auf **meist** **Fehlersituationen *Error* !**

Ist eine Klassenhierarchie (Vererbung!) der sogenannten built-in exceptions:

Superklasse: Exeption

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
```

exception hierarchy des Python Interpreters 3.6.2 (2)

Superklasse: Exeption

```
+-- NameError
|   +-- UnboundLocalError
+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|   |   +-- BrokenPipeError
|   |   +-- ConnectionAbortedError
|   |   +-- ConnectionRefusedError
|   |   +-- ConnectionResetError
|   +-- FileExistsError
|   +-- FileNotFoundError
```

Superklasse OSError

```
|   +-- InterruptedError
|   +-- IsADirectoryError
|   +-- NotADirectoryError
|   +-- PermissionError
|   +-- ProcessLookupError
|   +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
```


exception hierarchy des Python Interpreters 3.6.2 (3)

Superklasse: Exeption

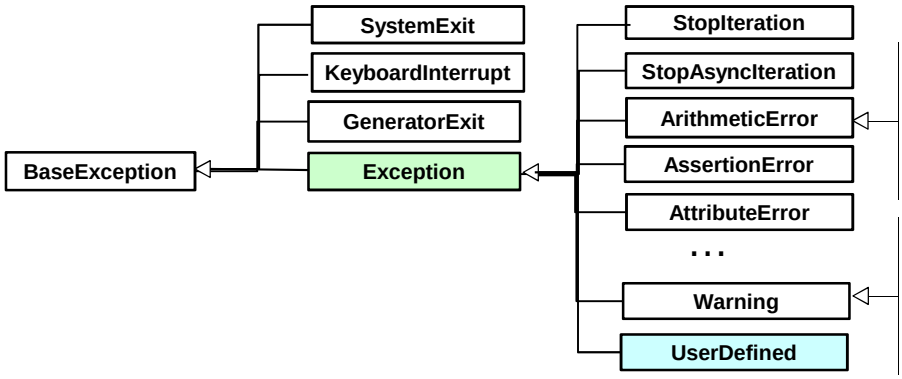
```
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
```

Superklasse: Exeption

```
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

Es lohnt sich, **die präzise Bedeutung der Exceptions** in der "Python Standard Library -- 5 Built-in Exceptions" nachzulesen, siehe:
https://docs.python.org/3.6/library/exceptions.html#concrete-exceptions

Die oberen Ebenen der exception hierarchy als UML Klassendiagramm



Warnings sind "spezielle" exceptions

- werden typischerweise in Situationen erzeugt, in denen (noch) kein echter Fehler aufgetreten ist, der zu einem Programmabbruch führen müsste.
- Trotzdem sollte der User (die ProgrammiererIn) darüber informiert sein.
- **Beispiel:** ein Programm nutzt ein obsoletes Modul.
- In Python ist dies als Exception eingeordnet und nutzt dieselben Mechanismen.

Ausnahmebehandlung (*exception handling*)

- Fast alle modernen Programmiersprachen wie C++, Objective-C, PHP, Java, … **und so auch Python** besitzen syntaktischen Strukturen um Ausnahmebehandlungen zu ermöglichen.
- Ausnahmebehandlungen sind "teuer", d.h. kosten erhebliche Laufzeiten ➔ nur Teile des gesamten Codes werden überwacht, nie alles!
- In Python wird der Code, der das Risiko für eine Ausnahme enthält, in einen
  - **`try`-Block eingebettet** und dem
  - except-Schlüsselwort mit dem Klassennamen derzu behandelnden Ausnahme
    Es folgt der Code zur Ausnahmebehandlung.

Beispiel

Die Eingabe von Daten sind typische Situationen, die man durch exceptions abfangen kann:

```
>>> n = int(input("Bitte geben Sie eine ganze Zahl ein: "))
Bitte geben Sie eine ganze Zahl ein: 1.25
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    n = int(input("Bitte geben Sie eine ganze Zahl ein: "))
ValueError: invalid literal for int() with base 10: '1.25'
>>>
```

Das wollen wir natürlich nicht!

