Fehlerbehandlung

>> **Robuste** Programme sind in der Lage, auf Fehler, die zur Laufzeit auftreten, angemessen zu reagieren (*error handling*) anstatt mit einer Fehlermeldung abzustürzen.

Lösungsansätze: **Exceptions** behandeln, **defensiv** programmieren und frühzeitig systematisch **testen**, um möglichst viele Fehler (*Bugs*) zu finden und zu beheben.

Exceptions

Problem zur Motivation von Exceptions

```python
def withdraw_money(balance, amount):
    return balance - amount


# Einfacher Aufruf der Funktion
new_balance = withdraw_money(100, 200)
print(f"New balance: {new_balance}")  # Gibt -100 aus
```

Naive Fehlerbehandlung

```python
def withdraw_money(balance, amount):
    if amount > balance:
        return -1  # Fehlerwert für "nicht genug Guthaben"
    return balance - amount


# Problem: Wie unterscheiden wir zwischen Fehler und echtem Kontostand?
new_balance = withdraw_money(100, 200)
if new_balance == -1:
    print("Not enough money")

# Was ist, wenn -1 ein gültiger Kontostand sein könnte? 🤔
```

Exceptions als bessere Lösung

```python
def withdraw_money(balance, amount):
    if amount > balance:
        raise ValueError("Not enough money in account")
    return balance - amount


# Verwendung mit try-except
try:
    new_balance = withdraw_money(100, 200)
except ValueError as e:
    print(f"Error: {e}")
    # Gibt "Error: Not enough money in account" aus
```

Es gibt verschiedene Exceptions für unterschiedliche Fehlerfälle.

```
try:
    number = int(input("Enter amount: "))    # ValueError möglich
    result = 100 / number                    # ZeroDivisionError möglich
except ValueError:
    print("Please enter a valid number")     # Behandlung ungültiger Eingabe
except ZeroDivisionError:
    print("Amount cannot be zero")           # Behandlung Division durch 0
```

Umgang mit beschränktem Variablen-Scope: try – except – else

```
# Problematisch:
try:
    result = int("not a number")
except ValueError:
    print("Conversion failed")
print(result)  # NameError! result existiert nicht


# Besser mit else:
try:
    result = int("123")
except ValueError:
    print("Conversion failed")
else: # Wird nur ausgeführt wenn kein Fehler auftritt
    print(result)
```

Kompakte Behandlung unterschiedlicher Fehlerfälle

```
# Mehrere Exception-Typen auf einmal behandeln
try:
    amount = float(input("Enter amount: "))
    result = 100 / amount
except (ValueError, ZeroDivisionError) as error:
    print(f"Invalid input: {error}")
    # error enthält die detaillierte Fehlermeldung
```

Zugriff auf Exception-Objekt: `except ... as name`

```python
try:
    file = open("config.txt")
except FileNotFoundError as e:
    print(f"Error type: {type(e)}")          # Typ der Exception
    print(f"Error message: {str(e)}")        # Fehlermeldung als String
    print(f"Error details: {e.args}")        # Weitere Details
```

Exceptions müssen nicht an Ort und Stelle behandelt werden: Weitergabe mit *raise*

```python
def process_data(filename):
    try:
        data = read_file(filename)     # Könnte FileNotFoundError werfen
        return analyze_data(data)      # Könnte ValueError werfen
    except FileNotFoundError:
        # Behandle Dateifehler hier
        raise  # Original-Exception weitergeben
    except ValueError as e:
        # Neue Exception mit zusätzlichen Infos werfen
        raise ValueError(f"Analysis failed: {e}") from e
```

Beispiel

```python
def read_amount():
    amount = float(input("Enter amount: "))
    if amount <= 0:
        raise ValueError("Amount must be positive")
    return amount


def process_payment():
    try:
        amount = read_amount()
        # Weitere Verarbeitung...
    except ValueError as e:
        print(f"Payment failed: {e}")


# Hauptprogramm
def main():
    try:
        process_payment()
    except Exception as e:
        print("A system error occurred!")
        log_error(e)  # Fehler loggen
```

Beispiel

```python
def read_amount():
    amount = float(input("Enter amount: "))
    if amount <= 0:
        raise ValueError("Amount must be positive")
    return amount


def process_payment():
    try:
        amount = read_amount()
        # Weitere Verarbeitung...
    except ValueError as e:
        print(f"Payment failed: {e}")


# Hauptprogramm
def main():
    try:
        process_payment()
    except Exception as e:
        print("A system error occurred!")
        log_error(e)  # Fehler loggen
```

```
main()
   └─> process_payment()
          └─> read_amount()
                 └─> ❌ ValueError
                     ↑
                     Exception wandert nach oben
                     bis sie behandelt wird
```

Systematisches Testen

```python
class BankAccount:
    def __init__(self, initial_balance):
        self.balance = initial_balance
        self.transactions = []


    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            self.transactions.append((-amount, "withdrawal"))
            return True
        return False


    def deposit(self, amount):
        self.balance += amount
        self.transactions.append((amount, "deposit"))


    def get_statement(self):
        return f"Balance: {self.balance}"
```

Warum systematisch testen?

