Лекция 5. Назначение и область применения исключений

**Оглавление**

[**Введение** 3](#_Toc188541623)

[**1.** **Принципы работы исключений** 4](#_Toc188541624)

[**2.** **Обработка исключений** 7](#_Toc188541625)

[**Обработка исключения** 7](#_Toc188541626)

[**Обработка множественных исключений** 9](#_Toc188541627)

[**3.** **Пользовательские исключения** 11](#_Toc188541628)

[**Заключение** 12](#_Toc188541629)

[**Список литературы** 13](#_Toc188541630)

**Назначение и область применения исключений**

**Цель:** формирование знаний по обработке исключений, возникающих при выполнении программного кода.

**План лекции:**Введение.1. Принципы работы исключений  
2. Обработка исключений.  
3. Пользовательские исключенияЗаключение.Список литературы.

# **Введение**

При разработке программ на любом языке программирования рано или поздно возникает необходимость обрабатывать ошибки и неожиданные ситуации. В Python для этих целей используется механизм исключений. Исключения позволяют программисту предсказать и обработать ошибки, которые могут возникнуть во время выполнения программы, не нарушая ее основного потока исполнения.

Представьте, что ваша программа пытается открыть файл, которого не существует, или делит число на ноль. В подобных ситуациях возникает ошибка, которая может привести к аварийному завершению программы. Однако, с помощью исключений, мы можем предвидеть такие ошибки и корректно их обрабатывать, обеспечивая стабильную работу программы.

Основные цели использования исключений включают:

* Повышение надежности программы за счет обработки ошибок.
* Упрощение отладки и поиска проблем.
* Сокращение количества ошибок, возникающих в уже работающей программе.

В Python используется конструкция try...except, которая позволяет «поймать» ошибку и выполнить определенные действия в ответ на эту ошибку. Также существуют конструкции else и finally, которые расширяют возможности обработки исключений, делая ее еще более гибкой.

В ходе лекции мы рассмотрим:

* Различные типы встроенных исключений в Python.
* Различные конструкции обработки исключений (try, except, else, finally).
* Как создавать свои собственные исключения.
* Практические примеры и рекомендации по использованию исключений для повышения надежности и читаемости кода.

# **Принципы работы исключений**

**Исключения** – это события, возникающие во время выполнения программы, которые могут привести к ошибкам или аномальному завершению программы. Они обычно возникают из-за некорректного кода или непредвиденных ситуаций, таких как деление на ноль, доступ к несуществующему файлу или использование неопределенной переменной.

Исключения (exceptions) в Python — это механизм обработки ошибок во время выполнения программы. Они позволяют программе продолжить работу после обнаружения ошибки, а не завершаться аварийно. В Python есть встроенные исключения, которые обрабатывают большинство типовых ошибок.

Рассмотрим иерархию встроенных в Python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для Python 3.3 и новее, в более ранних версиях есть незначительные изменения:

* **BaseException** - базовое исключение, от которого берут начало все остальные.
  + **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
  + **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
  + **GeneratorExit** - порождается при вызове метода close объекта generator.
  + **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
    - **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
    - **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.
    - **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**
      * **ChildProcessError** - неудача при операции с дочерним процессом.
      * **ConnectionError** - базовый класс для исключений, связанных с подключениями.
        + **BrokenPipeError**
        + **ConnectionAbortedError**
        + **ConnectionRefusedError**
        + **ConnectionResetError**
      * **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.
    - **SyntaxError** - синтаксическая ошибка.
      * **IndentationError** - неправильные отступы.
        + **TabError** - смешивание в отступах табуляции и пробелов.
    - **SystemError** - внутренняя ошибка.
    - **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** - предупреждение.

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try – except** и более сложные.