Das Abfangen fehlerhafter Eingaben mit `try … except`

```
while True:
    try:
        n = input("Bitte eine ganze Zahl eingeben: ")
        n = int(n)
        break
    except ValueError:
        print("Leider keine ganze Zahl eingegeben! \
Bitte nochmals versuchen ...")
print ("Genau so, danke! -- jetzt gehts weiter ... ")
```

```
========== RESTART: C:/Users/kroemker/My_Python_Projects/test.py ==========
Bitte eine ganze Zahl eingeben: 1.2
Leider keine ganze Zahl eingegeben! Bitte nochmals versuchen ...
Bitte eine ganze Zahl eingeben: 2
Genau so -- jetzt gehts weiter ...
>>>
```


Beispiel Fortsetzung: Mehrere Ausnahmen abfangen, unbekannter Fehler

- Man kann durch mehrere `except` auch mehrere Fehler abfangen.
- Auch das Abfangen irgendeines beliebigen Fehlers ist möglich.

```
while True:
    try:
        n = input("Eine ganze Zahl bitte != 0: ")
        n = int(n)
        n = 1/n
        break
    except ValueError:
        print("Leider keine ganze Zahl eingegeben! \
Bitte nochmals versuchen ...")
    except:
        print("Nicht erwarteter Fehler. Bitte noch einmal: ")
print ("Genau so, danke! -- jetzt gehts weiter ... ")
```

Beispiel Fortsetzung: der else-Zweig

- Wenn keine exception generiert wird, wird, wenn vorhanden, der else-Zeig ausgeführt.

```
while True:
    try:
        n = input("Eine ganze Zahl bitte != 0: ")
        n = int(n)
    except ValueError:
        print("Leider keine ganze Zahl eingegeben! \
Bitte nochmals versuchen ...")
    else:
        print("Prima, kein Fehler. Eingabe = ", n)
        break
print ("Genau so, danke! -- jetzt gehts weiter ... ")
```


Beispiel Fortsetzung: der finally-Zweig

- Der finally-Zweig wird in jedem Fall ausgeführt!

```
while True:
    try:
        n = input("Eine ganze Zahl bitte != 0: ")
        n = int(n)
    except ValueError:
        print("Leider keine ganze Zahl eingegeben! \
Bitte nochmals versuchen ...")
    else:
        print("Prima, kein Fehler. Eingabe = ", n)
        break
    finally:
        print("Dies wird bei jedem Durchgang ausgeführt!")
print ("Genau so, danke! -- jetzt gehts weiter ... ")
```

Bekannte Ausnahmen selbst erzeugen: `raise` (1)

- Fehler sind nicht immer zu vermeiden.
- Wenn Sie Skripte erstellen, die auf **Fehler vorbereitet sind** und darauf reagieren können, sparen Sie sich Zeit und Nerven.
- Die `raise`-Anweisung erlaubt der Programmierer*in, das Auslösen einer bestimmten Ausnahme zu erzwingen. Zum Beispiel:

```
>>> raise NameError('HeyDu') # einen der obigen Exeptions
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise NameError('HeyDu')
NameError: HeyDu
>>>
```


Bekannte Ausnahmen selbst erzeugen: `raise` (2)

- Das einzige Argument des Schlüsselwortes `raise` gibt den Klassennamen der Ausnahme an, die ausgelöst werden soll.
- Dies muss entweder eine Ausnahme-Instanz sein oder eine Ausnahmeklasse (eine Klasse, die von Exception erbt).

Bekannte Ausnahmen selbst erzeugen: `raise` (2)

- Wenn man "nur" wissen will, ob eine Ausnahme ausgelöst wurde, sie aber nicht behandeln willst, erlaubt es die `raise`-Anweisung (ohne Argument), dieselbe Ausnahme erneut auszulösen:

```
try:
    raise NameError('HeyDu')
except NameError:
    print('Eine Ausnahme flog vorbei!')
    raise
```

```
Eine Ausnahme flog vorbei!
Traceback (most recent call last):
  File "C:/Users/kroemker/AppData/Local/Programs/Python/Python36/
  test except.py", line 2, in <module>
    raise NameError('HeyDu')
NameError: HeyDu
```


Benutzerdefinierte Ausnahmen (1)

- Programme können ihre eigenen Ausnahmen **benennen**, indem sie eine neue Ausnahmeklasse erstellen.
- Ausnahmen sollten standardmäßig von der Klasse Exception erben, entweder direkt oder indirekt. Zum Beispiel:

```
class MyError(Exception):

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)
```

In diesem Beispiel wird die Methode `__init__()` der Klasse Exception überschrieben. Das neue Verhalten erzeugt schlicht das Attribut value, es ersetzt das Standardverhalten, ein Attribut args zu erzeugen.

