

Programowanie obiektowe

Wyjątki, metody specjalne, generatory

Paweł Daniluk

Wydział Fizyki

Jesień 2016



"It's easier to ask forgiveness than it is to get permission."

Co zrobić, jeżeli podczas wykonania programu zajdzie nietypowa sytuacja, która wykracza poza założony scenariusz?

Dwie możliwości

- 1 Sprawdzić przed każdym krokiem, czy można go wykonać.
- 2 Poczekać, aż wystąpi problem i wtedy go obsłużyć.

Drugi wariant pozwala na rozdzielenie miejsca, gdzie problem występuje, od miejsca gdzie jest obsługiwany.

Idea

Rzucanie

W momencie wystąpienia sytuacji nietypowej jest tworzony i rzucany wyjątek.

Przechwytywanie

W miejscu, gdzie jest możliwe obsłużenie błędu wyjątek jest przechwytywany.

Bez wyjątków

```
def czytaj1(f):
    line=f.readline()
    if line=='': # koniec pliku
        return (None, 'EOF')
    else:
        return (A(line), 'OK')

def czytaj2(f):
    ...

def czytaj(f):
    done=False
    while not done:
        (resA, status)=czytaj1(f)
        if status=='EOF':
            return (None, 'EOF')
        (resB, status)=czytaj2(f)
        if status=='EOF':
            return (None, 'EOF')
        res.extend((resA, resB))
        done=przetworz(resA, resB)

    return (res, 'OK')
```

Bez wyjątków c.d.

```
def main():  
    f=...  
    (res, status)=czytaj(f)  
    if status=='EOF':  
        print "Blad"  
    else:  
        print 'OK'
```

Z wyjątkami

```
class EOFException(Exception):  
    pass  
  
def czytaj1(f):  
    line=f.readline()  
    if line=='': # koniec pliku  
        raise EOFException()  
  
    return A(line)  
  
def czytaj2(f):  
    ...  
  
def czytaj(f):  
    done=False  
    while not done:  
        resA=czytaj1(f)  
        resB=czytaj2(f)  
        res.extend((resA, resB))  
        done=przetworz(resA, resB)  
  
    return (res)
```

Z wyjątkami c.d.

```
def main():
    f=...
    try:
        res=czytaj(f)
        print 'OK'
    except EOFException:
        print "Blad"
```

Co się może zdarzyć

- błędy arytmetyczne (dzielenie przez 0, pierwiastek z liczby ujemnej)
- pusta lista
- brak klucza w słowniku
- błędne dane na wejściu
- brak pliku, zła nazwa
- wyjście poza zakres
- ...

Przechwytywanie wyjątków

Instrukcja try

```
try :  
    f=open (...)  
    doSomething (...) # kod, który może wygenerować wyjątek  
except Exception1 as e:  
    process_exception(e) # przechwycono wyjątek klasy Exception1  
except Exception2:  
    print "Olaboga!!!" # przechwycono wyjątek klasy Exception2  
else:  
    jestSuper() # nie wystąpił żaden wyjątek  
finally:  
    f.close() # niezależnie co się stało trzeba posprzątać
```

Wyjątek jest przechwytywany przez najgłębiej zagnieżdżoną instrukcję try , która ma pasującą klauzulę except .

Catch only what you can handle.

Przykład

Brak klucza w słowniku

```
if k in d:  
    return d[k]  
else:  
    return None
```

```
try:  
    return d[k]  
except KeyError:  
    return None
```

Wyjątki standardowe

ArithmeticError nadklasa obejmująca wyjątki związane z błędami numerycznymi

AssertionError niespełniona asercja w instrukcji `assert`

AttributeError brak atrybutu

IndexError brak elementu o podanym indeksie

KeyError brak klucza w słowniku

NameError brak zmiennej

NotImplementedError nie zaimplementowana funkcjonalność

OverflowError przepełnienie

StopIteration koniec iteracji (podnoszony w metodzie `next()` iteratora)

ZeroDivisionError dzielenie przez 0

Podnoszenie wyjątku

W wielu sytuacjach wyjątek jest podnoszony samoczynnie (np. przez metody biblioteki standardowej).

Instrukcja raise

```
raise MyException("Message", data)
```

Własne wyjątki

Można definiować własne wyjątki. Muszą one dziedziczyć z klasy Exception .

```
class MyException(Exception):  
    def __init__(self, message, data):  
        self.message, self.data=message, data  
  
    def __str__(self):  
        return message+' :: '+str(data)
```

Metody specjalne

W Pythonie obiekty mogą mieć metody, które są wywoływane przy użyciu specjalnej składni.

Przykładem jest metoda `__init__`, która jest wywoływana automatycznie po stworzeniu obiektu.

