

Скачайте Яндекс Браузер для образования

Скачать

< Урок QT. Исключения

Понятие исключения. Обработка исключений. Собственные исключения

- 1 Что такое ошибка или исключение
- 2 Стек вызовов в Python
- 3 Работа с кодами возврата
- 4 Трудности работы с кодами возврата
- 5 Обработка исключений
- 6 Методики LBYL и EAFP
- 7 Полный синтаксис блока try...except
- 8 Создание пользовательских типов ошибок
- 9 Конструкция assert

Аннотация

Урок рассказывает о работе с исключениями в современных языках программирования, в частности, в языке Python. Сравнение методики исключений с методикой кодов возврата. Построение собственных классов исключений и их наследование, методики LBYL и EAFP.

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 такие обстоятельства **ошибками** или **исключениями**.

Чтобы было понятнее, рассмотрим пример алгоритма посещения магазина.

Родители посылают дочь в магазин и просят купить молока некоторого бренда 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 (все равно брать?), заканчивается срок годности (это важно?) и так далее.

Все 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. Все 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 дает 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 еще раз, что по умолчанию от необработанной ошибки программа на Python немедленно останавливается и выводит сообщение:

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

```
Traceback (most recent call last):  
  File "/home/02.py", line 2, in <module>  
    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 дает стек вызовов, когда указывает на место ошибки.

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

Проведем три эксперимента:

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

```
def div(a, b):  
    return a / b  
  
print(div(4, 10))
```

0.4

```
def run_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:

```
Введите a: 45  
Введите b: 0  
Traceback (most recent call last):  
  File "/home/01.py", line 11, in <module>  
    run_division()  
  File "/home/01.py", line 8, in run_division  
    print(div(a, b))  
  File "/home/01.py", line 2, in div  
    return a / b  
ZeroDivisionError: float division by zero
```

И еще раз:

```
Введите a: fd  
Traceback (most recent call last):  
  File "/home/01.py", line 11, in <module>
```

```
run_division()
File "/home/01.py", line 6, in run_division
    a = float(input("Введите a: "))
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 введет 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.

С кодами возврата вы, наверняка, достаточно часто сталкивались на предыдущих двух уроках при использовании библиотеки PyQT, так как QT написана на C++. Мы пишем строчку `sys.exit(app.exec())` при создании приложения в том числе и для того, чтобы в случае ошибки получить код возврата библиотеки PyQT.

Давайте рассмотрим еще один пример использования кодов возврата.

У нас есть приложение с возможностью оплаты картой каких-либо товаров или услуг (например, премиум-подписки). Просим пользователя ввести номер карты.





Примерно вот так может выглядеть **форма** такого приложения:

Программа для этого может быть тоже очень простой:

```
import sys

from PyQt5 import uic
from PyQt5.QtWidgets import QWidget, QApplication

class PayForm(QWidget):
    def __init__(self):
        super(PayForm, self).__init__()
        uic.loadUi('pay.ui', self)
        self.payButton.clicked.connect(self.get_data)

    def get_data(self):
        card_num = self.cardData.text()
        print(card_num)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    form = PayForm()
    form.show()
```

```
sys.exit(app.exec())
```

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?

Можно написать ему об этом:

```
self.hintLabel.setText('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 Лúна. 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 нечетной 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 четных 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 новые карты и генерируют номера для них, контрольную шестнадцатую цифру они подбирают так, чтобы алгоритм Лúна 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 (не забудьте поменять слот при нажатии на кнопку на `process_data`):

```
def get_card_number(self):
    card_num = self.cardData.text()
    return card_num

def process_data(self):
    number = self.get_card_number()
    self.errorLabel.setText("Ваша карта обрабатывается...")
```

