Debugging & Fehlerbehandlung in Python

1. Fehlerarten und deren Behebung

Python-Programme können verschiedene Arten von Fehlern enthalten, die sich auf unterschiedliche Weise äußern.

1.1 Syntaxfehler (SyntaxError)

- **Beschreibung:**
  Tritt auf, wenn der Code nicht den grammatikalischen Regeln von Python entspricht.
- **Beispiel:Fehler:** Fehlende schließende Klammer.

```
print("Hallo Welt"
```

- **Lösung:**
  Syntaxfehler durch genaue Überprüfung der Code-Struktur beheben. Der Python-Interpreter zeigt in der Regel die fehlerhafte Zeile an.

1.2 Laufzeitfehler (RuntimeError)

- **Beschreibung:**
  Fehler, die während der Programmausführung auftreten, obwohl der Code syntaktisch korrekt ist.
- **Beispiel:Fehler:** `ZeroDivisionError: division by zero`

```
x = 5 / 0  # Division durch Null
```

- **Lösung:**
  - Vorher prüfen, ob der Nenner 0 ist:

```
if y != 0:
    x = 5 / y
else:
    print("Fehler: Division durch Null")
```

1.3 Typfehler (TypeError)

- **Beschreibung:**
  Tritt auf, wenn ein Operator oder eine Funktion mit einem ungültigen Datentyp verwendet wird.
- **Beispiel:Fehler:** `TypeError: can only concatenate str (not "int") to str`

```
print("Das Ergebnis ist: " + 5)
```

- **Lösung:**
  - Vor der Konkatenation den Datentyp umwandeln:

```
print("Das Ergebnis ist: " + str(5))
```

1.4 Namensfehler (NameError)

- **Beschreibung:**
  Tritt auf, wenn auf eine nicht definierte Variable oder Funktion zugegriffen wird.
- **Beispiel:Fehler:** `NameError: name 'a' is not defined`

```
print(a)
```

- **Lösung:**
  - Sicherstellen, dass Variablen definiert wurden, bevor sie verwendet werden:

```
a = 10
print(a)
```

1.5 Indexfehler (IndexError)

- **Beschreibung:**
  Passiert, wenn versucht wird, auf eine nicht vorhandene Listenposition zuzugreifen.
- **Beispiel:Fehler:** `IndexError: list index out of range`

```
liste = [1, 2, 3]
print(liste[5])
```

- **Lösung:**
  - Prüfen, ob der Index innerhalb der Listenlänge liegt:

```
if 5 < len(liste):
    print(liste[5])
else:
    print("Fehler: Index außerhalb der Liste")
```

1.6 Schlüssel-Fehler (KeyError)

- **Beschreibung:**
  Tritt auf, wenn ein nicht vorhandener Schlüssel in einem Dictionary abgefragt wird.
- **Beispiel:Fehler:** `KeyError: 'alter'`

```
d = {"name": "Alice"}
print(d["alter"])
```

- **Lösung:**
  - `.get()` verwenden, um einen Standardwert zurückzugeben:

```
print(d.get("alter", "Schlüssel nicht vorhanden"))
```

1.7 Fehlerbehandlung mit `try-except`

- **Methode:** Fehler können mit `try-except` abgefangen werden, um das Programm vor Abstürzen zu schützen.
- **Beispiel:**

```
try:
    zahl = int(input("Gib eine Zahl ein: "))
    ergebnis = 10 / zahl
    print(f"Ergebnis: {ergebnis}")
except ZeroDivisionError:
    print("Fehler: Division durch Null!")
except ValueError:
    print("Fehler: Ungültige Eingabe!")
```

2. Debugging-Tools und Best Practices

2.1 Print-Debugging

- **Methode:** Einfügen von `print()` Befehlen, um Werte während der Programmausführung zu überprüfen.
- **Beispiel:**

```
def addiere(a, b):
    print(f"a: {a}, b: {b}")  # Debugging-Ausgabe
    return a + b

ergebnis = addiere(3, 4)
print("Ergebnis:", ergebnis)
```

2.2 Verwendung von `assert` für Tests

- **Methode:** Mit `assert` lassen sich Annahmen im Code testen. Falls eine Bedingung fehlschlägt, wird ein Fehler ausgelöst.
- **Beispiel:**

```
def teile(x, y):
    assert y != 0, "Division durch Null nicht erlaubt"
    return x / y
```

2.3 Nutzung des Debuggers (`pdb`)

- **Methode:** Der eingebaute Python Debugger `pdb` ermöglicht das Setzen von Haltepunkten und das schrittweise Debuggen.
- **Beispiel:**

```
import pdb

def testfunktion(x):
    y = x * 2
    pdb.set_trace()  # Debugging-Halt
    z = y + 3
    return z

print(testfunktion(5))
```

- **Wichtige Befehle im Debugger:**
  - `n` (next): Nächste Zeile ausführen.
  - `s` (step): In die aktuelle Funktion eintreten.
  - `c` (continue): Programm fortsetzen.