Weil unsystematisches manuelles Testen zeitaufwändig und fehleranfällig ist, oft nicht alle Randfälle abdeckt und Fehler übersieht, die erst im Produktivbetrieb auftreten.

Von manuellen Tests zum systematischen Testen

```python
def multiply_positive(a, b):
    if a <= 0 or b <= 0:
        raise ValueError("Numbers must be positive")
    return a * b


# Naiver Test-Ansatz mit if-Statements
if multiply_positive(2, 3) != 6:
    print("Test failed: 2 * 3 should be 6")


# Fehlerfall testen - umständlich!
try:
    multiply_positive(-2, 3)
    print("Test failed: should raise error "
          "for negative numbers!")
except ValueError:
    print("OK: Error correctly raised")
```

Verbesserung durch Nutzung von *assert*

```python
def test_multiply():
    assert multiply_positive(2, 3) == 6, "Basic multiplication failed"
    assert multiply_positive(1, 1) == 1, "Edge case failed"

    # Exception testen - immer noch umständlich
    try:
        multiply_positive(-2, 3)
        assert False, "Should have raised ValueError"
    except ValueError:
        pass  # Test erfolgreich
```

Weitere Verbesserung: Testen mit *pytest*

```python
# test_multiply.py
import pytest
from calculator import multiply_positive


def test_basic_multiplication():
    assert multiply_positive(2, 3) == 6


def test_edge_cases():
    assert multiply_positive(1, 1) == 1


def test_negative_number():
    with pytest.raises(ValueError):
        multiply_positive(-2, 3)
```

Testfunktionen beginnen mit `test_`

Jeder Test prüft genau einen Aspekt.

Mit `with pytest.raises()` lassen sich erwartete Exceptions besser testen.

Testen von Exceptions: vorher/nachher im Vergleich

```python
# Ohne with - umständlich und fehleranfällig
try:
    multiply_positive(-2, 3)
    assert False  # Sollten wir nie erreichen
except ValueError:
    pass  # Test erfolgreich


# Mit with - elegant und sicher
with pytest.raises(ValueError):
    multiply_positive(-2, 3)
```

Aufruf von pytest

```
$ pytest test_multiply.py  # Einzelne Testdatei ausführen

$ pytest     # Alle Tests im aktuellen Verzeichnis

$ pytest -v  # Mit detaillierter Ausgabe
```

```
============================= test session starts ==============================
platform linux -- Python 3.8.5, pytest-6.2.4
collecting ...
collected 3 items

test_multiply.py::test_basic_multiplication PASSED      [ 33%]
test_multiply.py::test_edge_cases PASSED                [ 66%]
test_multiply.py::test_negative_number PASSED           [100%]

============================== 3 passed in 0.02s ===============================
```

Wenn Tests fehlschlagen

```python
def test_will_fail():
    result = multiply_positive(2, 3)
    assert result == 7  # sollte fehlschlagen


# pytest Ausgabe:
# FAILED test_multiply.py::test_will_fail
# >       assert result == 7
# E       assert 6 == 7
# E         -6
# E         +7
```

Wenn Tests fehlschlagen: debug50 verwenden

Debuggen in VS Code:

1. Breakpoint setzen (roter Punkt)
2. `debug50 python script.py` (*mit* `python` aufrufen!)
3. Steuerung (genauso wie bei C):
   - Step Over (→): nächste Zeile
   - Step Into (↓): in Funktion springen
   - Continue (►): bis Breakpoint ausführen

Systematisches Testen zu Ende gedacht:

Test-Driven Development (TDD)

```python
def calculate_total(items, discount_code=None):
    # TODO: Mehrwertsteuer noch einbauen
    total = 0
    for item in items:
        if isinstance(item, dict):  # Irgendwann eingefügt
            total += item.get('price', 0)
        else:
            total += float(item)    # Legacy-Support

    # Rabatt-Logik (wurde später hinzugefügt)
    if discount_code:
        if discount_code.startswith('VIP'):  # Sonderfall
            total *= 0.8
        elif discount_code in DISCOUNT_CODES:
            total *= 0.9

    # Versandkosten (wurde noch später ergänzt)
    if total < 50:  # TODO: Länderabhängig machen
        total += 5.99

    return total  # TODO: Rundungsfehler beheben
```

