Wyjątki, typowanie, generatory oraz iteratory

Paweł Gliwny

Czym jest wyjątek?

Błąd, który występuje podczas działania programu

Może prowadzić do nagłego zakończenia działania kodu
Można go obsłużyć, aby uniknąć awarii

```
$ print(5 + "a")
```

TypeError: unsupported operand type(s)

Przykłady wyjątków: IndexError, ZeroDivisionError, ValueError, NameError

Obsługa wyjątków

- Zapobiega nagłemu zakończeniu działania programu
- Umożliwia obsługę błędów i podjęcie odpowiednich działań

```
try:
    print(5 + "a")
except TypeError:
    print("Nie można dodać liczby do tekstu, ale można je pomnożyć!")
finally:
    print(5 * "a")
```

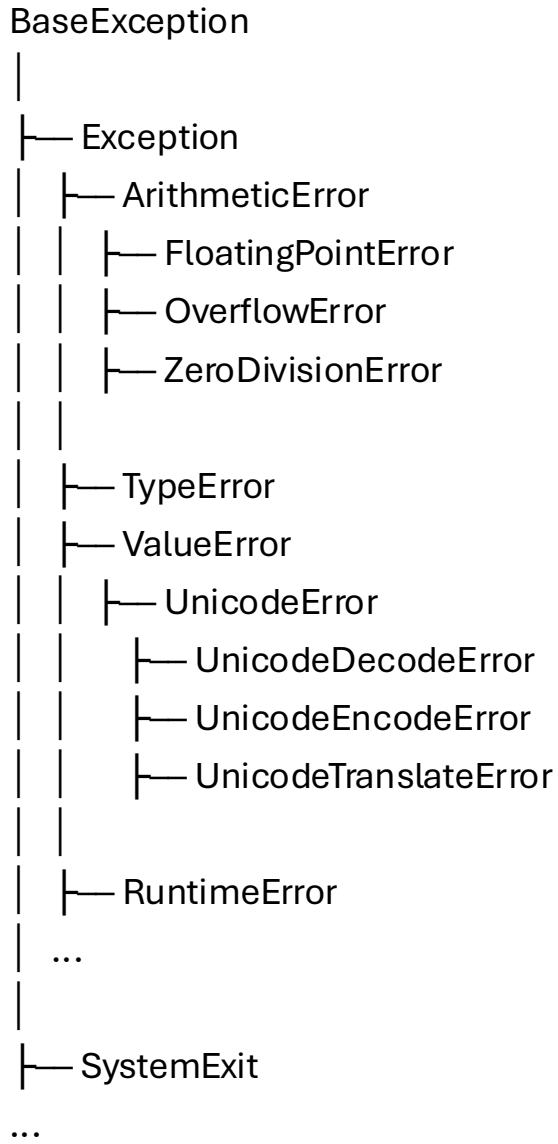
Obsługa wyjątków

- Zapobiega nagłemu zakończeniu działania programu
- Umożliwia obsługę błędów i podjęcie odpowiednich działań

```
try:
    print(5 + "a")
except TypeError:
    print("Nie można dodać liczby do tekstu, ale można je pomnożyć!")
finally:
    print(5 * "a")
```

Wyjątki są klasami

- Wszystkie Wyjątki W Pythonie Są Obiektami Klas
- Dziedziczą Z Klasy **Baseexception** Lub **Exception**
- Można Tworzyć Własne Wyjątki Poprzez Dziedziczenie



Własne wyjątki

- Własne wyjątki powinny dziedziczyć po **`Exception`** lub jednej z jej podklas
- Najczęściej są to klasy bez dodatkowej implementacji
- **raise** do rzucania wyjątku

```
class BalanceError(Exception):
    pass


class Customer:
    def __init__(self, name, balance):
        if balance < 0 :
            raise BalanceError("Balance has to be non-
  negative!")
        else:
            self.name = name
            self.balance = balance
```

Obsługa własnych wyjątków

- Po rzuceniu wyjątku program kończy działaniem błędu (exit code: 1)
- Wyjątek można złapać, tak aby program nie zakończył swojego działania błędem (exit code: 0)

```
cust = Customer("Larry Torres", -100)
```

```
try:
      cust = Customer("Larry Torres", -100)
except BalanceError as Err:
      print("Error in call Customer()")
      print(Err)
```

Zadanie

- Zadanie z wyjątkami do zrobienia

Poziom ***enterprise (produkcyjny)*** w programowaniu

Poziom enterprise odnosi się do oprogramowania, narzędzi lub umiejętności, które są odpowiednie dla dużych firm i organizacji.