Wir fügen hinzu:

```
class MyError(Exception):

    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print('Meine Ausnahme wurde ausgelöst, Wert:', e.value)


>>>
 RESTART: C:/Users/kroemker/AppData/Local/Programs/Python/Python36/test excep
Meine Ausnahme wurde ausgelöst, Wert: 4
```


Ausnahmeklassen …

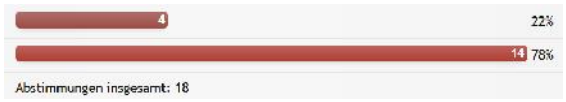
- können **alle** Möglichkeiten nutzen, die bei der Definition von Klassen zur Verfügung stehen, werden jedoch meist recht einfach gehalten.
- Oft bieten sie nur eine Reihe von Attributen, welche genauere Informationen über den Fehler bereitstellen.
- Beim Erstellen von Modulen, welche verschiedene Fehler auslösen können, wird oft eine Basisklasse für Ausnahmen dieses Moduls definiert und alle anderen Ausnahmen für spezielle Fehlerfälle erben dann von dieser Basisklasse.
- Meistens gibt man den Ausnahmeklassen Namen, die auf “Error” enden, ähnlich der Namensgebung der Standardausnahmen.
- **Viele Standardmodule definieren ihre eigenen Ausnahmen,** um Fehler zu melden, die in ihren Funktionen auftreten können.

Error Handling Umfrage: raise vs. return

- Im deutschen Python Forum findet sich unter

- https://www.python-forum.de/viewtopic.php?t=6689 folgende Umfrage:
- Error Handling Umfrage: raise vs. Return:
- Welche Methode benutzt ihr häufiger?

return ERROR_CODE
raise Exception



Rückgabe von Fehlermeldungen

```
def function(intval):
    if intval == 0: return -1
    #...
```

Klassische Fehlermeldung

oder

```
def function(intval):
    if intval == 0: raise ValueError
    #...
```

Exception Meldung

Python Prinzip EAFP

- **"It's Easier to Ask Forgiveness than Permission."**
  "*Es ist einfacher, um Vergebung zu bitten als um Erlaubnis*."
- Dieser allgemeine Python-Codierungsstil setzt die Existenz gültiger Schlüssel oder Attribute voraus und fängt Ausnahmen ab, wenn sich die Annahme als falsch erweist.
- Dieser saubere und schnelle Stil ist durch das Vorhandensein vieler `try` und `except` Anweisungen gekennzeichnet.
- Die Technik kontrastiert mit dem **LBYL-Stil**, der in vielen anderen Sprachen wie C häufig genutzt wird:
- LBYL: **Look Before You Leap**
  "*Schaue bevor du springst."*

Ein Beispiel zum EAFP-Prinzip

- EAFP:

```
try:
    x = my_dict["key"]
except KeyError:
    # handle missing key
```

Die LBYL-Version muss den Schlüssel innerhalb des dictionarys **zweimal** auflösen!

Ggf. auch etwas weniger lesbar!

LBYL:

```
if "key" in my_dict:
    x = my_dict["key"]
else:
    # handle missing key
```

Zusammenfassung

- Eine Ausnahme (exception) bezeichnet ein Verfahren, Informationen über bestimmte Programmzustände – häufig Fehlerzustände – an andere Programmebenen zur Weiterbehandlung weiterzureichen.
- Diese Verfahren werden in Python vergleichsweise häufig genutzt: EAFP-Programmierstil.
- Bitte nicht der "alten" C-Metapher LBYL stramm nachhängen.

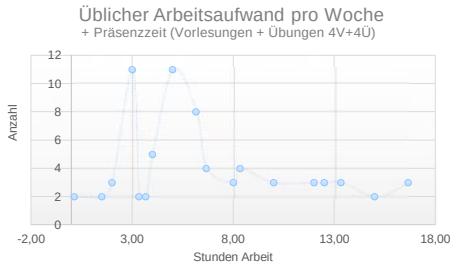
Anmeldung zur Klausur

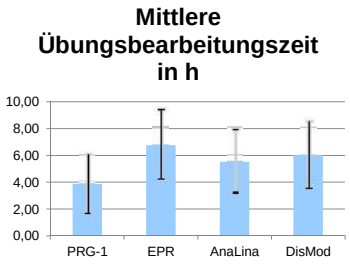
- Wichtig: Für alle Anmeldungen beim PA gilt:
  Anmeldeschluss: 9.2. (Ende dieser Woche!)
  Man kann sich wieder abmelden bis 16.2.

Die Alternative für Nicht- QIS/LSF Anmeldungen (sonstige Naturwissenschaften, Lehramt, etc.)
  Anmeldeschluss: 16.2.
  allerdings steht diese Frist noch auf: 28.01.2018. Wird korrigiert spätestens Mittwoch.

