# Исключения

**Исключения** — один из двух основных типов ошибок в программировании. В отличие от синтаксических ошибок, которые возникают во время написания, исключения могут появиться во время выполнения программы.

В программировании же исключения приводят к полному прекращению или к неверному выполнению программ. Для избегания прекращения работы или получения дополнительной информации об ошибке используют конструкции обработки исключений.

## Типы исключений

У каждого исключения есть собственный тип, который определяется тем, какая ошибка его вызвала, и дает возможность по-разному реагировать на различные виды ошибок.

Рассмотрим некоторые типы исключений, которые могут возникнуть во время прохождения этого курса:

BaseException — базовый тип, из которого происходят все остальные, в том числе системные:

Exception — базовый тип для «стандартных» и пользовательских исключений:

ArithmeticError — арифметическая ошибка:

OverflowError — возникает, когда результат арифметической операции слишком велик для представления;

На сегодняшний день практически невозможно получить OverflowError используя стандартную библиотеку Python, т.к. целые числа имеют динамическую длину и скорее вызовут MemoryError, а числа с плавающей запятой при пересечении граничных значений заменяются на 0.0, либо inf. Однако, эта ошибка может быть получена при работе со сторонними библиотеками или модулями, написанными на языках программирования с жесткой типизацией (числовые типы имеют четкие границы, выход за которые — ошибка).

ZeroDivisionError — деление на ноль.

ImportError — импортировать модуль или его атрибут не удалось;

LookupError — некорректный индекс или ключ:

IndexError — индекс не входит в диапазон элементов;

KeyError — несуществующий ключ (например, в словаре).

NameError — не найдено переменной с указанным именем;

RuntimeError — возникает, когда исключение не попадает ни под одну из других категорий;

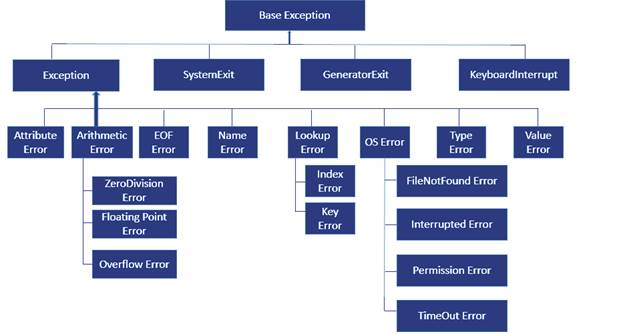
SyntaxError — синтаксическая ошибка:

IndentationError — неправильные отступы:

TabError — смешивание в отступах табуляции и пробелов.

TypeError — операция применена к объекту несоответствующего типа;

ValueError — функция получает аргумент правильного типа, но некорректного значения.



Список предусмотренных самим языком программирования («встроенных») типов исключений гораздо больше и приведен в [документации к языку.](https://docs.python.org/3/library/exceptions.html) Кроме того, пользователи сами могут определять новые типы исключений в своих программах и библиотеках.

## Перехват исключений

простой пример, деление на ноль:

5/0

приведет к ошибке ZeroDivisionError.

Или такие операции:

int("12abc") *#ValueError*

"2"+5 *#TypeError*

Давайте вначале предположим, что у нас имеется вот такая программа для вычисления деления двух целых чисел:

x = input("x: ")

y = input("y: ")

x = int(x)

y = int(y)

res = x/y

**print**(res)

Если мы будем вводить целые числа, то она работает корректно, но если ввести в качестве игрека ноль или не числовую величину, то возникнут соответствующие исключения.

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.

Так вот, в Python есть механизм для отслеживания (отлавливания) таких моментов в работе программы. Для этого критический по выполнению код следует поместить в блок try:

**try**:

    x = int(x)

    y = int(y)

    res = x/y

**except** ZeroDivisionError:

    res = "деление на ноль"

**print**(res)

А после него записать except и название исключения, которое требуется отследить. В данном случае мы отлавливаем деление на ноль, т.е. исключение с именем ZeroDivisionError.

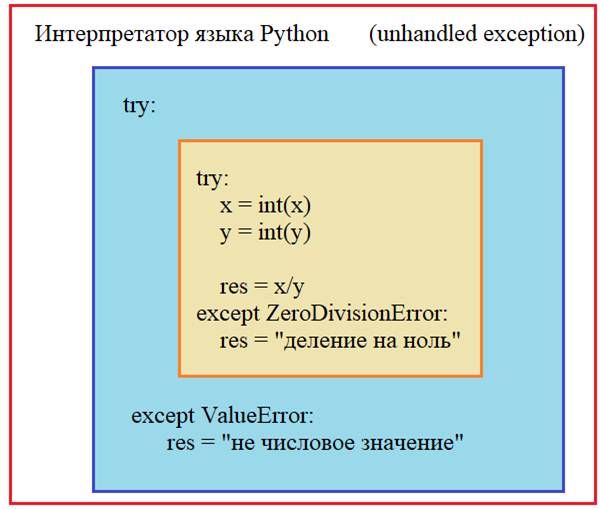
Давайте запустим эту программу и посмотрим как она теперь будет работать. Вводим 1 и 0, программа теперь не завершается аварийно и в консоли видим сообщение «деление на ноль».

Более подробно блок try except работает так. Сначала идет выполнение программы внутри блока try. Если все проходит в штатном режиме, то выполнение доходит до блока except и он пропускается, не выполняется. И далее уже вызывается функция print и печатается полученный результат.

Если же в процессе выполнения программы блока try возникает какое-либо исключение (любое), то выполнение программы прерывается и управление передается блоку except с соответствующим именем исключения.

Если нужное имя в блоке except отсутствует, то исключение переходит на более высокий уровень (в данном случае к среде выполнения Python).

