Шпаргалка: обработка исключений

Что такое исключение

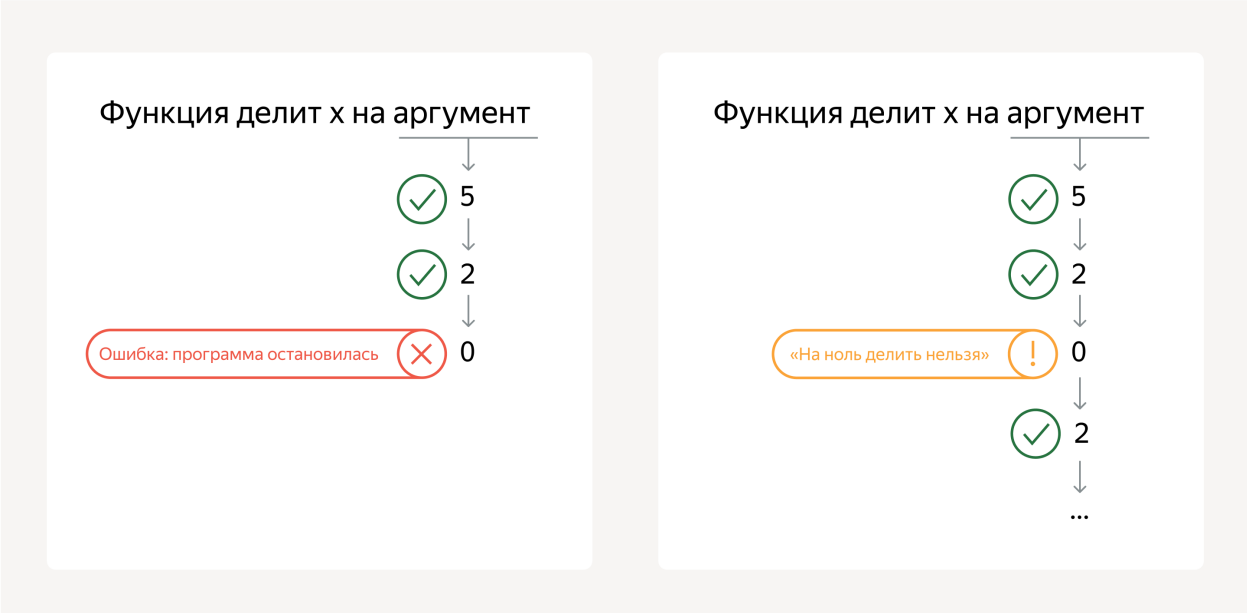
Есть такая разновидность ошибок, при которых программа может и не останавливаться: она сама их обработает и продолжит выполняться. Такие ошибки называются **исключениями**.

> 📌 **Исключения** — это ошибки, из-за которых не нужно останавливать программу. Программист решил, что они не критичные: программа выведет сообщение и продолжит работать.

Пример. Функция делит `x` на какое-либо число: его передают как аргумент. Функция работает хорошо — и тут получает на вход 0. Вроде бы ноль — это число, но делить на него нельзя.

Программа в замешательстве. Она завершается с ошибкой `ZeroDivisionError` в трейсбеке. Выглядит как ошибка, но ведь программа повела себя разумно: на ноль и правда поделить не получится. Это не ошибка, а просто особенный случай: нет смысла из-за него прерывать выполнение.

Хорошо будет сделать так, чтобы программа оповестила разработчика и продолжила работать с другими числами. Это тот случай и называется исключением.



В чём особенность исключений

📎 Важно понимать: исключения — это не отдельная сущность в коде, а как бы особенность проектирования. Разработчик понимает, что в каком-то месте программа может споткнуться, но это не критично, и останавливать выполнение не нужно. Он «подкладывает соломку»: закладывает обработку исключения.

При этом есть ситуации, когда ошибка критичная и перехватывать её не нужно: программа остановится, и это будет правильно. Например, неправильный синтаксис. Если разработчик забыл закрыть скобку, программа вырубится: так и должно быть. Нужно всё остановить и пойти пофиксить.

