

Obsługa błędów

Błędy składniowe, Kody błędów, Wyjątki

Błędy składniowe

Błędy składni wykrywane są podczas parsowania kodu, czyli analizy jego struktury, weryfikacji składni.

Przykładem takiego błędu jest niepoprawne wcięcie bloku kodu lub brak nawiasów.

```
>>> x := 4
      File "<stdin>", line 1
        x := 4
          ^
SyntaxError: invalid syntax
```



Obsługa błędów

Kody błędów

Obsługę błędów można realizować za pomocą **kodów błędów**, czyli specjalnych wartości przekazywanych z funkcji do wywołującego, oznaczających wystąpienie błędu podczas jej wykonywania.

Najczęściej kody błędów są implementowane za pomocą liczb całkowitych, chociaż może to być dowolny inny typ (np. napis). Dobrą praktyką jest ich grupowanie za pomocą typów wyliczeniowych.



Obsługa błędów

Kody błędów

W najprostszych przypadkach kodami błędów mogą być wartości `True`, `False` przekazywane do wywołującego funkcję za pomocą mechanizmu zwracania wartości.

```
import string

def check_digit(text):
    for x in text:
        if x not in string.digits:
            return False
    if len(text) == 0:
        return False
    return True
```



Obsługa błędów

Kody błędów

Wartości True, False nie niosą ze sobą informacji powodu z jakiego funkcja się zakończyła. Informują one tylko o tym czy dana funkcja zakończyła się poprawnie czy nie.

Bardziej szczegółową informację można zawrzeć w nazwanych kodach błędów.

```
import string

def check_digit(text):
    for x in text:
        if x not in string.digits:
            return False
    if len(text) == 0:
        return False
    return True
```



Obsługa błędów

Kody błędów

Kodami błędów mogą być dowolne wartości o określonych nazwach.

```
SUCCESS = 0
ERR_FOUND_LETTER = 1
ERR_EMPTY_STRING = 2

import string

def check_digit(text):
    for x in text:
        if x not in string.digits:
            return ERR_FOUND_LETTER
    if len(text) == 0:
        return ERR_EMPTY_STRING
    return SUCCESS
```



Obsługa błędów

Kody błędów

Stosując typ wyczerpujący (Enum) można powiązać ze sobą poszczególne kody błędów.

Funkcja `auto()` samoczynnie wygeneruje wartość liczbową dla kolejnych kodów.

```
import string
from enum import Enum, auto

class DigitErrors(Enum):
    SUCCESS = auto()
    ERR_FOUND_LETTER = auto()
    ERR_EMPTY_STRING = auto()

def check_digit(text):
    for x in text:
        if x not in string.digits:
            return DigitErrors.ERR_FOUND_LETTER
    if len(text) == 0:
        return DigitErrors.ERR_EMPTY_STRING
    return DigitErrors.SUCCESS
```



Obsługa błędów

Kody błędów

Zastosowanie typu
wyliczeniowego pozwoliło na:

- powiązanie wartości,
- grupowanie kodów przeznaczonych dla danego zastosowania,
- uniemożliwienie porównania różnych rodzajów błędów (np. błąd związany z otwarciem pliku i sprawdzeniem napisu).

```
import string
from enum import Enum, auto

class DigitErrors(Enum):
    SUCCESS = auto()
    ERR_FOUND_LETTER = auto()
    ERR_EMPTY_STRING = auto()

def check_digit(text):
    for x in text:
        if x not in string.digits:
            return DigitErrors.ERR_FOUND_LETTER
    if len(text) == 0:
        return DigitErrors.ERR_EMPTY_STRING
    return DigitErrors.SUCCESS
```



Obsługa błędów

Kody błędów

Podsumowanie:

- obsługa błędu może zostać pominięta świadomie lub nieświadomie,
- odseparowanie kodu obsługującego błędy od kodu wykonującego obliczenia jest trudne,
- obsłużenie błędu w innym miejscu w programie wymaga przekazywania kodu błędu za pomocą zwracanej wartości z kolejnych wywołań funkcji.



