Обработка ошибок

Технологии и языки программирования

Юдинцев В. В.

Кафедра теоретической механики
Самарский университет

2 декабря 2016 г.

Синтаксические ошибки

```
1 for i in range(5)
2     print(i)
```

```
SyntaxError: invalid syntax
```

Ошибки времени выполнения

Арифметическая ошибка:

```
1 a = 1.0
2 for i in range(5):
3     print(a/i)
```

```
...
ZeroDivisionError: division by zero
```

Ошибки времени выполнения

Ошибка при работе со словарём:

```
1 a = {'one': 'один', 'two': 'два'}
2 print(a['three'])
```

```
...
KeyError: 'three'
```

Обработка ошибок

Для самостоятельной обработки ошибок внутри программы, возникающих во время выполнения, используются ключевые слова `try ... except`:

```
1 try:
2     a = 1.0
3     s = 0
4     for i in range(5):
5         s = s + a/i
6     print(s)
7 except:
8     print('Произошло деление на ноль')
```

Любая ошибка во время выполнения программы внутри блока `try` приведет к выполнению кода в блоке `except`.

Обработка ошибки определённого типа

После ключевого слова `except` можно указать тип ошибки

```
try:
    f = open("datafile.txt", "r")
    a = f.readline()
except IOError:
    print('Невозможно открыть или прочитать файл')
```

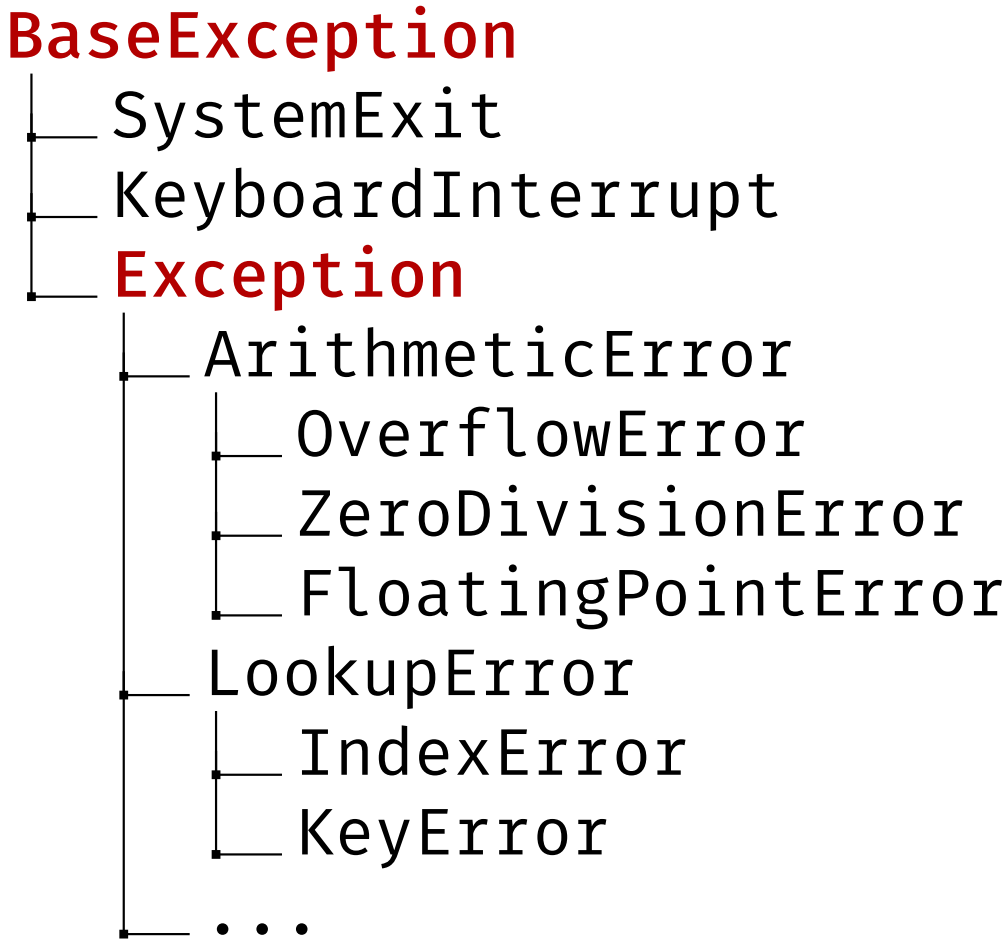
- Блок `except IOError:` выполнится только если произойдёт ошибка, связанная с вводом/выводом.
- Ошибки других типов будут обрабатываться объемлющим кодом.