Ошибки

🔴 Программа должна остановиться
🔴 На экран выведется трейсбек
🔴 Обнаруживает и обрабатывает программист уже после того, как программа остановилась
🔴 Часто вызываются неправильным синтаксисом или логическими несостыковками в коде

Исключения

🟢 Программа может обработать сама и продолжить выполняться
🟢 На экран может вывестись сообщение, которое заложит программист
🟢 Программист заранее пишет код, чтобы программа обработала сама
🟢 Часто вызываются непредвиденными обстоятельствами. Например — разрыв соединения. Надо дать программе попробовать ещё раз, восстановить соединение, и только потом можно завершиться

> 🙂 Одна и та же ситуация в разных случаях применения будет в первом — ошибкой, во втором — исключением. Просто потому, что разработчик так решил.

Пример. Уже знакомая тебе ошибка `AttributeError`. Она возникает, когда пытаются получить доступ к атрибутам объекта, которых у него на самом деле нет. В прошлых уроках эта ситуация всегда считалась ошибкой: программа останавливалась, тебе приходилось читать трейсбек. Но может случиться так, что ты решишь: «Да, вот тут попробовали обратиться к атрибуту и не нашли его — нормально, надо просто об этом сказать и продолжить».

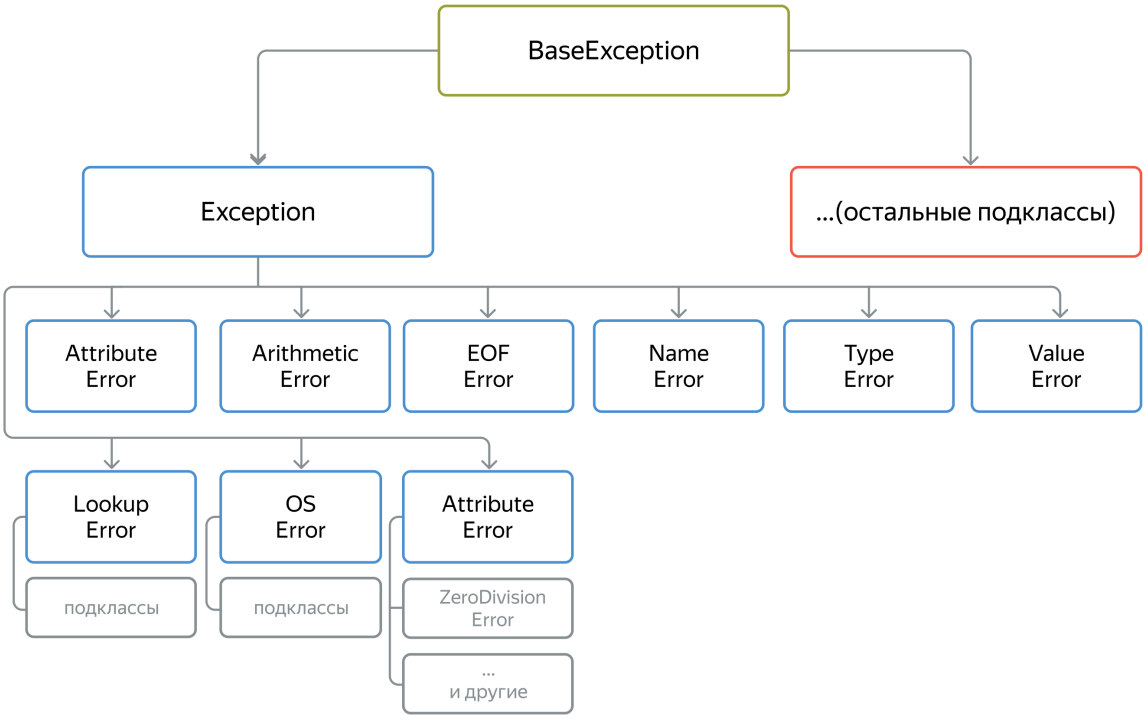
Тогда ты обработаешь эту ситуацию и вместо ошибки она станет исключением.

Разные виды исключений

Как ты уже из знаешь из урока про ООП, всё в Python — это классы и объекты. Ошибки и исключения — не исключение 🙂. Чтобы тоньше настраивать обработку исключений, познакомься с разными типами.

Верховный родитель для всех исключений и ошибок — `BaseException`. От него наследуется `Exception` и ещё несколько подклассов. О них пока говорить не будем, просто знай, что в теории они есть.

