České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Výjimky

Úvod

Zatímco v některých jazycích je typickým způsobem reportování chyb zahrnuto do návratových hodnot funkcí, Python patří k jazykům, kde je na to vyhrazena mašinérie *výjimek*.

Pár typických příkladů (přímo podle dokumentace):

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Výjimky standardně „probublávají“ postupně po hierarchii programu, ale každopádně na konci trasovacího výpisu (*stack traceback*) bude kdo a jakou výjimku vyvolal.

Ošetření výjimek

Výjimky je možné ošetřovat několika různými způsoby. Ukažme si základní z nich:

I. Základní použití – odchyť jednotlivé typy výjimek, ke kterým může dojít v *blok-1*:

```
try:
  # blok-1
except VÝJIMKA-1:
  # blok-2
except VÝJIMKA-2:
  # blok-3
```

II. Odchycení výjimky a předání jejího objektu pod vybraným jménem:

```
try:
  # blok-1
except VÝJIMKA as ex:
  # blok-2; je v něm možné použít objekt odchycené výjimky pod jmé
```

III. Nevyvolá-li kód v *blok-1* žádnou výjimku, provede se kód v *blok-3*:

```
try:
  # blok-1
except:
  # blok-2
else:
  # blok-3
```

IV. Kód v *blok-3* bude vykonán vždy při opouštění větve *try*, ať už vyvolala výjimku nebo ne:

```
try:
  # blok-1
except:
  # blok-2
finally:
  # blok-3
```

→ Větve *finally* se tak používají často pro „vyčišťovací“ kód, typicky pro uzavření otevřených zdrojů apod.

Příklad

Následující příklad ukazuje základní použití výjimek:

```
try:
    text = input('Enter something --> ')
except EOFError:   # Unix: ^D & Windows: ^z+↵
    print('\nWhy did you do an EOF on me?')
except KeyboardInterrupt:   # Unix: ^C & Windows: ^C
    print('\nYou cancelled the operation.')
else:   # něco jsme napsali a odentrovali
    print('You entered "{0}".'.format(text))
finally:   # vykoná se vždy při opouštění větve 'try'
    print('Completed.')
```

→ Upraveno podle http://www.swaroopch.com/notes/Python_en:Exceptions

- Zadáme-li nějaký text a poté odentrujeme, obdržíme následující výsledek:

```
Enter something --> ahoj
You entered "ahoj".
Completed.
```

- Přerušíme-li zadávání textu pomocí ^C, výstup se změní:

```
Enter something --> ahoj^C
You cancelled the operation.
Completed.
```

- Konečně zadáme-li místo vstupního textu rovnou EOF, uplatní se ještě jiná větev:

```
Enter something -->
Why did you do an EOF on me?
Completed.
```

Souhrnně tedy platí, že větev *finally* se provede vždy při opouštění větve *try*, zatímco větev *else* se provede pouze tehdy, nedojde-li ve větvi *try* k vyvolání výjimky.

Více výjimek v jedné větvi

Je též možné odchytit více výjimek najednou v podobě tuplu:

```
try:
  # blok-1
except (VÝJIMKA-1, VÝJIMKA-2, ...):
  # blok-2
```

→ Tohle je v ostrém kontrastu s chováním v Python'u 2.x, kde

uvedená konstrukce sloužila k zachycení výjimky pod jistým jménem (čili k tomu, co teď mnohem průhledněji a srozumitelněji dělá operátor as).

Objekt výjimky

Jelikož odchycená výjimka je objekt, má jistou vnitřní strukturu (ať už výchozí nebo uživatelem předefinovanou). Ukažme si několik příkladů:

I. Standardní výjimky mají nadefinované chování při pokusu o jejich výpis:

```
>>> try:
...     raise NameError('HiThere')
... except NameError as ex:
...     print(ex)
HiThere
```

→ To zajišťuje přítomnost „magické“ metody *__str__()*.

