Модуль 9. Обработка исключений

[Введение в обработку исключений. Блоки try / except 1](#_Toc148037939)

[Обработка исключений. Блоки finally и else 6](#_Toc148037940)

[Вложенные блоки try / except 9](#_Toc148037941)

[Распространение исключений (propagation exceptions) 11](#_Toc148037942)

[Инструкция raise и пользовательские исключения 14](#_Toc148037943)

[Оператор assert 15](#_Toc148037944)

[Когда использовать assert? 16](#_Toc148037945)

[Использование assert в функциях 16](#_Toc148037946)

[Менеджеры контекстов. Оператор with 17](#_Toc148037947)

## Введение в обработку исключений. Блоки try / except

Начиная с этого занятия, мы коснемся довольно важной темы в программировании – обработка исключений.  Давайте вначале я поясню, что стоит за этими «умными» словами. Предположим, мы в программе строчка за строчкой выводим некий многострочный текст:

**print**("Я к вам пишу – чего же боле?")

**print**("Что я могу еще сказать?")

**print**("Теперь, я знаю, в вашей воле")

**print**("Меня презреньем наказать.")

**print**("Но вы, к моей несчастной доле")

**print**("Хоть каплю жалости храня,")

**print**("Вы не оставите меня.")

И где-нибудь в серединке, после третьего print(), запишем команду:

**print**(a)

Разумеется, в момент выполнения этой программы, возникнет ошибка, а точнее – исключение с типом:

NameError

Причем, первые три строчки будут выполнены, то есть, до момента исключения интерпретатор Python выполняет программу строчка за строчкой. Это пример исключения в момент выполнения программы. Причем, типы исключений могут быть самыми разными, в зависимости от возникшей ошибки. Например, если попробовать выполнить деление на ноль:

10/0

то отобразится тип исключения:

ZeroDivisionError

которое также относится к моменту выполнения программы, так как первые три строчки (до ошибки) были успешно выполнены.

Другой вариант ошибок, это, например, неверный синтаксис в программе. Если, допустим, прописать строчку:

**if** True

**print**("True")

то увидим исключение типа:

SyntaxError

которое возникает в момент компиляции текста программы, поэтому, ни одной строчки не было отображено в консоли. То есть, исключения можно разбить на две группы:

* исключения в момент исполнения;
* исключения при компиляции (до исполнения кода).

И при обработке исключений нас будет интересовать первая группа – момента исполнения, так как именно из-за них исполнение программы может внезапно остановиться и, как вы понимаете, это не лучший сценарий развития.

На первый взгляд может показаться, что исключения возникают по вине программиста, из-за того, что он что-то не учел. Иногда такое бывает, но не всегда. Классический пример обработки исключений момента выполнения программы – открытие файлов на чтение или запись. Мы с вами уже об этом говорили, когда открывается файл:

file = open("myfile.txt")

то программист не может заранее знать, есть ли такой файл на диске или нет. Поэтому, такую строчку нужно выполнять в специальном блоке try с обработкой исключения типа FileNotFoundError:

**try**:

    file = open("myfile2.txt")

**except** FileNotFoundError:

**print**("Невозможно открыть файл")

Или, при делении одного числа на другое:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ZeroDivisionError:

**print**("Делить на ноль нельзя!")

Здесь заранее нельзя сказать чему будет равно значение y, поэтому деление лучше поместить в блок try с обработкой исключения ZeroDivisionError.

Кстати, здесь же может появиться и другое исключение ValueError, если мы введем с клавиатуры не целые числа, а, например, строки.

Чтобы отловить и это второе исключение, можно прописать еще один блок except:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ZeroDivisionError:

**print**("Делить на ноль нельзя!")

**except** ValueError:

**print**("Ошибка типа данных")

И таких блоков except может быть сколько угодно.

Чтобы отловить в блоке except сразу несколько типов исключений, их можно указать в круглых скобках через запятую после ключевого слова except:

**except** (ZeroDivisionError, ValueError):

    res = "деление на ноль или нечисловое значение"

Но все же чаще используют раздельную запись, поэтому я ее верну.

