



Исключения в Python

try:



except:



upd: 11.05.2023 Александр Попов 👁 46.6K 💬 2

Обработка исключений в Python (try except)

#управляющие конструкции

Содержание:

- Как устроен механизм исключений
- Как обрабатывать исключения в Python (try except)
 - As — сохраняет ошибку в переменную
 - Finally — выполняется всегда
 - Else — выполняется когда исключение не было вызвано
 - Несколько блоков except
 - Несколько типов исключений в одном блоке except
 - Raise — самостоятельный вызов исключений
 - Как пропустить ошибку
- Исключения в lambda функциях
- 20 типов встроенных исключений в Python
- Как создать свой тип Exception

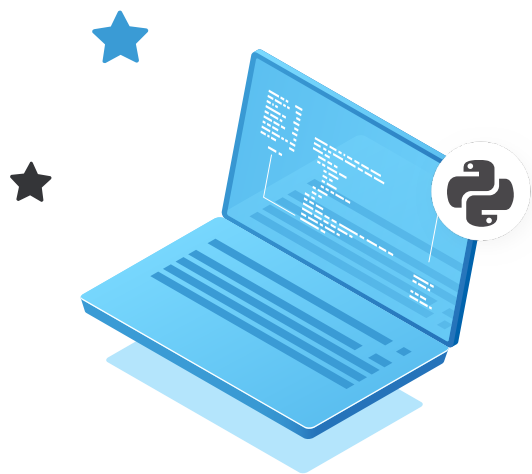
Программа, написанная на языке Python, останавливается сразу как обнаружит ошибку. Ошибки могут быть (как минимум) двух типов:

- **Синтаксические ошибки** — возникают, когда написанное выражение не соответствует правилам языка (например, написана лишняя скобка);
- **Исключения** — возникают во время выполнения программы (например, при делении на ноль).

Синтаксические ошибки исправить просто (если вы используете IDE, он их подсветит). А вот с исключениями всё немного сложнее — не всегда при написании программы можно сказать возникнет или нет в данном месте исключение. Чтобы приложение продолжило работу при возникновении проблем, такие ошибки нужно перехватывать и обрабатывать с помощью блока `try/except`.




Skillbox

реклама



Профессия "Python-разработчик"

 4 588 руб/мес  3 месяца бесплатно онлайн-курс

-  Курс на 75% состоит из практики
-  3 проекта в портфолио, гарантия трудоустройства
-  Командная стажировка под руководством тимлида

Перейти

Как устроен механизм исключений

В Python есть встроенные исключения, которые появляются после того как приложение находит ошибку. В этом случае текущий процесс временно приостанавливается и передает ошибку на уровень вверх до тех пор, пока она не будет обработана. Если ошибка не будет обработана, программа прекратит свою работу (а в консоли мы увидим Traceback с подробным описанием ошибки).



Пример: напишем скрипт, в котором функция ожидает число, а мы передаём строку (это вызовет исключение "TypeError"):

```
def b(value):  
    print("-> b")  
    print(value + 1) # ошибка тут
```

```
def a(value):  
    print("-> a")  
    b(value)
```

```
a("10")
```

```
> -> a  
> -> b  
> Traceback (most recent call last):  
>   File "test.py", line 11, in <module>  
>     a("10")  
>   File "test.py", line 8, in a  
>     b(value)  
>   File "test.py", line 3, in b  
>     print(value + 1)  
> TypeError: can only concatenate str (not "int") to str
```

В данном примере мы запускаем файл **"test.py"** (через консоль). Вызывается функция **"a"**, внутри которой вызывается функция **"b"**. Все работает хорошо до строчки `print(value + 1)`. Тут интерпретатор понимает, что нельзя конкатенировать строку с числом, останавливает выполнение программы и вызывает исключение **"TypeError"**.

Далее ошибка передается по цепочке в обратном направлении: **"b"** → **"a"** → **"test.py"**. Так как в данном примере мы не позаботились обработать эту ошибку, вся информация по ошибке отобразится в консоли в виде Traceback.

Traceback (трассировка) — это отчёт, содержащий вызовы функций, выполненные в определенный момент. Трассировка помогает узнать, что пошло не так и в каком месте это произошло.