И в случае отсутствия обработчика исключение считается **необработанным** (unhandled exception) и программа завершается аварийно.



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

**except** (ZeroDivisionError, ValueError):

    res = "деление на ноль или нечисловое значение"

Или, для раздельной обработки, в разных блоках except:

**except** ZeroDivisionError:

    res = "деление на ноль"

**except** ValueError:

    res = "одно из введенных значений не число"

Если мы хотим при возникновении ошибок указывать служебное сообщение, записанное в соответствующих исключениях, то это делается так:

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

    res = z

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

    res = v

То есть, после имени исключения (в действительности, это класс, но мы о них пока еще не говорили, поэтому будем воспринимать его просто как имя) ставится ключевое слово as и дальше переменная, которая будет ссылаться на класс ValueError, в котором хранится служебное сообщение о конкретной ошибке. Выводя его с помощью print, мы в консоли видим это сообщение.

Далее, 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:

**else**:

**print**("Исключений не произошло")

Теперь, при запуске программы, вводя корректные числа, мы увидим это сообщение. Если же возникает любое исключение, то этот блок не выполняется.

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

**Блок «finally» используется** в том случае, когда нам необходимо гарантировать окончание работы программы. Например, мы можем быть подключены к удаленному серверу по сети, работать с файлом или графическим интерфейсом пользователя (GUI). Во всех этих случаях мы должны освободить используемые программой ресурсы до того, как программа завершиться вне зависимости от того, успешно она выполнилась или нет. Такие действия (освобождение ресурсов) выполняются в блоке «finally».

**finally**:

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

Теперь, при запуске программы, мы всегда будем видеть это сообщение. 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.

Или, вот такой пример:

**def** getValues():

    x = input("x: ")

    y = input("y: ")

**try**:

        x = int(x)

        y = int(y)

**return** x,y

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

**print**(v)

**return** 0,0

**finally**:

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

x,y = getValues()

**print**(x,y)

Мы создаем функцию для ввода двух целых чисел и в блоке finally выводим сообщение. Этим сообщением мы покажем, что этот блок будет выполняться до операторов return, присутствующих в функции.

Наконец, мы можем прописывать except без указания имени класса исключения:

x = input("x: ")

y = input("y: ")

**try**:

    x = int(x)

    y = int(y)

    res = x/y

**except**:

**print**("Произошло исключение")

**else**:

**print**("Исключений не произошло")

**finally**:

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

**print**(res)

В этом случае оператор print("Произошло исключение") будет выполняться при любых ошибках в блоке try, но само исключение обработано не будет, т.е. программа завершится аварийно.

Вот так работает базовый механизм обработки исключений в Python. Конечно, мы здесь рассмотрели только отслеживание стандартных (встроенных) классов исключений. В действительности, можно добавлять свои и строить довольно гибкие программы по обработке исключительных ситуаций. Но для углубления в этот материал, нужно вначале изучить основы ООП в Python. Поэтому мы пока на этом остановимся, и в качестве заданий для самоподготовки попробуйте сделать следующие программы:

## Задания для самоподготовки

1. Напишите программу ввода трех чисел, нахождения их среднеарифметического и преобразования этого значения обратно в строку. Реализовать обработку возможных исключений при таком преобразовании.

### Особенности работы с try … except

### *Вызов исключений*

### 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. Раз это так, значит, можно заранее создать экземпляр класса:

e = ZeroDivisionError("Деление на ноль")

а, затем, сгенерировать это исключение:

**raise** e

***Комбинирование общего и конкретных блоков exept***

Мы так же можем комбинировать использование блоков «exept» для конкретных типов исключений и общий блок «exept». В таком случае общий блок должен быть размещен последним, он выполниться если тип полученного исключения не совпадает ни с одним из типов блоков «exept»:

|  |
| --- |
|  |
| try: |
| f = open("Some\_file.txt") |
| except ZeroDivisionError: |
| print("You cannot divide the delivery into 0 parts") |
| except:  print("Hmm... Something went wrong") |
|  |

В этом примере мы пытаемся открыть файл «Some\_file. txt», который не был создан ранее, что приводит к исключению типа «FileNotFoundError», однако для этого типа не предусмотрен отдельный блок «exept» и оно обрабатывается общим блоком.

***Порядок размещения блоков exept***

Одной из главных особенностей работы с «try … except» является правильное расположение блоков «except». Из таблицы типов исключений, приведенной в соответствующем разделе, видно, что «ZeroDivisionError» является подтипом «ArithmeticError», а «ValueError» — подтипом «Exception». Полную структуру иерархии встроенных типов исключений можно посмотреть в документации к языку.

## 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)

## Создание пользовательских исключений

Если мы посмотрим на иерархию классов исключений языка Python, то здесь во главе стоит базовый класс BaseException:

Остальные классы наследуются от него и имеют строгую специализацию, кроме, разве что, класса **Exception,** который является общим для большого разнообразия типов исключений в момент выполнения программы.

Так почему же мы выбрали класс Exception, а не BaseException? Дело в том, что классы SystemExit, GeneratorExit и KeyboardInterrupt являются весьма специфичными и, обычно, они не используются при обработке собственных исключений.

Поэтому, целесообразно выбирать именно класс Exception для формирования новых собственных классов исключений. Что мы сейчас и сделаем.

Итак, чтобы сформировать свой новый тип исключения, нужно прописать класс, который рекомендуется наследовать от класса Exception. В самом простом варианте достаточно просто описать иерархию:

**class** ExceptionPrintSendData(Exception):

    """Класс исключения при отправке данных принтеру"""

И далее в программе использовать этот новый класс:

**def** send\_data(self, data):

**if** **not** self.send\_to\_print(data):

**raise** ExceptionPrintSendData("принтер не отвечает")

**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.

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.