Если мы хотим при возникновении ошибок взаимодействовать с объектами классов исключений, то это делается так:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ZeroDivisionError **as** z:

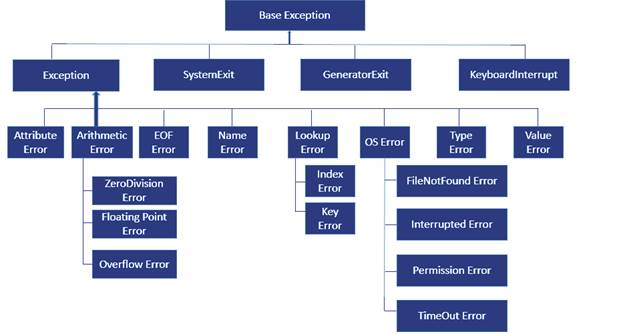
**print**(z)

**except** ValueError **as** z:

**print**(z)

То есть, после имени исключения ставится ключевое слово as и дальше переменная, которая будет ссылаться на объект класса ValueError, в котором хранится служебная информация о конкретной ошибке.

Разумеется, так обрабатывать можно только исключения на этапе исполнения. Про синтаксические ошибки мы сейчас не говорим. Так вот, существует определенная иерархия классов исключений, во главе которой стоит базовый класс BaseException:



Здесь приведен лишь ее фрагмент. Самих классов гораздо больше. Из этого рисунка хорошо видно, что большая часть классов наследуется от базового класса Exception и именно эта ветка нас будет интересовать.

На что влияет эта иерархия и как ее можно использовать при обработке исключений?

Например, мы видим обработку исключения деления на ноль, которое относится к базовому классу ArithmeticError, а он, в свою очередь, к базовому классу Exception.

Поэтому, вместо указания конкретного типа ZeroDivisionError можно указать любой из этих базовых классов:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ArithmeticError:

**print**("Делить на ноль нельзя!")

**except** ValueError:

**print**("Ошибка типа данных")

Но здесь есть один важный нюанс.

Если вместо ArithmeticError прописать класс Exception – общий для классов ArithmeticError и ValueError, то первым блоком exception будут отлавливаться все типовые исключения.

А блок с ValueError никогда выполнен не будет. Поэтому здесь есть одно простое правило. Сначала прописываются блоки со специализированными классами исключений, а затем с более общими (базовыми).

Если поменяем местами наши блоки except:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ValueError:

**print**("Ошибка типа данных")

**except** Exception:

**print**("Делить на ноль нельзя!")

То ValueError будет отлавливать свой тип исключения, а второй блок – все остальные, унаследованные от класса Exception.

И последнее, что я хочу отметить про блок except. Его можно прописывать без указания каких-либо классов, следующим образом:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except**:

**print**("Ошибка")

В этом случае он будет отлавливать все исключения, возникающие в блоке try.

На этом завершим первое знакомство с исключениями. Здесь я вас лишь хотел познакомить с блоками try и except, а также показать на что влияет иерархия классов исключений при их отслеживании.

## Задания на исключения

**1.** В программе вводятся в одну строчку через пробел некоторые данные, например:

"1 -5.6 2 abc 0 False 22.5 hello world"

Эти данные разбиваются по пробелу и представляются в виде списка строк:

lst\_in = input().split()

Ваша задача посчитать сумму всех целочисленных значений, присутствующих в списке lst\_in. Результат (сумму) вывести на экран. Реализовать обработку ошибок.

**Sample Input:**

8 11 abcd -7.5 2.0 -5

**Sample Output:**

14

lst\_in = input().split()

s = 0

for i in lst\_in:

try:

if type(int(i)) == int:

s += int(i)

except:

pass

print (s)

**Задание 2.** В программе вводятся в одну строчку через пробел некоторые данные, например:

"1 -5.6 True abc 0 23.56 hello"

Эти данные разбиваются по пробелу и представляются в виде списка строк:

lst\_in = input().split()

Ваша задача сформировать два новых списка **lst\_int и lst\_out** , в первом будут целые числа, во втором вещественные (тип float).

## Обработка исключений. Блоки finally и else

Мы продолжаем тему исключений. На прошлом занятии мы с вами познакомились с основами работы блоков try / except, а также увидели, как правильно использовать иерархию классов исключений для их корректной обработки.

