PYTHON

Лекция 10

ПЛАН ЛЕКЦИИ

- Исключения
- Генераторы

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Исключение обрабатывается ближайшим обработчиком
- Ближайшим по вложенности из обработчиков внутри функции
- Или по вложенности вызовов
- Сначала пытаемся найти внутри функции, потом идем вверх по стеку

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Внутри одного `try/except` может быть много веток `except`
- Перебираются все по очереди
- Выбирается первая подошедшая
- Возможна ветка `else`
- И ветка `finally`

ОБРАБОТКА ИСКЛЮЧЕНИЯ

- Подходящая `except`-ветка - та, в которой указан класс, экземпляром которого является обрабатываемое исключение
- Класс, указанный в одной ветке, может быть подклассом того, что указан в другой
- Это имеет смысл только, если ветка с подклассом идет раньше
- Иначе она никогда не исполнится

ВСТРОЕННЫЕ ИСКЛЮЧЕНИЯ

- В корне иерархии - класс `BaseException`
- У него пять прямых потомков, основной - `Exception`
- Другие - `BaseExceptionGroup, GeneratorExit, KeyboardInterrupt, SystemExit`
- Они все - в чем-то "особенные"

EXCEPTION VS BASEEXCEPTION

- Если в `except` не указать исключения, то это ветка ловит `BaseException`
- Иногда это может приводить к странностям
- Например, когда в программной логике есть основной цикл
- И мы сильно не хотим, чтобы программа выпала из-за ошибки
- Массово подавлять исключения не очень правильно - но на уровне главного цикла это может быть оправдано

КАРКАС ПРИЛОЖЕНИЯ

```
 1 import sys
 2
 3 def read_request():
 4     ...
 5
 6 def write_response(resp):
 7    ...
 8
 9 def process(req):
10     return ...
11
12
13 # ..................
```

КАРКАС ПРИЛОЖЕНИЯ

```
 1 # ...................
 2
 3 def main_loop():
 4     while True:
 5         try:
 6             req = read_request()
 7             resp = process(req)
 8             write_response(resp)
 9         except:
10             print('got exception')
11             pass
12
13 main_loop()
```

ЧТО НЕ ТАК

- Мы нее выйдем по Ctrl-C
- Если внутри `process` или `read_request` вызвать `sys.exit` - мы не выйдем
- Про `BaseExceptionGroup`, `GeneratorExit` - разберем попозже
- Кто потомок `BaseException`, но не потомок `Exception` - те, кого скорее не хотелось бы массово обрабатывать наравне со всеми
- Надо хорошо думать над каждым `'except:'` или `'except BaseException:'`

ПОТОМКИ EXCEPTION

- `ArithmethicError`: переполнение, деление на `0` и т.п.
- `AssertionError`: нарушение `assert`-критерия
- `AttributeError`: обращение к несуществующему атрибуту
- `EOFError`: конец файла
- `ImportError`: проблемы с импортированием (если модуль не компилируется - это `SyntaxError`, отдельная ветка)

ПОТОМКИ EXCEPTION

- `LookupError`: проблемы с индексацией (`IndexError/KeyError`)
- `NameError`: обращение к несуществующей переменной
- `OSError`: все про диски, сети и т.п.
- `ValueError`: некорректный аргумент
- Полная картина: https://docs.python.org/3/library/exceptions.html#exception-hierarchy

ПРИМЕР

```
 1 try:
 2     print(''[0])
 3 except IndexError as exc:
 4     print('(1) got', exc)
 5 except LookupError as exc:
 6     print('(2) got', exc)
 7
 8 try:
 9     print(''[0])
10 except KeyError as exc:
11     print('(3) got', exc)
12 except LookupError as exc:
13     print('(4) got', exc)
14 # ..............................
```

ПРИМЕР

```
# ..............................
try:
    print(''[0])
except LoopupError as exc:
    print('(5) got', exc)
except KeyError as exc:
    print('(6) got', exc)

try:
    print(''[0])
except LookupError as exc:
    print('(7) got', exc)
except IndexError as exc:
    print('(8) got', exc)
```

ДИНАМИЧНОСТЬ TRY/EXCEPT

- Синтаксически проверяется структура предложения
- Между `except` и `as` должно идти выражение
- Если его нет - это ошибка синтаксиса
- А если есть оно будет вычислено по мере необходимости

ДИНАМИЧНОСТЬ TRY/EXCEPT

- Можно написать

```
except 5 as exc:
```

- И это не будет проблемой, если исключение не будет брошено
- Или будет, но обработается раньше
- Можно написать

```
except f() as exc:
```

- И если дело дойдет и f вернет класс-наследник `BaseException` - все будет `OK`

ДИНАМИЧНОСТЬ TRY/EXCEPT

- А если не ОК - то возникнет новое исключение
- И в этом момент будет брошено уже оно
- А оригинальное будет хранится среди его атрибутов
- Чтобы можно было восстановить цепочку подавленных исключений
- (Но это можно сделать не всегда)