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

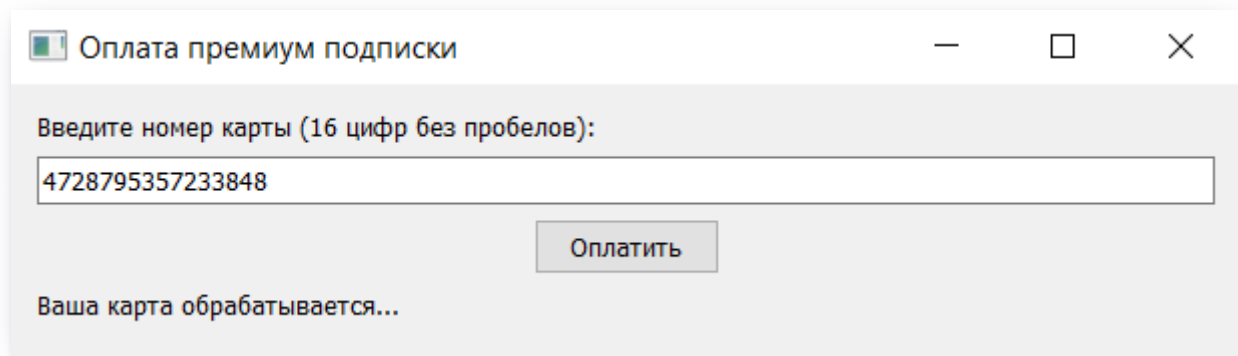
```
def get_card_number(self):
    card_num = self.cardData.text()
    return card_num

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

def luhn_algorithm(self, card):
    odd = map(lambda x: self.double(int(x)), card[::2])
    even = map(int, card[1::2])
    return (sum(odd) + sum(even)) % 10 == 0

def process_data(self):
    number = self.get_card_number()
    if self.luhn_algorithm(number):
        self.errorLabel.setText("Ваша карта обрабатывается...")
```

Проверим, введем номер карты: 4728795357233848.



Метод `luhn_algorithm` проверяет данные. Карта засчитается, только если данные корректны. Можно рассматривать этот метод как функцию с кодом возврата. Он говорит там, корректен ли номер карты.

Однако если пользователь введет не 16 цифр, а что-нибудь другое, или 16 цифр, разделенных пробелами, то он обрушит программу. Попробуйте, например, ввести «asdasdasd».

Программа упадет с таким сообщением:

```
Process finished with exit code -1073740791 (0xC0000409)
```

Не очень информативно... Это происходит из-за того, что мы используем PyQt, а она по умолчанию «замалчивает» необработанные ошибки. Внесем небольшие изменения в код нашей программы, чтобы это исправить (в дальнейшем вставляйте этот код во все свои приложения на PyQt):

```
def except_hook(cls, exception, traceback):
```

```

sys.__excepthook__(cls, exception, traceback)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    form = PayForm()
    form.show()
    sys.excepthook = except_hook
    sys.exit(app.exec())

```

Попробуем ввести неверные данные еще раз и получим в консоли уже более понятное сообщение:

```

Traceback (most recent call last):
  File "D:/projects/yandexlyceum/Lessons/QT2020/QT3/Sample/1/main.py", line 31, in process_data
    if self.luhn_algorithm(number):
  File "D:/projects/yandexlyceum/Lessons/QT2020/QT3/Sample/1/main.py", line 27, in luhn_algorithm
    return (sum(odd) + sum(even)) % 10 == 0
  File "D:/projects/yandexlyceum/Lessons/QT2020/QT3/Sample/1/main.py", line 25, in <lambda>
    odd = map(lambda x: self.double(int(x)), card[::2])
ValueError: invalid literal for int() with base 10: 'a'

```

Как с такими сообщениями работать поговорим чуть позже, а пока сделаем так, чтобы в ответ на некорректный запрос программа не «падала», а требовала ввести 16 цифр, произвольно разделенных пробелами.

