

Работа с ошибками

Цель занятия

После освоения темы вы:

- узнаете подходы к обработке ошибок;
- познакомитесь с механизмом формирования исключений;
- сможете использовать конструкции языка для генерации исключений на Python.

План занятия

1. [Обработка ошибок в программировании](#)
2. [Две стратегии обработки ошибок](#)
3. [Синтаксис обработки ошибок](#)
4. [Обработка исключений и производительность](#)
5. [Генерация исключений](#)
6. [Инструкция assert](#)
7. [Классы исключений](#)
8. [Создание пользовательских исключений](#)
9. [Практика работы с исключениями](#)

Используемые термины

Исключения или **исключительные ситуации** — механизм обработки нештатных ситуаций в программе. Исключения относят к ошибкам, которые произошли при исполнении программы.

LBYL (Look Before You Leap, «Семь раз отмерь, один раз отрежь») — стратегия обработки ошибок, которая сводится к тому, что при написании программы ошибок нужно избегать. Например, сначала проверить, правильные ли у нас данные и можем ли мы с ними работать, а потом уже переходить к их обработке.

EAFP (Easier to Ask for Forgiveness than Permission, «Легче попросить прощения, чем разрешения») — стратегия обработки ошибок, которая сводится к тому, чтобы попробовать выполнить какую-либо операцию; если выполнить действие не получится, тогда нужно уже предпринимать какие-либо меры.

Конспект занятия

1. Обработка ошибок в программировании

Виды ошибок в программах:

- **Синтаксические ошибки** возникают, если программа написана неправильно. Этот вид ошибок проще всего отследить.
- **Логические ошибки** возникают, если программа написана синтаксически правильно, но реализует неправильный алгоритм. Такие ошибки трудно отследить, поскольку никакие автоматизированные средства анализа кода не могут знать намерения программиста и логику, которую он реализует в своей программе.
- **Ошибки «времени выполнения»** (runtime) связаны с некоторыми нештатными ситуациями, когда программа написана правильно и реализует некоторую правильную логику, но какую-то ситуацию она не может обработать. Таких ошибок лучше избегать, поскольку их также трудно отследить. Причина — ошибки возникают не всегда, а только в некоторых особых ситуациях. Существуют средства, которые позволяют снизить вероятность появления таких ошибок, в частности — инструменты статического анализа кода.

Важно! Полностью избежать возникновения нештатных ситуаций принципиально невозможно, поскольку программа так или иначе взаимодействует с окружающим миром, а окружающий мир бывает непредсказуемым.

Примеры «непредсказуемых ситуаций»:

- обработка ввода данных пользователем, которые могут быть введены некорректно;
- попытка работы с сетью, когда сеть недоступна или сервер выдает неверный ответ.

Подобные ошибки существуют и на аппаратном уровне, например, при переполнении регистра. Для примера рассмотрим простое сложение двух чисел в двоичной системе (Рисунок 1). Происходит переполнение 8-битного регистра, и результат оказывается неправильным.

	10010100	148
+	10000101	133
<hr/>		
1	00011001	25

Рисунок 1. Пример ошибки при переполнении регистра

Рассмотрим, что мы можем делать с ошибками.

Вариант № 1. Не обращать внимания.

- + Самое простое решение.
- Программа выполняется некорректно, а мы об этом даже не знаем.
- Должен быть какой-то результат при некорректной операции — например, делении на ноль, недоступности сервера.

Вариант № 2. Записать где-то информацию об ошибке (используется в процессорах на аппаратном уровне).

- + Ошибку можно отследить.
- Но можно и не отследить (слишком легко об этом забыть).
- По-прежнему должен быть какой-то результат.

Вариант № 3. Вернуть специальное значение (используется на практике в языке Python).

- + Ошибка хорошо видна (относительно).
- Бывает сложно выбрать специальное значение (в Python часто используется `None`).
- Нет информации о том, какая именно ошибка произошла. В некоторых ситуациях этого достаточно и очевидно, какая ошибка произошла, но часто требуется детализация.

- По-прежнему можно забыть обработать специальное значение.

Вариант № 4. Вернуть два значения: результат операции и код ошибки (в Python не используется, используется в других языках программирования).

- + Сложно забыть ошибку.
- Код, выполняющийся в случае ошибки, перемешан с кодом основной логики, логика работы программы запутывается.

Вариант №5. Использовать «**исключения**» — отдельный механизм обработки нештатных ситуаций (активно используется в Python и других языках программирования). Исключения — ситуации, когда нормальная работа программы прерывается, и программа переходит в другой режим работы. Далее в уроке подробнее рассмотрим, как это может происходить.

- + Невозможно забыть обработать ошибку.
- + Основная логика и логика обработки ошибок может быть разделена.
- Требуется больше ресурсов.

Рассмотрим два способа обработки ошибок в Python на примере программы, которая ищет нужный символ в строке:

```
>>> s = 'Hello'
>>> s.find('o')
4
```

```
>>> s.index('o')
4
```

Методы `find` и `index` ведут себя одинаково, если символ есть в строке. Если символа в строке нет, то методы будут вести себя по-разному. Метод `find` возвращает специальное значение `-1`, а метод `index` возбуждает исключение:

```
>>> s = 'Hello'
>>> s.find('z')
-1
```

```
>>> s.index('z')
Traceback (most recent call last):
...
ValueError: substring not found
```

Выбор метода будет зависеть от того, какой способ предпочтительнее использовать в конкретной программе. Метод `find` будет более компактным и дешевым с точки зрения обработки, но при этом есть вероятность забыть обработать значение `-1` в случае нештатной ситуации.

2. Две основные стратегии обработки ошибок

LBYL (Look Before You Leap) — «Семь раз отмерь, один раз отрежь».

Данный подход предполагает, что мы должны избегать ошибок. То есть мы должны сначала проверить, правильные ли у нас данные и можем ли мы с ними работать, а потом уже переходить к их обработке.

EAFP (Easier to Ask for Forgiveness than Permission) — «Легче попросить прощения, чем разрешения».