Однако простой конструкции try / except часто бывает недостаточно для корректной обработки возникающих ошибок.

Еще на практике дополнительно прописывают необязательные блоки else и finally. Давайте посмотрим, как они работают и для чего нужны.

Итак, блок try поддерживает необязательный блок else, который выполняется при штатном выполнении кода внутри блока try, то есть, когда не произошло никаких ошибок. Например, его можно записать так:

**try**:

    x, y = map(int, input().split())

    res = x / y

**except** ZeroDivisionError **as** z:

**print**(z)

**except** ValueError **as** z:

**print**(z)

**else**:

**print**("Исключений не произошло")

Теперь, при запуске программы, вводя корректные числа, мы увидим сообщение «Исключений не произошло».

Если же возникает какое-либо исключение, то этот блок выполняться не будет.

Вторым необязательным блоком является блок finally, который, наоборот, выполняется всегда после блока try, вне зависимости произошла ошибка или нет:

**finally**:

**print**("Блок finally выполняется всегда")

Теперь, при запуске программы, мы будем видеть это сообщение вне зависимости от возникновения возможных ошибок.

И здесь часто возникает вопрос: зачем нужен этот блок, если он выполняется всегда после try? Мы с таким же успехом можем записать этот print сразу после этого блока и, вроде бы, все будет работать также?

В действительности, нет. Смотрите, если мы, например, уберем блок except с исключением ValueError, запустим программу и введем нечисловые значения, то, конечно, возникнет необработанное исключение, но при этом, блок finally все равно выполнился!

Этого не произошло бы, если просто записать print после try.

Но все же, в каких ситуация нам может понадобиться использовать блок finally? Классический пример – это работа с файлами.

Вначале, для открытия файла, нам следует прописать функцию open() в блоке try:

**try**:

    f = open("myfile.txt")

    f.write("hello")

**except** FileNotFoundError **as** z:

**print**(z)

**except**:

**print**("Другая ошибка")

Здесь ошибка произойдет после открытия файла в строчке f.write(), поэтому файл остается открытым. Это нехорошо.

Мы знаем, что любой файл нужно закрывать даже при возникновении ошибок. Как раз здесь может пригодиться блок finally:

**try**:

    f = open("myfile.txt")

    f.write("hello")

**except** FileNotFoundError **as** z:

**print**(z)

**except**:

**print**("Другая ошибка")

**finally**:

**if** f:

        f.close()

**print**("Файл закрыт")

Запустим программу и увидим, что файл закрывается (даже если убрать второй блок except).

Конечно, конкретно при работе с файлами такую громоздкую конструкцию заменяют более компактной с использованием файлового менеджера:

**try**:

**with** open("myfile.txt") **as** f:

        f.write("hello")

**except** FileNotFoundError **as** z:

**print**(z)

**except**:

**print**("Другая ошибка")

Который автоматически закрывает файл. Я привел первый вариант исключительно в учебных целях, чтобы вы увидели для чего в принципе можно использовать блок finally.

Следующая особенность работы блока finally связана с обработкой исключений внутри функций. Предположим, в функции вводятся два целых числа и возвращаются в виде кортежа:

**def** get\_values():

**try**:

        x, y = map(int, input().split())

**return** x, y

**except** ValueError **as** v:

**print**(v)

**return** 0, 0

**finally**:

**print**("finally выполняется до return")

x, y = get\_values()

**print**(x, y)

Здесь может возникнуть исключение ValueError, если были введены не целые числа. В этом случае возвращаются нули.

Также прописан блок finally с выводом сообщения «finally выполняется до return», чтобы мы убедились, что этот блок действительно выполняется до оператора return.

Запускаем программу, вводим два значения и вне зависимости от ошибок срабатывает блок finally до оператора return. Вот этот момент также нужно знать при реализации этого блока.