# **Обработка исключений**

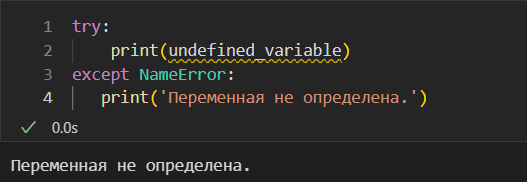
## **Обработка исключения**

Обработка исключений — это процесс написания кода для перехвата и обработки ошибок или исключений, которые могут возникать при выполнении программы. Это позволяет разработчикам создавать надёжные программы, которые продолжают работать даже при возникновении неожиданных событий или ошибок. Без системы обработки исключений подобное обычно приводит к фатальным сбоям.

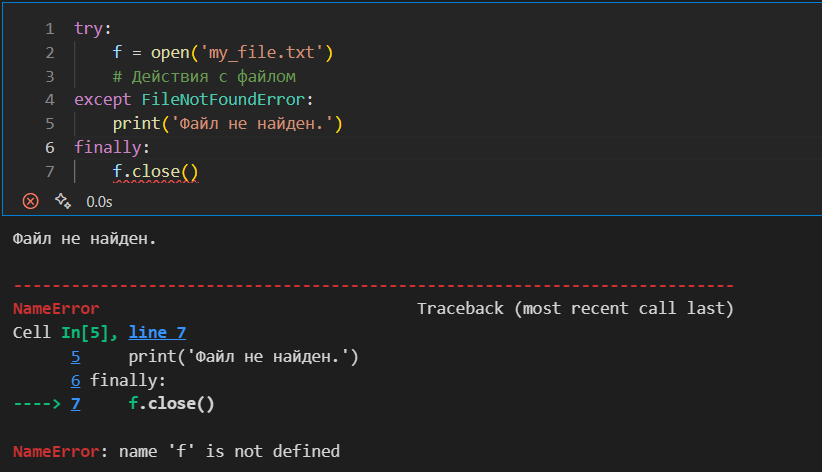
Когда возникают исключения — Python выполняет поиск подходящего обработчика исключений. После этого, если обработчик будет найден, выполняется его код, в котором предпринимаются уместные действия. Это может быть логирование данных, вывод сообщения, попытка восстановить работу программы после возникновения ошибки. В целом можно сказать, что обработка исключения помогает повысить надёжность Python-приложений, улучшает возможности по их поддержке, облегчает их отладку.

Для обработки исключений используются блоки try, except, finally, else и raise:

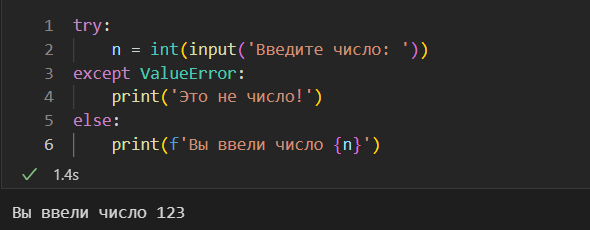
* С помощью блока try Python проверяет код на наличие исключений. Если в try встречается ошибка, выполнение переходит к первому блоку except.



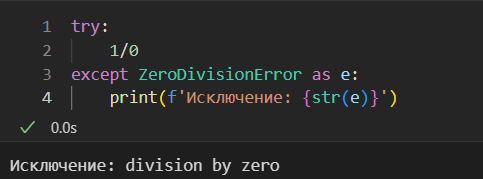
* В блоке except содержится код, который будет выполняться, если в блоке try нашлась ошибка.
* В finally помещают код, который будет выполняться независимо от того, была ли найдена ошибка или нет. Часто этот блок используют для работы с файлами, чтобы закрыть документ.



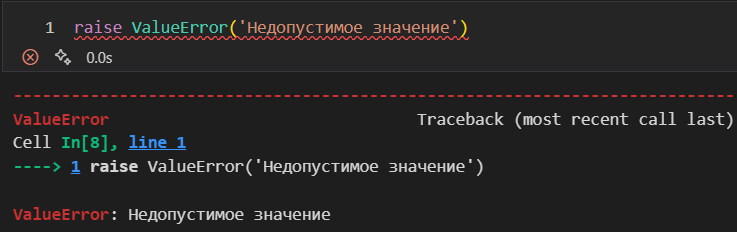
* Код в else выполняется, если try не нашёл исключений.



