Python Grundlagen

Julian Huber & Matthias Panny

Exceptions, Logging & Unit-Tests

🎯 Lernziele

- Studierende können Ausnahmebehandlung in ihren Code integrieren
- Studierende können Logfiles anlegen
- Studierenden können Unit-Test anlegen und auswerten

Ausnahmebehandlung

(engl. exception handling)

Ausnahmebehandlung

Warum?

- (Un)erwartete Fehler im Code zur Laufzeit
- Programme sollen nicht abstürzen
- Behandlung von Fehlern zur Laufzeit

Try-Except-Blöcke - `try_except_simple.py`

- Konzept identisch wie in C++ → teils leicht andere Syntax

```python
numbers = [1,2,0,4,5]

for number in numbers:
  # Versuche folgende Operation
  try:
    result = 10 / number
    print(f"{10} / {number} = {result}")

  # Wenn ein ZeroDivisionError auftritt
  except ZeroDivisionError:
    # Mache folgendes
    print("Division durch Null ist nicht erlaubt!")
    # Anstelle die Ausführung des Programms zu unterbrechen
```

Ausnahmebehandlung

- `try`-Block führt einen Code-Block versuchsweise aus
- `except`-Block wird ausgeführt, wenn ein Fehler auftritt → fängt ausgewählten Fehler (Exception Types) ab → lassen sich auch getrennt abfangen
- `else`-Block wird ausgeführt, wenn keine Exception auftritt
- `finally`-Block wird auf jeden Fall ausgeführt
- `raise` wirft eine Exception

Beispiel - try_except_full.py

```
numbers = [1,2,0,4,5]

for number in numbers:
  try:
    result = 10 / number
    print(f"{10} / {number} = {result}", end=" --> ")
  except ZeroDivisionError:
    print("Division durch Null ist nicht erlaubt!", end=" --> ")
  else:
    print("Operation erfolgreich!", end=" --> ")
  finally:
    print("Ende der Operation")
```

Ausnahmebehandlung

- In der Praxis wird oft nur der `try`- und `except`-Block verwendet
- Oft auch mit mehreren `except`-Blöcken

Beispiel - `try_except_multiple.py`

```
numbers = [1, 2, 0, 3, "4", 5]

for number in numbers:
  try:
    result = 10 / number
    print(f"{10}/{number}={result}")

  # Wenn ein ZeroDivisionError auftritt
  except ZeroDivisionError:
    # Mache folgendes
    print("Division durch Null ist nicht erlaubt!")
    # Anstelle die Ausführung des Programms zu unterbrechen

  # Wenn ein TypeError auftritt
  except TypeError:
    print("Division von String nicht erlaubt!")
```

Assertions

- Automatisierte Überprüfung von Annahmen → Exception wird geworfen, wenn Annahme falsch ist
- Keyword `assert` gefolgt von einem Bool'schen-Ausdruck
- Wird üblicherweise in Entwicklungsphasen verwendet → nicht in Produktionscode → kann mit `-O`-Flag deaktiviert werden

Beispiel - `assert_example.py`

```
numbers = [1, 2, 0, 3, "4", 5]

for number in numbers:
  try:
    assert isinstance(number, (int, float))
    assert number != 0.0
  except AssertionError:
    print("Assertion failed!")

print("Ran through all the numbers!")
```

- Deaktivierung mit `-O` Flag:

```
$ python ./assert_example.py
Assertion failed!
Assertion failed!
Ran through all the numbers!

$ python -O ./assert_example.py
Ran through all the numbers!
```

`with`-Statement

- Konstrukt aus `try-finally`-Blöcken wird oft für Ressourcenmanagement verwendet → egal ob eine Exception auftritt oder nicht, Ressourcen müssen freigegeben werden
- Kann mit `with`-Statement vereinfacht & lesbarer gemacht werden
- Kein Ersatz für `try-except`-Blöcke → `except` kann angehängt werden

Beispiel - `no_with_statement.py`

```
path = "my_file.txt"

file = open(path, "w")
try:
  file.write("Hello World!")
finally:
  file.close()
```

Beispiel - `with_statement.py`

```
path = "my_file.txt"

with open(path, "w") as file:
  file.write("Hello World!")
```

