Понятие исключения. Обработка исключений

План урока

Аннотация

Урок рассказывает о работе с исключениями в современных языках программирования и, в частности, в Python. Сравнение методики исключений с методикой кодов возврата.

1. Что такое ошибка или исключение

У вашей программы есть «главная» ветка — то, что относится к основному алгоритму. Именно ей вы уделяете основное внимание.

Во время работы программы иногда возникают непредвиденные обстоятельства («упс, что-то пошло не так») — как правило, внешние. Программа должна их верно обрабатывать. Мы будем называть их **ошибками** или **исключениями**.

Чтобы было понятнее, рассмотрим пример алгоритма посещения магазина.

Родители посылают дочь в магазин и просят купить молока определённой марки ценой 40 рублей. Это и есть для девочки основной алгоритм в данной ситуации. Однако, могут возникнуть непредвиденные обстоятельства: именно этот магазин сейчас закрыт (нужно ли тратить время и идти в следующий?), такого молока в продаже нет (вернуться обратно или купить другое?), молоко подорожало (всё равно брать?), заканчивается срок годности (это важно?) и так далее.

Всё это можно обсудить сразу, но нужно соблюдать меру и не раздувать алгоритм покупок до инструкции на 30 страниц, включающей подробные указания на случай стихийных бедствий.

Лучше всего решить, что делать в тех случаях, которые легко предугадать. Всё остальное свести к универсальному решению «вернуться обратно» или «позвонить родителям и спросить».

То же самое и в программировании.

Например, вы написали алгоритм, скачивающий плейлист из **ВКонтакте** в папку на компьютере. Запустили его. Что же может пойти не так?

- Проблемы с доступом в Интернет;
- Сервера ВКонтакте работают нестабильно и не отвечают;
- Если алгоритм даёт имена файлам по названиям треков, то ему могут попасться недопустимые символы;
- Закончилось свободное место на диске;
- ...

Есть несколько путей:

1. В любом непредвиденном случае программа остановится, а вы увидите сообщение об ошибке (так Python работает по умолчанию).
2. Вы добавляете автоматическую обработку некоторых известных вам проблем.
3. Программа будет каждый раз спрашивать вас о том, как поступить в возникшей нештатной ситуации.
4. Комбинация этих решений.

Если мы хотим, чтобы программа работала с широким диапазоном входных данных и внешних условий, то надо учитывать исключения.

Отметим ещё раз, что по умолчанию от необработанной ошибки программа на Python немедленно останавливается и выводит сообщение:

```python
for i in range(10):
    print(10 / i)
```

```
---------------------------------------------------------------
ZeroDivisionError            Traceback (most recent call last)
      1 for i in range(10):
----> 2     print(10 / i)

ZeroDivisionError: division by zero
```

2. Стек вызовов в Python

Теперь поговорим немного про стек вызовов (traceback). Программа на Python состоит из блоков, входящих один в другой, поэтому у любой точки программы есть вложенность. Давайте посмотрим, какую информацию даёт стек вызовов, когда указывает на место ошибки.

Например, вычисление частного двух чисел, введённых пользователем. Отметим, что огромное количество ошибок происходит из-за того, что пользователь ввёл неправильные данные.

Проведём 3 эксперимента:

1. Введём правильные числа
2. Введём второе число, равное 0
3. Введём не число, а строку

```python
def div(a, b):
    return a / b


print(div(4, 10))
```

```
0.4
```

```python
def test_division():
    a = float(input("Введите a: "))
```

```
    b = float(input("Введите b: "))
    print(div(a, b))
```

Запустим программу и посмотрим на результат:

```
Введите a: 45
Введите b: 11
4.090909090909091
```

Снова выполним программу и проанализируем результат:

```
----> 1 test_division()
Введите a: 45
Введите b: 0

      2     a = float(input("Введите a: "))
      3     b = float(input("Введите b: "))
----> 4     print(div(a, b))
      5
      6 test_division()

      1 def div(a, b):
----> 2     return a / b
      3
      4

ZeroDivisionError: float division by zero
```

И ещё раз:

```
---------------------------------------------------------------
ValueError                    Traceback (most recent call last)
----> 1 test_division()
Введите a: fd

      1 def test_division():
----> 2     a = float(input("Введите a: "))
      3     b = float(input("Введите b: "))
      4     print(div(a, b))
      5

ValueError: could not convert string to float: 'fd'
```

Печать стека вызовов в момент аварийного завершения программы помогает нам найти ошибку. Мы слишком понадеялись на то, что пользователь введёт только вещественные числа в европейском формате (с точкой в качестве десятичного разделителя). Кроме того, не все помнят, что на 0 делить нельзя (по крайней мере, в Python).