## Вложенные блоки try / except

Наконец, блоки try / except можно вкладывать один в другой, например, так:

**try**:

    x, y = map(int, input().split())

**try**:

        res = x / y

**except** ZeroDivisionError:

**print**("Деление на ноль")

**except** ValueError **as** z:

**print**("Ошибка ValueError")

Или, можно внутренний блок try вынести в функцию:

**def** div(a, b):

**try**:

**return** x / y

**except** ZeroDivisionError:

**return** "Деление на ноль"

А, затем, вызвать в первом блоке try:

**try**:

    x, y = map(int, input().split())

    res = div(x, y)

**except** ValueError **as** z:

**print**("Ошибка ValueError")

**print**(res)

Все эти комбинации, конечно же, допустимы и используются на практике. На следующем занятии мы продолжим эту тему и детальнее рассмотрим механизм возникновения и обработки исключений.

Практические задания

**Задание 1.** В программе вводятся два значения в одну строчку через пробел. Значениями могут быть числа, слова, булевы величины (True/False).

Необходимо прочитать эти значения из входного потока. Если оба значения являются числами, то вычислить их сумму, иначе соединить их в одну строку с помощью оператора + (конкатенации строк).

Результат вывести на экран (в блоке finally).

P.S. Реализовать программу с использованием блоков try/except/finally.

**Sample Input:**

8 11

**Sample Output:**

19

a, b = map(str, input().split())

try:

s = int(a) + int(b)

except:

try:

s = float(a) + float(b)

except:

s = a + b

finally:

print(s)

# Инструкция raise и пользовательские исключения

Мы продолжаем тему исключений. Во всех наших предыдущих примерах исключение возникало в результате ошибочных ситуаций во время работы программы, например, деления на ноль:

**print**("Куда ты скачешь, гордый конь,")

**print**("И где опустишь ты копыта?")

**print**("О мощный властелин судьбы!")

1/0

**print**("Не так ли ты над самой бездной")

**print**("На высоте, уздой железной")

**print**("Россию поднял на дыбы?")

Но как эта операция деления формирует само исключение? Для этого в языке Python имеется конструкция (оператор)

raise

которая и порождает указанные типы исключений. В самом простом варианте, мы можем вместо деления на ноль записать этот оператор и указать тип исключения ZeroDivisionError:

**raise** ZeroDivisionError("Деление на ноль")

Результат выполнения программы будет тем же – она остановится на конструкции raise. Только сообщение об ошибке теперь будет на русском языке – та строка, что мы указали при формировании объекта класса ZeroDivisionError. То есть, после оператора raise мы можем прописывать нужный нам класс исключения с собственными параметрами. Также можно просто указывать класс, не прописывая каких-либо параметров:

**raise** ZeroDivisionError

Здесь у нас также создается экземпляр, но без параметров. Раз это так, значит, можно заранее создать экземпляр класса:

e = ZeroDivisionError("Деление на ноль")

а, затем, сгенерировать это исключение:

**raise** e

Вообще, мы можем использовать любой класс в качестве исключения, унаследованного от базового класса:

BaseException

Например, если просто указать строку после оператора raise:

**raise** "деление на ноль"

то интерпретатор Python как раз это нам и укажет:

TypeError: exceptions must derive from BaseException

То есть, после raise должен находиться экземпляр класса исключения, а не какой-то произвольный объект.

Когда нам может понадобиться оператор raise? И разве сам язык Python не может генерировать нужные исключения при возникновении ошибок? Часто именно так и происходит. Например, если мы будем делать некорректные операции, вроде:

1 + "2"

[1, 2, 3][4]

то автоматически возникают ошибки заданного типа. Но прописать исключения на все случаи жизни невозможно.

# Оператор assert

Иногда по-настоящему полезное функциональное средство языка привлекает меньше внимания, чем оно того заслуживает. По некоторым причинам это именно то, что произошло со встроенной в Python инструкцией assert

x = 10

y = 5

**assert** x < y, "x должно быть меньше y"

В этом примере, поскольку

x

 не меньше

y

, генерируется исключение

AssertionError

 с сообщением «x должно быть меньше y».

## Когда использовать assert?

Оператор

assert

 обычно используется для отладки кода. Он позволяет быстро проверить предположения о вашем коде и выявить ошибки. Вам не следует использовать

assert

 для обработки ошибок в продакшен-коде, поскольку его можно отключить глобально во всем коде, используя ключ

-O

 при запуске программы.