У `Exception` много наследников: это конкретные типы ошибок и исключений. Например, уже знакомая тебе `AttributeError` — ошибка, связанная с атрибутом. Или `NameError` — имя не найдено.



То есть, когда ты перехватываешь исключение, ты «ловишь» объект какого-то из классов: `AttributeError`, `NameError` и так далее. Если пробуешь делить на ноль, поймается объект `ZeroDivisionError`. Если программа не нашла атрибут — объект `AttributeError`. И так далее.

Как обработать все исключения разом

С помощью специального оператора `try...except`. В него как бы оборачивают кусочек кода: «Попробуй сделать это. Если не получилось, поймай исключение, выведи это и продолжи выполнять программу».

```
def function(arg):
    try: # попробуй выполнить действие ниже
        # действие
    except Exception: # если не получилось, поймай исключение и выполни действие далее
       # действие
```

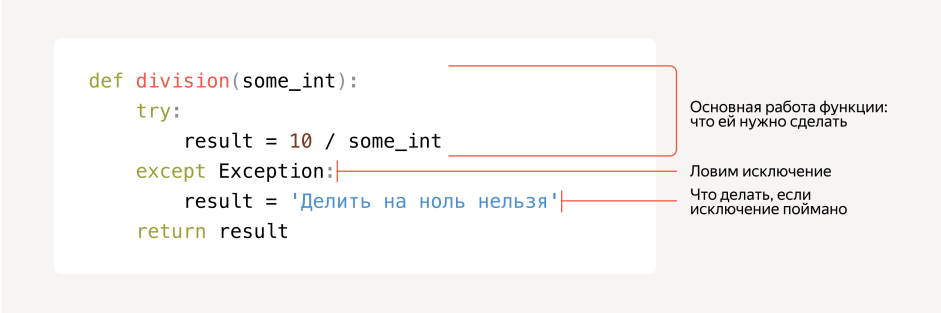
Пример с делением: функция делит x на переданный аргумент. Пусть для простоты x = 10. Изначальный код:

```
def division(some_int):
    result = 10 / some_int
    return result
```

В момент, когда функции передают ноль, появляется исключение: на ноль делить нельзя. Его хорошо бы обработать. Оборачиваем код в `try...except`:

```
def division(some_int):
    try: # попробуй выполнить действие ниже
        result = 10 / some_int
    except Exception: # если не получилось, поймай исключение и выполни действие далее
        result = 'Делить на ноль нельзя'
    return result
```

Что тут происходит: допустим, программа пробует выполнить деление. У неё не получается, потому что пробуют делить на 0. Python уже готов остановить программу и выбросить ошибку. И в этот момент строка `except Exception` «ловит» эту ошибку.



Вот что будет, если попробовать запустить программу с разными аргументами:

```
def division(some_int):
    try: # попробуй выполнить действие ниже
        result = 10 / some_int
    except Exception: # если не получилось, поймай исключение и выполни действие далее
        result = 'Делить на ноль нельзя'
    return result

print(division(0)) # 'Делить на ноль нельзя'
print(division(2)) # 5.0
```

Если не написать конструкцию `try...except`, после первого вызова функции программа завершится: передали ноль. Но если написать обработку, Python выдаст исключение, а программа продолжит работать.

Пример 2. Есть функция, которая складывает два числа: 2 и ещё какое-то, которое передали в аргументе.

Функция отработает корректно, только если аргументом передадут число. Если передадут строку или элемент списка, код остановится. Но можно «перехватить» это исключение — сделать так, чтобы программа не остановилась, а просто сказала, что так делать нельзя:

```
# функция принимает на вход аргумент любого типа
def addition(some_arg):
    try:
        result = some_arg + 2 # пытается прибавить к нему цифру 2
    except Exception:
        result = f'Складывать int и {type(some_arg).__name__} нельзя!'
    return result

print(addition('2')) # Складывать int и str нельзя!
print(addition([2])) # Складывать int и list нельзя!
print(addition(0)) # 2
```

☝️ Обрати внимание на разницу в проектировании этих двух примеров. В случае с делением на `some_int` тоже могут передать неправильный тип данных. Но обработку этого случая не заложили. Это потому, что программист решил: неправильный тип данных — критичная ошибка. Программа должна остановиться.