Typische Probleme beim Programmieren

Probleme:

- Unklare Anforderungen
- Vergessene Spezialfälle
- Gewachsene Struktur
- Verstreute TODO-Kommentare
- Inkonsistente Datenstrukturen

Test-Driven Development: erst die Tests schreiben, dann den Code, der sie erfüllt.

```python
def test_empty_cart():
    assert calculate_total([]) == 0


def test_single_item():
    assert calculate_total([{"price": 10.00, "quantity": 1}]) == 10.00


def test_multiple_items():
    cart = [
        {"price": 10.00, "quantity": 2},
        {"price": 5.00, "quantity": 1}
    ]
    assert calculate_total(cart) == 25.00


def test_discount_code():
    cart = [{"price": 100.00, "quantity": 1}]
    assert calculate_total(cart, "SAVE20") == 80.00
```

Beginn mit der Implementierung

```python
def calculate_total(items, discount_code=None):
    if not items:
        return 0.00

    total = 0.00
    for item in items:
        total += item["price"] * item["quantity"]
    return total
```

Test-Ausführung:

```
✓ test_empty_cart passes
✓ test_single_item passes
✓ test_multiple_items passes
✗ test_discount_code FAILS
  Expected 80.00, got 100.00
```

Fortsetzung der Implementierung um fehlgeschlagene Tests zu adresieren

```
DISCOUNT_CODES = {
    "SAVE20": 0.20  # 20% Rabatt
}


def calculate_total(items, discount_code=None):
    if not items:
        return 0.00

    total = 0.00
    for item in items:
        total += item["price"] * item["quantity"]

    # Rabatt anwenden, falls gültiger Code
    if discount_code in DISCOUNT_CODES:
        discount = DISCOUNT_CODES[discount_code]
        total = total * (1 - discount)

    return total
```

Programmieren gemäß TDD-Zyklus:

1. Test schreiben (erst einmal rot)
2. Code anpassen (bis Test grün)
3. Verbessern, wenn nötig
4. Gehe zu Schritt 1

Probleme antizipieren durch

Defensives Programmieren

Häufiges Problem: „Schönwetter“-Implementationen

```python
# Es wird so programmiert, dass es (nur) funktioniert,
# wenn alles genau so ist, wie man es sich vorstellt:
def process_student_data(data):
    total = 0
    for student in data:
        total += student['grade']
    return total / len(data)


# Viele mögliche Probleme:
data = [
    {'name': 'Max'},          # Fehlendes 'grade'
    {'grade': 'A'},           # Keine Zahl
    {'grade': -5},            # Ungültiger Wert
]
```

Defensives Programmieren

```python
class Grade:
    def __init__(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError("Grade must be a number")
        if not 0 <= value <= 100:
            raise ValueError("Grade not within 0-100")
        self.value = value


def process_student_data(data):
    if not data:
        raise ValueError("No data provided")

    grades = []
    for i, student in enumerate(data):
        if 'grade' not in student:
            raise KeyError(f"Student {i}: Missing grade")
        try:
            grade = Grade(student['grade'])
            grades.append(grade.value)
        except (TypeError, ValueError) as e:
            raise ValueError(f"Student {i}: {str(e)}")

    return sum(grades) / len(grades)
```

- Daten sofort beim *Eingang* prüfen
- Probleme frühzeitig melden
- Aussgekräftige Fehlermeldungen
- Unmögliche Zustände unmöglich machen

Weitere Best Practice: Annahmen dokumentieren

```
def calculate_average(values: list) -> float:
    """Calculate average of numbers.

    Args:
        values: Non-empty list of numbers

    Raises:
        ValueError: If list is empty
        TypeError: If values aren't numbers
    """
```

Zusätzliche Unterstützung: Type Hints

```python
from typing import List, Optional, Union

def calculate_average(numbers: List[float]) -> float:
    if not numbers:
        raise ValueError("List cannot be empty")
    return sum(numbers) / len(numbers)


# IDE zeigt Fehler:
values = ["1", "2", "3"]  # ← Warnung: List[str], nicht List[float]
result = calculate_average(values)

# Aber: Code läuft trotzdem!
values = ["1", "2", "3"]
result = calculate_average(values)  # → TypeError zur Laufzeit
```