Oznacza to, że rozwiązania tego typu muszą spełniać wysokie standardy dotyczące:

- Skalowalności
- Czytelności i utrzymania
- Bezpieczeństwa
- Wydajności
- Zgodnosci z dobrymi praktykami

Kod produkcyjny najważniejsze cechy

- **Skalowalności** – kod powinien działać sprawnie przy dużych ilościach danych i użytkowników.
- **Wydajności** – systemy enterprise muszą działać efektywnie nawet przy dużym obciążeniu.
- **Bezpieczeństwa** – aplikacje na poziomie enterprise często obsługują wrażliwe dane, więc muszą być odpowiednio zabezpieczone.

Dobre praktyki

- **Czytelności i utrzymania** – kod powinien być dobrze udokumentowany i łatwy do rozwijania przez różnych programistów.
- **Zgodności z dobrymi praktykami** – np. stosowanie podpowiedzi typów (`Type Hints`), testów jednostkowych, CI/CD, zarządzania zależnościami.

Programowanie na poziomie ***enterprise (produkcyjnym)*** w Python

- Używanie podpowiedzi **typów**,
- Stosowanie wzorców projektowych,
- Budowanie testowalnych, modułowych aplikacji,
- Integracja z systemami bazodanowymi i chmurowymi.
- To różni się od **skryptowego** użycia Pythona, gdzie kod często jest pisany szybciej, bez rygorystycznych standardów i z myślą o mniejszych projektach

Kod bez typowania

```
class Student:
     def __init__(self, name, student_id, tution_balance):
         self.name = name
         self.student_id = student_id
         self.tuition_balance = tuition_balance

walker = Student("Sarah Walker", 319921, 15000)
```

- Czy **`student_id`** powinno być liczbą całkowitą?
- Czy **`tuition_balance`** powinno być liczbą zmiennoprzecinkową, liczbą całkowitą, a może ciągiem znaków?
- Po utworzeniu **`walker`**, jak możemy później w kodzie zapamiętać, jakiego jest typu?

Podpowiedzi typów (ang. *type hints*)

Opcjonalne narzędzie, które pozwala na dodanie informacji o typie obiektu w kodzie.

- Ułatwia czytanie i debugowanie.
- Jedna z najlepszych metod demonstrowania umiejętności w Pythonie na poziomie ***enterprise***.
- Nie jest wymuszane przez interpreter.
- Wykorzystuje wbudowane słowa kluczowe typów, bibliotekę **`typing`** oraz **własne klasy.**

```
# Typowanie dla zmiennej
gpa: float = 3.92

# Typowanie dla funkcji/metody
def check_grades(year: str) -> List[int]:
    ...

# Można nawet użyć własną klasę
students: Dict[str, Student] = {...}
```

Podpowiedzi typów z wbudowanymi słowami kluczowymi

```
name: str = "Jan"
student_id: int = 2137
tuition_balance: float = 1745.75

def get_schedule(semester: str) -> dict:
    …
```

- Użyj składni **`zmienna: typ`** do określenia typu zmiennej.
- Deklaratywne, określa sygnaturę funkcji/metody.
- **`def .... () -> typ:`** pozwala określić typ zwracanej wartości.

Bibliotek ***typing***

- Biblioteka używana do zapewnienia większej ilość narzędzi do podpowiedzi typów
- `List, Dict, Tuple`
- Wysoko poziomowe obiekty

```
from typing import List, Dict

student_names: List[str] = ["Maciej", "Jan", "Anna"]
student_grades: Dict[str, float] = {
    "Adam":3.5,
    "Ola":4.5
}
```

Podpowiedzi typów z własnymi klasami

```
class Student:
      def __init__(self, name: str, student_id: int, tuition_balance: float) -> None:
          self.name: str = name
          self.student_id: int = student_id
          self.tuition_balance: float = tuition_balance

      def get_course(self, course_id: str) -> Course:
          …

          return course
```

Zobacz: student.py

```
>>> studnet_1: Student = Student("Jan Nowal", 3125, 2000)
>>> data_science: Course = walker.get_course("TDM-2000")
>>> print(type(student_1))
<class '__main__.Student'>
>>>print(type(data_science))
<class '__main__.Course'>
```