**def** test\_add():

**assert** add(2, 3) == 5

## Использование assert в функциях

Вот простой пример, чтобы дать вам понять, где инструкции assert могут пригодиться. Предположим, вы создаете интернет-магазин с помощью Python. Вы работаете над добавлением в систему функциональности скидочного купона, и в итоге пишете следующую функцию apply\_discount:

**def** apply\_discount(product, discount):   
 price = int(product[**'цена'**]) \* (1.0 - discount)  
 **assert** 0 <= price <= product[**'цена'**]  
 **return** price  
  
  
shoes = {**'имя'**: **'Модные туфли'**, **'цена'**: 14900}  
res = apply\_discount(shoes, 0.25)  
print(res) *# 11175*

Как видите программа останавливается с исключением AssertionError. Вы также можете видеть отчет об обратной трассировке этого исключения и то, как он указывает на точную строчку исходного кода, содержащую вызвавшее сбой утверждение.

Это значительно ускорит процесс отладки и в дальнейшем сделает ваши программы удобнее в поддержке. А в этом, заключается сила assert!

Оператор assert в Python — это удобный инструмент для отладки и тестирования. Он позволяет быстро проверять предположения о вашем коде и выявлять ошибки. Однако, имейте в виду, что его можно отключить, поэтому assert не должен использоваться для обработки ошибок в коде, предназначенном для продакшена.

# Менеджеры контекстов. Оператор with

На этом занятии речь пойдет о, так называемых, менеджерах контекста. С менеджером контекста мы с вами уже сталкивались, когда рассматривали работу с файлами. Дело в том, что когда открываем файловый поток с помощью функции open(), то в конце работы с ним, его желательно закрыть с помощью метода close(). Если реализовать эту логику через конструкцию try/except/finally, то получим примерно вот такой текст программы:

fp = None

**try**:

    fp = open("myfile.txt")

**for** t **in** fp:

**print**(t)

**except** Exception **as** e:

**print**(e)

**finally**:

**if** fp **is** **not** None:

        fp.close()

Благодаря блоку finally мы гарантированно закрываем файл, даже если в блоке try возникло какое-либо исключение. Но, если воспользоваться файловым менеджером контекста, то программа принимает вид:

**try**:

**with** open("myfile.txt") **as** fp:

**for** t **in** fp:

**print**(t)

**except** Exception **as** e:

**print**(e)

Видите, она фактически отличается от первой только тем, что не реализует блок finally. Но почему? Нам же нужно закрыть файл после работы с ним? Да, и файловый менеджер контекста делает это автоматически вне зависимости от возникновения возможных исключений. Как же в деталях это происходит? Для этого мы вначале посмотрим, как вообще создаются свои собственные менеджеры контекстов.

# Распространение исключений (propagation exceptions)

На предыдущих занятиях мы с вами в целом рассмотрели работу блоков try / except / finally / else на примере относительно простых программ. В реальности, программы куда сложнее, содержат вызовы различных функций и даже могут использовать многопоточную реализацию. Давайте посмотрим, как будут выглядеть сообщения об ошибках при использовании функций.

Для этого объявим вначале функцию:

**def** func1():

**return** 1/0

и вызовем ее также после третьего print():

**print**("Я к вам пишу – чего же боле?")

**print**("Что я могу еще сказать?")

**print**("Теперь, я знаю, в вашей воле")

func1()

**print**("Меня презреньем наказать.")

**print**("Но вы, к моей несчастной доле")

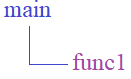
**print**("Хоть каплю жалости храня,")

**print**("Вы не оставите меня.")

При попытке выполнить эту программу, Python выдаст следующие строчки:

Traceback (most recent call last):  
  File "D:/Python/Projects/p\_course/ex1.py", line 7, in <module>  
    func1()  
  File "D:/Python/Projects/p\_course/ex1.py", line 2, in func1  
    return 1/0  
ZeroDivisionError: division by zero

Что примечательного в этом сообщении? Мы здесь видим указание об ошибке типа ZeroDivisionError для строчки 2 и строчки 7. В строчке 7 происходит вызов функции, которая выдает исключение, а в строчке 2 записано непосредственное деление на ноль. Почему строчки отображаются именно в таком порядке? Здесь нам нужно вспомнить, что при вызове функций формируется стек их вызова. И в нашем случае, этот стек можно представить, следующим образом:



Далее, возникшее исключение на уровне func1, последовательно распространяется по всему стеку вызова, доходя до верхнего уровня main. Именно этот стек распространения исключения и выдает интерпретатор языка Python.

