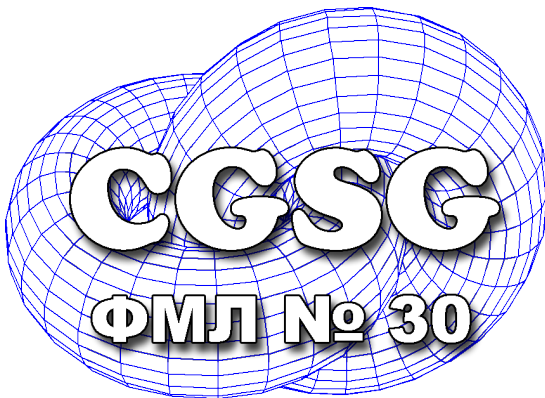


Язык программирования



Лекция 6. Исключения, итераторы



Владимир Руцкий
<altsysrq@gmail.com>



План лекции

- Исключения
- Оператор with
- Итераторы
- Генераторы
- lambda-функции

Типы ошибок

- Синтаксические ошибки - возникают при загрузке модуля

```
1 >>> while True print "test"
2     File "<stdin>", line 1
3         while True print "test"
4             ^
5 SyntaxError: invalid syntax
6 >>>
```

- Ошибки внутри Python или библиотек
 - «Программа совершила недопустимую операцию и будет закрыта»
- Ошибки времени выполнения - **исключительные ситуации**

```
1 >>> def div(a, b):
2     ...     return a / b
3     ...
4 >>> div(1, 0)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   File "<stdin>", line 2, in div
8 ZeroDivisionError: integer division or modulo by zero
9 >>>
```

Исключительные ситуации

- Популярный механизм обработки ошибок
- Деление на ноль, закончилась память, не удалось открыть файл и др. - исключительные ситуации
- Все исключения имеют **тип** и **объект-исключение**
 - обычно **тип** = `type(объект-исключение)`
- Для обработки используется конструкция
`try: ... except: ...`

Обработка исключительных ситуаций

try:

<блок кода>

except ТипИсключения1:

<блок обработки исключения1>

except ТипИсключения2:

<блок обработки исключения2>

- Если в <блоке кода> происходит необработанная исключительная ситуация, то:
 - <блок кода> прерывается
 - среди конструкций **except** последовательно ищется блок с подходящим типом исключения (имеющий тот же тип, что и **ТипИсключения** или родительский для него)
 - _ если найден, то он выполняется
 - _ если не найден, то исключение не обработано и произойдёт переход к внешнему обработчику исключений (если его нет, то программа завершится с ошибкой)

```
1 >>> while True:
2 ...     try:
3 ...         x = int(raw_input("Please enter a number: "))
4 ...         break
5 ...     except ValueError:
6 ...         print "Oops! That was no valid number. Try again..."
7 ...
```

Синтаксис

```
1 # -*- encoding: utf-8 -*-
2 import sys
3 try:
4     # <Блок кода>
5     f = open('myfile.txt')
6     s = f.readline()
7     v = float(s.strip())
8     r1 = 1 / int(v)
9     r2 = v**v
10
11 # <ТипИсключения> as <Объект-Исключение>
12 except IOError as exc:
13     errno, strerror = exc
14     print "I/O error({0}): {1}".format(errno, strerror)
15 except ValueError:
16     print "Could not convert data to an integer."
17 # Обработчик для нескольких исключений
18 except ZeroDivisionError, OverflowError:
19     print "Floating point operation exception."
20 # Обработчик для всех типов исключений
21 except:
22     # В sys можно найти всю информацию об исключении, включая
23     # стек вызовов.
24     print "Unexpected error:", sys.exc_info()[0]
25     raise # Оставляем текущее исключение необработанным --- произойдёт поиск
26           # обработчика в следующем вложенном try: ... except: ...
27 # Блок, который выполняется если не произошло никакого (непойманного в
28 # <блоке кода>) исключения.
29 else:
30     print "No exceptions!"
```

Иерархия исключений

- Обработчик с типом **Тип** подходит для исключения с типом **ТипИсключения**, если `isinstance(ТипИсключения, Тип)`
- Встроенные исключения образуют иерархию по типам

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

Порождение исключений

- *Вызвать (породить, бросить) исключение можно явно:*

raise some_object

- ИЛИ

raise SomeType, some_object

- ЧТО ЭКВИВАЛЕНТНО

raise SomeType.__class__, some_object

```
1 >>> try:
2 ...     # Бросаем исключение стандартного типа (можно любого своего).
3 ...     raise Exception('spam', 'eggs')
4 ... except Exception as inst:
5 ...     print type(inst)      # объект-исключение
6 ...     print inst.args      # Класс Exception для удобства сохраняет свои
7 ...                         # аргументы в .args
8 ...     print inst           # У Exception переопределён __str__()
9 ...     x, y = inst          # и адресация __getitem__()
10 ...    print 'x =', x
11 ...    print 'y =', y
12 ...
13 <type 'exceptions.Exception'>
14 ('spam', 'eggs')
15 ('spam', 'eggs')
16 x = spam
17 y = eggs
```