Wyjątki

Wyjątki są rodzajem błędów wykrytych podczas działania programu.

Przykładem takiego błędu jest dzielenie liczby przez 0.

```
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



Obsługa błędów

Wyjątki

Wyjątek jest obiektem będącym instancją klasy `BaseException`, obowiązuje ustalona hierarchia wyjątków.

BaseException

- +-- `SystemExit`

- +-- `KeyboardInterrupt`

- ...

- +-- `Exception`



Obsługa błędów

Wyjątki

Wyjątek jest obiektem będącym instancją klasy `BaseException`, obowiązuje ustalona hierarchia wyjątków.

```
+-- Exception
```

```
...
```

```
    +-- ArithmeticError
```

```
    |   +-- FloatingPointError
```

```
    |   +-- OverflowError
```

```
    |   +-- ZeroDivisionError
```

```
    +-- AssertionError
```

```
...
```



Obsługa błędów

Wyjątki

Wyjątek jest obiektem będącym instancją klasy `BaseException`, obowiązuje ustalona hierarchia wyjątków.

+- - Exception

+- - ImportError

| +- - ModuleNotFoundError

...

+- - RuntimeError

| +- - NotImplementedError

| +- - RecursionError



Obsługa błędów

Wyjątki

Wyjątek jest obiektem będącym instancją klasy `BaseException`, obowiązuje ustalona hierarchia wyjątków.

```
+-- Exception
```

```
    +-- SyntaxError
```

```
    |    +-- IndentationError
```

```
    |          +-- TabError
```

```
...
```

```
    +-- ValueError
```

```
...
```



Obsługa błędów

Wyjątki

Wyjątki mogą być przechwycone,
czyli w pewien sposób
przewidziane i obsługowane
przez programistę.

W tym celu powstała
specjalna składnia
try-except-else-finally.

```
try:
    # kod, w którym może wystąpić wyjątek
except KlasaOczekiwanegoWyjatku as nazwa:
    # kod obsługujący sytuację, gdy wystąpił
    # podany wyjątek
except InnaKlasaOczekiwanegoWyjatku as nazwa:
    # kod obsługujący sytuację, gdy wystąpił
    # podany wyjątek
else:
    # kod wykonywany, gdy wyjątek nie wystąpił
finally:
    # kod wykonywany zawsze
```



Obsługa błędów

Wyjątki

Wystąpienie sytuacji
wyjątkowej powoduje
**przerwanie wykonywania
kodu** umieszczonego w bloku
`try`.

```
try:
    # kod, w którym może wystąpić wyjątek
except KlasaOczekiwanegoWyjatku as nazwa:
    # kod obsługujący sytuację, gdy wystąpił
    # podany wyjątek
except InnaKlasaOczekiwanegoWyjatku as nazwa:
    # kod obsługujący sytuację, gdy wystąpił
    # podany wyjątek
else:
    # kod wykonywany, gdy wyjątek nie wystąpił
finally:
    # kod wykonywany zawsze
```



Obsługa błędów

Wyjątki

W podanym przykładzie zostanie rzucony wyjątek w wyniku próby podzielenia liczby przez 0.

Na ekranie pojawi się komunikat o próbie dzielenia oraz informacja o przechwyceniu wyjątku.

```
try:
    print("Proba dzielenia przez 0.")
    x = 2 / 0
    print("Wynik operacji:", x)
except ArithmeticError as e:
    print("Zlapano wyjatek:", e)
```



Obsługa błędów

Wyjątki

Wyjątek może zostać rzucony również w przypadku próby dokonania niedozwolonej konwersji typów - np. zamiany napisu na liczbę.

```
try:
    print("Proba zamiany napisu 'E' na liczbę.")
    x = int('E')
    print("Wynik operacji:", x)
except ValueError as e:
    print("Zlapano wyjatek:", e)
```



Obsługa błędów

Wyjątki

Rzucone wyjątki powodują przerwanie wykonywania wszystkich funkcji z sekwencji wywołań, aż do napotkania bloku try-except-else-finally, jeśli nie zostanie on napotkany następuje przerwanie wykonywania programu.