ИЗВЛЕКАЕМ ПОДРОБНОСТИ

ИЗВЛЕКАЕМ ПОДРОБНОСТИ

ИЗВЛЕКАЕМ ПОДРОБНОСТИ

- Этим занимаются некоторые стандартные исключения
- Например, `OSError` хранит код ошибки и системное сообщение о ней
- А файловые исключения хранят имя файла
- Но `args` надо передать наверх

ИЗВЛЕКАЕМ ПОДРОБНОСТИ

ПРИМЕР

```
 1 def char_at(s, ind):
 2     return s[ind]
 3
 4 def exc_details(exc):
 5     print('got:', exc)
 6     print('type:', type(exc))
 7     print('args type:', type(exc.args))
 8     print('args:', exc.args)
 9     print('context:', exc.__context__)
10
11 # ..........................
```

ПРИМЕР

```
# ..........................
try:
    print(char_at('hello', 10))
except Exception as exc:
    exc_details(exc)

print()

try:
    open('strange-file-name')
except Exception as exc:
    exc_details(exc)
    print(exc.errno)
    print(exc.strerror)
```

ПРИМЕР

```
 1 def char_at(s, ind):
 2     return s[ind]
 3
 4 def exc_details(exc):
 5     print('got:', exc)
 6     print('type:', type(exc))
 7     print('args type:', type(exc.args))
 8     print('args:', exc.args)
 9     print('context:', exc.__context__)
10     if exc.__context__ is not None:
11         print('context type:', type(exc.__context__))
12         print('context args:', exc.args)
13
14 # ......................
```

ПРИМЕР

```
# ......................

    if (isinstance(exc, OSError)):
        try:
            print(exc.errno)
            print(exc.strrerror)
        except Exception as exc:
            exc_details(exc)

# ......................
```

ПРИМЕР

```
 1 # ......................
 2
 3 try:
 4     print(char_at('hello', 10))
 5 except Exception as exc:
 6     exc_details(exc)
 7
 8 print()
 9
10 try:
11     open('strange-file-name')
12 except Exception as exc:
13     exc_details(exc)
```

СВОИ КЛАССЫ ИСКЛЮЧЕНИЙ

- Это хорошая практика
- И лучше бросать высокоуровневые исключения
- Соответствующие уровню абстракции кода
- Если они бросаются как результат обработки более низкоуровневых исключений - те могут быть включены в контекст

СВОИ КЛАССЫ ИСКЛЮЧЕНИЙ

- Например, мы скачали какие-то данные и их обрабатываем
- Они хранятся в каталоге
- И в нем хранится файл `metadata.json`
- Вы на это рассчитываете и читаете этот файл
- А в реальности файла нет и бросается `FileNotFoundError`

СВОИ КЛАССЫ ИСКЛЮЧЕНИЙ

- Если есть класс, абстрагирующий схему хранения данных
- То лучше, чтобы бросаемые его методами исключения соответствовали его уровню абстракции
- Не `FileNotFoundError`, а что-то типа `BrokenDataError`

СВОИ КЛАССЫ ИСКЛЮЧЕНИЙ

- Возможно, нужен суперкласс всех ошибок вашего класса
- И конкретные подклассы

Ветка `else`

- Ветка `else` может идти после всех `except`
- Исполняется только если исключение не было брошено
- Это НЕ то же самое, что поместить ее код в конец `try`-блока
- Исключение, брошенное в `else`-блоке, не будет обработано тем же `try/except`

ПРИМЕР

```
 1 def f(s):
 2     return s[0]
 3
 4 try:
 5     f('a')
 6 except KeyError:
 7     pass
 8 else:
 9     print('OK')
10
11 # ............................
```

ПРИМЕР

```
1 # ............................
2
3 try:
4     f('')
5 except KeyError:
6     pass
7 else:
8     print('OK')
```

БЛОК FINALLY

- Может идти в конце
- Исполняется всегда в конце, независимо ни от чего
- Брошено исключение или нет, обработано или нет
- Даже если брошено исключение в ходе поиска except-блока

ПРИМЕР

```
 1 def f(s):
 2     return s[0]
 3
 4 try:
 5     f('a')
 6 except KeyError:
 7     pass
 8 else:
 9     print('OK')
10 finally:
11     print('done')
12
13 # ....................
```

ПРИМЕР

```
# ....................

try:
    f('')
except KeyError:
    pass
else:
    print('OK')
finally:
    print('done')
```

БЛОК FINALLY

- Если в `finally` бросается исключение, то начинается стандартная обработка
- Если необработанных нет - то его контекст будет `None`
- Если есть, то текущее запишется в контекст
- И дальше летит новое исключение

БЛОК FINALLY

- `return` в `finally`-блоке не запрещен
- И может приводить к неочевидному поведению
- Если было необработанное исключение, оно бесследно исчезнет
- Если был `return` в основном блоке или в `except` - `return` в `finally` будет "главнее"

ПРИМЕР

```
 1 def f(s):
 2     return s[0]
 3 def f2(s):
 4     try:
 5         f(s)
 6         return 0
 7     except KeyError:
 8         return 1
 9     finally:
10         print('done')
11         return 2
12
13 print(f2('a'))
14 print(f2(''))
```

