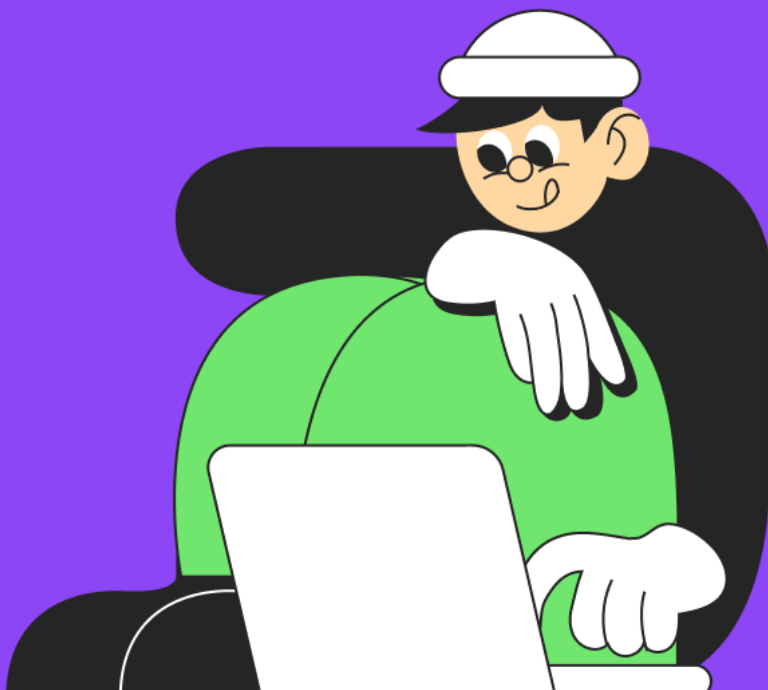
GeekBrains

Лекция 13. Исключения

Погружение в Python



Оглавление

На этой лекции мы

1. Разберёмся с обработкой ошибок в Python
2. Изучим иерархию встроенных исключений
3. Узнаем о способе принудительного поднятия исключения в коде
4. Разберёмся в создании собственных исключений

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались в превращении объекта в функцию

2. Изучили способы создания итераторов
3. Узнали о создании менеджеров контекста
4. Разобрались в превращении методов в свойства
5. Изучили работу дескрипторов
6. Узнали о способах экономии памяти

Термины лекции

- **Исключение** — это любое состояние ошибки или непредвиденное поведение, возникающее при выполнении программы

Подробный текст лекции

1. Обработка исключительных ситуаций в Python

На протяжении курса мы регулярно сталкивались с ошибками в программах. Python выдавал несколько строк красного текста в консоль — результат трассировки ошибки. И после такого программа переставала работать.

Встроенный в язык механизм обработки исключений позволяет изменить поведение программы при появлении ошибки. Впрочем, некоторые ошибки действительно должны завершать программу. О них подробнее чуть позже в этой лекции.
А пока рассмотрим простой пример кода и результат его выполнения
Код:

```
num = int(input('Введите целое число: '))
```

Результат

```
Введите целое число: сорок два
Traceback (most recent call last):
  File "C:\Users\main.py", line 1, in <module>
    num = int(input('Введите целое число: '))
ValueError: invalid literal for int() with base 10: 'сорок два'
```

```
Process finished with exit code 1
```

Программа запросила целое число, пользователь написал “сорок два” и мы получили ошибку ValueError. Красный текст — результат трассировки ошибки. Читать его лучше построчно снизу вверх. Самая нижняя строка указывает какую именно ошибку мы получили и почему. Строкой выше указывается строчка кода, повлёкшая ошибку. Отдельно Python указывает название файла с номером строки для быстрого перехода к месту ошибки. При этом строчек с кодом и указаниями на строки ошибки может быть несколько. Зависит от того как долго ошибка распространялась по цепочке кода.