3. Работа с кодами возврата

Этот тип работы с исключениями первым появился в истории программирования. Его следы остались в некоторых функциях Python — языка, который унаследовал механику у C.

Например, метод find строки ищет позицию вхождения подстроки в заданную строчку.
Он возвращает либо номер позиции, либо −1, если такой подстроки нет.

```
s = "Привет, мир!"
print(s.find(","))
print(s.find("Д"))
```

Программа выведет:

```
6
-1
```

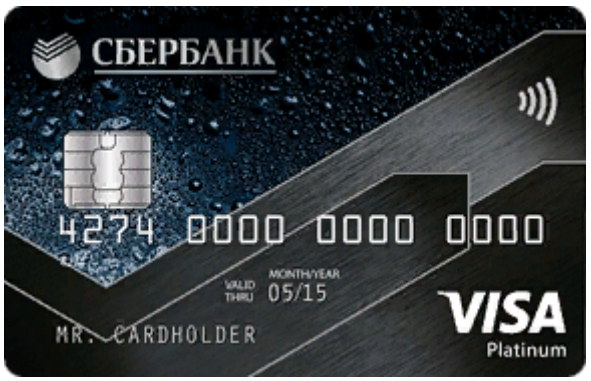
Из-за того, что в исходной строке не было символа «Д», нам было возвращено значение −1. Это и есть **код возврата**. Так как любая функция в Python возвращает значения, мы можем использовать их для кодирования информации об ошибке.

Для каждой ошибки можно придумать свой код возврата. Коды не должны совпадать с возможными обычными ответами.

В больших информационных системах у каждой ситуации, в том числе и у нештатной, есть свой номер. Например, у ошибок в интернете: 404, 503. А 200 — это код, возвращаемый серверами при успешном выполнении задания.

Давайте рассмотрим пример использования кодов возврата.

У нас есть интернет-магазин с оплатой картами. Просим ввести пользователя номер карты.



Программа для этого может быть очень простой:

```
card_number = input("Введите номер карты: ")
```

Но здесь нас подстерегает несколько неожиданностей. Пользователь не знает формата, в котором нужен номер карты. Просто цифры без пробела? Или группами по четыре с пробелами, как на карте?

Пишем ему об этом:

```
card_number = input("Введите номер карты (16 цифр без пробелов): ")
```

Но и теперь мы не застрахованы от ошибок — пользователь может набрать букву О вместо нуля, перепутать цифры, не прочитать внимательно инструкцию и ввести пробелы или просто начать целенаправленно взламывать нашу программу, набирая заведомо некорректные данные (это называется фаззинг (fuzzing).

Ситуацию усложняет то, что номера карт не просто случайный набор из 16 цифр.

Первая цифра обозначает тип карты. Например, номера карт Visa начинаются с 4, а MasterCard — с 5. Следующие пять обозначают банк, который выпустил карту. Другие девять цифр — уникальный номер конкретной карты. Последнюю, шестнадцатую цифру, называют контрольной.

Номер должен проходить проверку специальным алгоритмом Лу́на. Его придумал немецкий инженер Ганс Питер Лун.

Чтобы проверить, нужно взять номер карты и вычислить для него специальное число — контрольную сумму. Вот как это делают:

1. Каждую цифру в нечётной позиции, начиная с первого числа слева, умножаем на два. Если результат больше 9, складываем обе цифры этого двузначного числа. Или вычитаем из него 9 и получаем тот же результат. Например, если у нас 18, при сложении 1 + 8 получится 9, при вычитании 18 — 9 — тоже 9.
2. Затем мы складываем все результаты и цифры на чётных позициях — в том числе и последнюю контрольную цифру.

3. Если сумма кратна 10, то номер карты правильный. Именно последняя контрольная цифра делает общую сумму кратной 10.