Если, к примеру, добавить в программу еще один вложенный вызов функции:

**def** func2():

**return** 1/0

**def** func1():

**return** func2()

То в консоли появятся уже три строчки об ошибке, так как стек вызова стал длиннее:

Traceback (most recent call last):  
  File "D:/Python/Projects/p\_course/ex1.py", line 10, in <module>  
    func1()  
  File "D:/Python/Projects/p\_course/ex1.py", line 5, in func1  
    return func2()  
  File "D:/Python/Projects/p\_course/ex1.py", line 2, in func2  
    return 1/0  
ZeroDivisionError: division by zero

Все это пример того, как исключение, зародившееся на одном из уровней стека вызова, постепенно поднимается на самый верх. Это называется распространением исключений. По-английски:

propagation exceptions

Причем, обработать (перехватывать) исключение можно на любом уровне этого стека. Например, сделаем это на самом верхнем. Поместим вызов функции в блок try/except:

**try**:

    func1()

**except**:

**print**("Error for func1")

Теперь, все исключения для func1, которые дойдут до верхнего уровня, будут обработаны и программа выполнится в полном объеме.

Но, мы также можем обрабатывать исключения и на более глубоких уровнях, например, непосредственно в функции func2() при выполнении деления на ноль:

**def** func2():

**try**:

**return** 1/0

**except**:

**return** "-- деление на ноль --"

А на вершине стека, в глобальной области, просто вызовем функцию func1:

**print**("Я к вам пишу – чего же боле?")

**print**("Что я могу еще сказать?")

**print**("Теперь, я знаю, в вашей воле")

**print**(func1())

**print**("Меня презреньем наказать.")

**print**("Но вы, к моей несчастной доле")

**print**("Хоть каплю жалости храня,")

**print**("Вы не оставите меня.")

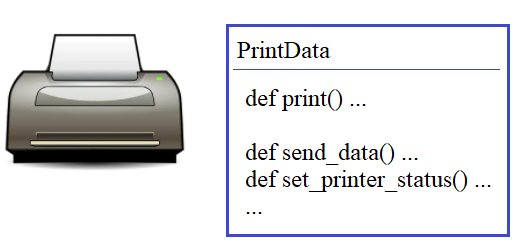
При выполнении программы также не будет возникать никаких ошибок, так как деление на ноль обрабатывается непосредственно в функции func2 и исключение уже не всплывает на более высокие уровни.

То есть, мы можем обрабатывать исключения на разных уровнях стека вызова, что очень удобно. Этот механизм обработки исключений позволяет программистам писать независимый, модульный, красивый код.

В критических функциях достаточно генерировать исключения, а их обработку выполнять на другом, более глобальном уровне.

Например, создается класс для печати данных на принтере. Тогда все ошибки, связанные с принтером (нет бумаги, нет подключения, не тот режим печати и т.п.) можно обрабатывать единым образом на верхнем, глобальном уровне.

А нижние уровни только сигнализируют о проблемах и не более того. В результате, получается разделение ролей: функции нижних уровней сосредоточены исключительно на обработке данных, а функции верхних – на формировании сервисной информации для пользователя.



Вот так, благодаря механизму распространения и обработки исключений можно создавать гибкий и безопасный программный код.

**Задание 3.** Объявите функцию с сигнатурой:

def input\_int\_numbers(): ...

которая бы считывала строку из введенных целых чисел, записанных через пробел, и возвращала кортеж из введенных чисел (в виде целых чисел, а не строк).

Если хотя бы одно значение не является целым числом, то генерировать исключение, командой:

raise TypeError('все числа должны быть целыми')

Вызовите эту функцию в цикле до тех пор, пока пользователь не введет в строке все целочисленные значения (то есть, цикл завершается, когда функция отработает штатно, без генерации исключения).

Выведите на экран прочитанные значения, записанные в виде строки через пробел.

**Sample Input:**

1 abc 3 5

2.4 -5 4 3 2

0 -5 8 11

1 2 3 4

**Sample Output:**

0 -5 8 11