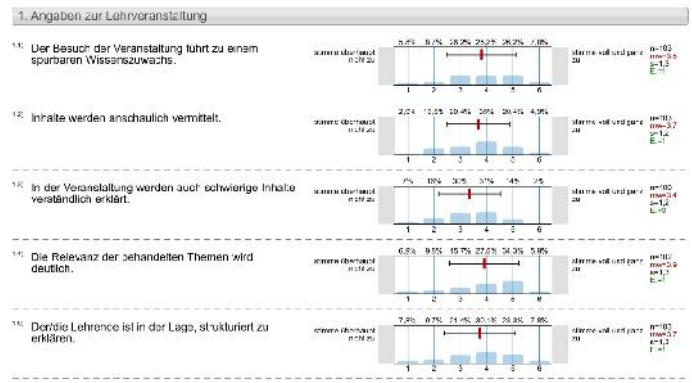
Rückblick Veranstaltungsevaluation PRG1

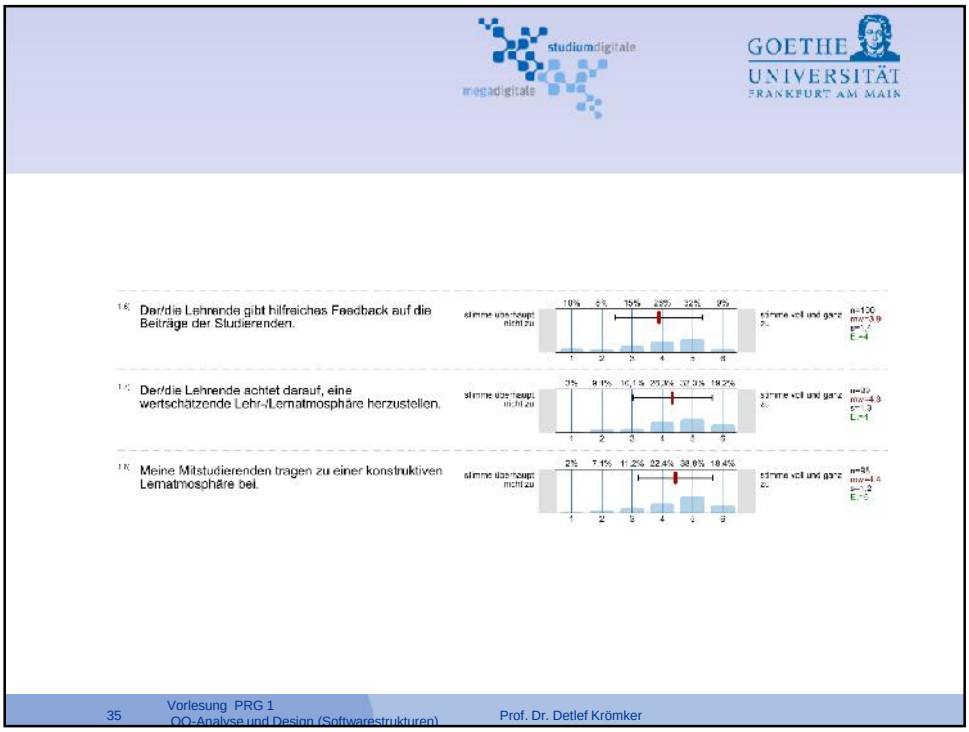
- Diese Evaluation ist dieses Semester deutlich (zu) spät (wg. allgemeiner Evaluation. Danke für die Teilnahme
- Eine Übersicht zum Aufwand der Übungen: Soll wäre 12-14 h