Traceback лучше читать снизу вверх ↑

```
Traceback (most recent call last):  
  File "test.py", line 11, in <module>  
    a("10")  
  File "test.py", line 8, in a  
    b(value)  
  File "test.py", line 3, in b  
    print(value + 1)  
TypeError: can only concatenate str (not "int") to str
```

4
3
2
1


Пример Traceback в Python

В нашем примере Traceback содержит следующую информацию (читаем снизу вверх):

1. `TypeError` — тип ошибки (означает, что операция не может быть выполнена с переменной этого типа);
2. `can only concatenate str (not "int") to str` — подробное описание ошибки (конкатенировать можно только строку со строкой);
3. Стек вызова функций (1-я линия — место, 2-я линия — код). В нашем примере видно, что в файле "test.py" на 11-й линии был вызов функции "a" со строковым аргументом "10". Далее был вызов функции "b". `print(value + 1)` это последнее, что было выполнено — тут и произошла ошибка.
4. `most recent call last` — означает, что самый последний вызов будет отображаться последним в стеке (в нашем примере последним выполнялся `print(value + 1)`).

В Python ошибку можно перехватить, обработать, и продолжить выполнение программы — для этого используется конструкция `try ... except ...`.






Find out more about deployment here:
<https://crn.link/deployment>
→ Book in 6.58s...

**Углубленное
изучение JavaScript
и Node.js.**

Узнать больше



**Бесплатный курс
по изучению
JavaScript
с практикой**

Узнать больше



**Курсы разработки
Python, Java, Fullstack
от GeekBrains!**

Узнать больше


Как обрабатывать исключения в Python (try except)

В Python исключения обрабатываются с помощью блоков `try/except`. Для этого операция, которая может вызвать исключение, помещается внутрь блока `try`. А код, который должен быть выполнен при возникновении ошибки, находится внутри `except`.

Например, вот как можно обработать ошибку деления на ноль:

```
try:
    a = 7 / 0
except:
    print('Ошибка! Деление на 0')
```

Здесь в блоке `try` находится код `a = 7 / 0` — при попытке его выполнить возникнет исключение и выполнится код в блоке `except` (то есть будет выведено сообщение "Ошибка! Деление на 0"). После этого программа продолжит свое выполнение.

 [PEP 8](#) рекомендует, по возможности, указывать конкретный тип исключения после ключевого слова `except` (чтобы перехватывать и обрабатывать конкретные исключения):

```
try:
    a = 7 / 0
except ZeroDivisionError:
    print('Ошибка! Деление на 0')
```

Однако если вы хотите перехватывать все исключения, которые сигнализируют об ошибках программы, используйте тип исключения `Exception`:

```
try:
    a = 7 / 0
except Exception:
    print('Любая ошибка!')
```

As — сохраняет ошибку в переменную

Перехваченная ошибка представляет собой объект класса, унаследованного от "BaseException". С помощью ключевого слова `as` можно записать этот объект в переменную, чтобы обратиться к нему внутри блока `except`:

```
try:
    file = open('ok123.txt', 'r')
except FileNotFoundError as e:
    print(e)

> [Errno 2] No such file or directory: 'ok123.txt'
```

В примере выше мы обращаемся к объекту класса "FileNotFoundError" (при выводе на экран через `print` отобразится строка с полным описанием ошибки).

У каждого объекта есть поля, к которым можно обращаться (например если нужно логировать ошибку в собственном формате):

```
import datetime

now = datetime.datetime.now().strftime("%d-%m-%Y %H:%M:%S")

try:
    file = open('ok123.txt', 'r')
except FileNotFoundError as e:
    print(f"{now} [FileNotFoundError]: {e.strerror}, filename: {e.filename}")

> 20-11-2021 18:42:01 [FileNotFoundError]: No such file or directory, filename:
```

Finally — выполняется всегда

При обработке исключений можно после блока `try` использовать блок `finally`. Он похож на блок `except`, но команды, написанные внутри него, выполняются обязательно. Если в блоке `try` не возникнет исключения, то блок `finally` выполнится так же, как и при наличии ошибки, и программа возобновит свою работу.