А вот во втором случае программа должна принимать на вход аргумент любого типа. Поэтому аргумент, отличный от числа, — исключение, а не ошибка.

Чем `try...except` отличается от `if`

Тебе могло показаться, что логика работы `try...except` похожа на `if...else`. Это действительно так. Но вот проектировали эти две конструкции для совершенно разных случаев, поэтому одно не заменяет другое.

Оператор `if` проверяет условие. Он выполняет блоки кода, основываясь на том, истинно условие или ложно. Эта конструкция работает с булевыми значениями.

Конструкция `try...except` создана специально для перехвата исключений: она не завязана на булевых значениях.

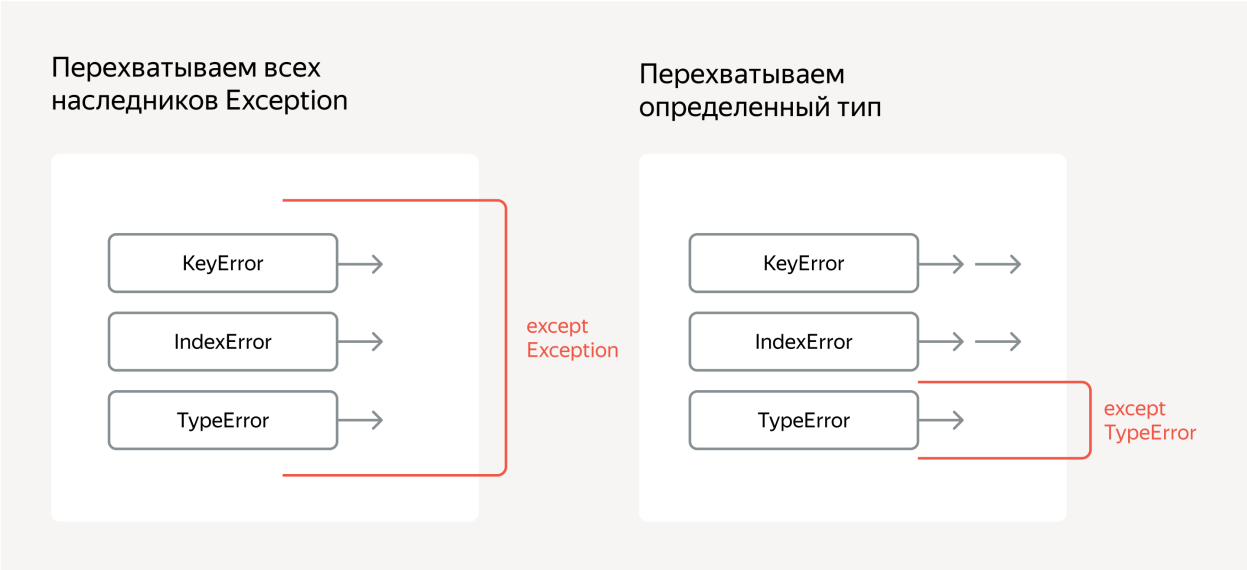
Условный оператор — для сложных структур, «развилок» в коде. Обработка исключений с `try...except` — про **отказоустойчивость**: свойство системы сохранять работоспособность, если какая-то её часть перестанет работать.

📌 Оператор `if` не сможет перехватить исключение.

Как перехватить определённый вид исключений

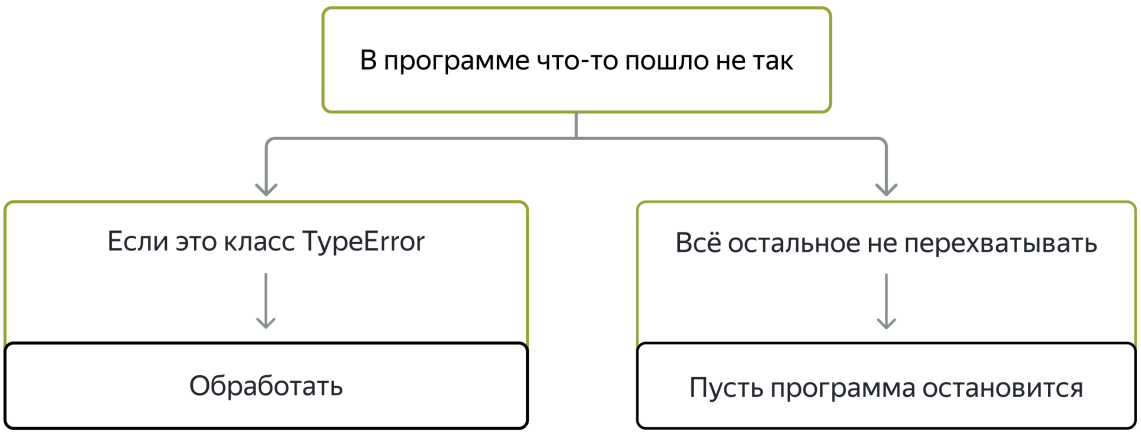
📍 Написать конкретный вид после `except`. Например, `except TypeError`

