Zaawansowana obsługa wyjątków i debugowanie

Maj 2025

Wprowadzenie

- **Cel prezentacji**: Omówienie zaawansowanych technik obsługi wyjątków i debugowania.
- **Dlaczego to ważne?**
  - Wyjątki mogą przerwać działanie programu.
  - Debugowanie pozwala szybko identyfikować i naprawiać błędy.
- **Plan prezentacji**:
  1. Podstawy wyjątków
  2. Zaawansowana obsługa wyjątków
  3. Techniki debugowania
  4. Praktyczne przykłady
  5. Najlepsze praktyki

Podstawy wyjątków

- **Czym są wyjątki?**
  - Błędy występujące w czasie działania programu.
- **Hierarchia wyjątków** (Python, Java, C++):
  - Klasy bazowe: `Exception`, `Throwable`, `std::exception`.

Python:

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Nie mozna dzielic przez zero!")
```

Wielopoziomowa obsługa wyjątków

- **Wielopoziomowa obsługa wyjątków**:
  - Łapanie wielu wyjątków w jednym bloku.

```
try:
    value = int(input("Podaj liczbe: "))
    result = 10 / value
except (ValueError, ZeroDivisionError) as e:
    print(f"Błąd: e)
```

- **Wskazówka**: Zawsze łap najpierw bardziej specyficzne wyjątki, potem ogólne (np. `Exception`).

Własne wyjątki

- Możemy tworzyć własne klasy wyjątków:

```
1 class CustomError(Exception):
2     pass
3
4 def check_value(value):
5     if value < 0:
6         raise CustomError("Wartosc nie moze byc
              ujemna!")
```

Klauzule else i finally

```
try:
    file = open("data.txt", "r")
except FileNotFoundError:
    print("Plik nie istnieje!")
else:
    print("Plik otwarty poprawnie.")
    file.close()
finally:
    print("Zakonczono operacje na pliku.")
```

Propagacja i traceback

```python
def risky_function():
    raise ValueError("Cos poszlo nie tak!")

try:
    risky_function()
except ValueError as e:
    print(f"Przechwycono: e)
```

```python
import traceback
try:
    1 / 0
except Exception as e:
    traceback.print_exc()
```

Print debugging

```
1 def calculate(x, y):
2     print(f"x: x, y: y)  Debugreturn x / y
```

Debuggery IDE i pdb

```
1 import pdb
2
3 def faulty_function(x):
4     pdb.set_trace()  # Punkt przerwania
5     return x / 0
```

Logowanie w Pythonie

```
1 import logging
2
3 x = 5
4 logging.basicConfig(level=logging.DEBUG)
5 logging.debug("Zmienna x = %s", x)
6 logging.error("Wystapil blad!")
```

Testy jednostkowe

```
1 import unittest
2
3 def divide(x, y):
4     return x / y
5
6 class TestDivide(unittest.TestCase):
7     def test_zero(self):
8         with self.assertRaises(ZeroDivisionError):
9             divide(10, 0)
```

Przykład: Aplikacja sieciowa

```
import requests

def fetch_data(url):
    try:
        response = requests.get(url, timeout=5)
        response.raise_for_status()
    except requests.Timeout:
        print("Timeout.")
    except requests.HTTPError as e:
        print(f"HTTP błąd: e)except
            requests.ConnectionError:print("Błąd
            połączenia.")else:return
            response.json()
```

Przykład: Debugowanie pętli

```
1 def process_list(data):
2     import pdb; pdb.set_trace()
3     for item in data:
4         if item == 0:
5             raise ValueError("Zero w danych!")
6     return sum(data)
```

Najlepsze praktyki

- Obsługuj tylko znane wyjątki.
- Unikaj pustych bloków `except`.
- Stosuj testy jednostkowe.
- Używaj loggera zamiast `print`.
- Dokumentuj złożone fragmenty.

Podsumowanie i pytania

- Wyjątki = kontrola błędów.
- Debugowanie = identyfikacja i naprawa błędów.
- Ćwiczenie czyni mistrza!
- Pytania?