rozdział 8:

WYJĄTKI

ostatnia modyfikacja: 27.11.17

Podstawy programowania

wyjątek (ang. exception):

- **wyjątek** to specyficzna dana, powstająca w sposób *automagiczny* w chwili **wystąpienia błędu**, który uniemożliwia kontynuowanie wykonania programu
- sytuację taką nazywa się **podniesieniem** (ang. raise) wyjątku

Podstawy programowania

wyjątek został podniesiony… i co dalej?

- podniesiony wyjątek oczekuje, że ktoś go zauważy i obsłuży
- jeśli to się nie stanie, wyjątek spowoduje awaryjne zakończenie wykonania programu

Podstawy programowania

a co, jeśli wyjątek został zauważony?

- zauważenie wyjątku pozwala zidentyfikować błąd i go naprawić (albo zignorować)
- w takiej sytuacji program może wykonywać się dalej

jak identyfikować wyjątki?

- wszystkie wyjątki mają nazwy i są ułożone w hierarchicznej strukturze, pozwalającej wygodnie kategoryzować błędy o różnej naturze

(poezja mimowolna...)

Podstawy programowania

Jak rozpoznać wyjątek?

```
napis = 'fantasmagoria'
liczba = int(napis)
```

```
Traceback (most recent call last):
  File "r.py", line 2, in <module>
    liczba = int(napis)
ValueError: invalid literal for int() with base 10: 'fantasmagoria'
```

Podstawy programowania

Jak rozpoznać wyjątek?

```
liczba = 10
liczba /= 0
```

```
Traceback (most recent call last):
  File "r.py", line 2, in <module>
    liczba /= 0
ZeroDivisionError: division by zero
```

Podstawy programowania

Jak rozpoznać wyjątek?

```
l = [ ]
x = l[0]
```

```
Traceback (most recent call last):
  File "r.py", line 2, in <module>
    x = l[0]
IndexError: list index out of range
```

Podstawy programowania

jak obsługiwać wyjątki?

- najpierw trzeba spróbować (ang. **try**) coś zrobić…
- a potem zobaczyć, czy nie stało się coś złego

Podstawy programowania

dygresja:

- generalnie (i niezbyt formalnie) można wyróżnić dwa style programowania:
  - **defensywny** – zanim coś zrobimy, sprawdźmy, czy to się uda
  - **ofensywny** – zróbmy coś w ciemno i sprawdźmy, czy się udało

Podstawy programowania

Styl defensywny:

```
a = int(input())
b = int(input())
if b != 0:
   print(a / b)
else:
   print('Nic z tego!')
print('To koniec')
```

Podstawy programowania

Styl ofensywny:

```
a = int(input())
b = int(input())
try:
    print(a / b)
except:
    print('Nic z tego!')
print('To koniec')
```

Podstawy programowania

Najpierw próbujemy:

```
a = int(input())
b = int(input())
try:
   print(a / b)
except:
   print('Nic z tego!')
print('To koniec')
```

try:
tak informujemy Pythona, że zamierzamy zrobić coś, co może skończyć się **niepowodzeniem**;

jeżeli faktycznie stanie się tu coś złego, pozostałe instrukcje z części try zostaną **pominięte** i zostanie **podniesiony wyjątek…**

i zacznie się szukanie kogoś, kto byłby skłonny ten wyjątek **obsłużyć**

Podstawy programowania

Potem sprawdzamy:

```
a = int(input())
b = int(input())
try:
   print(a / b)
except:
   print('Nic z tego!')
print('To koniec')
```

except:
tak informujemy Pythona, że chcemy obsłużyć **dowolny** wyjątek, jaki ewentualnie został podniesiony w **poprzedzającej** części **try**

teraz możemy na spokojnie zająć się problemem i go rozwiązać... albo go zignorować

Podstawy programowania

I kontynuujemy pracę:

```
a = int(input())
b = int(input())
try:
   print(a / b)
except:
   print('Nic z tego!')
print('To koniec')
```

pracujemy dalej, jak gdyby się nic nie wydarzyło

Podstawy programowania

zapamiętaj:

- najpierw podejmuje się próbę wykonania **wszystkich** instrukcji stojących pomiędzy try i except

Podstawy programowania

zapamiętaj:

- jeżeli nie stanie się nic złego, wykonanie przeskakuje za ostatnią instrukcję gałęzi except i program wykonuje się dalej

Podstawy programowania

zapamiętaj:

- jeżeli w gałęzi try stanie się coś złego, wykonanie **natychmiast** przeskakuje do gałęzi except
- oznacza to, że pewne instrukcje w gałęzi try mogą zostac pominięte

Podstawy programowania

Jak przebiega „wyskok“ z bloku try?

```
try:
    print('1')
    x = 1 / 0
    print('2')
except:
    print('coś poszło źle')
print('3')
```

```
1
coś poszło źle
3
```

uwaga:

- w takiej postaci bloku try-except pewną niedogodnością jest to, że wszystkie, nawet bardzo różne wyjątki, wpadają do tego samego miejsca
- i nie mamy jak sprawdzić, co naprawdę się stało

Podstawy programowania

Jak przebiega „wyskok“ z bloku try?

```
try:
   x = int(input())
   y = 1 / x
except:
   print('coś poszło źle')
print('To koniec')
```

Pojawienie się tego komunikatu nie pozwala nam stwierdzić, co się tak naprawdę stało:
- czy wprowadzono błędną daną?
- czy dana była poprawna, ale równa zero?

co robić?

- można postąpić na dwa sposoby:
  - zbudować dwa, kolejno po sobie następujące, bloki try-except (niestety, spowoduje to znaczny rozrost kodu)
  - wykorzystać bardziej złożoną postać naszej nowej instrukcji

Podstawy programowania

```
try:
   :
except exc1:
   :
except exc2:
   :
except:
   :
```

Tu trafimy, gdy zostanie podniesiony wyjątek **exc1**

Tu trafimy, gdy zostanie podniesiony wyjątek **exc2**

A tu trafimy, gdy zostanie podniesiony wyjątek **nie wymieniony wcześniej**

Podstawy programowania

Selektywna obsługa wyjątków:

```
try:
    x = int(input())
    y = 1 / x
except ZeroDivisionError:
    print('nie potrafię dzielić przez zero!')
except ValueError:
    print('a miałeś podać liczbę!')
except:
    print('Och....')
print('To koniec')
```

Tutaj trafimy np. wtedy, gdy zamiast wprowadzić daną naciśniemy klawisze **Ctrl-C**

Podstawy programowania

zapamietaj:

- gałęzie except są przeszukiwane w takiej **kolejności**, w jakiej zostały **zapisane w programie**
- liczba różnych gałęzi except jest **dowolna**, ale jeśli wystąpiło try, to musi pojawić się **choć jedno** except
- except nie może wystąpić bez poprzedzającego try

Podstawy programowania

zapamietaj:

- jeśli **wykonała się** jakaś gałąź except, to już nie wykona się **żadna inna**
- jeśli żadna z gałęzi except **nie będzie pasować** do podniesionego wyjątku, to wyjątek zostanie uznany za **nieobsłużony**
- except **bez nazwy wyjątku** (tzw. except domyślny) musi być wymieniony jako **ostatni**

Podstawy programowania

Usunęliśmy jedną z gałęzi – i co teraz?

```
try:
    x = int(input())
    y = 1 / x
except ValueError:
   print('a miałeś podać liczbę!')
except:
   print('Och....')
print('To koniec')
```

```
0
Och....
To koniec
```

Ponieważ nie mamy już dedykowanej gałęzi dla wyjątki **ZeroDivisionError**, dzielenie przez zero trafi do gałęzi **domyślnej**

Podstawy programowania

Usunęliśmy jeszcze jedną:

```
try:
    x = int(input())
    y = 1 / x
except ValueError:
   print('a miałeś podać liczbę!')
print('To koniec')
```

Wyjątek ZeroDivisionError nie doczekał się obsługi :(

```
0
Traceback (most recent call last):
  File "r.py", line 3, in <module>
    y = 1 / x
ZeroDivisionError: division by zero
```