Посмотрите на код ниже.

```
def get(text: str = None) -> int:
    data = input(text)
    num = int(data)
    return num


if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

Попробуем и тут ввести “сорок два”

```
Введите целый делитель: сорок два
Traceback (most recent call last):
  File "C:\Users\main.py", line 8, in <module>
    number = get('Введите целый делитель: ')
  File "C:\Users\main.py", line 3, in get
    num = int(data)
ValueError: invalid literal for int() with base 10: 'сорок два'

Process finished with exit code 1
```

Ошибка точно такая же. Но теперь трассировка показывает, что ошибка возникла в третьей строке кода при приведении данных к целому типу. А чуть ранее мы вызвали функцию get в строке 8, которая и заставила выполнятся третью строчку кода.

Чтение трассировки помогает найти источник ошибки. Python и тут проявляет максимальное дружелюбие к разработчику и даёт максимально возможное количество информации об ошибке. Если вы писали на других языках, скорее всего вы уже заметили это дружелюбие.

Команда try

Что делать, если мы не хотим завершать программу в случае появления ошибки? Команда try позволяет обернуть блок кода с возможной ошибкой и отловить её.

Поступим так с третьей строкой нашей программы.

```
def get(text: str = None) -> int:
    data = input(text)
    try:
        num = int(data)
    return num


if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

Команда try: должна завершать строку двоеточием, а вложенный блок кода пишется с отступами. Всё точно так же как с if, for, while и т.п.

🔥 **Важно!** Блок кода внутри try в идеале должен состоять из одной строки кода — потенциального источника ошибки. Оборачивание всей программы в блок try считается плохим стилем программирования.

Запустим код и получим самую частую ошибку новичка — SyntaxError.

```
  File "C:\Users\main.py", line 5
    return num
    ^^^^^^
SyntaxError: expected 'except' or 'finally' block
```

Ошибка синтаксиса говорит о том, что Python не понял вас. “Дружище, твой код не похож на Python программу. Я указал на место, откуда надо начинать искать ошибку. Но она может быть и раньше этой строки”. Синтаксические ошибки обязательно исправлять в процессе создания программы, а не пытаться их обрабатывать.

Команда except

Как вы верно догадались из текста синтаксической ошибки, команда try должна работать в связке с командой except или finally.

Рассмотрим вариант обработки ошибки в нашем коде:

```
def get(text: str = None) -> int:
    data = input(text)
    try:
        num = int(data)
    except ValueError as e:
        print(f'Ваш ввод привёл к ошибке ValueError: {e}')
        num = 1
        print(f'Будем считать результатом ввода число {num}')
    return num


if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

После зарезервированного слова except указывается класс ошибки, которую мы хотим обработать. Обычно используется запись вида except NameError as e: Таким образом в переменную e попадает информация об ошибке. Например для вывода её в консоль, сохранения в логи и т.п. В нашем случае при невозможности получить целое число из пользовательского ввода сообщаем ему об этом. Далее делаем вид, что была введена единица. Программа продолжает работать несмотря на ошибку.

- **Цикл while для обработки ошибок ввода**

Иногда необходимо получить данные повторно, если попытка не удалась. В случае с пользователем можем спрашивать его бесконечно, пока не добъёмся ввода целого числа.

```
def get(text: str = None) -> int:
    while True:
        try:
            num = int(input(text))
            break
        except ValueError as e:
            print(f'Ваш ввод привёл к ошибке ValueError: {e}\n'
                  f'Попробуйте снова')
    return num


if __name__ == '__main__':
```

```
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

Бесконечный цикл while True можно прервать командой break. В случае преобразования ввода пользователя к целому без ошибок она завершит цикл и вернёт число из функции. При появлении ошибки дальнейшие строки не выполняются, сразу переходим в блок except. Обработав ошибку мы возвращаемся к началу цикла, следовательно повторяем запрос.

Подобное поведение можно применить не только к функции input, но и к любой ситуации получения данных. Код ниже имитирует получение данных из источника (база данных, сайт, удалённыё сервер и т.п.).

```
MAX_COUNT = 5

count = 0
while count < MAX_COUNT:
  count += 1
  try:
    data = get_data()
    break
  except ConnectionError as e:
    print(f'Попытка {count} из {MAX_COUNT} завершилась ошибкой {e}')
```

- **Несколько except для одного try**

Как вы уже догадались при вводе нуля в примерах на деление выше мы получим ошибку. Это ZeroDivisionError: division by zero.
Вспомним высшую математику, а именно то, что при делении любого числа на ноль получаем бесконечность.

```
def hundred_div_num(text: str = None) -> tuple[int, float]:
    while True:
        try:
            num = int(input(text))
            div = 100 / num
            break
        except ValueError as e:
            print(f'Ваш ввод привёл к ошибке ValueError: {e}\n'
                  f'Попробуйте снова')
        except ZeroDivisionError as e:
            div = float('inf')
            break
    return num, div
```

```
if __name__ == '__main__':
    n, d = hundred_div_num('Введите целый делитель: ')
    print(f'Результат операции: "100 / {n} = {d}"')
```

В приведённом примере блок try обрабатывает сразу несколько строчек, которые способны вызвать разные ошибки. Не лучший вариант. Правильнее было бы разделить код на отдельные try блоки со своими возможными исключениями. Но зато мы познакомились с обработкой нескольких ошибок разом.
Если не удалось получить целое число, обрабатываем ошибку значения и даём ещё один шанс. А если делим на ноль, вместо ошибки возвращаем бесконечность.