Подход сводится к тому, чтобы попробовать выполнить какую-либо операцию. Если выполнить действие не получится, тогда нужно уже предпринимать какие-либо меры.

На бытовом уровне может казаться, что первый подход является наиболее правильным. В обычной жизни пытаться что-то сделать, а потом уже разрешать последствия — не лучшая стратегия. В программировании не так. Посмотрим на конкретном примере, что второй подход гораздо проще и лучше.

Разберем пример программы ввода пользователем целого числа. Если число не будет целым, программа «упадет»:

```
number = int(input('Введите целое число: '))
# Программа упадет, если будет введено не число
```

Давайте для начала проверим, ввел ли пользователь число. Если это так — будем с ним работать, если нет — сообщим об этом. На первый взгляд проблема решается очень просто с помощью функции `isdigit`:

```
number = input('Введите целое число: ')
if number.isdigit():
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

После запуска программы мы можем наблюдать:

```
Введите целое число: 123
Это не целое число, попробуйте еще раз...
```

Проблема может состоять в том, что пользователь ввел лишний пробел. Функция `int` с этим бы справилась:

```
>>> int(' 123')
123
```

Давайте сделаем программу более дружелюбной и «обрежем» лишние пробелы с помощью функции `strip`:

```
number = input('Введите целое число: ')
if number.strip().isdigit():
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

Продолжая тестировать программу, мы можем встретиться с еще одной проблемой — пользователь вводит отрицательное число:

```
Введите целое число: -123
Это не целое число, попробуйте еще раз...
```

Проверку введенное число не проходит, так как мы проверяем, что число полностью состоит из цифр, а знак «-» — это не цифра. Учтем это в программе:

```
number = input('Введите целое число: ').strip()
if (number[0] == '-' or number[0].isdigit()) and
number[1:].isdigit():
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

Но пользователь может ввести знак «+» в начале числа, и это будет вполне корректно с точки зрения функции `int`. Но наша программа опять не сможет принять такое число. Добавим в программу еще одну проверку:

```
number = input('Введите целое число: ').strip()
if (number[0] in '+-' or number[0].isdigit()) and
number[1:].isdigit():
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

Но пользователь по-прежнему может ввести строку, похожую на число, которое наша программа обработать не сможет:

```
Введите целое число: 1_234
```

Это не целое число, попробуйте еще раз...

Python позволяет разделять разряды знаком «_», например, функция `int` справится с таким вводом числа:

```
>>> int('1_234')
1234
```

Добавим ещё одну проверку в программу. Проверка становится более сложной: первый символ «+» или «-», все остальные — цифры или знаки подчеркивания. В коде используется функция `all` совместно с генераторным выражением:

```
number = input('Введите целое число: ').strip()
if (number[0] in '+-' or number[0].isdigit()) and \
    all(number[i].isdigit() or number[i] == '_'
        for i in range(1, len(number))):
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

Пользователь просто нажал клавишу `Enter` и ничего не ввел:

```
Введите целое число: ↵
Traceback (most recent call last):
...
if (number[0] in '+-' or number[0].isdigit()) and \
IndexError: string index out of range
```

Программа не дошла до конца проверки условия, а выдала ошибку до проверки — число ли ввел пользователь. Учтем это в программе, проверив, что `number` не является пустой строкой:

```
number = input('Введите целое число: ').strip()
if number and (number[0] in '+-' or number[0].isdigit()) and \
    all(number[i].isdigit() or number[i] == '_'
        for i in range(1, len(number))):
    number = int(number)
else:
    print('Это не целое число, попробуйте еще раз...')
```

На этом можно остановиться, но пользователь может быть злонамеренным и специально искать уязвимости в программе. Например, введет что-то такое, что пройдет проверку, но вызовет ошибки в программе:

```
Введите целое число: ②①₪₀⁴□۱۸€₪๑๘
```

```
Traceback (most recent call last):
```

```
...
```

```
    number = int(number)
```

```
ValueError: invalid literal for int() with base 10:
```

```
'20504189608'
```

Здесь можно использовать функцию `isdecimal`, которая хорошо работает в паре с функцией `int`. Заменим функцию `isdigit`:

```
number = input('Введите целое число: ').strip()
```

```
if number and (number[0] in '+-' or number[0].isdecimal()) and \
    all(number[i].isdecimal() or number[i] == '_'
        for i in range(1, len(number))):
```

```
    number = int(number)
```

```
else:
```

```
    print('Это не целое число, попробуйте еще раз...')
```

Можно найти еще один способ вызвать ошибку в программе:

```
Введите целое число: 1_
```

```
Traceback (most recent call last):
```

```
...
```

```
    number = int(number)
```

```
ValueError: invalid literal for int() with base 10: '1_'
```

Подчеркивание можно использовать как разделитель разрядов, но знак подчеркивания не может быть последним символом. То есть проверка в программе пройдена, а функция `int` по-прежнему выдает ошибку. Добавим в программу еще одно условие – последний символ в строке должен быть цифрой:

```
number = input('Введите целое число: ').strip()
```

```
if number and (number[0] in '+-' or number[0].isdecimal()) and \
    all(number[i].isdecimal() or number[i] == '_'
        for i in range(1, len(number))) and \
    number[-1].isdecimal():
```

```
    number = int(number)
```

```
else:
```

```
    print('Это не целое число, попробуйте еще раз...')
```

Но пользователь может ввести, например:

```
Введите целое число: -_1
```

```
Traceback (most recent call last):
```

```
...
```



```
number = int(number)
ValueError: invalid literal for int() with base 10: '-_1'
```

На самом деле, еще есть много вариантов, которые пройдут проверку в программе, но вызовут ошибку в функции `int`. Задача начинает казаться сложнее, чем она выглядела на первый взгляд.

Давайте попробуем использовать другой подход — EAFP — «Легче попросить прощения, чем разрешения»:

```
try:
    number = int(input('Введите целое число: '))
except ValueError:
    print('Это не целое число, попробуйте еще раз...')
```

Мы просто попробуем ввести целое число, а дальше пусть функция `int` решает, сможет ли она с ним работать. Это делается с помощью конструкции `try-except`, про которую мы будем говорить далее.

3. Синтаксис обработки ошибок

Синтаксис оператора `try-except`:

```
try:
    # код, который может вызвать ошибку
except Exception [as e]: # указываем тип ошибки
    # обработать ошибку
[else:]
    # что сделать, если ошибок не было
[finally:]
    # что сделать в любом случае
```

Если в блоке `try` возникнет ошибка, которую мы не описали в блоке `except`, то программа будет работать как обычно — без генерации соответствующего исключения.

Обрабатываемую ошибку можно сохранить в специальный объект с помощью оператора `as`.

После блоков `try` и `except` следуют блоки `else` и `finally`, которые не являются обязательными.

Блок `else` редко используется на практике — обычно все, что требуется сделать, описывают в блоке `try`. Но нужно понимать, что если в блоке `try` есть несколько

строчек кода, и ошибка произошла во второй строке, то все оставшиеся строки не будут выполнены. Поэтому это то же самое, что и написать строки в блоке `else`.

Блок `else` делает структуру кода более понятной и наглядной. Также в блок можно прописать ситуации для ошибок, которые мы не хотим отлавливать.

Блок `finally` часто используется для того, чтобы совершить действия, которые необходимо выполнить в любом случае — закрыть файлы, освободить ресурсы. То есть сделать все то, что программа должна сделать, если произошла ошибка.

Рассмотрим типичные ошибки новичков при обработке ошибок:

- **Помещать в блок `try` слишком много кода.** В этом случае трудно определить, где произошла ошибка. В идеале блок `try` должен быть предельно коротким и прицельным — то есть проверять конкретную операцию, которая может вызвать конкретную ошибку:

```
try:
    # 100 lines of code
except ValueError:
    # где именно возникла ошибка?
```

- **Относить обработку ошибки слишком далеко от места, где она возникла.** Эта ошибка аналогична предыдущей. Отличие состоит в том, что блок кода вынесен в функцию `main`.

```
try:
    main()
except ValueError:
    # где именно возникла ошибка?
```

- **Не указывать класс ошибки (или указать слишком широкий класс ошибок).** Синтаксис языка позволяет записать `except`, не указывая класс исключений. Приведенный в примере блок `except` эквивалентен тому, чтобы написать `except BaseException`, то есть программа будет отлавливать все исключения. Нужно отметить, что в Python исключениями являются и такие ситуации, как опечатка в программе или ее закрытие. В некоторых случаях подобные исключения имеет смысл отлавливать, но не в примере ниже:

```
try:
    number = int(input('Введите целое число: '))
except: # = except BaseException:
    # мы окажемся здесь при любых исключениях:
```

```
# опечатка: number = int(inupt()) или int(input)
# пользователь нажал ^C и пытается выйти
# ...
print('Это не целое число, попробуйте еще раз...')
```

- **Не обрабатывать ошибку.**

```
try:
    ...
except:
    pass
```

Такой код имеет место — например, в ситуации, когда мы считываем данные из файла и знаем, что некоторые данные могут быть повреждены. То есть мы считываем файл построчно, а при возникновении ошибки просто пропускаем ее.

Если подобная конструкция используется как временная заглушка в коде, и вы планируете в дальнейшем прописать обработку исключения, лучше будет прописать в блоке `except` вывод на печать какого-либо сообщения в виде `print`. Это акцентирует внимание и позже, вы сможете вернуться к доработке кода.

Принципы обработки ошибок или что можно делать:

- **Можно не писать блоки `else` и `finally`, а только `except`.**
- **Можно не писать `except`, а только `finally`.** Если ошибка произойдет, то она произойдет как обычно. Возможно, возникшая ошибка будет обработана где-то в другом месте, а сейчас нужно выполнить какие-то завершающие действия. Именно так работает контекстный менеджер:

```
f = open('filename')
try:
    # делаем что-то с файлом
    ...
finally:
    f.close()
```

- **Можно обрабатывать несколько типов исключений.** При этом нужно указать их.

Обрабатывать исключения можно одновременно. В примере в кортеже после слова `except` перечислены обрабатываемые типы исключений. Скобки принципиальны, без скобок пример работать не будет:

```
try:
    ...
except (TypeError, ValueError):
    ...
```

Исключения можно обрабатывать по отдельности. В примере несколько блоков `except` следуют последовательно после блока `try`:

```
try:
    ...
except TypeError:
    ...
except ValueError:
    ...
```

Рассмотрим порядок выполнения блоков.

В примере ниже последовательность выполнения блоков очевидна. Сначала выполняется обработка ошибки, затем — блок `finally`:

```
try:
    5 / 0
except ZeroDivisionError:
    print('EXC')
finally:
    print('FIN')
```

В следующем примере нет блока `except`. Подобная ситуация возникает, когда блок `except` присутствует в коде. В программе возникает ошибка, но в данном блоке `except` она не обрабатывается:

```
try:
    5 / 0
finally:
    print('FIN')
```

Python видит, что есть блок `finally`. И видит, что возникло исключение. В этом случае программа сначала выполнит блок `finally`, а затем вернется к исключению:

```
FIN
Traceback (most recent call last):
```

```
...
ZeroDivisionError: division by zero
```

Блок `finally` выполняется даже после `return` в функции:

```
def f(x, y):
    try:
        return x / y
    finally:
        print('This runs even after return!')
```

```
>>> f(1, 2)
This runs even after return!
0.5
>>> f(1, 0)
This runs even after return!
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Рассмотрим пример обработки исключения во время другого исключения:

```
try:
    int('abc')
except ValueError:
    int([])
finally:
    5 / 0
```

В примере возникает ошибка в блоке `try`, мы пытаемся ее обработать. Затем возникает ошибка в блоке `except`, а затем возникает ошибка и в блоке `finally`.