Типы исключений при работе с файлами

Другие типы исключений для работы с файлами:

- `FileNotFoundError`
  Открываемый файл или каталог не существует
- `FileExistsError`
  Создаваемый файл или каталог уже существует
- `PermissionError`
  Доступ к файлу или каталогу при недостаточном уровне прав

Иерархия исключений



Все исключения кроме `SystemExit` и `KeyboardInterrupt` являются потомками базового класса `Exception`.

KeyError

Код в стиле EAFP: Easier to ask for forgiveness than permission

```
1 def print_dict_val(d, my_key):
2     try:
3         print(d[my_key])
4     except KeyError:
5         print('Ключ не найден:', my_key)
```

или в стиле LBYL: Look before you leap

```
1 def print_dict_val(d, my_key):
2     if my_key in d:
3         print(d[my_key])
4     else:
5         print('Ключ не найден:', my_key)
```

Дополнительная информация об ошибке

В блоке `except` можно указать имя переменной, которая будет иметь тип ошибки и содержать информацию об ошибке:

```
try:
    f = open("datafile.txt", "r")
    a = f.readline()
except IOError as err:
    print('Невозможно открыть или прочитать файл')
    print('Имя файла:', err.filename)
```

```
Невозможно открыть или прочитать файл
Имя datafile.txt
```

Несколько блоков except

Блок `try` может вызывать ошибки различных типов. Для каждого типа ошибки можно создать свой блок `except`, указав тип ошибки:

```
try:
    f = open("datafile.txt", "r")
    str_value = f.readline()
    a = int(str_value)
except FileNotFoundError as err:
    print("Невозможно открыть или прочитать файл")
except ValueError as err:
    print("Ошибка преобразования")
except:
    print("Неизвестная ошибка")
```

Блок except для нескольких исключений

```
try:
    f = open("datafile.txt", "r")
    str_value = f.readline()
    a = int(str_value)
except (FileNotFoundError, ValueError) as err:
    print("Ошибка загрузки данных из файла")
except:
    print("Неизвестная ошибка")
```

Конструкция try ... except ... else

В “защищаемом” участке кода делается попытка открыть файл для чтения. Если файл не существует, но генерируется исключение и управление передаётся блоку `except`, иначе выполняется блок `else`:

```
try:
    f = open("datafile.txt", "r")
except FileNotFoundError as err:
    print('Невозможно открыть или прочитать файл')
    print('Имя файла:', err.filename)
else:
    a = f.readline()
```

Переменная `f`, объявленная в блоке `try`, доступна и в блоке `else`.

Блок `finally`

После блоков `except` и `else` может быть определён блок `finally`, который выполняется в любом случае:

```
f = open("datafile.txt", "r")
try:
    str_value = f.readline()
    a = int(str_value)
except ValueError as err:
    print("Ошибка преобразования")
finally:
    f.close()
```

Файл закроется при любом исходе.

Ввод данных с клавиатуры

```
1 s = float(input('Введите основание треугольника'))
2 h = float(input('Введите высоту треугольника'))
3
4 print('Площадь треугольника равна {:1.2f}'.format(0.5*s*h))
```

При вводе не числовых значений программа сообщит об ошибке и остановится:

```
Введите основание треугольника: a
...
...
ValueError: could not convert string to float: 'a'
```

Это плохая реакция программы на ошибку: нет возможности исправить ошибку не перезапуская программу.

Контроль ввода данных

```
1  def input_as(text, type_of_value):
2   is_bad_input = True
3   while is_bad_input:
4    try:
5     val = input(text)
6     val = type_of_value(val)
7     is_bad_input = False
8    except ValueError as err:
9     print("Это не "+type_of_value.__name__+",
      попробуйте еще раз.")
10  return val
```

```
val = input_as("Введите целое число: ", int)
print("Введено значение", val)
```

Оператор `raise`

Если необходимо после обработки ошибки передать управление обработчику ошибок верхнего уровня, необходимо использовать оператор `raise`:

```
1 def divide(a, b):
2  try:
3   res = a/b
4  except:
5   print('divide: b=0!')
6   raise
7  return res
```

```
1 try:
2   a = float(input('a='))
3   b = float(input('b='))
4   divide(a, b):
5 except
6   print('Неверные
     исходные данные!')
```

```
a: 1
b: 2
divide: b=0!
Неверные исходные данные!
```

Оператор `raise`

При помощи оператора `raise` можно вызывать исключения любых типов:

```
try:
    msg = 'Введите число в диапазоне от {} до {}:'
    a, b = 1, 10
    x = int(input(msg.format(a,b)))
    if x<a or x>b :
        msg = 'Значение {} вне диапазона {}, {}'
        raise Exception(msg.format(x,a,b))
    print(x)
except Exception as err:
    print(err.args[0])
```

```
Введите число в диапазоне от 1 до 10: 16
Значение 16 вне диапазона 1, 10
```