Внимание! Язык Python поддерживает такие математические числа как бесконечность и минус бесконечность. Записываются они как особая форма вещественного числа:

- float('inf') - бесконечность
- float('-inf') - минус бесконечность

Команда else

Если надо выполнить код только в случае успешного завершения блока try, можно воспользоваться командой else в связке try-except-else

```
MAX_COUNT = 5

result = None
for count in range(1, MAX_COUNT + 1):
    try:
        num = int(input('Введите целое число: '))
        print('Успешно получили целое число')
    except ValueError as e:
            print(f'Попытка {count} из {MAX_COUNT} завершилась
ошибкой {e}')
    else:
        result = 100 / num
        break

print(f'{result = }')
```

В приведённом примере пользователь вводит число. Если получаем ошибку, сообщаем о ней пользователю. Всего даём MAX_COUNT попыток. Но стоит успешно преобразовать текст в целое как сработает блок else. Он сохранит результат деления и завершит цикл попыток.
Если внутри блока try произойдёт одно из событий ниже, блок else не будет вызван:

- возбуждено исключение
- выполнена команда return
- выполнена команда break
- выполнена команда continue

Именно по этой причине в нашем примере break перекочевал из try в else.
Блок else может быть лишь один и обязан следовать за блоком except.

Вложенные блоки обработки исключений

При необходимости одни try блоки могут включать другие. Аналогично работа.т вложенные циклы или вложенные if — сложные ветвления.
Перепишем код выше так, чтобы ошибка деления на ноль обрабатывалась внутри блока else верхнего try.

```
MAX_COUNT = 5

result = None
for count in range(1, MAX_COUNT + 1):
    try:
        num = int(input('Введите целое число: '))
        print('Успешно получили целое число')
    except ValueError as e:
            print(f'Попытка {count} из {MAX_COUNT} завершилась
ошибкой {e}')
    else:
        try:
            result = 100 / num
        except ZeroDivisionError as e:
            result = float('inf')
        break

print(f'{result = }')
```

Если удалось получить целое число, заглядываем в else и пытаемся делить. Но если деление на ноль, возвращаем бесконечность вместо ошибки.

Команда finally

Ещё одна команда для обработки исключений — finally. Она срабатывает во всех случаях. И если была ошибка и отработал блок except. И если ошибки не было.

```
def get(text: str = None) -> int:
    num = None
    try:
        num = int(input(text))
    except ValueError as e:
        print(f'Ваш ввод привёл к ошибке ValueError: {e}')
    finally:
        return num if isinstance(num, int) else 1


if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    try:
        print(f'100 / {number} = {100 / number}')
    except ZeroDivisionError as e:
        print(f'На ноль делить нельзя. Получим {e}')
```

Даём пользователю одну попытку на ввод числа. Независимо от результата сработает блок finally. Он вернёт либо целое число, либо единицу, если получить целое не удалось. Обработку деления на ноль вынесли в основной код.
Обычно finally используют для действий, которые обязательны независимо от того была ошибка или нет.

Блок finally без except

Вполне допустимо использовать только связку try-finally. Например мы хотим прочитать информацию из файла. И независимо от результат чтения закрыть его.

```
file = open('text.txt', 'r', encoding='utf-8')
try:
    data = file.read().split()
    print(data[len(data)])
finally:
    print('Закрываю файл')
    file.close()
```

Открываем файл для чтения, скачиваем его и сохраняем каждую строку в отдельную ячейку списка. Но внезапно “забываем”, что нумерация начинается с нуля, а для доступа к последнему элементу можно использовать индекс -1. В результате попытка прочитать информацию из ячейки, следующей за последней

выдаёт ошибку IndexError: list index out of range. Но блок finally закрывает файл раньше, чем программа завершит свою работу.