✍️ Aufgabe

- Wir wollen eine Funktion schreiben, die eine Ganzzahldivision durchführt und das Ergebnis, sowie den Rest zurückgibt
- Es soll im Falle einer Division durch Null eine Exception geworfen werden

Musterlösung - `division_excepction.py`

- Variante 1: Excpetion wird geworfen und weitergegeben
- Variante 2: Exception wird abgefangen und `None` wird zurückgegeben

Logging

Logging

- Protokollieren von Ereignissen im Programm → z.B. Fehler, Warnungen, aber auch nur Informationen
- Aus vielen Gründen hilfreich für Entwickler:
  - Fehleranalyse
  - Performanceanalyse
  - Überwachung
  - Debugging

Wann sind Logfiles besonders wichtig?

- Wenn der Anwender nicht direkt Entwickler des Programms ist
- Wenn das Programm auf einem anderen Rechner, Server oder Embedded-System läuft
- etc.

Logfiles

- Python bietet ein eingebautes Logging-Modul
- Konfiguration des Loggings über `logging.basicConfig()` → es kann unterschieden werden was geloggt wird

Definition in Python - `logging_example.py`

```
import logging

# Setzt die Konfiguration für das Log
# Name des Log-Files, in das die Logs alles was kritischer ist als INFO)
logging.basicConfig(filename='app.log', level=logging.INFO)

# Erstellt eine neue Zeile im Log-File mit dem String als Text
logging.info('Programm gestartet')
```

Inhalt des Logfiles `app.log`

```
INFO:root:Programm gestartet
```

Logfiles

- Loglevel bestimmt, welche Nachrichten geloggt werden → alle gleich oder kritischeren Nachrichten werden geloggt
- z.B. `CRITICAL` und `ERROR` werden geloggt, wenn das Level auf `WARNING` gesetzt wurde

Logging Levels

Level	Num	Bedeutung
`logging.DEBUG`	10	Nur für Entwickler:innen relevant
`logging.INFO`	20	Bestätigung, dass alles läuft, wie geplant
`logging.WARNING`	30	Hinweis auf etwas unerwartetes, das zu Problemen führen könnte (e.g. ‘disk space low’). Noch läuft die Software, wie geplant
`logging.ERROR`	40	Ernsthaftes Problem, eine Funktion der Software konnte nicht erfüllt werden
`logging.CRITICAL`	50	Ernsthaftes Problem, das Programm ist nicht mehr lauffähig.

Logging von Exceptions

- Exceptions können auch geloggt werden → `logging.exception()` loggt eine Exception mit dem Level `ERROR`

Logging von Exceptions - `log_exception.py`

```
import logging
logging.basicConfig(
  filename="app.log",
  level=logging.INFO,
  format='%(asctime)s %(message)s'
)

logging.info('Programm gestartet')

try:
  assert 1 == 0
except AssertionError as err:
  logging.exception("My assert failed :( ")
  raise err
```

app.log

```
2024-10-29 09:36:25,823 Programm gestartet
2024-10-29 09:36:25,824 My assert failed :(
Traceback (most recent call last):
  File "logs.py", line 14, in <module>
    assert 1 == 0
AssertionError
```

Unit Tests

(dt: Modultests, Komponententests)

Unit Tests

- Tests, die einzelne Einheiten von Code auf ihre Funktionalität testen
- Prüfen ob jede Funktion korrekt arbeitet
- Sind automatisiert und wiederholbar → sollten nach jeder Änderung im Code ausgeführt werden
- Daher oft Teil des *Continuous Integration* (CI) Prozesses

Unit Tests

- Es wird für jede Funktion ein Test-Funktion geschrieben, in der die Funktionalität mittels `assert`-Statements überprüft wird
- `AssertionError` signalisiert fehlgeschlagenen Test

Beispiel - `unit_test_example.py`

```python
# Implementierung der Funktion
def add(a: float, b: float) -> float:
  return a + b

# Testmethode für Addition mit add()
def test_add():
  assert add(2, 3) == 5
  assert add(0, 0) == 0
  assert add(-1, 1) == 0

# Aufruf der Test-Funktion
try:
  test_add()
except AssertionError:
  print("Test failed!")
else:
  print("Test passed!")
```

Unit Test

- Um Handling von Tests zu vereinfachen, gibt es Test-Frameworks
- `unittest` ermöglicht die Definition von Test-Cases in eigenen Skripten → Teil der Python Standard Library
- Automatisches Ausführen von Tests für bestimmte Module
- Testergebnisse werden angezeigt

