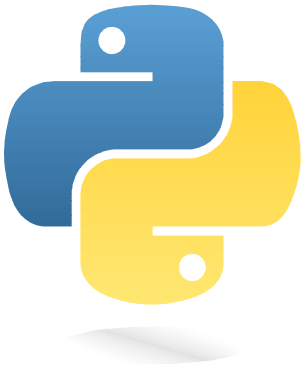
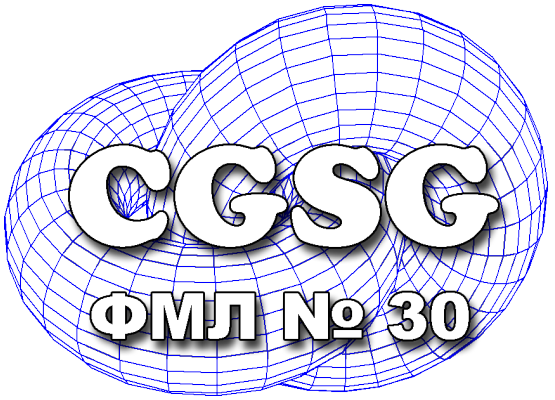
Язык программирования



python™

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

```
>>> while True print "test"
  File "<stdin>", line 1
    while True print "test"
                     ^
SyntaxError: invalid syntax
>>>
```

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

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

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)

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

Синтаксис

```
# -*- encoding: utf-8 -*-
import sys
try:
    # <Блок кода>
    f = open('myfile.txt')
    s = f.readline()
    v = float(s.strip())
    r1 = 1 / int(v)
    r2 = v**v

# <ТипИсключения> as <Объект-Исключение>
except IOError as exc:
    errno, strerror = exc
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
# Обработчик для нескольких исключений
except ZeroDivisionError, OverflowError:
    print "Floating point operation exception."
# Обработчик для всех типов исключений
except:
    # В sys можно найти всю информацию об исключении, включая
    # стек вызовов.
    print "Unexpected error:", sys.exc_info()[0]
    raise   # Оставляем текущее исключение необработанным --- произойдёт поиск
            # обработчика в следующем вложенном try: ... except: ...
# Блок, который выполняется если не произошло никакого (непойманного в
# <блоке кода>) исключения.
else:
    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 some_object.__class__, some_object

```
>>> try:
...     # Бросаем исключение стандартного типа (можно любого своего).
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # объект-исключение
...     print inst.args      # Класс Exception для удобства сохраняет свои
...                          # аргументы в .args
...     print inst           # У Exception переопределён __str__()
...     x, y = inst          # и адресация __getitem__()
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Собственные исключения

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__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)

```
>>> a = [1, "2", "three"]
>>> it = iter(a)
>>> it.next()
1
>>> it.next()
'2'
>>> it.next()
'three'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

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]