II. Struktura objektu výjimek je ale mnohem složitější:

```
>>> try:
...     raise NameError('HiThere')
... except NameError as ex:
...     print( dir(ex) )
['__cause__', '__class__', '__context__', '__delattr__', '__dict__
```

Především vidíme, že (standardní) výjimky mohou dostávat argumenty:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # instance výjimky
...     print(inst.args)     # argumenty výjimky jsou uloženy v par
...     print(inst)          # díky '__str__' můžeme argumenty vyti
...     x, y = inst.args     # n-ticové rozbalení argumentů
...     print('x =', x)
...     print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Vyvolání výjimky

K vyvolání výjimky slouží příkaz `raise`. Jeho použití je standardní (přímo podle dokumentace):

I. Konkrétní výjimku můžeme uměle vyvolat:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

II. Výjimku můžeme po jejím odchycení a částečném zpracování předat po hierarchii dále:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
An exception flew by!
Traceback (most recent call last):
  File "<pyshell#23>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

Hierarchie standardních výjimek

→ Viz http://docs.python.org/release/3.1.3/library/exceptions.html

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EnvironmentError
      |    +-- IOError
      |    +-- OSError
      |         +-- WindowsError (Windows)
      |         +-- VMSError (VMS)
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
```

```
|    +-- KeyError
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
     +-- PendingDeprecationWarning
     +-- RuntimeWarning
     +-- SyntaxWarning
     +-- UserWarning
     +-- FutureWarning
     +-- ImportWarning
     +-- UnicodeWarning
     +-- BytesWarning
```

Vlastní výjimky

Samozřejmě kromě výše uvedených předdefinovaných výjimek si můžete nadefinovat výjimky vlastní. Typicky to budou potomci třídy *Exception* a není-li třeba, postačí velmi jednoduché, např.:

```
class MyError(Exception):
    pass
```

Ve složitějších případech můžete nadefinovat vlastní hierarchii výjimek a i předefinovat jejich výchozí chování. Ukázka přímo z dokumentace:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is n
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

assert

Kromě výjimek, které slouží především k odchytávání chyb v běžících a již nasazených programech, máme v Python'u k dispozici i standardní ladící prostředek – příkaz `assert`. Kromě plného použití v rámci testování nachází uplatnění i při rychlém odchytávání chyb při ladění programu.

> → Při spuštění interpretru s parametrem `-O` jsou příkazy *assert* odstraněny.

Struktura příkazu *assert* je `assert VÝRAZ [, ARGUMENTY]`. Použití ilustruje následující příklad:

```
>>> xs = [1, 2, 3]

# A) Tohle je pravda, není co řešit:
>>> assert len(xs) == 3

# B1) Tohle pravda ale není:
>>> assert len(xs) == 2
Traceback (most recent call last):
  File "<pyshell#31>", line 2, in <module>
    assert len(xs) == 2
AssertionError

# B2) Tohle také není pravda. Navíc přidáme komentář:
>>> assert len(xs) == 2, 'Špatný počet argumentů!'
Traceback (most recent call last):
  File "<pyshell#32>", line 2, in <module>
    assert len(xs) == 2, 'Špatný počet argumentů!'
AssertionError: Špatný počet argumentů!

# B2) Nebo ještě víc údajů:
>>> assert len(xs) == 2, ('Špatný počet argumentů!', len(xs) )
Traceback (most recent call last):
  File "<pyshell#33>", line 2, in <module>
    assert len(xs) == 2, ('Špatný počet argumentů!', len(xs) )
AssertionError: ('Špatný počet argumentů!', 3)
```

Příklad – kontrola vstupních parametrů:

Program *debug_assert.py* **:**

```
import math

def vector_length( vector ):
  """This function calculates l
  vector should therefore be a

  assert len(vector) == 3
  x,y,z = vector
  return math.sqrt( x**2 + y**2

print( vector_length([1,2,3]) )
print( vector_length((0,2,0)) )
print( vector_length([1,2])   )
```

Výstup:

```
3.74165738677
2.0
Traceback (most recent call las
  File "debug_assert.py", line
    print( vector_length([1,2])
  File "debug_assert.py", line
    assert len(vector) == 3
AssertionError
```