Zamiana na napis

`object.__repr__(self)` zwraca reprezentację obiektu, która jest zgodna ze składnią Pythona, lub jeśli to niemożliwe napis postaci '`<opis obiektu>`', który jest użyteczny przy debugowaniu – wywoływana przez `repr(object)`

`object.__str__(self)` zwraca “ładny” napis – wywoływana przez `str(object)`

Kontenery

`object.__len__(self)` liczba elementów – wywoływana przez `len(object)`

`object.__getitem__(self, key)` pobieranie wartości – wywoływana przez `object[key]`

`object.__setitem__(self, key, value)` zmiana – wywoływana przez `object[key]=value`

`object.__delitem__(self, key)` usuwanie – wywoływana przez `del object[key]`

`object.__iter__(self)` iterator

`object.__reversed__(self)` iterator iterujący w przeciwnym kierunku – wywoływana przez `reversed(object)`

`object.__contains__(self, item)` zawieranie – wywoływana przez `item in object`

Operacje arytmetyczne

object.__add__(self, other) object + other
object.__sub__(self, other) object - other
object.__mul__(self, other) object * other
object.__div__(self, other) object / other
object.__floordiv__(self, other) object // other
object.__mod__(self, other) object % other
object.__divmod__(self, other) divmod(object, other)
object.__pow__(self, other) object ** other
object.__lshift__(self, other) object << other
object.__rshift__(self, other) object >> other
object.__and__(self, other) object & other
object.__xor__(self, other) object ^ other
object.__or__(self, other) object | other

Operacje arytmetyczne c.d.

```
object.__neg__(self)  -object  
object.__pos__(self)  +object  
object.__abs__(self)  abs(object)  
object.__invert__(self) ~object
```

Porównywanie

```
object.__lt__(self, other)  object < other
object.__le__(self, other) object <= other
object.__eq__(self, other) object == other
object.__ne__(self, other) object != other
object.__gt__(self, other) object > other
object.__ge__(self, other) object >= other
object.__cmp__(self, other) zwraca wartość ujemną jeżeli object <
                             other , zero jeżeli object==other i liczbę dodatnią jeżeli
                             object > other
```

Generatory

Problem

Jak napisać funkcję (metodę), która oblicza i zwraca wiele (dowolnie wiele) wartości?

Można zwracać listę, ale to nie rozwiązuje problemu nieograniczonej liczby wyników. Dodatkowo jest czasochłonne.

Idealnym rozwiązaniem byłaby funkcja, która może wielokrotnie wykonać instrukcję `return`.

Przykład (życzeniowy)

```
def kwadraty( start ):
    n=start
    while True:
        return n * n
        n += 1
```

Generatory c.d.

Trzeba jakoś pamiętać stan funkcji i ją wznawiać. Można to emulować definiując odpowiednią klasę.

```
class Kwadraty:
    def __init__(self, start):
        self.n = start

    def next(self):
        res = self.n * self.n
        self.n += 1
        return res

k = Kwadraty(5)

for i in range(3):
    print k.next()
```

Generatory c.d.

Python ma magiczną instrukcję `yield`, która znacząco upraszcza sytuację.

```
def kwadraty(start):  
    n = start  
    while True:  
        yield n * n  
        n += 1
```

```
k = kwadraty(5)  # k jest instancja generatora
```

```
for i in range(3):  
    print k.next()
```

Generatory c.d.

Każda funkcja zawierająca instrukcję `yield` zwraca instancję generatora. Taka funkcja może mieć wyłącznie bezargumentowe instrukcje `return`, które kończą pracę generatora wyjątkiem `StopIteration`.

Generatory doskonale nadają się do implementowania iteratorów, ponieważ mają metody `__iter__()` i `next()`.

Generatory mają również metody `send(value)`, `throw(exception)` i `close()`, które służą do przesyłania danych do generatora i podnoszenia w nim wyjątków.

Coroutines

```
import random

def get_data():
    """Return 3 random integers between 0 and 9"""
    return random.sample(range(10), 3)

def consume():
    """Displays a running average across lists of integers sent to it"""
    running_sum = 0
    data_items_seen = 0

    while True:
        data = yield
        data_items_seen += len(data)
        running_sum += sum(data)
        print('The running average is {}'.format(running_sum / float(data_items_seen)))

def produce(consumer):
    """Produces a set of values and forwards them to the pre-defined consumer function"""
    while True:
        data = get_data()
        print('Produced {}'.format(data))
        consumer.send(data)
        yield

if __name__ == '__main__':
    consumer = consume()
    consumer.send(None)
    producer = produce(consumer)

    for _ in range(10):
        print('Producing ...')
        next(producer)
```