

Zaawansowana obsługa wyjątków i debugowanie

Wiktor Banek, Daniel Kubiela

Maj 2025

- **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
 - 6 Omówienie projektu
- **Znaczenie:** Poprawna obsługa błędów zwiększa niezawodność kodu.

- **Czym są wyjątki?**
 - Błędy występujące w czasie działania programu.
- **Hierarchia wyjątków** (Python, Java, C++):
 - Klasy bazowe: `Exception`, `Throwable`, `std::exception`.
- **Przykład w Pythonie:** Obsługa błędu dzielenia przez zero.
- **Znaczenie:** Umożliwia kontrolowanie błędów bez awarii programu.

Python:

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

Wielopoziomowa obsługa wyjątków

- **Wielopoziomowa obsługa wyjątków:**
 - Łapanie wielu wyjątków w jednym bloku.
- **Przykład:** Obsługa błędów wejścia i dzielenia przez zero.
- **Wskazówka:** Zawsze łap najpierw bardziej specyficzne wyjątki, potem ogólne (np. `Exception`).
- **Korzyść:** Umożliwia zwarte i czytelne zarządzanie błędami.

```
1 try:  
2     value = int(input("Podaj liczbę: "))  
3     result = 10 / value  
4 except (ValueError, ZeroDivisionError) as e:  
5     print(f"Błąd: e")
```

Własne wyjątki

- **Cel:** Możemy tworzyć własne klasy wyjątków.
- **Zastosowanie:** Umożliwia precyzyjne sygnalizowanie błędów.
- **Przykład:** Własny wyjątek dla ujemnych wartości.

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

- **Cel:** `else` wykonuje kod, gdy nie ma wyjątków; `finally` zawsze się wykonuje.
- **Przykład:** Obsługa pliku z zapewnieniem zamknięcia.

```
1 try:
2     file = open("data.txt", "r")
3 except FileNotFoundError:
4     print("Plik nie istnieje!")
5 else:
6     print("Plik otwarty poprawnie.")
7     file.close()
8 finally:
9     print("Zakończono operacje na pliku.")
```

Propagacja i traceback

- **Propagacja:** Wyjątki przekazywane w górę stosu wywołań.
- **Traceback:** Śledzenie szczegółów błędu za pomocą traceback.

```
1 def risky_function():
2     raise ValueError("Cos poszlo nie tak!")
3
4 try:
5     risky_function()
6 except ValueError as e:
7     print(f"Przechwycono: e)
```

```
1 import traceback
2 try:
3     1 / 0
4 except Exception as e:
5     traceback.print_exc()
```

Print debugging

- **Cel:** Prosta metoda śledzenia wartości zmiennych.
- **Zastosowanie:** Szybkie debugowanie w trakcie .

```
1 def calculate(x, y):  
2     print(f"x: {x}, y: {y}")  
    return x / y
```


- **Cel:** Interaktywne debugowanie z punktami przerwania.
- **Przykład:** Użycie `pdb` do analizy kodu w trakcie wykonania.

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

- **Cel:** Rejestrowanie informacji o działaniu programu.
- **Zalety:** Trwałe zapisywanie komunikatów zamiast `print`.

```
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!")
```

- **Cel:** Automatyzacja weryfikacji poprawności kodu.
- **Przykład:** Testowanie obsługi błędu dzielenia przez zero.

```
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: Przetwarzanie i walidacja pliku

- **Cel:** Pokazanie obsługi błędów przy przetwarzaniu pliku CSV.
- **Funkcjonalność:** Walidacja liczby kolumn i wartości liczbowych.

```
1 def process_csv_file(file_path):
2     try:
3         with open(file_path, 'r') as file:
4             lines = file.readlines()
5             if not lines:
6                 raise ValueError("Plik pusty!")
7             for i, line in enumerate(lines, 1):
8                 values = line.strip().split(',')
9                 if len(values) < 2:
10                    raise ValueError(f"Linia {i}: Za mało
11                                   kolumn!")
12                    float(values[0]) # Validate number
13 except FileNotFoundError:
14     print("Plik nie istnieje!")
15 except ValueError as e:
16     print(f"Błąd: {e}")
17 else:
18     print("Przetworzono poprawnie.")
```

- **Projektowanie wyjątków:**
 - Twórz specyficzne wyjątki (np. `FileFormatError`).
 - Nie używaj wyjątków do sterowania przepływem.
 - Dokumentuj wyjątki w docstringach.
- **Strategie debugowania:**
 - *Binary search*: Podziel kod, by znaleźć błąd.
 - *Hypothesis-driven*: Testuj hipotezy o błędzie.
 - Używaj `cProfile` do analizy wydajności.
- **Wskazówka:** Unikaj zbyt ogólnych wyjątków.

Przykład: Debugowanie pętli

- **Cel:** Pokazanie interaktywnego debugowania pętli.
- **Zastosowanie:** pdb do analizy błędów w danych.

```
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)
```

- Obsługuj tylko znane wyjątki.
- Unikaj pustych bloków `except`.
- Stosuj testy jednostkowe.
- Używaj loggera zamiast `print`.
- Dokumentuj złożone fragmenty.
- **Znaczenie:** Poprawia czytelność i niezawodność kodu.
- **Praktyka:** Regularnie przeglądaj logi i testy.

- Wyjątki = kontrola błędów.
- Debugowanie = identyfikacja i naprawa błędów.
- Ćwiczenie czyni mistrza!
- **Wnioski:** Skuteczna obsługa błędów wymaga praktyki.