Когда банки выпускают новыю карты и генерируют номера для них, контрольную шестнадцатую цифру они подбирают так, чтобы алгоритм Лу́на давал кратное десяти число. Поэтому у всех банковских карт в мире — номера с такой контрольной суммой.
Когда пользователь вводит номер своей карты, мы применяем алгоритм Луна. Если получится число, не кратное 10, — значит, пользователь ошибся.

Допустим, мы получаем номер карты от пользователя, а потом эта карта поступает в обработку:

```
def get_card_number():
    card_number = input("Введите номер карты (16 цифр): ")
    return card_number


def process():
    number = get_card_number()
    print("Ваша карта обрабатывается...")


process()
```

На этом этапе мы никак не контролируем пользователя. Проверяем номер карты по алгоритму Луна:

```
def get_card_number():
    card_number = input("Введите номер карты (16 цифр): ")
    return card_number


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


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

    return (sum(odd) + sum(even)) % 10 == 0


def process():
```

```
    number = get_card_number()
    if luhn_algorithm(number):
        print("Ваша карта обрабатывается...")


process()
```

Что мы видим, когда запускаем программу:

```
Введите номер карты (16 цифр): 4728795357233848
Ваша карта обрабатывается...
```

Функция **luhn_algorithm** проверяет данные. Карта засчитается, только если данные корректны. Можно рассматривать эту функцию как функцию с кодом возврата. Она говорит там, корректен ли номер карты.

Если же номер неправильный:

```
def get_card_number():
    card_number = input("Введите номер карты (16 цифр): ")
    return card_number


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


def process():
    while True:
        number = get_card_number()
        if luhn_algorithm(number):
            print("Ваша карта обрабатывается...")
            break
        else:
            print("Номер недействителен. Попробуйте снова.")


process()
```

Теперь диалог будет выглядеть вот так:

```
Введите номер карты (16 цифр): 4332432343434341
Номер недействителен. Попробуйте снова.
Введите номер карты (16 цифр): 3445212323123232
Номер недействителен. Попробуйте снова.
Введите номер карты (16 цифр): 4728795357233848
Ваша карта обрабатывается...
```

Однако, если пользователь введёт не 16 цифр, а что-нибудь другое, или 16 цифр, разделенных пробелами, то он обрушит программу:

```
Введите номер карты (16 цифр): asdasdasd
---------------------------------------------------------------
ValueError                         Traceback (most recent call last)
----> 1 process()

<ipython-input-13-f73e5219d5f3> in process()
      2     while True:
      3         number = get_card_number()
----> 4         if luhn_algorithm(number):
      5             print("Ваша карта обрабатывается...")
      6             break

<ipython-input-12-1a0f185e782e> in luhn_algorithm(card)
     15     even = map(int, card[1::2])
     16
---> 17     return (sum(odd) + sum(even)) % 10 == 0
     18
     19

<ipython-input-12-1a0f185e782e> in <lambda>(x)
     12
     13 def luhn_algorithm(card):
---> 14     odd = map(lambda x: double(int(x)), card[::2])
     15     even = map(int, card[1::2])
     16

ValueError: invalid literal for int() with base 10: 'a'
```

Сделаем так, чтобы в ответ на некорректный запрос программа не «падала», а требовала ввести 16 цифр, произвольно разделённых пробелами.

Для этого функция **get_card_number** теперь будет возвращать специальный код — например, 404, как в интернете.

```
def get_card_number():
    card_number = input("Введите номер карты (16 цифр): ")

    card_number = "".join(card_number.strip().split())
    if card_number.isdigit() and len(card_number) == 16:
        return card_number
    else:
        return 404


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


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

    return (sum(odd) + sum(even)) % 10 == 0


def process():
    while True:
        number = get_card_number()
        if number == 404:
            print("Введите только 16 цифр. Допускаются пробелы")
            continue
        if luhn_algorithm(number):
            print("Ваша карта обрабатывается...")
            break
        else:
            print("Номер недействителен. Попробуйте снова.")

process()
```