Podstawy programowania

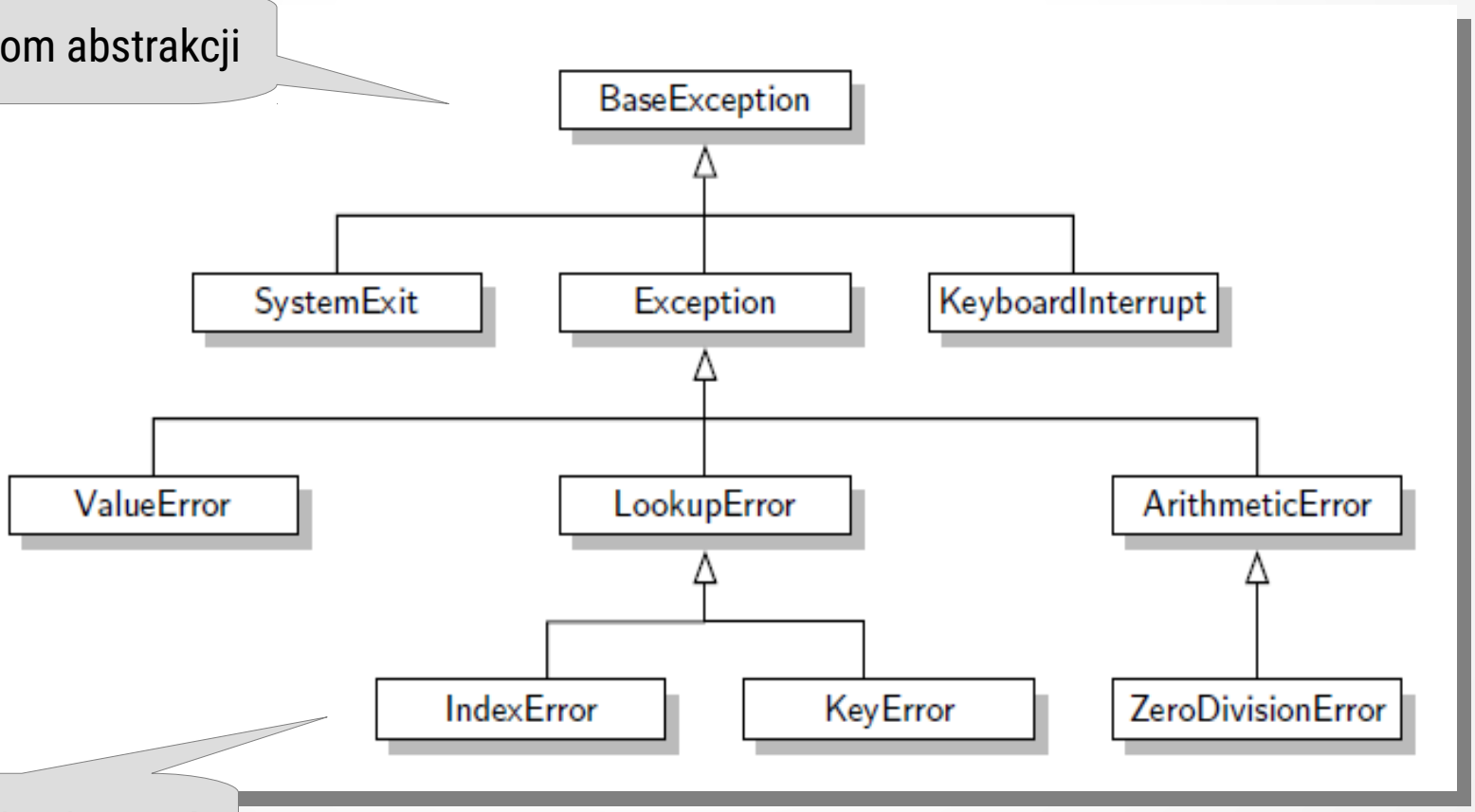
wyjątki w Pythonie 3:

- Python 3 wyróżnia **63** wyjątki wbudowane
- wyjątki w Pythonie tworzą **hierarchię drzewiastą**
- oznacza to, że pewne wyjątki są bardziej **ogólne**, a pewne bardziej **szczególne**
- innymi słowami, pewne są **abstrakcyjne**, a pewne **konkretne**

Podstawy programowania

- **Oto (skromny) fragment drzewa wyjątków Pythona:**

wysoki poziom abstrakcji



wysoki poziom konkretności

Podstawy programowania

hierarchia wyjątków na przykładzie ZeroDivisionError:

- ZeroDivisionError jest szczególnym przypadkiem wyjątku
- ArithmeticError
- ArithmeticError jest szczególnym przypadkiem wyjątku Exception
- Exception jest szczególnym przypadkiem wyjątku BaseException
- czyli (zwróć uwagę na zwrot strzałek):
  BaseException ← Exception ← ArithmeticError ← ZeroDivisionError

Podstawy programowania

To już znamy:

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print('problem?')
print('To koniec')
```

Obsłużyliśmy wyjątek ZeroDivisionError

```
problem?
To koniec
```

Podstawy programowania

A teraz drobna – ale znacząca - zmiana:

```
try:
    y = 1 / 0
except ArithmeticError:
    print('problem?')
print('To koniec')
```

Działa tak samo – czemu?

```
problem?
To koniec
```

Podstawy programowania

Kolejna zmiana:

```
try:
    y = 1 / 0
except Exception:
    print('problem?')
print('To koniec')
```

Ciągle tak samo!

```
problem?
To koniec
```

Podstawy programowania

zapamiętaj:

- każdy wyjątek wpada w **pierwszą gałąź** except, która specyfikuje **pasujący** wyjątek
- to niekoniecznie musi być dokładnie taki sam wyjątek – może to być również wyjątek **bardziej ogólny** (bardziej abstrakcyjny)

Podstawy programowania

Prosimy o uwagę – co się stanie teraz?

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print('dzielenie przez zero')
except ArithmeticError:
    print('błąd arytmetyczny')
print('To koniec')
```

Podstawy programowania

Oczywiście!

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print('dzielenie przez zero')
except ArithmeticError:
    print('błąd arytmetyczny')
print('To koniec')
```

```
dzielenie przez zero
To koniec
```

Podstawy programowania

A co teraz się stanie?

```
try:
    y = 1 / 0
except ArithmeticError:
   print('błąd arytmetyczny')
except ZeroDivisionError:
   print('dzielenie przez zero')
print('To koniec')
```

Podstawy programowania

Co z tego wynika?

```
try:
    y = 1 / 0
except ArithmeticError:
   print('błąd arytmetyczny')
except ZeroDivisionError:
   print('dzielenie przez zero')
print('To koniec')
```

```
błąd arytmetyczny
To koniec
```

Podstawy programowania

zapamiętaj:

- kolejność gałęzi except ma **znaczenie**!
- nie umieszczaj wyjątku bardziej abstrakcyjnego **przed** bardziej konkretnym
- no chyba, że wiesz co robisz

Podstawy programowania

Jeszcze jedna forma:

Gdy więcej niż jednemu wyjątkowi chcesz przypisać taką samą obsługę, wymień je wszystkie rozdzielając przecinkami i ujmując w nawiasy

```
try:
   :
except ( exc1,exc2 ):
   :
```

Podstawy programowania

wyjątki i funkcje:

- jeśli pewien wyjątek zostanie podniesiony w funkcji, to może zostać **obsłużony**:
  - w **tej** funkcji
  - **poza** tą funkcją
  - **i tu, i tu**

Podstawy programowania

Obsługa wewnątrz funkcji:

```
def badfun(n):
    try:
        return 1/n
    except ArithmeticError:
        print('Sorry, taką mamy arytmetykę!')
        return None

nn = 0
badfun(nn)
print('To koniec')
```

```
Sorry, taką mamy arytmetykę!
To koniec
```

Podstawy programowania

Obsługa poza funkcją:

```
def badfun(n):
   return 1/n

nn = 0
try:
    badfun(nn)
except ArithmeticError:
        print('Funkcjo, cóżeś mi uczyniła?')
print('To koniec')
```

```
Funkcjo, cóżeś mi uczyniła?
To koniec
```

Podstawy programowania

instrukcja raise:

- `raise` *wyjątek*
- powoduje podniesienie wyspecyfikowanego wyjątku tak, jakby wystąpił naprawdę

Podstawy programowania

Wyjątek podniesiony na żądanie:

```
def badfun(n):
   raise ZeroDivisionError

nn = 0
try:
    badfun(nn)
except ArithmeticError:
    print('Funkcjo, cóżeś mi uczyniła?')
print('To koniec')
```