2.4 Logging statt `print()`

Warum `logging` statt `print()`?

Viele Anfänger nutzen `print()`, um sich Debugging-Informationen oder Fehler ausgeben zu lassen. Das Problem: `print()` ist nicht flexibel, bietet keine Protokollierungsstufen und ist schwer zu verwalten, wenn das Programm wächst.

Das `logging`-Modul in Python bietet dagegen:

- Unterschiedliche **Log-Level** zur Priorisierung von Meldungen (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`)
- Möglichkeit, Log-Einträge in **Dateien** oder andere Ausgabekanäle zu schreiben
- Eine einfache Möglichkeit, das **Logging-Verhalten zentral zu konfigurieren**
- **Bessere Wartbarkeit** und **Fehlersuche** in großen Projekten

---

Grundlegende Nutzung des `logging` Moduls

Beispiel für eine einfache Logging-Konfiguration:

```
import logging

# Grundlegende Konfiguration
logging.basicConfig(level=logging.DEBUG)

def multipliziere(a, b):
    logging.debug(f"Multipliziere {a} mit {b}")
    return a * b

ergebnis = multipliziere(3, 4)
logging.info(f"Ergebnis: {ergebnis}")
```

Erklärung:

- `logging.basicConfig(level=logging.DEBUG)`: Setzt die minimale Log-Stufe auf `DEBUG`, sodass alle Log-Nachrichten ausgegeben werden.

- `logging.debug()`: Gibt eine Debug-Nachricht aus (wird nur angezeigt, wenn das Level auf `DEBUG` oder niedriger gesetzt ist).
- `logging.info()`: Gibt eine Info-Nachricht aus, nützlich für allgemeine Informationen.

Log-Level und ihre Bedeutung

Level	Bedeutung
`DEBUG`	Detaillierte Debugging-Informationen
`INFO`	Allgemeine Informationen zum Programmablauf
`WARNING`	Warnungen über potenzielle Probleme
`ERROR`	Fehler, die das Programm beeinträchtigen
`CRITICAL`	Kritische Fehler, die zum Absturz führen können

Beispiel zur Verwendung mehrerer Log-Level:

```
import logging

logging.basicConfig(level=logging.DEBUG)

def divide(a, b):
    if b == 0:
        logging.error("Division durch Null ist nicht erlaubt!")
        return None
    logging.info(f"Teile {a} durch {b}")
    return a / b

divide(10, 2)
divide(10, 0)
```

Logging in eine Datei schreiben

Standardmäßig schreibt `logging` in die Konsole. Um Logs in eine Datei zu speichern:

```
logging.basicConfig(filename='app.log', level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
```

- `filename='app.log'` speichert Logs in `app.log`.
- `format='%(asctime)s - %(levelname)s - %(message)s'` gibt an, wie die Logs formatiert werden.

Beispiel-Logdatei (`app.log`):

```
2025-03-04 14:30:01,123 - INFO - Teile 10 durch 2
2025-03-04 14:30:01,124 - ERROR - Division durch Null ist nicht erlaubt!
```

Tests mit `unittest` und `logging`

Beim Testen kann es hilfreich sein, das Logging zu prüfen.

Beispiel mit `unittest`:

```
import logging
import unittest

logging.basicConfig(level=logging.DEBUG)

def addiere(a, b):
    logging.debug(f"Addiere {a} mit {b}")
    return a + b

class TestLogging(unittest.TestCase):
    def test_addiere(self):
        self.assertEqual(addiere(2, 3), 5)
        self.assertEqual(addiere(-1, 1), 0)

if __name__ == '__main__':
    unittest.main()
```

Beim Testlauf sieht man sowohl die Log-Ausgaben als auch die Testergebnisse.

Fazit

- `logging` bietet **strukturierte, skalierbare und flexible** Debugging- und Fehlermeldungen.
- `print()` sollte nur für schnelle Tests genutzt werden, nicht für langfristige Fehlerprotokolle.
- `unittest` kann mit `logging` kombiniert werden, um Fehlersuche in Tests zu verbessern.

Mit diesen Techniken bleibt dein Code übersichtlich, professionell und leichter wartbar!

2.5 Unit-Tests zur Fehlervermeidung

Einführung in Unit-Tests

Unit-Tests sind eine essenzielle Technik zur Qualitätssicherung in der Softwareentwicklung. Sie helfen dabei, Fehler frühzeitig zu erkennen, indem sie sicherstellen, dass einzelne Funktionen oder Methoden eines Programms erwartungsgemäß arbeiten. Python bietet mit dem `unittest`-Modul eine leistungsfähige Möglichkeit, automatisierte Tests zu schreiben und auszuführen.