Обычно `try/except` используется для перехвата исключений и восстановления нормальной работы приложения, а `try/finally` для того, чтобы гарантировать выполнение определенных действий (например, для закрытия внешних ресурсов, таких как ранее открытые файлы).

В следующем примере откроем файл и обратимся к несуществующей строке:

```
file = open('ok.txt', 'r')

try:
    lines = file.readlines()
    print(lines[5])
finally:
    file.close()
    if file.closed:
        print("файл закрыт!")

> файл закрыт!
> Traceback (most recent call last):
>   File "test.py", line 5, in <module>
>     print(lines[5])
> IndexError: list index out of range
```

Даже после исключения "IndexError", сработал код в секции `finally`, который закрыл файл.

p.s. данный пример создан для демонстрации, в реальном проекте для работы с файлами лучше использовать менеджер контекста `with`.

Также можно использовать одновременно три блока `try/except/finally`. В этом случае:

- в `try` — код, который может вызвать исключения;
- в `except` — код, который должен выполняться при возникновении исключения;
- в `finally` — код, который должен выполняться в любом случае.

```
def sum(a, b):
    res = 0

    try:
        res = a + b
    except TypeError:
        res = int(a) + int(b)
    finally:
        print(f"a = {a}, b = {b}, res = {res}")

sum(1, "2")
```

```
> a = 1, b = 2, res = 3
```

Else — выполняется когда исключение не было вызвано

Иногда нужно выполнить определенные действия, когда код внутри блока `try` не вызвал исключения. Для этого используется блок `else`.

Допустим нужно вывести результат деления двух чисел и обработать исключения в случае попытки деления на ноль:

```
b = int(input('b = '))
c = int(input('c = '))
try:
    a = b / c
except ZeroDivisionError:
    print('Ошибка! Деление на 0')
else:
    print(f"a = {a}")

> b = 10
> c = 1
> a = 10.0
```

В этом случае, если пользователь присвоит переменной "с" ноль, то появится исключение и будет выведено сообщение "'Ошибка! Деление на 0'", а код внутри блока `else` выполняться не будет. Если ошибки не будет, то на экране появятся результаты деления.

Несколько блоков except

В программе может возникнуть несколько исключений, например:

1. Ошибка преобразования введенных значений к типу `float` (`"ValueError"`);
2. Деление на ноль (`"ZeroDivisionError"`).

В Python, чтобы по-разному обрабатывать разные типы ошибок, создают несколько блоков `except` :

```
try:
    b = float(input('b = '))
    c = float(input('c = '))
    a = b / c
```



```
except ZeroDivisionError:
    print('Ошибка! Деление на 0')
except ValueError:
    print('Число введено неверно')
else:
    print(f"a = {a}")
```

```
> b = 10
> c = 0
> Ошибка! Деление на 0

> b = 10
> c = питон
> Число введено неверно
```

Теперь для разных типов ошибок есть свой обработчик.

Несколько типов исключений в одном блоке except

Можно также обрабатывать в одном блоке except сразу несколько исключений. Для этого они записываются в круглых скобках, через запятую сразу после ключевого слова except . Чтобы обработать сообщения "ZeroDivisionError" и "ValueError" в одном блоке записываем их следующим образом:

```
try:
    b = float(input('b = '))
    c = float(input('c = '))
    a = b / c
except (ZeroDivisionError, ValueError) as er:
    print(er)
else:
    print('a = ', a)
```

При этом переменной `er` присваивается объект того исключения, которое было вызвано. В результате на экран выводятся сведения о конкретной ошибке.

Raise — самостоятельный вызов исключений

Исключения можно генерировать самостоятельно — для этого нужно запустить оператор `raise` .

```
min = 100
if min > 10:
    raise Exception('min must be less than 10')
```

```
> Traceback (most recent call last):  
> File "test.py", line 3, in <module>  
>     raise Exception('min value must be less than 10')  
> Exception: min must be less than 10
```

Перехватываются такие сообщения точно так же, как и остальные:

```
min = 100  
  
try:  
    if min > 10:  
        raise Exception('min must be less than 10')  
except Exception:  
    print('Моя ошибка')  
  
> Моя ошибка
```

