# **Инструкция raise и пользовательские исключения**

Курс по Python ООП: <https://stepik.org/a/116336>

[Смотреть материал на видео](https://www.youtube.com/watch?v=XR-16WMXZOY&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

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. И если в качестве примера взять все тот же класс печати данных:

То, в частности, метод send\_data() может генерировать свое исключение, если по каким-то причинам данные не были отправлены принтеру. В качестве демонстрации я приведу гипотетический класс PrintData для работы с принтером:

**class** PrintData:     **def** **print**(self, data):         self.send\_data(data)         **print**(f"печать: {str(data)}")       **def** send\_data(self, data):         **if** **not** self.send\_to\_print(data):             **raise** Exception("принтер не отвечает")       **def** send\_to\_print(self, data):         **return** False

Как раз здесь мы генерируем исключение, если данные не могут быть отправлены принтеру. Затем, это исключение может быть обработано на любом уровне стека вызова. Например, если далее создать экземпляр этого класса и вызвать метод print():

p = PrintData() p.**print**("123")

То мы увидим сформированное нами исключение. Как вы понимаете, в язык Python не встроена по умолчанию возможность генерации исключения при взаимодействии с принтером. Это приходится делать уже самому программисту с помощью оператора raise. Вот для этого он и нужен.

## **Создание пользовательских исключений**

В нашем гипотетическом классе PrintData исключение генерируется с помощью класса Exception. Почему именно он? 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")

Соответственно, ниже в программе, мы можем обработать этот тип ошибки, просто указав имя нашего нового класса:

p = PrintData()   **try**:     p.**print**("123")**except** ExceptionPrintSendData:     **print**("Ошибка печати")

Видите, как это здорово! Мы создали новый тип исключения, просто прописав новый класс. И благодаря этому можем отличить ошибку передачи данных принтеру от каких-либо других ошибок.

Разумеется, мы поймаем эту же ошибку, если укажем базовый класс Exception, но пропустим, если указать какой-либо другой независимый класс исключения, иерархически не связанный с нашим, например, ArithmeticError.

Кроме того, мы можем расширить функционал класса ExceptionPrintSendData. Давайте добавим в него конструктор. Он прописывается для произвольного числа аргументов:

**class** ExceptionPrintSendData(Exception):     """Класс исключения при отправке данных принтеру"""     **def** \_\_init\_\_(self, \*args):         self.message = args[0] **if** args **else** None

А также магически метод\_\_str\_\_ для представления ошибки в консоли:

**def** \_\_str\_\_(self):         **return** f"Ошибка: {self.message}"

Если теперь убрать блок try/except и вызвать метод print(), то увидим наш вариант отображения ошибки в консоли:

p = PrintData() p.**print**("123")

Это лишь пример расширения функционала класса исключения. В каждом конкретном случае программист может написать любую свою реализацию.

Наконец, пользовательские классы исключений дают возможность создавать свою иерархию исключений. В частности, в нашем примере, можно прописать общий класс исключений для принтера ExceptionPrint:

**class** ExceptionPrint(Exception):     """Общий класс исключения принтера"""

А, затем, остальные, более конкретные типы наследовать от него:

**class** ExceptionPrintSendData(ExceptionPrint):     """Класс исключения при отправке данных принтеру"""

В результате, в блоке except мы можем отлавливать как конкретные типы ошибок, так и общие, связанные с принтером:

p = PrintData()   **try**:     p.**print**("123")**except** ExceptionPrintSendData **as** e:     **print**(e)**except** ExceptionPrint:     **print**("Ошибка печати")

Такой подход дает гибкий механизм обработки собственных типов исключений, благодаря чему, программа становится более понятной и структурированной.

Курс по Python ООП: <https://stepik.org/a/116336>