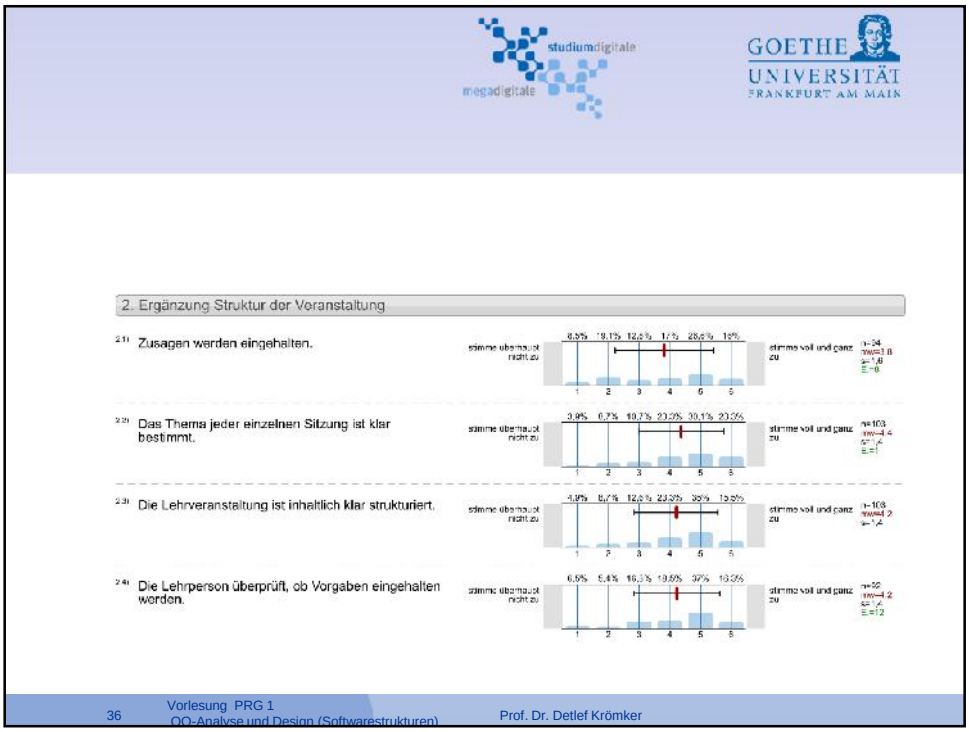




Übersicht

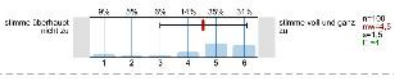



1.8 Der/die Lehrende gibt hilfreiches Feedback auf die Beiträge der Studierenden.
1.9 Der/die Lehrende achtet darauf, eine wertschätzende Lehr-/Lernatmosphäre herzustellen.
1.10 Meine Mitstudierenden tragen zu einer konstruktiven Lernatmosphäre bei.


2. Ergänzung Struktur der Veranstaltung
2.1 Zusagen werden eingehalten.
2.2 Das Thema jeder einzelnen Sitzung ist klar bestimmt.
2.3 Die Lehrveranstaltung ist inhaltlich klar strukturiert.
2.4 Die Lehrperson überprüft, ob Vorgaben eingehalten werden.

3. Ergänzung eLearning

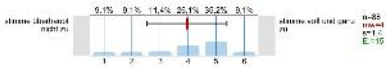
3.1 Ich habe das eLearning-Angebot aktiv genutzt und damit gelernt/gearbeitet (d.h., nicht nur heruntergeladen, durchgeschaut oder oberflächlich gelesen).



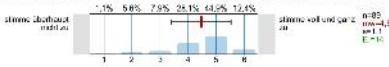
Allgemeine Beurteilung des E-Learning Angebotes der Lehrveranstaltung

3.2 Das eLearning-Angebot der Lehrveranstaltung bringt für das Lernen/Arbeiten eine Verbesserung.

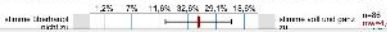
3.3 Das eLearning-Angebot und die Präsenzveranstaltung sind gut aufeinander abgestimmt.



3.4 Die Lehrinhalte des eLearning-Angebots waren gut verständlich.

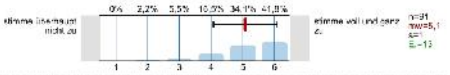


3.5 Die investierte Zeit in die Arbeit mit dem eLearning-Angebot ist im Verhältnis zum Lernerfolg

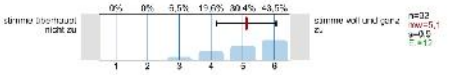


Bedarf an weiteren E-Learning Angeboten

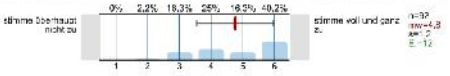
3.13 Man sollte Möglichkeiten ausbauen, Lernmaterialien und Lernstoff über das Internet zu bekommen.



3.14 Man sollte Möglichkeiten zur Anwendung und Einübung des Lernstoffes über das Internet ausbauen (z.B. Online-Übungen, Selbsttests).



3.15 Man sollte Möglichkeiten zur Kommunikation und Kooperation über das Internet ausbauen (z.B. durch Foren, Lernplattform, Wiki).



Ausblick

... ersteinmal viel Erfolg in DisMod

... und spätestens dann eine gute Vorbereitung für die Klausur in PRG1

und, herzlichen Dank für Ihre Aufmerksamkeit!