* Ключевое слово as при обработке ошибок используется для присвоения исключению переменной. К примеру, напишем собственное исключение для обработки деления на ноль. Ошибку назовём ZeroDivisionError и присвоим переменной e. Теперь к ней можно получить доступ для печати названия ошибки в консоль.



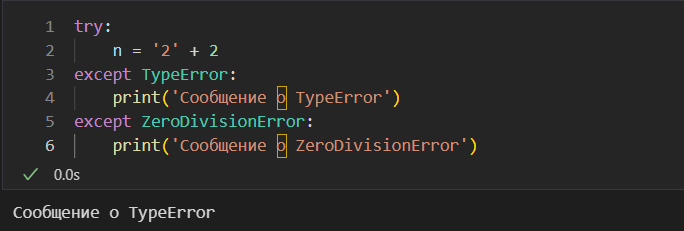
* Команда raise в Python используется для принудительного вызова исключения. Это может быть полезно, если мы столкнулись с условием, которое должно остановить выполнение программы или вызвать ошибку.



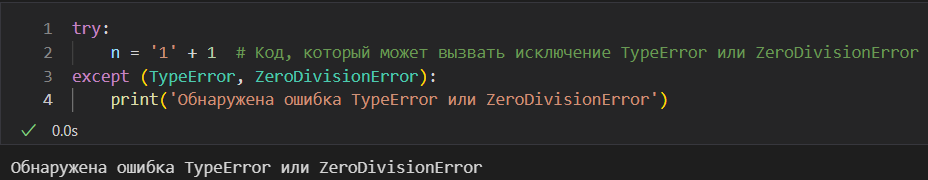
## **Обработка множественных исключений**

С помощью блоков except можно обрабатывать несколько исключений разными способами. Каждый except соответствует определённому типу ошибки.

Например, если в блоке try происходит исключение TypeError, будет выполнен первый блок except, и аналогично для ZeroDivisionError:



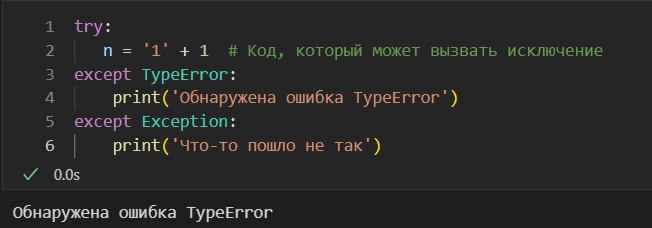
Иногда возникает необходимость обрабатывать несколько типов исключений одинаковым образом. В этом случае можно перечислить эти исключения в одном блоке except, разделив их запятыми:



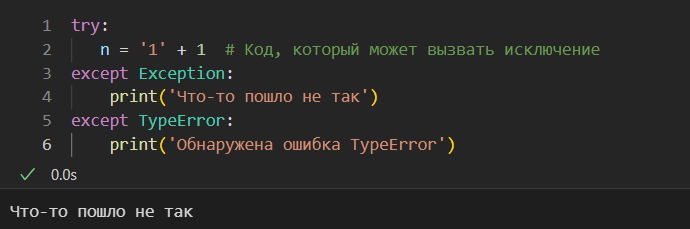
Блок except без указания конкретного типа исключения будет обрабатывать все исключения, которые не были обработаны в предыдущих блоках except, в том числе прерывание с клавиатуры, системный выход и другое.

Такая форма конструкции практически не используется. Вместо этого разработчики предпочитают except Exception. Так можно сначала обработать конкретные исключения, а потом уже всё остальное.