ПРИМЕР

```
1 def f(v)
2     try:
3         return v
4     finally:
5         return 0
6
7
8 print(f(10))
9 print(f('hello'))
```

ГЕНЕРАТОРЫ

- Есть ключевое слово `yield`
- Его наличие в функции полностью меняет смысл происходящего
- Вызов такой функции начинает работать как конструктор специального объекта
- Код такой функции в момент вызова даже не исполняется

ПРИМЕР

```
 1 def f():
 2     print('hello')
 3     yield 123
 4
 5 v1 = f()
 6 print(v1)
 7 print(type(v1))
 8
 9 v2 = f()
10 print(v2)
11 print(type(v2))
```

МОДЕЛЬ ИСПОЛНЕНИЯ

- Вызов метода `__next__` над любым из полученных объектов запускает исполнение
- Код исполняется до `yield` или до выхода
- Если дошли до `yield`, то исполнение приостанавливается
- Выражение при `yield` вычисляется и передается в точку вызова `__next__` как его результат

ПРИМЕР

```
1 def f():
2     print('hello')
3     yield 123
4
5 v1 = f()
6 print(v1.__next())
7
8 v2 = f()
9 print(v2.__next__())
```

МОДЕЛЬ ИСПОЛНЕНИЯ

- Если дошли до конца - бросается исключение `StopInteration`
- Вместо `__next__` можно использовать встроенную функцию `next`
- Значение, передаваемое через `yield`, не обязано быть константой
- Оно может зависеть от параметров, от предыдущих действий

ПРИМЕР

```
 1 def fibo():
 2     prev, curr = 0, 1
 3     while True:
 4         yield prev
 5         prev, curr = curr, prev + curr
 6
 7 v1 = fibo()
 8 v2 = fibo()
 9 for _ in range(10):
10     print(next(v1))
11     print(next(v2))
12     print(next(v1))
```

ПРИМЕР

```
 1 def recur(prev=0, curr=1):
 2     while True:
 3         yield prev
 4         prev, curr = curr, prev + curr
 5
 6 v1 = recur(2, 3)
 7 v2 = fibo(-1, 2)
 8 for _ in range(10):
 9     print(next(v1))
10     print(next(v2))
11     print(next(v1))
```

ОБЩАЯ СХЕМА ИТЕРИРОВАНИЯ

- Вызовы `next` создают эффект итерирования
- Это в чем-то похоже на итерирование по списку или множеству
- Но есть важное отличие - для итерирования по списку или множеству нужно состояние
- Которое не является частью объекта `list`

ПРИМЕР

```
 1 def iter_list(data):
 2     index = 0
 3     while index < len(data): # специально без for
 4         yield data[index]
 5         index += 1
 6
 7 v = iter_list([1, 56, 10])
 8 print(next(v))
 9 print(next(v))
10 print(next(v))
11
12 # ..............................
```

ПРИМЕР

```
1 # ..............................
2 try:
3     next(v)
4 except StopIteration:
5     print('ok')
```

ТЕРМИНОЛОГИЯ

- Объект, получающийся при вызове функции с `yield` - генератор
- Объект, реализующий метод `__next__` - итератор
- Объект, реализующий метод `__iter__` - итерируемый (`iterable`)
- Подразумевается, что `__iter__` возвращает итератор

ФАКТЫ

- Генератор является итератором
- Генератор является итерируемым
- `__iter__` генератора возвращает ссылку на него самого
- Не всякий итерируемый является итератором

ФАКТЫ

- Список НЕ является итератором
- Список является итерируемым
- `__iter__` списка возвращает ссылку объекту служебного класса-итератора
- К методу `data.__iter__` можно обратиться через `iter(data)`

ПРИМЕР

```
1 v = iter([1, 56, 10])
2 print(next(v))
3 print(next(v))
4 print(next(v))
5
6 try:
7     next(v)
8 except StopIteration:
9     print('ok')
```

ЦИКЛ FOR

- Справа от `in` - итерируемый объект
- В частности - может быть генератор
- Можно создать wrapper-класс для списка
- В нем определить `__iter__`, возвращающий итератор с обходом в другом порядке
- Или сделать это через генератор

ПРИМЕР

```
1 def filter(data, f):
2     for v in data:
3         if f(v):
4             yield(v)
5
6 data = filter([1, 56, 10], lambda v: v % 2 == 0)
7 for v in data:
8     print(v)
```

ПРИМЕР

```
 1 def zip_with_next(data):
 2     it = iter(data)
 3     prev = next(it)
 4     for v in it:
 5         yield((prev, v))
 6         prev = v
 7
 8 data = zip_with_next([1, 56, 10, 12, 21])
 9 for v in data:
10     print(v)
```

ВСТРОЕННЫЕ ГЕНЕРАТОРЫ

- Много встроенных функций: `map, filter, enumerate, zip`
- Отдельный пакет: `itertools`
- `reduce` - в `functoools`
- `fold` - из коробки нет (?!)