Собственные исключения

```
1 >>> class MyError(Exception):
2 ...     def __init__(self, value):
3 ...         self.value = value
4 ...     def __str__(self):
5 ...         return repr(self.value)
6 ...
7 >>> try:
8 ...     raise MyError(2*2)
9 ... except MyError as e:
10 ...     print 'My exception occurred, value:', e.value
11 ...
12 My exception occurred, value: 4
13 >>> raise MyError('oops!')
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in ?
16 __main__.MyError: 'oops!'
```

try: ... except: ... finally: ...

- Есть синтаксис для гарантированного выполнения кода

try:

<блок кода>

except:

...

finally:

<блок кода 2>

- Если произошло исключение в <блоке кода>, то после его обработки вызовется <блок кода 2>
- Иначе после выполнения <блока кода> вызовется <блок кода 2>

f = **open**("file.txt")

try:

...

finally:

f.close() # выполнится всегда после завершения блока **try**

Оператор **with** (кратко)

- **with** <выражение>:
 <блок кода>
- **with** <выражение> **as** <переменная>:
 <блок кода>
- Вычисляет: <менеджер> = <выражение>
- Вызывает:
 <результат> = <менеджер>.__enter__()
 - если есть **as**, то <переменная> = <результат>
- Выполняется <блок кода>
 - Если возникло исключение:
 вызывает <менеджер>.__exit__(ТипИскл., ОбъектИскл., стек)
 - Иначе:
 вызывает <менеджер>.__exit__(None, None, None)

Оператор **with** (кратко) 2

- Можно создать свои `scope-guard`'ы, гарантированно выполняющие некоторые операции после завершения блока кода
- Некоторые объекты поддерживают синтаксис **with** сразу:
 - # В конце **with** гарантированно вызовется
`f.close()`
with open("file.txt") as f:
 `f.read()`
- Есть синтаксис для вложенных **with**:
 - **with A() as a:**
 with B() as b:
 ...
• **with A() as a, B() as b:**
 ...

Итераторы

- **Итератор** последовательности - объект, имеющий операцию **next()**, возвращающую следующий элемент последовательности или порождающий исключение **StopIteration**, если последовательность иссякла

- Для получения итератора по объекту используется **iter()**

it = **iter**(obj)

- Реализация:

- Если obj содержит метод **__iter__()**, то

it = obj.**__iter__**()

- Если obj поддерживает obj[i], то создаётся объект-обёртка, который будет вызывать obj[i] начиная с i = 0

- (Ещё есть **iter(o, sentinel)**, см. документацию)

```
1 >>> a = [1, "2", "three"]
2 >>> it = iter(a)
3 >>> it.next()
4 1
5 >>> it.next()
6 '2'
7 >>> it.next()
8 'three'
9 >>> it.next()
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 StopIteration
13 >>>
```

Итераторы. Пример работы **for**

for val in obj:

 <блок>

- практически эквивалентно

it = **iter**(obj)

try:

while True:

 val = it.**next**()

 <блок>

except StopIteration:

pass

Реализация итератора

```
1 class Reverse(object):
2     """Итератор по последовательности в обратном порядке"""
3     def __init__(self, data):
4         self.data = data
5         self.index = len(data)
6     def __iter__(self):
7         # Вернём себя в качестве объекта-итератора
8         return self
9     def next(self):
10        if self.index == 0:
11            raise StopIteration
12        self.index = self.index - 1
13        return self.data[self.index]
```

Генераторы (кратко)

- **Генератор** - функция, использующая для возвращения одного значения оператор **yield**
- **def** reverse(data):
 for index in **range**(**len**(data)-1, -1, -1):
 yield data[index]
- it = reverse(data)
 - Создаст объект-генератор
- it.next()
 - Выполнит функцию reverse() до первого **yield**
 - Результат **yield** будет возвращён в качестве следующего элемента

lambda-функции

- Можно создать неименованную функцию с помощью **lambda**:

```
>>> f = lambda arg1, arg2: arg1**arg2
```

```
>>> f(2, 3)
```

```
8
```

- Это удобно для обработки списков:

- ```
>>> filter(lambda s: len(s) == 3, ['one', 'two', 'three'])
```

```
['one', 'two']
```

- фильтрует только те элементы, у которых длина 3

- ```
>>> map(lambda x: x**3, range(4))
```



```
[0, 1, 8, 27]
```