К примеру, в первом блоке except обработаем исключение TypeError — будем выводить в консоль сообщение Обнаружена ошибка TypeError. Второй блок except будет отлавливать остальные исключения и выводить «Что-то пошло не так»:



Начинать обработку следует с более узких классов исключений, например TypeError. Если начать с более широкого класса, такого как Exception, то всегда будет срабатывать первый блок except:

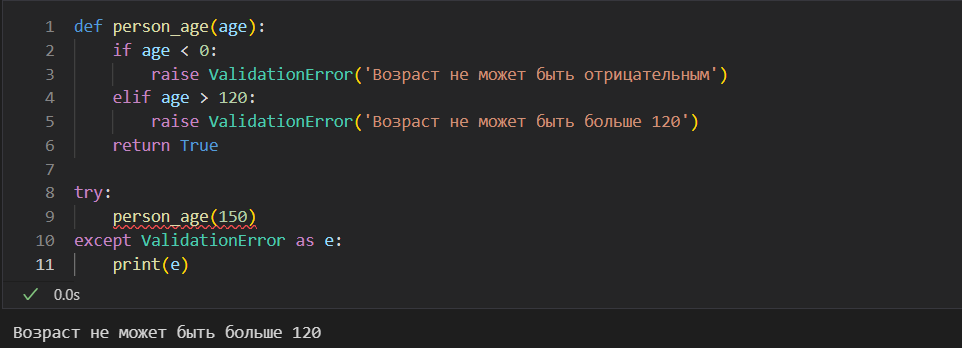


# **Пользовательские исключения**

Иногда надо реализовать собственный обработчик ошибок с помощью исключений. Это делает код более безопасным и поддерживаемым. Для создания собственного исключения достаточно определить новый класс, который наследуется от базового класса Exception или от любого другого встроенного исключения:



В примере выше ValidationError — исключение, которое не делает ничего, кроме наследования поведения стандартного исключения Exception.



В этом примере функция person\_age использует созданное нами исключение ValidationError для проверки корректности введённого возраста. Вызов функции с некорректными данными приводит к генерации исключения ValidationError, которое затем можно перехватить и обработать.

Исключения в Python полезны для написания отказоустойчивого кода. Они позволяют обрабатывать ошибки при выполнении программы и гарантировать, что код продолжит свою работу даже в неожиданных ситуациях. Однако важно понимать, что не все ошибки можно или нужно обрабатывать, — иногда лучше позволить программе завершиться с ошибкой, чтобы можно было быстро узнать о проблеме и исправить её. С другой стороны невозможно исправить ошибки, связанные с действиями пользователей при вводе данных или любом другом взаимодействии с программой.

# **Заключение**

Исключения и их обработка в Python являются важными инструментами для создания устойчивых и надежных программ. В ходе этой лекции мы узнали, как исключения помогают нам справляться с непредвиденными ситуациями и ошибками, которые могут возникнуть во время выполнения программы. Мы рассмотрели основные типы встроенных исключений, а также научились создавать свои собственные исключения для более точного контроля над ошибками.

Конструкция try...except позволяет перехватывать и обрабатывать ошибки, не нарушая основного потока программы. Мы также познакомились с расширенными возможностями обработки исключений, такими как блоки else и finally, которые делают обработку ошибок еще более гибкой и эффективной.

Использование исключений в Python позволяет повысить надежность кода, облегчить процесс отладки и обеспечить более предсказуемую работу программ. Теперь у вас есть инструменты и знания, которые помогут вам создавать более устойчивые и удобные для пользователя программы, минимизируя влияние ошибок на их выполнение.

Помните, что обработка исключений – это не просто способ справляться с ошибками, но и важная часть процесса проектирования программного обеспечения. Уделяйте внимание качественной обработке ошибок, и ваши программы станут более стабильными, понятными и удобными в использовании.

# **Список литературы**

1. https://habr.com/ru/companies/wunderfund/articles/736526/
2. https://www.geeksforgeeks.org/python-exception-handling/
3. https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html
4. https://sky.pro/media/chto-takoe-isklyucheniya-i-kak-ih-obrabatyvat-v-python/
5. https://skillbox.ru/media/code/isklyucheniya-v-python-chto-eto-takoe-i-kak-s-nimi-rabotat/
6. https://ru.hexlet.io/blog/posts/chto-takoe-isklyucheniya-v-python-i-kak-ih-obrabatyvat
7. https://metanit.com/python/tutorial/2.11.php