Создание собственных исключений. Способы обработки ошибок в программах

План урока

Аннотация

Урок продолжает тему работы с исключениями: полный синтаксис конструкции try...except, построение собственных классов исключений и их наследование, методики LBYL и EAFP.

1. Методики LBYL и EAFP

У работы с исключениями есть преимущества перед кодами возврата. В современных языках программирования используются именно объекты-исключения. Поговорим подробнее про механизмы работы с ними.

Когда мы изучаем исключения, то с их помощью хочется обрабатывать вообще все внештатные ситуации, максимально очищая код от дополнительных условий-проверок.

Есть два крайних подхода: **LBYL** (Look Before You Leap — *Посмотри перед прыжком*) и **EAFP** (Easier to Ask Forgiveness than Permission — *Проще извиниться, чем спрашивать разрешение*).

Например, при работе со словарями, когда доступ по ключу, а ключа нет, генерируется стандартное исключение **KeyError**:

```
mydict = {4: 34}
mydict[4354]

    ---------------------------------------------------------------

    KeyError                         Traceback (most recent call last)

    <ipython-input-3-d2c5c3da9fa7> in <module>()
          1 mydict = {4: 34}
    ----> 2 mydict[4354]


    KeyError: 4354
```

С одной стороны, можно перестраховываться, заранее проверяя, что всё получится. Это идеология **LBYL**-подхода. Сначала посмотрели, убедились, что всё в порядке, только потом сделали. Как при переходе улицы: поглядели на светофор, потом по сторонам. Если горит зелёный свет и нет препятствий, то можно переходить.

```
mydict = {'Elizabeth': 12, 'Ivan': 145}
if 'Ivan' in mydict:
    mydict['Ivan'] += 1
```

С другой стороны, мы можем описывать только главный алгоритм, рассчитывая, что всё будет хорошо. Но при таком подходе необходимо прописать действия с исключениями (иногда и разных типов). Это суть подхода **EAFP**.

```
try:
    mydict['Ivan'] += 1
except KeyError:
    pass
```

В Python преобладает **EAFP**-подход, особенно если речь идёт о стандартных исключениях и действиях с данными внутри них. Но это не значит, что методику **LBYL** вообще нельзя использовать. Всегда нужно рассматривать конкретный случай. Иногда есть и третий вариант. Например, в нашем случае можно было воспользоваться словарём, содержащим для всех ключей значение по умолчанию (в примере — 0):

```
from collections import defaultdict
s = defaultdict(lambda: 0)
s[34]

  ------------------------------------------------------------

 0
```

```
s[12] += 11
s[12]

  ------------------------------------------------------------

11
```

В коде многопоточной программы лучше использовать **EAFP**. Если процессов несколько, один из них может неожиданно изменить данные, которые только что проверил и собирается использовать другой. Но сейчас мы не будем на этом останавливаться.

2. Полный синтаксис блока try...except

На предыдущем уроке мы рассмотрели базовый синтаксис блока исключений, но вообще try...except может выглядеть существенно сложно:

```
s = [(1, 2), (4, 7), (1, 0), (13, None)]

for i in range(10):
    try:
        x, y = s[i]
        print(x / y)
    except IndexError:
```

```
        print('Мы за границей списка')
    except ZeroDivisionError as e:
        print('Поделили на 0')
    except Exception as e:
        print('Непредвиденная ошибка %s' % e)
    finally:
        print('Идём дальше')

------------------------------------------------------------

    0.5
    Идём дальше
    0.5714285714285714
    Идём дальше
    Поделили на 0
    Идём дальше
    Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'
    Идём дальше
    Мы за границей списка
    Идём дальше
    Мы за границей списка
    Идём дальше
    Мы за границей списка
    Идём дальше
    Мы за границей списка
    Идём дальше
    Мы за границей списка
    Идём дальше
    Мы за границей списка
    Идём дальше
```

К одному **try** может быть прикреплено несколько **except**. Блок **finally** выполняется независимо от того, создано ли исключение и какого оно типа. Даже если программа прервётся внешним исключением, переходом по break или continue, блок **finally** будет выполнен.

Порядок перечисления исключений важен. Перебор закончится на первом же подходящем по условию блоке:

```
s = [(1, 2), (4, 7), (1, 0), (13, None)]

for i in range(10):
    try:
        x, y = s[i]
        print(x / y)
    except Exception as e:
```

```
        print('Непредвиденная ошибка %s' % e)
    except IndexError:
        print('Мы за границей списка')
    except ZeroDivisionError as e:
        print('Поделили на 0')
    finally:
        print('Идём дальше')

-------------------------------------------------------------

    0.5
    Идём дальше
    0.5714285714285714
    Идём дальше
    Непредвиденная ошибка division by zero
    Идём дальше
    Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
    Непредвиденная ошибка list index out of range
    Идём дальше
```

Все ошибки получились **непредвиденными**, потому что **Exception** — класс-родитель для большинства встроенных исключений. Все они могут быть приведены к типу **Exception** и провалились в первый же блок except.