```
def f():  
    print("Proba zamiany napisu 'E' na liczbę.")  
    x = int('E')  
    print("Wynik operacji:", x)  
  
def g():  
    print("Wywołuje f()")  
    f()  
    print("Zakonczono f()")  
  
try:  
    g()  
except ValueError as e:  
    print("Zlapano wyjatek:", e)
```



Obsługa błędów

Wyjątki

Blok else jest wykonywany tylko w przypadku, gdy wyjątek nie został rzucony.

W przytoczonym przykładzie na ekranie pojawi się:

Proba zamiany napisu
'15' na liczbę.

Wynik operacji: 15

Wyjątek nie wystąpił

```
try:
    print("Proba zamiany napisu '15' na liczbę.")
    x = int('15')
    print("Wynik operacji:", x)
except ArithmeticError as e:
    print("Zlapano wyjatek:", e)
else:
    print("Wyjątek nie wystapil")
```



Obsługa błędów

Wyjątki

Kod umieszczony w **bloku finally** jest wykonywany zawsze, niezależnie od tego, czy wyjątek został rzucony, czy nie.

```
try:
    print("Proba zamiany napisu '15' na liczbę.")
    x = int('15')
    print("Wynik operacji:", x)
except ArithmeticError as e:
    print("Zlapano wyjatek:", e)
finally:
    print("Blok kodu wykonywany zawsze.")
```



Obsługa błędów

Wyjątki

Pytanie:

Czy w przypadku wykonania instrukcji `return` blok `finally` również zostanie wykonany?

Zwróćmy uwagę na instrukcję `return` zwracającą podaną wartość z funkcji.

```
def funkcja():  
    try:  
        print("Proba zamiany napisu '15' \\  
na liczbę.")  
        x = int('15')  
        print("Wynik operacji:", x)  
        return x  
    except ArithmeticError as e:  
        print("Zlapano wyjatek:", e)  
    finally:  
        print("Blok kodu wykonywany zawsze.")  
        print("ale czy na pewno?")  
  
funkcja()
```



Obsługa błędów

Wyjątki

Pytanie:

Czy w przypadku wykonania instrukcji `return` blok `finally` również zostanie wykonany?

TAK, blok `finally` zostanie wykonany.

```
def funkcja():  
    try:  
        print("Proba zamiany napisu '15' \\  
na liczbę.")  
        x = int('15')  
        print("Wynik operacji:", x)  
        return x  
    except ArithmeticError as e:  
        print("Zlapano wyjatek:", e)  
    finally:  
        print("Blok kodu wykonywany zawsze.")  
        print("ale czy na pewno? TAK")  
  
funkcja()
```



Obsługa błędów

Wyjątki

Blok finally może występować razem z blokiem try (pominięta konstrukcja except). Na ekranie zobaczymy:

Proba zamiany napisu '15' na liczbę.

Wynik operacji: 15

Blok kodu wykonywany zawsze.

```
try:
    print("Proba zamiany napisu '15' na liczbę.")
    x = int('15')
    print("Wynik operacji:", x)
finally:
    print("Blok kodu wykonywany zawsze.")
```



Obsługa błędów

Wyjątki

Pominięcie bloku except w przypadku konstrukcji else jest nieprawidłowe.

```
try:  
    print("Proba zamiany napisu '15' na liczbę.")  
    x = int('15')  
    print("Wynik operacji:", x)  
else:  
    print("Wyjątek nie wystąpił")
```



Obsługa błędów

Wyjątki

O rzuceniu wyjątku może zdecydować programista wykorzystując **instrukcję raise**.

Argumentem do instrukcji jest obiekt będący instancją klasy reprezentującej odpowiednią klasę błędu.

Wyjątki rzucone przez programistę również mogą zostać przechwycone za pomocą bloku try-except-else-finally.