```
Funkcjo, cóżeś mi uczyniła?
To koniec
```

Podstawy programowania

instrukcja raise:

- `raise`

- powoduje podniesienie **takiego samego** wyjątku, jaki jest właśnie **obsługiwany**
- co oznacza, że instrukcji tej można użyć tylko **w ramach obsługi wyjątku**

Podstawy programowania

Obsługa wyjątku w funkcji i poza funkcją:

```
def badfun(n):
    try:
        return n/0
    except:
        print('Znowu się nie udało...')
        raise

nn = 0
try:
    badfun(nn)
except ArithmeticError:
    print('Funkcjo, cóżeś mi uczyniła?')
print('To koniec')
```

```
Znowu się nie udało...
Funkcjo, cóżeś mi uczyniła?
To koniec
```

Podstawy programowania

dygresja:

- zapoznamy się z kolejną instrukcją Pythona 3:

```
assert wyrażenie
```

- jest to tzw. **asercja**

Podstawy programowania

jak działa asercja?

- instrukcja assert oblicza swoje wyrażenie
- jeśli jego wartością jest True albo wartość **różna od zera i od None** albo **niepusty napis**, to nie dzieje się **NIC**
- w przeciwnym przypadku zostaje **podniesiony** wyjątek AssertionError (mówimy wtedy, że asercja zawiodła)

Podstawy programowania

jak się używa asercji?

- asercje wstawia się w kod w miejsca, w których chcemy **zabezpieczyć** się przed przeniknięciem ewidentnie błędnych danych
- łatwiej jest ustalić, czemu pewna asercja zawiodła, niż szukać błędu w całym kodzie
- asercje nie zastępują wyjątków ani sprawdzania poprawności danych – są ich **uzupełnieniem**

Podstawy programowania

Na przykład:

```
import math
x = float(input())
assert x>=0.0
x = math.sqrt(x)
print(x)
```

```
-1
Traceback (most recent call last):
  File "r.py", line 3, in <module>
    assert x>=0.0
AssertionError
```

Podstawy programowania

Na przykład:

```
import math
x = float(input())
assert x>=0.0
x = math.sqrt(x)
print(x)
```

```
0
0
```

wybrane wyjątki wbudowane Pythona 3

`ArithmeticError`

- BaseException ← Exception ← ArithmeticError
- wyjątek abstrakcyjny, zawierający w sobie wyjatki wywoływane przez błędy operacji arytmetycznych

`AssertionError`

- BaseException ← Exception ← AssertionError
- podnoszony przez instrukcję assert, gdy asercja zawodzi

BaseException

- BaseException
- wyjątek najbardziej abstrakcyjny, obejmuje sobą wszelkie możliwe wyjątki

`Exception`

- BaseException ← Exception
- wyjątek abstrakcyjny, zawierający w sobie te wyjątki wbudowane, które są podnoszone w wyniku błędów

`FloatingPointError`

- BaseException ← Exception ← ArithmeticError ← FloatingPointError
- podnoszony przez błędy operacji arytmetycznych na liczbach rzeczywistych

IndexError

- BaseException ← Exception ← LookupError ← IndexError
- podnoszony przez użycie niepoprawnego indeksu sekwencji

KeyboardInterrupt

- BaseException ← KeyboardInterrupt
- podnoszony w wyniku użycia kombinacji klawiszy przerywających program (najczęściej Ctrl-C)

LookupError

- BaseException ← Exception ← LookupError
- wyjątek abstrakcyjny, zawierający w sobie wszystkie wyjątki wywoływane przez błędy dostępu do tzw. kolekcji (sekwencje są jednymi z kolekcji)

MemoryError

- BaseException ← Exception ← MemoryError
- podnoszony gdy operacja nie może zostać zakończona z powodu braku dostępnej pamięci

`OverflowError`

- BaseException ← Exception ← ArithmeticError ← OverflowError
- podnoszony, gdy wynik operacji arytmetycznej jest zbyt duży (co do modułu), by można go było zachować

RuntimeError

- BaseException ← Exception ← RuntimeError
- wyjątek abstrakcyjny, zawierający w sobie te wyjątki wbudowane, które są podnoszone w na skutek błędów w wykonaniu kodu

`RecursionError`

- BaseException ← Exception ← RuntimeError ← RecursionError
- podnoszona na skutek (potencjalnie) nieskończonej rekurencji