

## Лабораторная работа 3. Обработка исключений

Во время работы программы иногда возникают непредвиденные обстоятельства - как правило, внешние. Программа должна их верно обрабатывать. Такие обстоятельства называют ошибками или исключениями.

### 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 строку. 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
```

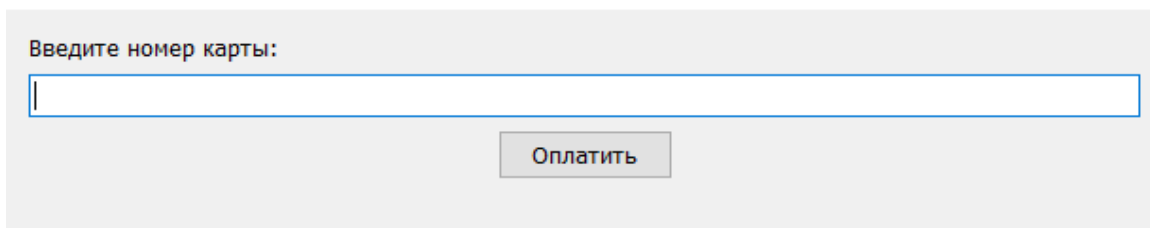
Из-за того, что в исходной строке не было символа «Д», нам было возвращено значение `-1`. Это и есть **код возврата**. Так как любая функция в Python возвращает значения, мы можем использовать их для кодирования информации об ошибке.

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.

Рассмотрим пример использования кодов возврата.

У нас есть приложение с возможностью оплаты картой каких-либо товаров или услуг. Просим пользователя ввести номер карты.



Введите номер карты:

Программа может быть 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 прочитав внимательно инструкцию и ввести пробелы.

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

Первая цифра обозначает тип карты. 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.

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 Луна 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.

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...")
```

Введите номер карты (16 цифр без пробелов):

4728795357233848

Оплатить

Ваша карта обрабатывается...

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

Однако если пользователь введет не 16 цифр, а что-нибудь другое, или 16 цифр, разделенных пробелами, то он обрушит программу. Программа «упадет».

Внесем небольшие изменения в код нашей программы, чтобы это исправить:

```
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())
```

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()
    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.")

```

Введите номер карты (16 цифр без пробелов):

Оплатить

Введите только 16 цифр. Допускаются пробелы

Мы видим, что даже простая функция для обработки пользовательских данных обрastaет дополнительным кодом, проверкой многих условий и 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.

## 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.

Поймаем исключения:

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

```
Traceback (most recent call last):  
  File "/home/06.py", line 2, in <module>  
    s.index("9")  
ValueError: substring not found
```

«Поймав» исключение, очень легко определить его тип. Он выведется на экран.

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))
```

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}")

```

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.

### Методики LBYL и EAFP

С помощью исключений 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, считывая, 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.

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.

### Полный синтаксис блока `try...except`

`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.

### 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 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.

### Перехват системных исключений

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

Их можно использовать. Например, нажатие комбинации `Ctrl+C` обычно прекращает работу консольного приложения 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 комбинации `Ctrl+C` мы попадем в блок `except` и зададим пользователю вопрос о прерывании работы программы.

### Конструкция `assert`

Конструкция `assert` - это часть Python, связанная с тестированием.

В любое место программы можно 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:

```
-----

Traceback (most recent call last):
  File "/home/01.py", line 2, in <module>
    assert len(s) == 4
```

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.

Исключения - прекрасный инструмент, который позволит 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.

### Задание 1.

Напишите программу, проверяющую корректность введенного номера сотового телефона в РФ по следующим критериям:

- Номер может начинаться как с **+7**, так и с **8**
- Допускается любое количество любых пробельных символов в любом месте, например, `+7 905 3434 341`.
- Допускается наличие в любом месте одной пары скобок (обязательно пары), например: `8 (905) 3434 341`.
- Допускается наличие любого количества знаков `-`, только не подряд (`--`), не в начале и не в конце. Например, `+7 905-34-34-341`.

Если введенный номер корректен, он преобразуется к формату `+79053434341`. То есть `8` заменяется на `+7`, а все другие символы-НЕцифры убираются. В итоговой записи остается 11 цифр.

Если же номер не удовлетворяет перечисленным условиям, выводится слово `error`.

Пример:

Ввод	Вывод
<code>+7(902)123-4567</code>	<code>+79021234567</code>
<code>567))7776553</code>	<code>error</code>

### Задание 2.

Напишите программу, которая будет требовать у пользователя ввода нового пароля до тех пор, пока не будет введен корректный, либо пока пользователь не прекратит программу с клавиатуры, то есть нажмет комбинацию клавиш `Ctrl-Break` или аналогичную ей.

Критерии правильности пароля:

- Длина пароля больше 8 символов.
- В нем присутствуют большие и маленькие буквы любого алфавита.



- В нем имеется хотя бы одна цифра.
- В пароле нет ни одной комбинации из 3 буквенных символов, стоящих рядом в строке клавиатуры независимо от того, русская раскладка выбрана или английская. Например, недопустимы , «QwE», «TYU», «йцу», «Hjk», «ЛДЖ» и т.д. А «QWу», «хъф» и т.д. - вполне подходят. Причем, надо учесть как раскладку РС-совместимых компьютеров, так и раскладку MAC'ов.

Если пользователь вводит неправильный пароль, то необходимо вывести имя класса того типа исключения, который будет «выброшен» вашей программой. После этого ввод продолжается.

Как только будет введен правильный пароль следует вывести ок и тотчас же прекратить выполнение программы.