🔥 **Важно!** Для ситуации выше правильным будет использовать менеджер контекста для гарантированного закрытия файла.

Как вы можете видеть комбинации try-except-else-finally дают большой простор для отлова исключений и изменения логика кода в зависимости от ситуаций.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты. Четыре попытки ввода пользователя указаны ниже кода

```
d = {'42': 73}
try:
    key, value = input('Ключ и значение: ').split()
    if d[key] == value:
        r = 'Совпадение'
except ValueError as e:
    print(e)
except KeyError as e:
    print(e)
else:
    print(r)
finally:
    print(d)
```

>>> Ключ и значение: 42 13
>>> Ключ и значение: 42 73 3
>>> Ключ и значение: 73 42
>>> Ключ и значение: 42 73

2. Иерархия исключений в Python

Любое исключение в Python является классом. При этом одни классы наследуются от других. Общая [иерархия классов для встроенных исключений Python] выглядит так:

```
BaseException
```

```
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
└── Exception
     ├── ArithmeticError
     │    ├── FloatingPointError
     │    ├── OverflowError
     │    └── ZeroDivisionError
     ├── AssertionError
     ├── AttributeError
     ├── BufferError
     ├── EOFError
     ├── ExceptionGroup [BaseExceptionGroup]
     ├── 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
     ├── StopAsyncIteration
     ├── StopIteration
     ├── SyntaxError
     │    └── IndentationError
     │         └── TabError
     ├── SystemError
     ├── TypeError
     ├── ValueError
     │    └── UnicodeError
```

```
      │          ├── UnicodeDecodeError
      │          ├── UnicodeEncodeError
      │          └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

BaseException и его ближайшие родственники

Базовым исключением является класс BaseException. Все остальные исключения являются его потомками. Не рекомендуется его перехватывать или использовать в создании своих исключений. Оно нужно для правильной иерархии исключений, ожидаемого поведения Python.
Помимо BaseException в подавляющем большинстве случаев не перехватывают и следующие исключения:

- **BaseExceptionGroup** — базовое исключение для объединения исключений в группу
- **GeneratorExit** — исключение создаёт генератор или корутина при закрытии. Подобное ситуация не является ошибкой с технической точки зрения.
- **KeyboardInterrupt** — возникает при нажатии комбинации клавиш, прерывающих работу программы. Например Ctrl-C в терминале посылает сигнал, требующий завершить процесс
- **SystemExit** — при выходе из программы через функцию exit() поднимается данное исключение.

Основной Exception и его наследники

Исключение Exception является базовым для всех исключений Python. Также оно используется при создании своих собственных исключений.
Перечислим некоторые часто встречающиеся исключения и дадим пояснения по ним.

- ArithmeticError — арифметическая ошибка.
    - FloatingPointError — порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
    - OverflowError — возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
    - ZeroDivisionError — деление на ноль.
- AssertionError — выражение в функции assert ложно. Подробнее об assert поговорим на следующей лекции, когда будем разбирать тестирование кода.
- AttributeError — объект не имеет данного атрибута (значения или метода).
- BufferError — операция, связанная с буфером, не может быть выполнена.
- EOFError — функция наткнулась на конец файла и не смогла прочитать то, что хотела.
- ImportError — не удалось импортирование модуля или его атрибута.
- LookupError — некорректный индекс или ключ.
    - IndexError — индекс не входит в диапазон элементов.
    - KeyError — несуществующий ключ (в словаре, множестве или другом объекте).
- MemoryError — недостаточно памяти.
- NameError — не найдено переменной с таким именем.
- UnboundLocalError — сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
- OSError — ошибка, связанная с системой.
    - BlockingIOError — возникает, когда операция блокирует объект (например, сокет), установленный для неблокирующей операции
    - ChildProcessError — неудача при операции с дочерним процессом.
    - ConnectionError — базовый класс для исключений, связанных с подключениями.
        - BrokenPipeError — возникает при попытке записи в канал, в то время как другой конец был закрыт, или при попытке записи в сокет, который был отключен для записи
        - ConnectionAbortedError — попытка соединения прерывается узлом
        - ConnectionRefusedError — партнер отклоняет попытку подключения
        - ConnectionResetError - соединение сбрасывается узлом

    - FileExistsError — попытка создания файла или директории, которая уже существует.
    - FileNotFoundError — файл или директория не существует.
    - InterruptedError — системный вызов прерван входящим сигналом.
    - IsADirectoryError — ожидался файл, но это директория.
    - NotADirectoryError — ожидалась директория, но это файл.
    - PermissionError — не хватает прав доступа.
    - ProcessLookupError — указанного процесса не существует.
    - TimeoutError — закончилось время ожидания.
- ReferenceError — попытка доступа к атрибуту со слабой ссылкой.
- RuntimeError — возникает, когда исключение не попадает ни под одну из других категорий.
    - NotImplementedError — возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
    - RecursionError — превышена глубина стека вызова функций.
- StopAsyncIteration — порождается в корутине (асинхронной функции), если в итераторе больше нет элементов.
- StopIteration — порождается встроенной функцией next, если в итераторе больше нет элементов.
- SyntaxError — синтаксическая ошибка.
    - IndentationError — неправильные отступы.
        - TabError — смешивание в отступах табуляции и пробелов.
- SystemError — интерпретатор находит внутреннюю ошибку, но ситуация не выглядит настолько серьезной, чтобы заставить его отказаться от всякой надежды. Возвращаемое значение представляет собой строку, указывающую, что пошло не так.
- TypeError — операция применена к объекту несоответствующего типа.
- ValueError — функция получает аргумент правильного типа, но некорректного значения.
    - UnicodeError — ошибка, связанная с кодированием / раскодированием unicode в строках.
        - UnicodeEncodeError — исключение, связанное с кодированием unicode.
        - UnicodeDecodeError — исключение, связанное с декодированием unicode.
        - UnicodeTranslateError — исключение, связанное с переводом unicode.
- Warning — группа для предупреждений.

Группа Warning

Warning включает в себя ряд базовых предупреждений. Предупреждающие сообщения обычно выдаются в ситуациях, когда полезно предупредить пользователя о каком-либо состоянии в программе, когда это условие не требует возбуждения исключения и завершения программы. Например, может потребоваться выдать предупреждение, когда программа использует устаревший модуль.

3. Ключевое слово raise

Отдельно рассмотрим команду raise для поднятия исключений. С ней мы уже сталкивались на прошлой лекции. Команда поднимает исключение, указанное после неё. Используется для случаев, когда вы явно хотите сообщить о неправильной работе вашего кода.

Например у нас есть функция, которая запрещает изменять значение у существующих ключей.