Для этого метод `get_card_number` теперь будет возвращать специальный код — например, 404, как в Интернете.

```

def get_card_number(self):
    card_num = self.cardData.text()
    card_num = ''.join(card_num.split())
    if card_num.isdigit() and len(card_num) == 16:
        return card_num
    else:
        return 404

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

def luhn_algorithm(self, card):
    odd = map(lambda x: self.double(int(x)), card[::2])
    even = map(int, card[1::2])
    return (sum(odd) + sum(even)) % 10 == 0

def process_data(self):

```

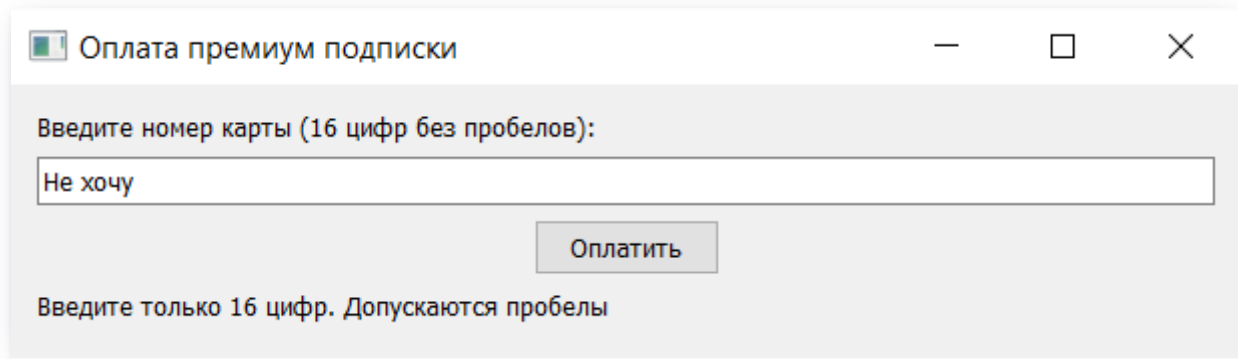


```

number = self.get_card_number()
if number == 404:
    self.errorLabel.setText(
        "Введите только 16 цифр. Допускаются пробелы")
elif self.luhn_algorithm(number):
    self.errorLabel.setText(
        "Ваша карта обрабатывается...")
else:
    self.errorLabel.setText(
        "Номер недействителен. Попробуйте снова.")

```

Теперь при вводе неверных данных пользователь может получить еще одно сообщение:



4. Трудности работы с кодами возврата

Мы видим, что даже простая функция для обработки пользовательских данных обрastaет дополнительным кодом, проверкой многих условий и «магическими» кодами возврата. Если функция с кодом возврата находится глубоко в стеке вызовов, то придется сделать так, чтобы ее 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 все остальное. Исключение создается в любом месте и поднимается по стеку вызовов, пока его не отловит какой-нибудь код-обработчик.

Сейчас поймаем исключения:

```

Traceback (most recent call last):
  File "/home/06.py", line 2, in <module>
    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 удастся выполнить строку `a = ...`, управление переходит к обработчику исключений (блок `except`).

Так как исключения — это классы, 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__', '__doc__', '__
```

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 метод `get_card_number`:

```
def get_card_number(self):
    card_num = self.cardData.text()
    if not (card_num.isdigit() and len(card_num) == 16):
        raise ValueError("Неверный формат номера")
    return card_num
```

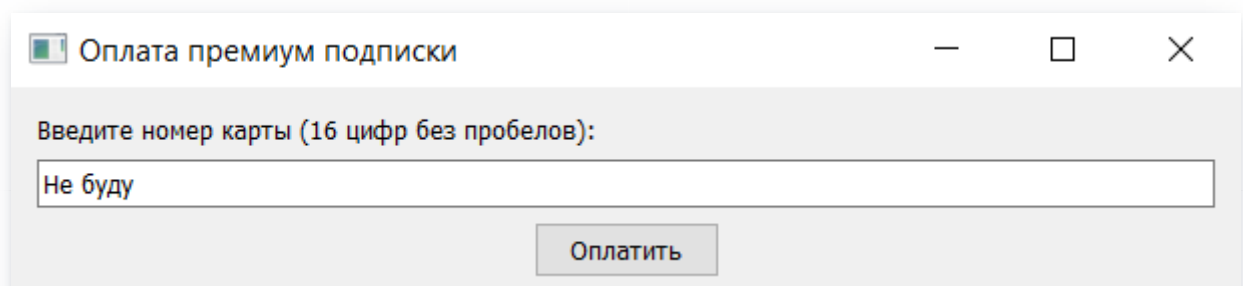
А остальные методы нашего класса станут такими:

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

def luhn_algorithm(self, card):
    odd = map(lambda x: self.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_data(self):
    try:
        number = self.get_card_number()
        if self.luhn_algorithm(number):
            print("Ваша карта обрабатывается...")
    except ValueError as e:
        self.errorLabel.setText(f"Ошибка! {e}")
```

Результат работы программы:



Оплата премиум подписки

Введите номер карты (16 цифр без пробелов):

Не буду

Оплатить

Ошибка! Неверный формат номера

Работа с исключениями избавляет от некоторых недостатков кодов возврата: он становится короче и не надо ставить конструкции `if` во всем 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.

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

6. Методики LBYL и EAFP

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]
```

```
Traceback (most recent call last):
  File "/home/01.py", line 2, in <module>
    mydict[4354]
KeyError: 4354
```

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

7. Полный синтаксис блока try...except

Мы уже пользовались блоком try...except, но вообще try...except может выглядеть существенно сложнее:

```
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('Идем дальше')
```

0.5
Идем дальше
0.5714285714285714
Идем дальше
Поделили на 0
Идем дальше

Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем 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 прервется 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

Идем дальше

0.5714285714285714

Идем дальше

Непредвиденная ошибка division by zero

Идем дальше

Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'

Идем дальше

Непредвиденная ошибка list index out of range

Идем дальше

Непредвиденная ошибка list index out of range

Идем дальше

Непредвиденная ошибка list index out of range

Идем дальше

Непредвиденная ошибка list index out of range

Идем дальше

Непредвиденная ошибка list index out of range

Идем дальше

Непредвиденная ошибка list index out of range

Идем 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 подошел:

```
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

Все хорошо

Идем дальше

0.5714285714285714

Все хорошо

Идем дальше

Поделили на 0

Идем дальше

Непредвиденная ошибка unsupported operand type(s) for /: 'int' and 'NoneType'

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка

Идем дальше

Мы за границей списка


```
Идем дальше
Мы за границей списка
Идем дальше
Мы за границей списка
Идем дальше
```

Перечислим еще 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 все в порядке.

8. 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 про проверку номеров банковской карты, в который мы добавили несколько собственных исключений:

```
import sys

from PyQt5 import uic
from PyQt5.QtWidgets import QWidget, QApplication

class CardError(Exception):
    pass

class CardFormatError(CardError):
    pass
```

```
class CardLuhnError(CardError):
    pass

class PayForm(QWidget):
    def __init__(self):
        super(PayForm, self).__init__()
        uic.loadUi('pay.ui', self)
        self.hintLabel.setText(
            'Введите номер карты (16 цифр без пробелов):')
        self.payButton.clicked.connect(self.process_data)

    def get_card_number(self):
        card_num = self.cardData.text()
        if not (card_num.isdigit() and len(card_num) == 16):
            raise CardFormatError("Неверный формат номера")
        return card_num

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

    def luhn_algorithm(self, card):
        odd = map(lambda x: self.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_data(self):
        try:
            number = self.get_card_number()
            if self.luhn_algorithm(number):
                print("Ваша карта обрабатывается...")
        except CardError as e:
            self.errorLabel.setText(f"Ошибка! {e}")

def except_hook(cls, exception, traceback):
    sys.__excepthook__(cls, exception, traceback)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    form = PayForm()
    form.show()
    sys.excepthook = except_hook
```

```
sys.exit(app.exec())
```

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

9. Конструкция `assert`

Конструкция `assert` — это часть Python, связанная с тестированием. Тестирование мы еще не проходили, но это не мешает 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 — создается 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.

Исключения — прекрасный инструмент, который позволит вам писать более простые и красивые программы, однако, использовать его надо с умом, выбирая, когда лучше использовать их, а когда добавить дополнительное условие.

Сам механизм исключений достаточно медленный, поэтому, например, не очень хорошей идеей будет кидать исключение внутри цикла, когда мы точно знаем, что их будет достаточно большое количество.

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках сервиса, принадлежат АНО ДПО «Образовательные технологии Яндекса». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «Образовательные технологии Яндекса».

[Пользовательское соглашение.](#)

© 2018 – 2024 ООО «Яндекс»