В блоке try...except можно применять конструкцию **else**. Этот блок выполняется, если ни один из блоков except не подошёл:

```
s = [(1, 2), (4, 7), (1, 0), (13, None)]

for i in range(10):
    try:
        x, y = s[i]
        print(x / y)
    except IndexError:
```

```
        print('Мы за границей списка')
    except ZeroDivisionError as e:
        print('Поделили на 0')
    except Exception as e:
        print('Непредвиденная ошибка %s' % e)
    else:
        print('Всё хорошо')
    finally:
        print('Идём дальше')

-------------------------------------------------------------

0.5
Всё хорошо
Идём дальше
0.5714285714285714
Всё хорошо
Идём дальше
Поделили на 0
Идём дальше
Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'
Идём дальше
Мы за границей списка
Идём дальше
Мы за границей списка
Идём дальше
Мы за границей списка
Идём дальше
Мы за границей списка
Идём дальше
Мы за границей списка
Идём дальше
Мы за границей списка
Идём дальше
```

Перечислим ещё несколько особенностей работы с исключениями.

— Сам блок **except** может быть источником исключений;

— Так как обработчик исключения и код, который его вызвал, могут находиться на разных уровнях стека, код может **разорваться**. В такой ситуации сложно уследить за общей логикой работы программы;

— Возможно вы захотите сделать что-то такое:

```
try:
    someting()
except Exception:
    pass
```

Если вы не логируете (в файл, базу данных) ошибки в этом случае и не регистрируете сам их факт, программу тяжело отлаживать: она может выдавать неверные результаты, но неизменно отчитываться, что всё в порядке.

3. Создание пользовательских типов ошибок

Исключение — это объект. Мы можем дополнять дерево исключений собственными так же, как делаем это с любыми другими классами и объектами.

Если мы пишем большой модуль, нам почти всегда требуется собственная иерархия объектов-исключений. Обычно методы и свойства для собственных объектов не переопределяются и не дополняются. То, что нужно — прозрачные названия ошибок и корректная работа блока try...except с нужными классами — обеспечивается простым наследованием.

Как правило, новые объекты наследуют классу **Exception**.

Рассмотрим пример из прошлого урока, в который мы добавили несколько собственных исключений:

```
class CardError(Exception):
    pass


class CardFormatError(CardError):
    pass


class CardLuhnError(CardError):
    pass


def get_card_number():
    card_number = input("Введите номер карты (16 цифр): ")
    card_number = "".join(card_number.strip().split())
    if not (card_number.isdigit() and len(card_number) == 16):
        raise CardFormatError("Неверный формат номера")
    return card_number
```

```
def double(x):
    res = x * 2
    if res > 9:
        res = res - 9
    return res


def luhn_algorithm(card):
    print(card)
    odd = map(lambda x: double(int(x)), card[::2])
    even = map(int, card[1::2])

    if (sum(odd) + sum(even)) % 10 == 0:
        return True
    else:
        raise CardLuhnError("Недействительный номер карты")


def process():
    while True:
        try:
            number = get_card_number()
            if luhn_algorithm(number):
                print("Ваша карта обрабатывается...")
                break
        except CardError as e:
            print("Ошибка! %s" % e)


process()

------------------------------------------------------------

    Введите номер карты (16 цифр): вапвап
    Ошибка! Неверный формат номера
    Введите номер карты (16 цифр): 8768768768768768
    8768768768768768
    Ошибка! Недействительный номер карты
```

Родительское исключение CardError позволяет нам перехватывать все нештатные ситуации, связанные с номером карты, а не только те, для которых есть специальные обработчики.

4. Перехват системных исключений

В иерархии классов-исключений есть исключения **SystemExit** и **KeyboardInterrupt**.

Их можно использовать. Например, нажатие комбинации Ctrl+C обычно прекращает работу программы и генерирует исключение **прерывание от клавиатуры**.

Обработаем этот случай:

```
a = 0

while True:
    try:
        a += 1
    except KeyboardInterrupt:
        res = input('Действительно остановить? ')
        if res == 'yes':
            break
```

В этом примере после «волшебной» комбинации Ctrl+C мы попадем в блок except и зададим пользователю вопрос о прерывании работы программы.

5. Конструкция assert

Конструкция **assert** — это часть Python, связанная с тестированием. Тестирование мы ещё не проходили, но это не помешает понять, как же **assert** работает.

В любое место программы вы можете вставить блок такого вида:

```
assert логическое выражение
```

Например:

```
s = [1, 2, 34, 54, 3]
assert len(s) == 4
```

Когда мы попытаемся выполнить приведенный код, то получим:

```
---------------------------------------------------------------

AssertionError                Traceback (most recent call last)

<ipython-input-1-c1fdb0a01058> in <module>()
      1 s = [1, 2, 34, 54, 3]
----> 2 assert len(s) == 4


AssertionError:
```

Блок работает так: если выражение верное, ничего не происходит. Если нет — создаётся исключение **AssertionError**.

Можно снабдить программу разными вспомогательными проверками. Они помогут контролировать правильность исполнения. Если какое-то условие не будет выполнено, то программа аварийно остановится. Это помогает тестировать и отлаживать ПО.

Интерпретатор Python можно запустить с опцией -O — тогда он будет пропускать блоки **assert.**

```
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
```

Это позволяет включать и выключать отладочные проверки.

На этом мы закончим тему **Введение в исключения**.

© 2018 – 2019 ООО «Яндекс»