```
import string

def check_digit(text):
    for x in text:
        if x not in string.digits:
            raise ValueError(x + ' is not a digit')
    if len(text) == 0:
        raise ValueError('Empty string')

check_digit('5')
check_digit('3a')
```



Obsługa błędów

Wyjątki

Wbudowane klasy wyjątków nie zawsze są wystarczające, dlatego **programista ma możliwość utworzenia wyjątku własnej klasy.**

W tym celu należy:

1. określić odpowiedzialność wyjątku, zdefiniować nazwę,
2. wybrać klasę nadrzędną,
3. utworzyć dokumentację dokumentacji, jeśli istnieje taka potrzeba,
4. utworzyć metodę `__init__()`.
5. wywołać metodę `__init__()` klasy nadrzędnej.



Obsługa błędów

Wyjątki

Po jakiej klasie dziedziczyć podczas tworzenia własnej klasy wyjątku?

- **BaseException** **NIE**

*The base class for all built-in exceptions. **It is not meant to be directly inherited by user-defined classes.***

- **Exception** **TAK**

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

Opisy odpowiedzialności dla poszczególnych wyjątków:

<https://docs.python.org/3/library/exceptions.html>



Obsługa błędów

Wyjątki

Wyjątki zdefiniowane przez użytkownika muszą dziedziczyć po klasie **Exception** - bezpośrednio lub pośrednio za pomocą innej klasy.

Utworzenie nowego rodzaju wyjątku wiąże się z utworzeniem nowej klasy.

```
class EmptyString(ValueError):  
    """Rzucany, gdy napis jest pusty."""  
    def __init__(self):  
        super().__init__("String is empty")  
  
class NotDigit(ValueError):  
    """Rzucany, gdy napis nie jest cyfrą."""  
    def __init__(self, wrong_char):  
        super().__init__("Not a digit: " +  
                        wrong_char)
```



Obsługa błędów

Wyjątki

Nowo utworzony rodzaj wyjątku może zostać rzucony w taki sam sposób, jak wyjątki wbudowane.

```
def check_digit(string):  
    for x in string:  
        if x not in string.digits:  
            raise NotDigit(x)  
if len(string) == 0:  
    raise EmptyString
```



Obsługa błędów

Wyjątki

Istnieje możliwość utworzenia **wielu bloków except**, jednak rzucony **wyjątek trafi tylko do pierwszego pasującego**.

```
try:
    check_digit('5')
    check_digit('3a')
except EmptyString as e:
    print(e)
except NotDigit as e:
    print(e)
```



Obsługa błędów

Wyjątki

Wyjątki mogą być ze sobą pogrupowane i obsługiwane w jednym bloku `except`.

```
try:
    check_digit('5')
    check_digit('3a')
except (EmptyString, NotDigit) as e:
    print(e)
```



Obsługa błędów

Wyjątki

Wykorzystując hierarchię klas można przechwycić wyjątek na podstawie klasy nadrzędnej.

```
try:  
    check_digit('5')  
    check_digit('3a')  
except Exception as e:  
    print(e)
```



Obsługa błędów

Wyjątki

Podsumowanie:

- **obsługa wyjątku nie może być pominięta**, w przeciwnym wypadku program może przerwać działanie,
- wyjątki powodują przerwanie wykonywania wszystkich funkcji z sekwencji wywołań,
- obsługa wyjątków jest w jasny sposób odizolowana od kodu wykonującego obliczenia,
- wyjątki mogą przechowywać dodatkowe informacje na temat napotkanego błędu.



Obsługa błędów

Pytania

1. Czym jest wyjątek?
2. Po jakiej klasie muszą dziedziczyć wyjątki zdefiniowane przez użytkownika?
3. Czy blok instrukcji finally zostanie wykonany zawsze?
4. Czy bloków except może być wiele?



Literatura

1. Errors and Exceptions,
<https://docs.python.org/3/tutorial/errors.html>



Obsługa błędów