Если тебе нужно перехватить исключение определённого вида, после `except` вместо суперкласса `Exception` напиши конкретного наследника. Например, `TypeError`.



Пример. Представь такую ситуацию:

- Если что-то не так с типом переменной, хотим остановить, сделать это исключением и его обработать. То есть ловим `TypeError`.
- Если ошибка `KeyError` или `IndexError` — это критично. Нужно прерывать выполнение функции.



Вот как это будет в коде:

```
# функция принимает на вход аргумент любого типа
def addition(some_arg):
    try:
        result = some_arg[0] + 2 # пытается прибавить к первому элементу цифру 2
    except TypeError: # перехватывает не все исключения, а только связанные с типом переменной
        result = f'Складывать int и {type(some_arg).__name__} нельзя!'
    return result

print(addition('2')) # Складывать int и str нельзя!
print(addition([2])) # 4
print(addition({1: '2'})) # ошибка KeyError!
```

Как задать разную обработку разным типам

📍 Нужно написать несколько операторов `except`

Пример. Нужно отловить несколько видов исключений и каждому задать свою обработку.

Допустим, нужно:

- если исключение `TypeError`, вывести `Узнать размер числа нельзя!`;
- если получаем `KeyError`, вывести `В словаре нет ключа 0!`.

В таком случае понадобится несколько операторов `except`:

```
def get_size(some_arg):
    try:
        # пробуем получить и напечатать размер первого элемента
        print(len(some_arg[0]))

    except TypeError:
        print(f'Узнать размер числа нельзя!')

    except KeyError:
        print('В словаре нет ключа 0!')
```

Теперь программа перехватывает два типа исключений. Для каждого — своя обработка:

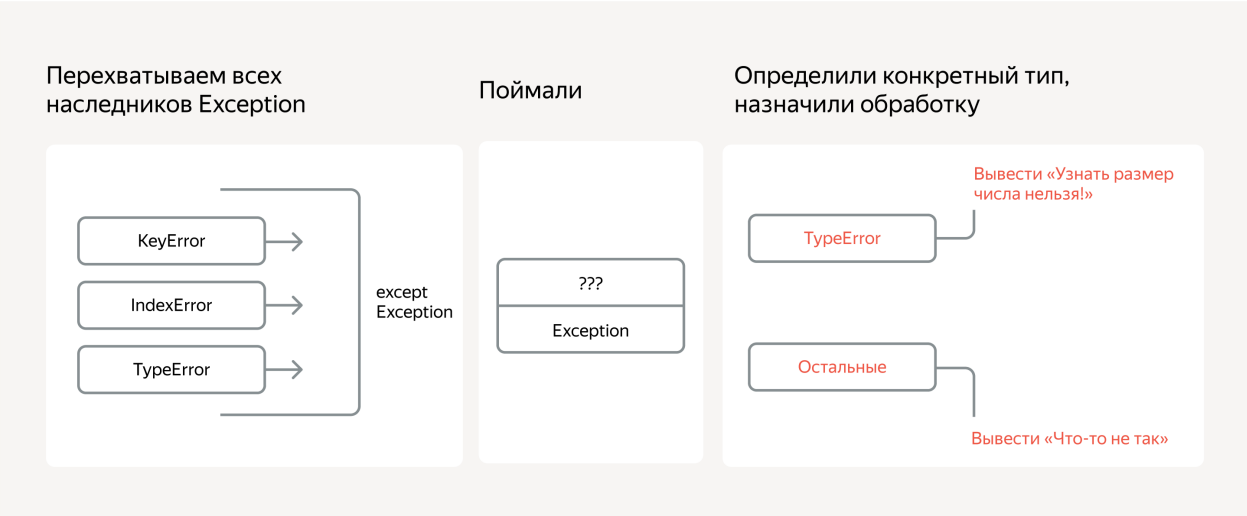
```
# функция принимает на вход аргумент любого типа
def get_size(some_arg):
    try:
        # пробуем получить и напечатать размер первого элемента нашего аргумента
        print(len(some_arg[0]))
    # при получении исключения, связанного с типом аргумента, печатаем следующий текст
    except TypeError:
        print(f'Узнать размер числа нельзя!')
    # при получении исключения, связанного отсутствием ключа в словаре, печатаем другой текст
    except KeyError:
        print('В словаре нет ключа 0!')

get_size('это строка') # 1
get_size(['это', 'список']) # 3
get_size(0) # Узнать размер числа нельзя!
get_size({'это': 'словарь'}) # В словаре нет ключа 0!
```