Python отобразит ошибки в следующем виде. Сначала будет показана ошибка в блоке `try`, поскольку мы эту ошибку не смогли обработать:

```
Traceback (most recent call last):
...
    int('abc')
ValueError: invalid literal for int() with base 10: 'abc'
```

Дальше выводится информация о следующем исключении:

```
During handling of the above exception, another exception
occurred:
```

```
Traceback (most recent call last):
```

```
...
```

```
    int([])
```

```
TypeError: int() argument must be a string, a bytes-like object or  
a real number, not 'list'
```

И далее информация об исключении, возникшем в блоке `finally`:

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
```

```
...
```

```
    5 / 0
```

```
ZeroDivisionError: division by zero
```

Если в коде возникло несколько исключений подряд, мы можем в другом месте кода обработать их все:

```
try:
```

```
    try:
```

```
        int('abc')
```

```
    except ValueError:
```

```
        int([])
```

```
    finally:
```

```
        5 / 0
```

```
except Exception as e:
```

```
    while e is not None:
```

```
        print(e)
```

```
        e = e.__context__
```

Большой блок кода, например, функцию, мы помещаем в блок `try`. Исключения перехватываются следующим образом: мы сохраняем их в переменную `e`. При этом первое исключение, которое мы получаем, является последним. У объекта `e` есть атрибут `__context__`, обратившись к которому мы получаем историю, когда исключение возникло:

```
division by zero
```

```
int() argument must be a string, a bytes-like object or a real  
number, not 'list'
```

```
invalid literal for int() with base 10: 'abc'
```

Если у исключения нет контекста, атрибут `__context__` будет хранить значение `None`.

Исключения, как и все в Python, являются объектами. Чтобы узнать, как устроен этот объект, можно написать код, записав исключение в переменную `e`. И далее с ним поработать:

```
>>> try:
...     5 / 0
... except Exception as e:
...     ex = e
...
>>> ex
ZeroDivisionError('division by zero')
>>> dir(ex)
['__cause__', '__class__', '__context__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getstate__', '__gt__',
 '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__setstate__', '__sizeof__', '__str__', '__subclasshook__',
 '__suppress_context__',
 '__traceback__', 'add_note', 'args', 'with_traceback']
>>> type(ex)
<class 'ZeroDivisionError'>
```

Важно записать исключение в переменную, иначе мы дальше не сможем с ним работать.

4. Обработка исключений и производительность

Сначала рассмотрим, как Python компилирует функцию.

```
def f(x):
    if x < 0:
        raise ValueError('x must be positive')
    return x ** 0.5

def g(x, y):
    return f(x) - f(y)

def h(x):
    return g(x, x - 2)
```

```
h(1)
```

Python компилирует функцию в так называемый байт-код, который хранится в специальном атрибуте функции в виде последовательности байтов. Чтобы превратить байт-код в читабельный вид, используют модуль `dis` — дизассемблер, и в нем функцию с таким же названием — `dis`:

```
>>> import dis
>>> dis.dis(f)    # python 3.11
```

Рассмотренный пример демонстрирует работу в версии Python 3.11. Байт-код функции относится к деталям реализации языка, и поэтому он может меняться от версии к версии. Сейчас в стандарте языка прописано, что, если исключение не возникает, то наличие `try` никак не влияет на производительность кода. Это можно увидеть из приведенного ниже примера:

```
1           0 RESUME                               0

2           2 NOP

3           4 LOAD_FAST                            0 (x)
           6 LOAD_FAST                            1 (y)
           8 BINARY_OP                           11 (/)

5          12 LOAD_GLOBAL                          1 (NULL + print)
          24 LOAD_CONST                          1 ('This runs even after
return!')
          26 PRECALL                              1
          30 CALL                                1
          40 POP_TOP
          42 RETURN_VALUE
      >>   44 PUSH_EXC_INFO
          46 LOAD_GLOBAL                          1 (NULL + print)
          58 LOAD_CONST                          1 ('This runs even after
return!')
          60 PRECALL                              1
          64 CALL                                1
          74 POP_TOP
          76 RERAISE                              0
      >>   78 COPY                                3
          80 POP_EXCEPT
          82 RERAISE                              1
```



```
ExceptionTable:  
  4 to 10 -> 44 [0]  
 44 to 76 -> 78 [1] lasti
```

Из байт-кода видно, что происходит загрузка переменных `x` и `y`, а далее операция деления. Затем загружается функция `print`, которая выводит на экран сообщение, и функция `f` возвращает значение.

Блок `finally` в конце функции компилятор встроил внутрь самого кода. Но в байт-коде видно, что после инструкции `return` есть что-то еще — это блок обработки ошибок. Блок описывается в специальной таблице рядом с телом функции.

Из блока следует, что если ошибка произошла в определенном диапазоне операций, например, с 4 по 10, то следует перейти к операции 44. Если ошибка произошла с 44 операции и далее, то есть внутри блока `finally`, обработчик передает эту ошибку далее с помощью оператора `RE_RAISE`. Таким образом, если никаких ошибок не происходит, функция выполняется так, как будто бы не было оператора `try`.

Если ошибка произошла, интерпретатор возвращается к таблице исключений и смотрит, есть ли обработчик функции для данной ошибки. В этом случае требуются дополнительные действия, что будет влиять на производительность.

Рассмотрим конкретный пример. Напишем одну и ту же программу в разных стилях.

Мы хотим провести частотный анализ текста — то есть подсчитать количество, которое каждая буква встречается в тексте. Алгоритм довольно известный: создаем словарь, разбираем текст по буквам. Если буквы в словаре не было, записываем в словарь 1. Если буква уже была в словаре, то добавляем 1.

В коде приведен пример стратегии «Семь раз отмерь, один раз отрежь», то есть, прежде чем обратиться к словарю, мы проверяем, есть ли в нем буква:

```
text = open('Анна Каренина.txt').read()  
  
with timeit('LBYL'):  
    letters = {}  
    for letter in text:  
        if letter not in letters:  
            letters[letter] = 1  
        else:  
            letters[letter] += 1
```

Этот же код мы можем записать в стратегии «Легче попросить прощения, чем разрешения». Мы пытаемся прибавить 1 к значению в словаре, если значения нет, то записать 1:

```
text = open('Анна Каренина.txt').read()

with timeit('EAFP'):
    letters = {}
    for letter in text:
        try:
            letters[letter] += 1
        except KeyError:
            letters[letter] = 1
```

Результаты выполнения кода:

- LBYL: 0.551 sec
- EAFP: 0.464 sec

Код, который использует блоки `try-except`, оказался значительно быстрее, чем код, который эти блоки не использует. Чтобы понять, почему так получилось, нужно вспомнить, как работают словари. Словарь — это хеш-таблица. При обращении к определенному ключу в словаре Python вычисляет хеш-значение ключа и определяет, в каком месте хеш-таблицы должно лежать это значение. Во втором примере эти операции выполняются один раз.

В первом примере рассмотренные операции выполняются дважды. Каждый раз оператор `in` вычисляет хеш-значение и смотрит, что лежит в словаре. Если значение есть, мы повторно вычисляем хеш-значение и обращаемся к хеш-таблице. Из-за этого возникает проигрыш в производительности.

Результаты не всегда будут такими. Производительность зависит от того, насколько часто происходят ошибки. Само понятие «исключение» предполагает, что происходит что-то исключительное. Если исключения происходят относительно редко, то блок `try-except` вряд ли повлияет на производительность. Возможно, даже повысит ее.

Если вы предполагаете, что исключения происходить будут часто, то следует отказаться от `try-except` и переписать код как-то иначе.

5. Генерация исключений

Создать исключение можно с помощью инструкции `raise`:

```
>>> raise ValueError
Traceback (most recent call last):
...
ValueError

>>> raise ValueError('Hello world!')
Traceback (most recent call last):
...
ValueError: Hello world!
```

Можно просто указать тип ошибки и ничего не писать, а можно создать экземпляр класса и передать ему аргументы. Набор аргументов может быть произвольным, но чаще всего используется один аргумент — сообщение об ошибке, которое будет выведено на экран.

Рассмотрим класс для создания битового массива. В этом классе используется механизм исключений, чтобы обработать ситуацию получения массива по индексу, и индекс не является корректным:

```
class Bitarray:
    def __getitem__(self, index):
        if index >= len(self) or index < -len(self):
            raise IndexError('Bitarray index out of range')
```

Почему мы используем исключение `IndexError` именно с таким сообщением? Если есть аналог обрабатываемой ситуации во встроенных типах исключений, тогда полезно сделать исключение похожим на то, которое возникает в стандартной ситуации. Такие исключения удобнее обрабатывать.

На практике можно посмотреть, как ведет себя список при обращении к некорректному индексу:

```
>>> a = [1, 2, 3]
>>> a[3]
...
IndexError: list index out of range
```

В пользовательском коде мы делаем то же самое. Отличие — заменяем тип `list` на тип `Bitarray`:

```
>>> b = Bitarray(3)
>>> b[3]
...
IndexError: Bitarray index out of range
```

Разберем пример, когда ошибка происходит в глубине программы:

```
def f(x):  
    if x < 0:  
        raise ValueError('x must be positive')  
    return x ** 0.5  
  
def g(x, y):  
    return f(x) - f(y)  
  
def h(x):  
    return g(x, x - 2)  
  
h(1)
```

В коде есть три функции, которые вызывают друг друга. Функция `f` вычисляет квадратный корень из числа, но сначала проверяет, что число является положительным. Если число отрицательное — генерирует ошибку.

Здесь мы не можем положиться на то, что вычисление квадратного корня из числа выдаст ошибку. Python-вычисление корня из отрицательного числа вернет комплексное число.

История возникновения ошибки может быть запутанной. Если мы увидим ошибку на экране, то увидим так называемый `traceback` — историю, как появилась ошибка:

```
Traceback (most recent call last):  
  File ..., line 12, in <module>  
    h(1)  
  File ..., line 10, in h  
    return g(x, x - 2)  
             ^^^^^^^^^  
  File ..., line 7, in g  
    return f(x) - f(y)  
           ^^^  
  File ..., line 3, in f  
    raise ValueError('x must be positive')  
ValueError: x must be positive
```

В примере показан пример `traceback` для версии Python 3.11. Мы видим номер строки файла с первым вызовом функции, который привел к ошибке. Далее видим, что внутри функции произошел еще один вызов и так далее.

Рассмотрим случай, когда мы не можем обработать полностью исключение. Для этого можно использовать функцию `raise` внутри блока `except`:

```
try:
    ...
except ConnectionError:
    # do something
    raise
```

В примере `raise` указан без каких-либо дополнительных аргументов. Это означает, что мы заново вызываем одну и ту же ошибку. Подобный прием может использоваться для дополнительной обработки — например, записать информацию об ошибке в лог-файл.

Новая возможность Python 3.11 — у исключений появился метод `add_note`, который позволяет добавить заметку к исключению. Список заметок будет доступен в специальном атрибуте `__notes__`:

```
try:
    ...
except ConnectionError as e:
    e.add_note('I was here')
    raise

e.__notes__
['I was here']
```

Такой прием будет полезен, если мы знаем контекст, который привел к ошибке. Также будет полезен в дальнейшей обработке ошибки.

Иногда возникает необходимость заменить исключение. Это можно сделать с помощью инструкции `raise`. Например, это может понадобиться, когда мы хотим добавить немного инкапсуляции для логики программы:

```
try:
    return users[user_id]
except KeyError:
    # UserNotFound - пользовательское исключение
    raise UserNotFound(f'user id {user_id} not found')
```

В примере выше функция ищет пользователя по его идентификатору. Функция обращается к некоторому словарию, если пользователя нет в словаре, мы получаем ошибку `KeyError`. Но мы не хотим, чтобы кто-то видел эту ошибку, которая говорит

о внутренних деталях реализации программы — наличии словаря. Поэтому мы создаем исключение, которое уже никак не связано с внутренней реализацией.

Но с точки зрения интерпретатора такой код будет выглядеть, что как будто мы не смогли обработать ошибку и произошла другая ошибка:

```
Traceback (most recent call last):
```

```
...
```

```
KeyError: 17843691
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
```

```
...
```

```
UserNotFound: user id 17843691 not found
```

Можно изменить код следующим образом:

```
try:
```

```
    return users[user_id]
```

```
except KeyError as e:
```

```
    # поясняем, что это не отдельное исключение
```

```
    raise UserNotFound(f'user id {user_id} not found') from e
```

Ошибка `KeyError` описывается как причина той ошибки, которую мы генерируем.

Вывод будет похожим:

```
Traceback (most recent call last):
```

```
...
```

```
KeyError: 17843691
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
```

```
...
```

```
UserNotFound: user id 17843691 not found
```

Если мы хотим «спрятать» `KeyError`, можно изменить код: `from None` удаляет информацию о произошедшем исключении и создает другое:

```
try:
```

```
    return users[user_id]
```

```
except KeyError:
    # удаляем информацию о предыдущем исключении
    raise UserNotFound(f'user id {user_id} not found') from None
```

Таким образом, при выводе мы получим:

```
Traceback (most recent call last):
...
UserNotFound: user id 17843691 not found
```

То есть мы скрыли детали реализации и показали только информацию, нужную для пользователя.

6. Инструкция `assert`

Инструкция `assert`, так же как и инструкция `raise`, может использоваться для генерации исключений. Но между ними есть существенные отличия. Инструкция `assert` служит для проверки базовых инвариантов, нарушение которых явно свидетельствует об ошибке в логике работы программы.

Инструкция `assert` нужна исключительно как средство отладки во время разработки программы, но она не должна работать в реально работающей программе:

```
assert 2 * 2 == 4, 'something really weird happened'
```

После слова `assert` записывается условие, которое с точки зрения программиста является очевидным. Через запятую записывается сообщение, которое выводится, если условие окажется ложным:

```
assert condition, message
```

Синтаксис инструкции почти эквивалентен записи:

```
if not condition:
    raise AssertionError(message)
```

Рассмотрим, почему нельзя говорить о полной эквивалентности. В примере после инструкции `assert` следует инструкция `print`, которая не должна выполняться при неверном условии:

```
# assertion.py
assert 2 * 2 == 5, 'We should stop here'
print('This should never happen')
```

Однако, запустив программу в терминале, мы получим:

```
$ python assertion.py
Traceback (most recent call last):
  File ...
    assert 2 * 2 == 5, 'We should stop here'
AssertionError: We should stop here
```

Но при запуске программы в реальном рабочем окружении результат будет другой. Программа запускается с флагом «-O», который означает «оптимизация». Python оптимизирует код, чтобы он запускался быстрее, а для этого отключает инструкции `assert`:

```
$ python -O assertion.py
This should never happen
```

Важно! Не используйте инструкцию `assert` для проверки ситуаций, которые могут возникнуть в ходе работы программы. `Assert` используется только для проверки явного нарушения работы логики программы.

Рассмотрим пример из реальной программы:

```
...
# tree_dict и nodes сгенерированы нашей программой
roots = [x for x in tree_dict if x not in nodes]
assert len(roots) == 1, f'tree has more than one roots: {"",
".join(roots)}'
root = roots[0]
```

Программа работает с неким деревом, которое сама же и строит. Цель программы – найти корень дерева. Инструкция `assert` проверяет, действительно ли в дереве один корень, то есть не нарушена ли логика программы. Важно отметить, что дерево генерируется самой программой, и если программа правильно сформировала дерево, инструкция `assert` никогда не должна сработать.

Инструкция `assert` может использоваться для простых тестов:

```
def is_prime(n):
    """ Определяет, является ли число простым """
    ...

assert is_prime(2)
assert is_prime(13)
assert is_prime(1999993)
assert not is_prime(1)
```



```
assert not is_prime(6)
assert not is_prime(25)
```

Мы написали функцию и делаем несколько проверок, которые должны пройти, если функция написана правильно.

Также `assert` может использоваться для того, чтобы убедиться, что список условий является исчерпывающим:

```
# Python 3.10
match code:
    case 0: ...
    case 1: ...
    case 2: ...
    case _: assert False, f'impossible code {code}'
```

Строка, содержащая `assert False`, точно должна сработать. Такая проверка нужна для того, чтобы убедиться, что мы точно никогда не дойдем до этого места в программе.

7. Классы исключений

В [документации](#) к языку достаточно подробно описана иерархия классов исключений.

Все исключения наследуются от базового класса `BaseException` (Рисунок 2), а все «обычные» исключения наследуются от класса `Exception`.

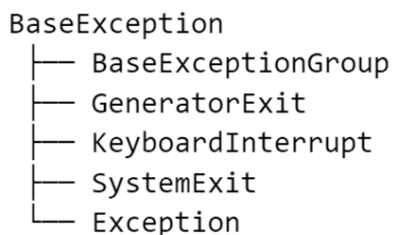


Рисунок 2. Иерархия исключений

Помимо этого есть некоторые особые исключения, которые перехватывать блоком `except` не нужно. Например, `KeyboardInterrupt` возникает, когда пользователь нажал сочетание клавиш `Ctrl+C`, `SystemExit` — при вызове функции `Exit`:

```
try:
    exit()
except SystemExit:
```

```
print('I want to exit!')
```

Класс `Exception` содержит также несколько уровней иерархии (Рисунок 3).

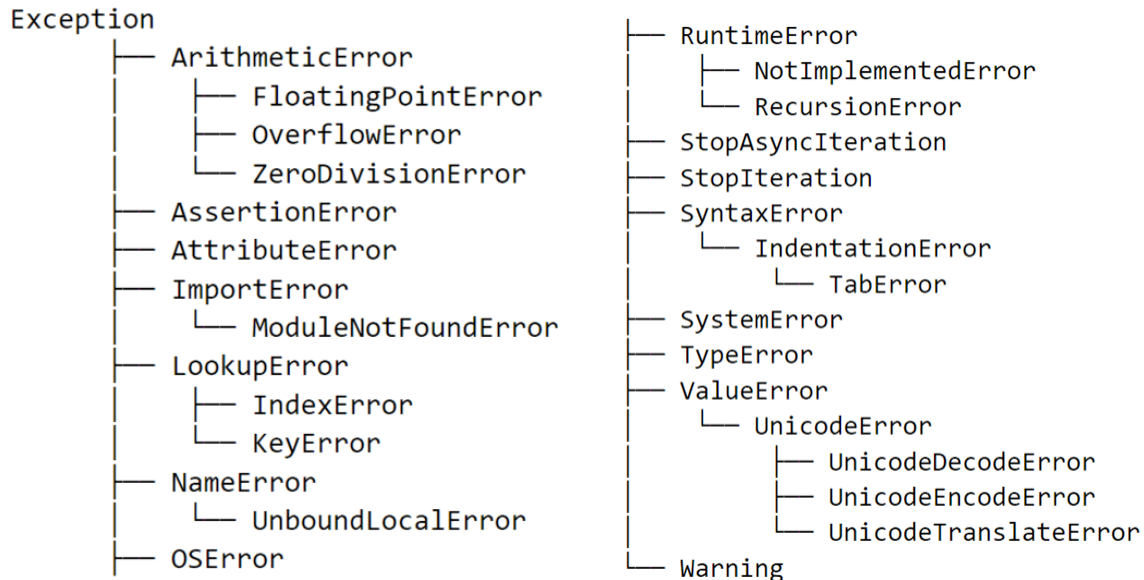


Рисунок 3. Производные классы `Exception`

Рассмотрим некоторые классы исключений:

- `ArithmeticError` — описывает арифметические ошибки.
- `AssertionError` — описывает ошибки, связанные с инструкцией `assert`.
- `AttributeError` — возникает при обращении к несуществующему атрибуту.
- `ImportError` — ошибка импорта.
- `NameError` — возникает при использовании необъявленной переменной.
- `OSError` — ошибки связанные со взаимодействием с операционной системой (рисунок 4).
- `Warning` — предупреждения, которые не приводят к остановке программы, а выводят сообщения для программиста.

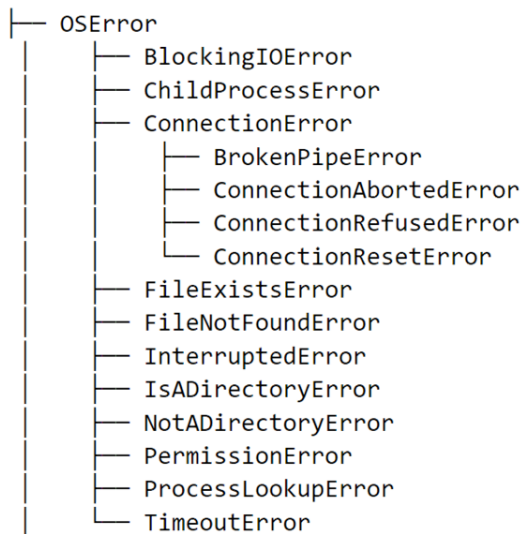


Рисунок 4. Ошибки, связанные со взаимодействием с операционной системой

8. Создание пользовательских исключений

Чтобы создать собственный тип исключения, достаточно создать класс в иерархии исключений с помощью наследования:

```
class FatalException(BaseException): pass
class ConnectionError(Exception): pass
class UserNotFound(LookupError): pass
```

Чаще всего исключение наследуется от класса `Exception`.

Класс исключения — полноценный класс, мы можем описывать в нем дополнительные атрибуты и методы.

С помощью дополнительных атрибутов можно записать информацию в исключение, что именно произошло. Далее в примере в исключение записывается информация о коде ответа веб-сервиса. Затем в программе проверяется код, который вернул сервис — например, код 404 означает, что сервис недоступен.

```
class ServiceUnavailable(Exception):
    def __init__(self, *args, status_code):
        super().__init__(*args)
        self.status_code = status_code
```

...

```
try:
```

```
...
except ServiceUnavailable as ex:
    if ex.status_code == 404:
        print('Недоступен')
    else:
        print('Ошибка', ex.status_code)
```

Рассмотрим более развернутый пример. В программе функция `f` проверяет, входит ли число в диапазон от 0 до 100. Если условие не выполняется, генерируется исключение `NotInRangeError`.

Исключение `NotInRangeError` также описано в программе. Исключение принимает на вход три аргумента – границы диапазона (два числа) и само значение:

```
class NotInRangeError(Exception):
    def __init__(self, min_limit, max_limit, value):
        super().__init__(
            f'value {value} is not in range [{min_limit},
{max_limit}]'
        )
        self.min_limit = min_limit
        self.max_limit = max_limit
        self.value = value

def f(x):
    if not 0 <= x <= 100:
        raise NotInRangeError(0, 100, x)
    ...
```

Обработка исключения описывается в блоке `try-except`. В первую очередь программа проверяет, введено ли корректное число (исключение `ValueError`). При корректном числе может сработать исключение `NotInRangeError`. Атрибуты исключения `NotInRangeError` используются, чтобы указать, в каком диапазоне должно быть введенное число:

```
try:
    f(int(input()))
except ValueError:
    print('Вы должны ввести целое число')
except NotInRangeError as e:
    print(f'Число должно быть в диапазоне от {e.min_limit} до
{e.max_limit}')
```

9. Практика работы с исключениями

Разберем практический пример работы с исключениями.

Главный вопрос, который встает перед нами в этой ситуации: где именно обрабатывать исключение? Общий принцип — чем ближе к месту, тем лучше. Но не всегда можно обработать исключение непосредственно там, где оно возникает.

Исключения — это альтернативный способ общения функций друг с другом. Часть программы, которая обрабатывает исключения, должна иметь достаточно «полномочий» для этого. То есть она должна знать, что хочет пользователь, и почему возникло исключение.

Структуру программы можно сравнить со структурой организации. В организации есть генеральный директор, который запускает все остальные процессы. Есть функции, которые запускают другие функции — например, у генерального директора в подчинении директор по производству и т. д.

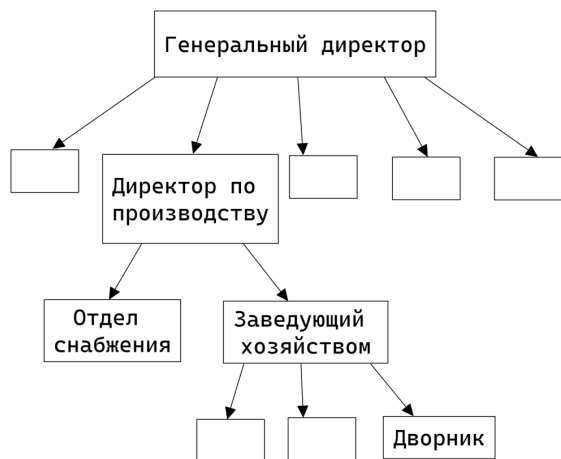


Рисунок 5. Структура организации

В конце иерархии есть сотрудник без подчиненных — например, дворник. Если вдруг его метла будет сломана, то он должен понять, может ли он сам справиться с этой проблемой.

- 1) ДА: он может починить метлу самостоятельно и продолжить работу. Тогда никаких дополнительных действий совершать не нужно.
- 2) НЕТ: он обращается к своему начальнику. Если тот не может решить проблему, то он обращается к своему начальнику. И так выше, не доводя до сведения генерального директора.

Важно! Важно найти правильный уровень, который будет достаточно высоким, чтобы решить проблему и справиться с исключениями (имеет достаточно полномочий). Но

не будет слишком высоким, чтобы внутренние детали реализации мелких функций не влияли на код, который находится на высоком уровне.

В качестве примера рассмотрим небольшую программу, в которой нет обработки ошибок — ведение домашней бухгалтерии. Нам нужно ее сделать.

[Первоначальный код и итоговый код программы находятся в дополнительных материалах]

Как устроен первоначальный код

- `DB_FILE = 'db.scv'` — класс, описывающий базу данных, внутри которого методы `get`, `add`, `get_next_id` и `save`.
- В самой программе есть методы `add_op` и `view_op` — добавления и просмотра операции.
- Словарь с функциями:

```
funcs = {  
    1. add_op,  
    2. view_op,  
}
```

- Текст для вывода и пользовательский ввод:

```
print('Домашняя бухгалтерия')  
print('В базе данных {db.n_records} записей')  
while True:  
    print('Введите команду:')  
    print('0. выйти')  
    print('1. добавить операцию')  
    print('2. посмотреть информацию об операции')  
    cmd = int(input())  
    if cmd == 0:  
        db.save()  
        print('Всего доброго!')  
        break  
    funcs[cmd]()
```

Запуская программу, пользователь может совершить множество ошибок. Например, ввести неправильную сумму, не тот id или вызвать некорректную команду. Кроме того, программа обращается к файлу с базой данных. И если в нее внести изменения, ошибочные данные или совсем удалить, то программа так же выдаст ошибку.

Как это исправить?

Ошибка в классе базы данных.

Если файла нет:

1. Создадим несколько классов пользовательских ошибок:
`Class DatabaseError(Exception)` (базовый класс),
`Class DatabaseNotFound(DatabaseError)`
(класс-наследник), чтобы показать пользователю, что база данных не найдена
2. Добавим сообщение для пользователя:

```
ans = input('Создать новый файл? [д/н]')
if ans != 'д':
    print('Программа не может работать без базы
    данных')
    exit()
else:
    db = DataBase.create_new() #метод класса
```
3. Для создания нового пустого файла используем декоратор:

```
@classmethod
def create_new(cls):
    open(cls.DB_FILE, 'w').close()
    return cls()
```

Если в файле некорректные данные. Создадим класс `Class`

`DatabaseCorrupted(DatabaseError)` (класс-наследник), чтобы показать пользователю, что файл содержит некорректные данные.

Ошибка ввода команды.

Ввод некорректного числа:

1. Напишем метод функции `read_int()`, которая будет копировать метод функции `input()`. Вместе с циклом `while` мы будем считывать ввод бесконечно, пока не получим на вход корректное число.

```
def read_int(prompt=''):
    while True:
        try:
            value = int(input(prompt))
        except ValueError:
            print('Некорректное число')
        else:
            return value
```

2. Используем `read_int()` во всех местах кода, где раньше использовался `input()`.
3. Воспользуемся приемом валидации значения для функции `read_int()`, чтобы она не принимала отрицательные значения или другие, не подходящие нам:

```
def read_int(prompt='', validator: Callable[[int], None] = None):
```

`validator` — функция, выполняющая проверку корректности введенного значения, должна возбудить ошибку `ValidationError` с понятным для пользователя сообщением или вернуть `None`.
Добавим функцию `positive_validator()`, которая проверит, чтобы число было положительным:

```
def positive_validator(x: int) -> None:
```

```
    if x <= 0:
```

```
        raise ValidationError('Число должно быть
```

```
                               положительным')
```

Ввод некорректного id. Можно использовать встроенное исключение `KeyError`, если запись в базе данных не найдена:

```
def view_op():
    op_id = read_int('Введите id операции: ')
    try:
        amount, name = db.get(op_id)
    except KeyError:
        print('Запись не найдена')
    else:
        print(f'{amount} руб. за {name}')
```

Ввод некорректной операции. Добавим метод выхода в словарь функций, запишем функцию метода и проверим, есть ли такое число в словаре:

```
def quit():
    db.save()
    print('Всего доброго!')
    exit()

funcs = {
    0. quit,
    1. add_op,
```



```
2. view_op,  
}
```

```
def funcs_validator(x: int) -> None:  
    if x not in funcs:  
        raise ValidationError('Такой операции нет')
```

Чтобы база данных сохраняла данные, нужно написать блок `finally`:

```
finally:  
    db.save()
```

Дополнительные материалы для самостоятельного изучения

1. [Built-in Exceptions](#)