```
def add_key(dct, key, value):
    if key in dct:
            raise KeyError(f'Перезаписывание существующего ключа запрещено. {dct[key] = }')
    dct[key] = value
    return dct


data = {'one': 1, 'two': 2}
print(add_key(data, 'three', 3))
print(add_key(data, 'three', 3))
```

Первый раз мы смогли добавить пару ключ-значение в словарь. Но строкой ниже получили ошибку. Ключ уже добавлен в словарь и изменить его нельзя. Поднимаем исключение KeyError.

Поднимаем исключения в классах

Более сложный пример поднятия исключений — контроль создания класса. Подобное было при создании дескрипторов. В нашем случае класс отслеживает переданные аргументы в методе инициализации.

```
class User:
    def __init__(self, name, age):
        if 6 < len(name) < 30:
            self.name = name
        else:
             raise ValueError(f'Длина имени должна быть между 6 и
30 символами. {len(name) = }')
        if not isinstance(age, (int, float)):
                raise TypeError(f'Возраст должен быть числом.
Используйте int или float. {type(age) = }')
        elif age < 0:
            raise ValueError(f'Возраст должен быть положительным.
{age = }')
        else:
            self.age = age


user = User('Яков', '-12')
```

Класс контролирует длину имени пользователя. Далее убеждаемся, что возраст — число. И если верно, то проверяем больше ли нуля число.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
def func(a, b, c):
    if a < b < c:
        raise ValueError('Не перемешано!')
    elif sum((a, b, c)) == 42:
        raise NameError('Это имя занято!')
    elif max(a, b, c, key=len) < 5:
        raise MemoryError('Слишком глуп!')
    else:
        raise RuntimeError('Что-то не так!!!')


func(11, 7, 3)  # 1
```

```
func(3, 2, 3)  # 2
func(73, -40, 9)  # 3
func(10, 20, 30)  # 4
```