Возможный диалог с программой:

```
Введите номер карты (16 цифр): 34543
Введите только 16 цифр. Допускаются пробелы
Введите номер карты (16 цифр): fr
Введите только 16 цифр. Допускаются пробелы
Введите номер карты (16 цифр): sd sdfsdf
```

```
Введите только 16 цифр. Допускаются пробелы
Введите номер карты (16 цифр): 4728 7952 5723 3848
Номер недействителен. Попробуйте снова.
Введите номер карты (16 цифр): 4728 7953 5723 3848
Ваша карта обрабатывается...
```

4. Трудности работы с кодами возврата

Даже простая функция для пользовательских данных обрастает дополнительным кодом, проверкой многих условий и «магическими» кодами возврата. Если функция с кодом возврата находится глубоко в стеке вызовов, то придётся сделать так, чтобы её правильно обрабатывала вся вышестоящая цепочка функций. Каждая из них должна принимать код и возвращать свой.

Чтобы упростить обработку ошибок, программисты стали работать с исключениями как с объектами.

5. Обработка исключений

В Python и других объектно-ориентированных языках **исключения** — такие же объекты в программе, как и всё остальное. Исключение создаётся в любом месте и поднимается по стеку вызовов, пока его не отловит какой-нибудь код-обработчик.

Сейчас поймаем исключения:

```
s = "3434"
s.index("9")
```

```
---------------------------------------------------------------
ValueError                       Traceback (most recent call last)
<ipython-input-19-d0c0f2962a8b> in <module>()
      1 s = "3434"
----> 2 s.index("9")

ValueError: substring not found
```

Такое сообщение об ошибке означает, что метод **index** породил исключение — объект типа **ValueError**. Все функции стека вызовов получили уведомление о нештатной ситуации. Если ни одна из них не отреагирует, программа аварийно завершится.

Исключения ловят в специальном блоке try...except:

```
try:
    a = int(input("Введите целое число: "))
    print(a + 10)
except ValueError:
    print("Неверное число")
```

```
Введите целое число: 11df
Неверное число
```

Функция **int** порождает исключение **ValueError** (неверное значение), когда в строке есть посторонние символы (например, буквы). Как только не удаётся выполнить строку a = ... , управление переходит к обработчику исключений (блок **except**). Так как исключение — это класс, то они могут быть наследниками друг друга. Обработчик поймает не только указанные исключения, но и всех их наследников. Дерево встроенных исключений выглядит так:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
           +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
```

```
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Если нужен доступ к исключению как к объекту, пригодится такая конструкция:

```
try:
    a = int(input("Введите целое число: "))
    print(a + 10)
except ValueError as ve:
    print("Неверное число")
    print(ve)
    print(dir(ve))
```

```
Введите целое число: fff
Неверное число
invalid literal for int() with base 10: 'fff'
['__cause__', '__class__', '__context__', '__delattr__', '__dict__', '__dir__', '
```

В данном примере в переменную ve попадает объект класса **ValueError**, а функция **dir** позволяет увидеть содержимое этого объекта. Перепишем задачу ввода карты вместе с исключениями. Там, где нужно было использовать код возврата, вставим конструкцию **raise** (генерация объекта — исключения заданного типа). Вот во что превратится функция **get_card_number:**

```
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 ValueError("Неверный формат номера")
    return card_number
```

Оставшийся код станет таким:

```
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 ValueError("Недействительный номер карты")


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


process()
```

Результат работы программы:

```
Введите номер карты (16 цифр): 34543
Ошибка! Неверный формат номера
Введите номер карты (16 цифр): 4728 7952 5723 3848
4728795257233848
Ошибка! Недействительный номер карты
Введите номер карты (16 цифр): 4728 7953 5723 3848
4728795357233848
Ваша карта обрабатывается...
```

Работа с исключениями избавляет от некоторых недостатков кодов возврата: он становится короче и не надо ставить конструкции if во всём стеке вызовов. Работа с ошибками становится гибче благодаря дереву исключений. Мы можем работать с системными исключениями (например, с прерыванием по нажатию на клавишу) так же легко, как с собственными.

Однако для работы с исключениями надо тренироваться. Код легко наводнить бесконечными проверками условий в ущерб основному алгоритму. На втором уроке мы ещё больше поговорим об исключениях.

Помощь

© 2018 – 2019 ООО «Яндекс»