Type Hints sind nützlich, aber kein Allheilmittel

```python
def set_age(age: int) -> None:
    if age < 0 or age > 150:  # Diese Prüfung kann nicht durchType Hints ersetzt werden!
        raise ValueError("Invalid age")


# IDE: Alles OK
age: int = -42
set_age(age)  # Aber trotzdem falsch!


# Type Hints helfen hier nicht:
data = load_from_file() # Daten aus externer Quelle
set_age(data['age'])    # Type unknown, nichts zu prüfen
```

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Aufgabe 1

Was ist der Unterschied zwischen
diesen beiden Implementierungen?

```python
# Version A:
def divide_numbers_a(a, b):
    if b == 0:
        return None
    return a / b


# Version B:
def divide_numbers_b(a, b):
    if b == 0:
        raise ValueError("Division by zero")
    return a / b
```

Aufgabe 2

Was könnte hier schiefgehen?

```
def process_age():
    age = int(input("Enter age: "))
    if age < 0:
        print("Invalid age")
    return age
```

Aufgabe 3

Welcher der beiden Tests ist besser und warum?

```python
# Test A:
def test_user_data_a():
    assert validate_user("Max", 25)
    assert validate_user("", -5)
    assert validate_user("A", 0)

# Test B:
def test_user_name():
    assert validate_user("Max", 25)

def test_user_empty_name():
    with pytest.raises(ValueError):
        validate_user("", 25)

def test_user_invalid_age():
    with pytest.raises(ValueError):
        validate_user("Max", -5)
```

Übung zu Test-Driven Development:

Implementieren Sie die Funktion format_time basierend auf diesen Tests:

```
def test_basic_format():
    assert format_time(8, 30) == "08:30"
    assert format_time(14, 5) == "14:05"

def test_invalid_hours():
    with pytest.raises(ValueError):
        format_time(24, 30)
    with pytest.raises(ValueError):
        format_time(-1, 30)

def test_invalid_minutes():
    with pytest.raises(ValueError):
        format_time(12, 60)
    with pytest.raises(ValueError):
        format_time(12, -1)
```

Eine mögliche Lösung

```
def format_time(hours, minutes):
    # Eingaben prüfen
    if not (0 <= hours <= 23):
        raise ValueError("Hours must be between 0 and 23")
    if not (0 <= minutes <= 59):
        raise ValueError("Minutes must be between 0 and 59")


    # Zeit formatieren
    return f"{hours:02d}:{minutes:02d}"
```

```python
def outer():
    print("1: Entering outer")
    try:
        middle()
        print("2: This line is never reached")
    except TypeError:
        print("3: Caught TypeError in outer")
    print("4: Exiting outer")

def middle():
    print("5: Entering middle")
    try:
        inner()
        print("6: This line is never reached")
    except ValueError:
        print("7: Caught ValueError in middle")
    print("8: This line is never reached")

def inner():
    print("9: Entering inner")
    x = int("not a number")  # ValueError
    print("10: This line is never reached")
```

Und was passiert hier?

Was wird ausgegeben, wenn wir `outer()` aufrufen?

Das hier?

```
1: Entering outer
5: Entering middle
9: Entering inner
```

```python
def outer():
    print("1: Entering outer")
    try:
        middle()
        print("2: This line is never reached")
    except TypeError:
        print("3: Caught TypeError in outer")
    print("4: Exiting outer")

def middle():
    print("5: Entering middle")
    try:
        inner()
        print("6: This line is never reached")
    except ValueError:
        print("7: Caught ValueError in middle")
    print("8: This line is never reached")

def inner():
    print("9: Entering inner")
    x = int("not a number")  # ValueError
    print("10: This line is never reached")
```

Korrekte Ausgabe

Was wird ausgegeben,
wenn wir `outer()` aufrufen?

```
1: Entering outer
5: Entering middle
9: Entering inner
7: Caught ValueError in middle
4: Exiting outer
```

Exceptions sind besser als Rückgabewerte für Fehler.

Sie machen den Code klarer und verhindern Verwechslungen mit gültigen Werten.

Defensive Programmierung bedeutet, Fehler früh zu erkennen und klar zu kommunizieren – lieber einmal zu viel prüfen als einmal zu wenig.

pytest vereinfacht das Testen erheblich – besonders das Testen von Exceptions wird mit with pytest.raises() elegant und lesbar.

Type Hints sind nützliche Hinweise für Entwickler und IDEs, ersetzen aber keine Laufzeitprüfungen.

Test-Driven Development (TDD) hilft uns, systematisch von Anforderungen zu Code zu kommen – erst der Test, dann die Implementierung.

Ein systematischer Ansatz zur Fehlerbehandlung spart später viel Zeit beim Debugging und macht den Code robuster.