4. Создание собственных исключений

В финале пару примеров создания собственных исключений. Попробуем для класса User из прошлого примера создать свои исключения.
Писать код исключений будем в отдельном файле error.py Начнём с того, что создадим своё собственное базовое исключение. От него будут наследоваться остальные наши исключения. Родительское исключение займёт пару строк кода

```
class UserException(Exception):
    pass
```

А теперь добавим исключения для ошибок имени и возраста пользователя. Пока это будут исключения на минималках.

```
class UserAgeError(UserException):
    pass


class UserNameError(UserException):
    pass
```

Теперь внесём правки в код инициализации пользователя. Заодно избавимся от магических чисел для минимальной и максимальной длины имени.

```
from error import UserNameError, UserAgeError


class User:
    MIN_LEN = 6
    MAX_LEN = 30

    def __init__(self, name, age):
        if self.MIN_LEN < len(name) < self.MAX_LEN:
            self.name = name
        else:
            raise UserNameError
        if not isinstance(age, (int, float)) or age < 0:
```

```
            raise UserAgeError
        else:
            self.age = age


user = User('Яков', '-12')
```

Подобный код отлично справляется с поставленной задачей. Но стал менее информативен в случае возникновения ошибок.

```
error.UserNameError
```

Понятно, что ошибка в имени. Но не очень информативно. Исправим ситуацию.

Методы __init__ и __str__ в классах своих исключений

Чтобы исключение давало подробную информацию об ошибке, будем передавать ему проблемную переменную. Класс User доработаем в строках подъёма ошибок.

```
from error import UserNameError, UserAgeError


class User:
    def __init__(self, name, age):
        if 6 < len(name) < 30:
            self.name = name
        else:
            raise UserNameError(name)
        if not isinstance(age, (int, float)) or age < 0:
            raise UserAgeError(age)
        else:
            self.age = age


user = User('Яков', '-12')
```

Благодаря наследованию переданные в исключение переменные могут выводится в тексте ошибки.

```
    raise UserNameError(name)
```

```
error.UserNameError: Яков
```

Уже лучше. Но без пары дандер методов в классах ошибок пока не идеально. Дорабатываем код в файле error.

```
class UserException(Exception):
    pass


class UserAgeError(UserException):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f'Возраст пользователя должен быть целым int() или вещественным float() больше нуля.\n' \
                    f'У вас тип {type(self.value)}, а значение {self.value}'


class UserNameError(UserException):
    def __init__(self, name, lower, upper):
        self.name = name
        self.lower = lower
        self.upper = upper

    def __str__(self):
        text = 'попадает в'
        if len(self.name) < self.lower:
            text = 'меньше нижней'
        elif len(self.name) > self.lower:
            text = 'больше верхней'
            return f'Имя пользователя {self.name} содержит {len(self.name)} символа(ов).\n' \
                    f'Это {text} границы. Попадите в диапазон ({self.lower}, {self.upper}).'
```

В случае с возрастом просто получаем текущее значение в переменную value. Далее выводим информацию об ошибке без явного уточнения проблемы. Просто сообщаем о допустимых типе и значении, а также выводим переданное значение и его тип.

При обработки ошибки имении дополнительно принимаем в инициализацию граничные значения длины. Переменная text внутри дандер __str__ получает значение в зависимости от границы: “меньше нижней” или “больше верхней”. Вывод точно указывает на то, в какую из границ мы не попали.

🔥 **Внимание!** В классе User надо исправит строку вызова ошибки имени, чтобы код сработал верно. Иначе исключение вернёт нам исключение TypeError: UserNameError.__init__() missing 2 required positional arguments: 'lower' and 'upper'

```
    ...
    def __init__(self, name, age):
        if self.MIN_LEN < len(name) < self.MAX_LEN:
            self.name = name
        else:
            raise UserNameError(name, self.MIN_LEN, self.MAX_LEN)
    ...
```

Вывод

На этой лекции мы:

1. Разобрались с обработкой ошибок в Python
2. Изучили иерархию встроенных исключений
3. Узнали о способе принудительного поднятия исключения в коде
4. Разобрались в создании собственных исключений

Краткий анонс следующей лекции

1. Разберёмся с написанием тестов в Python
2. Изучим возможности doctest
3. Узнаем о пакете для тестирования unittest
4. Разберёмся с тестированием через pytest

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий. Добавьте обработку исключений так, чтобы код продолжал работать и выполнял задачу. Для любителей сложностей создайте собственные исключения для каждой из задач.