Gliederung bei kleinen Projekten - `test_example`

- Test-Datei wird im gleichen Verzeichnis wie die zu testende Datei abgelegt

```
test_example
├── geometries.py
└── test_geometries.py
```

- Ausführen mittels:

```
$ cd test_example
$ python -m unittest test_geometries
```

Aufbau von Unit Test

geometries_simple.py

```python
# Our code to be tested
class Rectangle:
  def __init__(self, width, height):
    self.width = width
    self.height = height

  def get_area(self):
    return self.width * self.height

  def set_width(self, width):
    self.width = width
  def set_height(self, height):
    self.height = height
```

test_geometries_simple.py

```python
import unittest
from geometries_simple import Rectangle
# Der TestCase basierend auf der unittest Klasse
# --> kann mehrere Testfunktionen enthalten
class TestGetAreaRectangle(unittest.TestCase):
  def runTest(self):
    rectangle = Rectangle(2, 3)
    self.assertEqual(rectangle.get_area(), 6, "incorrect area")
    # Könnte auch weitere Assertions enthalten
```

✍️ Aufgabe

- Wir wollen die Abdeckung unserer Unit Tests erweitern → mehr Funktionalität soll automatisch getestet werden
- Ergänzen Sie eine Methode, die den Umfang ausrechnet und einen Unit-Test-Case dazu, der zwei verschiedene Abfragen enthält.

Musterlösung

- `geometries.py`
- `test_geometries.py`

Unit Test

- Bei Großen Projekten werden Test-Dateien in eigenen Verzeichnissen abgelegt
- Test-Dateien werden als Pakete behandelt → `__init__.py`-Datei wird benötigt

Gliederung großen Projekten - `test_example_subfolders`

```
test_example_subfolders
├── geometries
│   ├── __init__.py          # make it a package
│   └── geometries.py
└── test
    ├── __init__.py          # also make test a package
    └── test_geometries.py

$ cd test_example_subfolders
$ python -m unittest test.test_geometries
```

✍️ Aufgabe

- Wir wollen uns Beispiel noch weiter ausbauen
- Schreiben Sie ein Skript, dass es einem User erlaubt ein neues Rechteck, über die den user-input `input()` anzulegen → der Nutzer soll aufgefordert werden Länge und Breite einzugeben
- Es soll der Versuch gestartet werden, Eingaben, die z.B. als String getätigt wurden in eine Zahl umzuwandeln. Tritt dabei ein `ValueError` auf, so soll der Nutzer erneut aufgefordert werden Länge und Breite des Rechtecks einzugeben
- Passen Sie den Code so an, dass auch keine Seitenlänge oder -Höhe von 0 zulässig ist, indem Sie ein `assert`-Statement in den `try`-Block einbauen und einen zweiten `except`-Block für einen `AssertionError` einfügen.

Musterlösung - `user_interface.py`

- Nutzt `Rectangle`-Klasse aus `geometries.py`
- Implementiert `ask_user_for_dimensions()`-Funktion

✍️ Aufgabe

- Das vorherige Beispiel kann nun auch noch um das Logging von Fehlern, etc. erweitert werden
- Es soll jedes erzeugte Rechteck mit Info-Level geloggt werden
- Jedes nicht erfolgreich erzeugte Rechteck soll mit Warning-Level geloggt werden

Musterlösung - `user_interface_log.py`

- Fügt im `else`-Block ein `logging.info()` hinzu
- Fügt bei allen `except`-Blöcken ein `logging.warning()` hinzu

🏆 Hausübung

- Es soll mit Hilfe des Maschenstromverfahrens eine Netzwerkanalyse durchgeführt werden
- Bei diesem Verfahren können die Ströme in einem elektrischen Netzwerk berechnet werden. Es basiert auf der Anwendung des Kirchhoff'schen Maschengesetzes auf die einzelnen Maschen des Netzwerks.

Aufgabe

- Stellen Sie die Maschen- & Knotengleichungen für das Beispiel im Notebook auf
- Bilden Sie daraus ein Gleichungssystem
- Lösen Sie dieses Gleichungssystem

Für Interessierte

- Automatisertes Durchführen einer Netzwerkanalyse mit dem Maschenstromverfahren mittels LTSpice & Python