Кроме того, ошибку можно обработать в блоке `except` и пробросить дальше (вверх по стеку) с помощью `raise` :

```
min = 100  
  
try:  
    if min > 10:  
        raise Exception('min must be less than 10')  
except Exception:  
    print('Моя ошибка')  
    raise  
  
> Моя ошибка  
> Traceback (most recent call last):  
> File "test.py", line 5, in <module>  
>     raise Exception('min must be less than 10')  
> Exception: min must be less than 10
```

Как пропустить ошибку

Иногда ошибку обрабатывать не нужно. В этом случае ее можно пропустить с помощью `pass` :

```
try:  
    a = 7 / 0  
except ZeroDivisionError:  
    pass
```

Исключения в lambda функциях

Обрабатывать исключения внутри lambda функций нельзя (так как lambda записывается в виде одного выражения). В этом случае нужно использовать именованную функцию.

20 типов встроенных исключений в Python

Иерархия классов для встроенных исключений в Python выглядит так:

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
    ArithmeticError
    AssertionError
    ...
    ...
    ...
    ValueError
    Warning
```

Все исключения в Python наследуются от базового `BaseException` :

- `SystemExit` — системное исключение, вызываемое функцией `sys.exit()` во время выхода из приложения;
- `KeyboardInterrupt` — возникает при завершении программы пользователем (чаще всего при нажатии клавиш `Ctrl+C`);
- `GeneratorExit` — вызывается методом `close` объекта `generator` ;
- `Exception` — исключения, которые можно и нужно обрабатывать (предыдущие были системными и их трогать не рекомендуется).

От `Exception` наследуются:

- 1 `StopIteration` — вызывается функцией `next` в том случае если в итераторе закончились элементы;
- 2 `ArithmeticError` — ошибки, возникающие при вычислении, бывают следующие типы:

- `FloatingPointError` — ошибки при выполнении вычислений с плавающей точкой (встречаются редко);
 - `OverflowError` — результат вычислений большой для текущего представления (не появляется при операциях с целыми числами, но может появиться в некоторых других случаях);
 - `ZeroDivisionError` — возникает при попытке деления на ноль.
- 3 `AssertionError` — выражение, используемое в функции `assert` неверно;
- 4 `AttributeError` — у объекта отсутствует нужный атрибут;
- 5 `BufferError` — операция, для выполнения которой требуется буфер, не выполнена;
- 6 `EOFError` — ошибка чтения из файла;
- 7 `ImportError` — ошибка импортирования модуля;
- 8 `LookupError` — неверный индекс, делится на два типа:
- `IndexError` — индекс выходит за пределы диапазона элементов;
 - `KeyError` — индекс отсутствует (для словарей, множеств и подобных объектов);
- 9 `MemoryError` — память переполнена;
- 10 `NameError` — отсутствует переменная с данным именем;
- 11 `OSError` — исключения, генерируемые операционной системой:
- `ChildProcessError` — ошибки, связанные с выполнением дочернего процесса;
 - `ConnectionError` — исключения связанные с подключениями (`BrokenPipeError`, `ConnectionResetError`, `ConnectionRefusedError`, `ConnectionAbortedError`);
 - `FileExistsError` — возникает при попытке создания уже существующего файла или директории;
 - `FileNotFoundError` — генерируется при попытке обращения к несуществующему файлу;
 - `InterruptedError` — возникает в том случае если системный вызов был прерван внешним сигналом;
 - `IsADirectoryError` — программа обращается к файлу, а это директория;
 - `NotADirectoryError` — приложение обращается к директории, а это файл;

- `PermissionError` — прав доступа недостаточно для выполнения операции;
- `ProcessLookupError` — процесс, к которому обращается приложение не запущен или отсутствует;
- `TimeoutError` — время ожидания истекло;

12 `ReferenceError` — попытка доступа к объекту с помощью слабой ссылки, когда объект не существует;

13 `RuntimeError` — генерируется в случае, когда исключение не может быть классифицировано или не подпадает под любую другую категорию;

14 `NotImplementedError` — абстрактные методы класса нуждаются в переопределении;

15 `SyntaxError` — ошибка синтаксиса;

16 `SystemError` — сигнализирует о внутренне ошибке;


17 `TypeError` — операция не может быть выполнена с переменной этого типа;

18 `ValueError` — возникает когда в функцию передается объект правильного типа, но имеющий некорректное значение;

19 `UnicodeError` — исключение связанное с кодирование текста в `unicode`, бывает трех видов:

- `UnicodeEncodeError` — ошибка кодирования;
- `UnicodeDecodeError` — ошибка декодирования;
- `UnicodeTranslateError` — ошибка перевода `unicode`.

20 `Warning` — предупреждение, некритическая ошибка.

 Посмотреть всю цепочку наследования конкретного типа исключения можно с помощью модуля `inspect`:

```
import inspect
```

```
print(inspect.getmro(TimeoutError))
```

```
> (<class 'TimeoutError'>, <class 'OSError'>, <class 'Exception'>, <class 'Base
```



Подробное описание всех классов встроенных исключений в Python смотрите в [официальной документации](#).

Как создать свой тип Exception

В Python можно создавать свои исключения. При этом есть одно обязательное условие: они должны быть потомками класса `Exception` :

```
class MyError(Exception):
    def __init__(self, text):
        self.txt = text

try:
    raise MyError('Моя ошибка')
except MyError as er:
    print(er)

> Моя ошибка
```

• • •

С помощью `try/except` контролируются и обрабатываются ошибки в приложении. Это особенно актуально для критически важных частей программы, где любые "падения" недопустимы (или могут привести к негативным последствиям). Например, если программа работает как "демон", падение приведет к полной остановке её работы. Или, например, при временном сбое соединения с базой данных, программа также прервёт своё выполнение (хотя можно было отловить ошибку и попробовать соединиться в БД заново).

Вместе с `try/except` можно использовать дополнительные блоки. Если использовать все блоки описанные в статье, то код будет выглядеть так:

```
try:
    # попробуем что-то сделать
except (ZeroDivisionError, ValueError) as e:
    # обрабатываем исключения типа ZeroDivisionError или ValueError
except Exception as e:
    # исключение не ZeroDivisionError и не ValueError
    # поэтому обрабатываем исключение общего типа (унаследованное от Exception)
    # сюда не сходят исключения типа GeneratorExit, KeyboardInterrupt, SystemE
else:
    # этот блок выполняется, если нет исключений
```

```
# если в этом блоке сделать return, он не будет вызван, пока не выполнен  
finally:  
# этот блок выполняется всегда, даже если нет исключений else будет проигн  
# если в этом блоке сделать return, то return в блоке
```

Подробнее о работе с исключениями в Python можно ознакомиться в [официальной документации](#).

4

40



Если вам понравилась статья, поделитесь ссылкой
на нее



Комментарии (2)

Оставить комментарий..

Код

Представьтесь (имя или ник)..

Опубликовать

Сначала популярные N

Сашка-Поломашка


1 год назад

Хороший детальный разбор для освоения и закрепления. Спасибо большое за ваш труд.
Очень признателен.

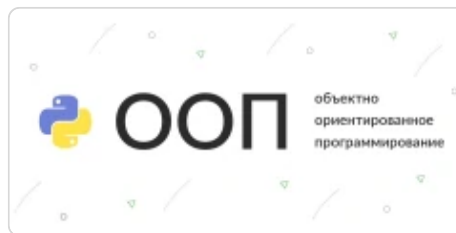
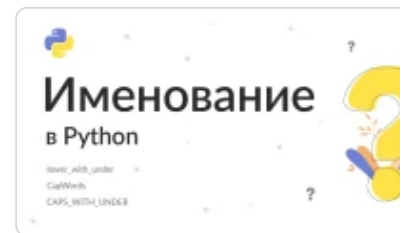
 8 [ОТВЕТИТЬ](#)**Иван**

11 мес. 6 дней назад

молодцы!

 2 [ОТВЕТИТЬ](#)

Может понравиться

[Основы](#) upd: 11.05.2023**Оператор выбора в Python
(if else)**[Основы](#) upd: 11.05.2023**Основы ООП в Python —
классы, объекты, методы**[Основы](#) upd: 11.05.2023**Именованное в Python — и
выбирать имена и почему
это важно**

Py

© pythonchik.ru, 2023

Использование материалов сайта pythonchik.ru
разрешено только с указанием dofollow-
ссылки.

[контакты](#)[политика конфиденциальности](#)