Warum Unit-Tests wichtig sind

- **Früherkennung von Fehlern**: Probleme werden entdeckt, bevor sie größere Auswirkungen haben.
- **Erhöhte Code-Stabilität**: Durch Tests kann sichergestellt werden, dass Änderungen keine unerwarteten Fehler verursachen.
- **Dokumentation der erwarteten Funktionalität**: Tests zeigen, welche Eingaben und Ausgaben für eine Funktion erwartet werden.
- **Erleichterte Wartung**: Wenn Tests vorhanden sind, können Entwickler sicherer Refactorings oder Erweiterungen vornehmen.

Grundlegendes Beispiel für Unit-Tests mit `unittest`

```
import unittest

def addiere(a, b):
    """Einfache Funktion zur Addition zweier Zahlen"""
    return a + b

class TestAddition(unittest.TestCase):
    def test_addiere(self):
        """Testet die Addition von zwei Zahlen."""
        self.assertEqual(addiere(2, 3), 5)
        self.assertEqual(addiere(-1, 1), 0)
        self.assertEqual(addiere(0, 0), 0)

if __name__ == "__main__":
    unittest.main()
```

Erweiterte Unit-Tests mit mehreren Fällen

Um ein komplexeres Verhalten zu testen, können wir verschiedene Edge Cases berücksichtigen, z. B.:

```
class TestAdditionErweitert(unittest.TestCase):
    def test_negative_zahlen(self):
        self.assertEqual(addiere(-5, -7), -12)

    def test_gleitkommazahlen(self):
        self.assertAlmostEqual(addiere(2.5, 3.1), 5.6)

    def test_grosse_zahlen(self):
        self.assertEqual(addiere(10**6, 10**6), 2 * 10**6)
```

Hier nutzen wir `assertAlmostEqual()`, um Rundungsungenauigkeiten bei Gleitkommazahlen zu berücksichtigen.

Anwendung von Unit-Tests in Wissenschaft und Ingenieurwesen

Unit-Tests helfen:

- numerische Fehler zu vermeiden
- Korrektheit von Algorithmen überprüfen

Beispiel: Physikalische Berechnungen testen

Angenommen, wir haben eine Funktion zur Berechnung der kinetischen Energie:

```
def kinetische_energie(masse, geschwindigkeit):
    """Berechnet die kinetische Energie: E = 0.5 * m * v^2"""
    return 0.5 * masse * geschwindigkeit ** 2
```

Dazu passende Unit-Tests:

```
class TestPhysikBerechnungen(unittest.TestCase):
    def test_kinetische_energie(self):
        self.assertEqual(kinetische_energie(2, 3), 9.0)
        self.assertAlmostEqual(kinetische_energie(1.5, 4.2), 13.23, places=2)
        self.assertEqual(kinetische_energie(0, 10), 0)
```

Beispiel: Berechnung der Standardabweichung

In der Ingenieurwissenschaft und Statistik ist die Standardabweichung eine häufig verwendete Größe zur Analyse von Messwerten. Hier ein Beispiel:

```
import numpy as np

def standardabweichung(arr):
    """Berechnet die Standardabweichung einer Zahlenliste."""
    n = len(arr)
    if n == 0:
        raise ValueError("Liste darf nicht leer sein")
    mittelwert = sum(arr) / n
    varianz = sum((x - mittelwert) ** 2 for x in arr) / n
    return np.sqrt(varianz)
```

Dazu ein passender Test:

```
class TestStandardabweichung(unittest.TestCase):
    def test_standardabweichung(self):
        self.assertAlmostEqual(standardabweichung([1, 2, 3, 4, 5]), 1.414, places
=3)
        self.assertAlmostEqual(standardabweichung([10, 10, 10, 10]), 0.0)
        self.assertAlmostEqual(standardabweichung([2, 4, 4, 4, 5, 5, 7, 9]), 2.0)

    def test_leere_liste(self):
        with self.assertRaises(ValueError):
            standardabweichung([])
```

Fazit

Unit-Tests mit `unittest` sind ein wertvolles Werkzeug zur Fehlervermeidung und Code-Qualitätssicherung, besonders in wissenschaftlichen und ingenieurtechnischen Anwendungen. Sie ermöglichen eine frühzeitige Validierung mathematischer Berechnungen, Algorithmen und Modelle und helfen dabei, unbemerkte Fehler zu vermeiden.

Fazit

- Fehler lassen sich in Python in verschiedene Kategorien einteilen.
- Debugging-Methoden wie `print()`, `assert`, `pdb`, `logging` und `try-except` helfen, Probleme effizient zu finden und zu lösen.
- Best Practices wie Unit-Tests und strukturierte Fehlerbehandlung reduzieren Fehler langfristig.