Leniwe wartościowanie

- Koncepcja, zgodnie z którą, zwłaszcza gdy mamy do czynienia z **dużymi ilościami danych**, nie chcemy przetwarzać wszystkich danych zanim nie będą potrzebne wyniki.
- Znany przykład polecenie **`range`** dla którego zużycie pamięci jest takie samo nawet wtedy, gdy zakres reprezentuje duży zbiór liczb

Generatory

- Generatorów można używać w podobny sposób do obiektów range.
- Generatory wykonują kilka operacji na danych w porcjach — na żądanie.
- Pomiędzy wywołaniami generatory zachowują swój stan.
  - Można zapisać zmienne, które są potrzebne do obliczenia wyniku, i sięgać do nich za każdym razem, gdy jest wywoływany generator.

Funkcje generatorów w Pythonie

- Zamiast `return`, używamy słowa kluczowego **`yield`**.
- Każde wywołanie generatora zwraca wartość z **`yield`** i **zapamiętuje swój stan**.
- Wykonanie funkcji wznawia się od miejsca, w którym zostało przerwane.

```
def count():
    n = 0
    while True:
        n +=1
        yield n


>>> counter = count()
>>> next(counter)
1
>>> next(counter)
2
```

Funkcja next()

- **`next()`** to wbudowana funkcja w Pythonie, która pobiera kolejny element z iteratora.
- Każde wywołanie **`next(iterator)`** zwraca kolejny element.
- Jeśli iterator się skończy, rzuca wyjątek **`StopIteration`**.
- Działa w tle w pętlach **`for`**, automatycznie pobierając kolejne wartości.

```
# Tworzymy iterator z listy
numbers = iter([1, 2, 3])

print(next(numbers))  # 1
print(next(numbers))  # 2
print(next(numbers))  # 3

print(next(numbers))
# Błąd: StopIteration
```

Własne iteratory

- Klasy umożliwiające przeglądanie kolekcji obiektów lub strumieni danych, zwracając po jednym elemencie na raz.
- Podobne do list czy krotek, ale działają inaczej.
- Mogą nawigować, transformować i generować dane.
- Iterowane za pomocą pętli **for**.
- Obsługiwane przez funkcję **next()**.

```
# Kolekcja
Chuck = NameIterator("Jan Nowak")
for letter in chuck:
    print(letter)
j
a
n
…

# strumień danych
fun_gama = DiceGame(rolls=3)
next(fun_gama)
4
```

Protokół iteratora

`__iter__()`

- Zwraca iterator, w tym przypadku referencję do samego siebie.
- **`return self`**

`__next__()`

- Zwraca kolejną (**next**) wartość z kolekcji lub strumienia danych.
- W tej metodzie odbywa się iteracja, transformacja i generowanie danych.

Aby klasa była uznana za iterator, musi definiować zarówno `__iter__()`, jak i `__next__()` !

Przykład

```
Class CoinFlips:
    def __init__(self, number_of_flips):
       self.number_of_flips = number_of_flips
       self.counter = 0

    def __iter__(self):
        return self # zwrca referncję iteratora

    # Rzut monetą, zwrca wynik orzeł lub reszka
    def __next__(self):
        if self.counter < self.number_of_flips:
            self.counter += 1
            return random.choice(["H", "T"])
```

Użycie iteratora

```
>>> three_flips = CoinFlips(3)

>>> next(three_flips)
H
>>> next(three_flips)
H
>>> next(three_flips)
T
```

Użycie pętli for dla naszego iteratora

```
three_flips = CoinFlips(3)

for flip in three_flips:
    print(flip)
```

```
T
H
T
None
None
...
```

StopIteration

```
def __next__(self):
    if self.counter < self.number_of_flips:
        self.counter += 1
        return random.choice(["H", "T"])
    else:
        raise StopIteration
```

- Sygnał końca kolekcji/strumienia danych
- Zabezpiecza przed nieskończoną pętlom
- Łatwy w obsłudze

Iteracja po dodaniu StopIteration

```
three_flips = CoinFlips(3)

for flip in three_flips:
    print(flip)
```

```
T
H
T
```

Obsługa wyjątku StopIteration

```
while True:
    try:
        next(three_flips)
    except StopIteration:
        print("Completed all coin flips!")
        break
```

H
H
C
Completed all coin flips!