Как определить тип исключения

📍 Сохранить в переменную, а потом применить `type()`

Иногда заранее непонятно, какое исключение перехватится. То есть ты пишешь обработку для всех `Exception`, что-то ловится: тебе надо понять, что именно. Если это `KeyError` — выводить одно. Если `AttributeError` — другое.



Итак, перехватили исключение: нужно узнать, какой у него тип. Для этого его нужно сохранить в переменную, а потом применить `type()`.

```
def addition(some_arg):
    try:
        result = some_arg + 2
    except Exception as e: # сохранили исключение в переменную e
    #... тут будет продолжение кода
```

1. **Напечатать исключение.** Это нужно, чтобы получить его описание в удобном виде.

Дело в том, что в базовом классе всех исключений есть метод, который выводит полное описание исключения, — `__str__`. Без этой функции вернётся что-то вроде `<ClassObject.;5830h6rq1855>`, а с ней — название и описание `TypeError: type str can not …`.

Когда ты вызываешь `print()` для исключения, `__str__` автоматически вызывается под капотом. И вот у тебя есть понятное описание: как раз то, что нужно.

```
def addition(some_arg):
    try:
        result = some_arg + 2
    # назначаем переменной полученное исключение, чтобы иметь доступ к его параметрам
    except Exception as e:
        # получаем полное сообщение об ошибке
        print(e)
```

2. **Узнать тип.** Ты уже знаешь, как узнать тип переменной: тут всё то же самое. Нужно вызвать функцию `type()`. Как аргумент передай переменную, в которую сохранили перехваченное исключение.

```
def addition(some_arg):
    try:
        result = some_arg + 2
    # назначаем переменной полученное исключение, чтобы иметь доступ к его параметрам
    except Exception as e:
        # получаем полное сообщение об ошибке
        print(e)
        # получаем тип исключения, добавляем '__name__', чтобы получить само название
        print(type(e).__name__)
```

Обработать разные виды исключений одинаково

📍 Перечислить все виды в скобках после `except`. Например, `except (TypeError, KeyError)`

Пример. «Отбираем» только `KeyError` и `TypeError`. Текст выводим один и тот же. Можно написать два блока `except`, но код в них будет одинаковый. Это плохо: строчки дублируются.

Можно перехватить сразу несколько видов исключений в одном блоке. Для этого нужно просто перечислить все виды в скобках после `except`. Например, `except (TypeError, KeyError)`.

В коде:

```
# функция принимает на вход аргумент любого типа
def get_size(some_arg):
    try:
        # пробуем получить и напечатать размер первого элемента нашего аргумента
        print(len(some_arg[0]))
    # если получим исключение, связанное с типом аргумента или отсутствием
    # ключа, печатаем:
    except (TypeError, KeyError) as e:
        print(f'Произошла ошибка {type(e).__name__}. Узнать размер нельзя!')

get_size('это строка') # 1
get_size(['это', 'список']) # 3
get_size(0) # Произошла ошибка TypeError. Узнать размер нельзя!
get_size({'это': 'словарь'}) # Произошла ошибка KeyError. Узнать размер нельзя!
```

Оператор `else`

📍 Отвечает за действия, которые произойдут, если исключение не перехвачено. Часто его опускают для лаконичности: так тоже можно

В конструкции с `if` этот оператор означает: «Если ничего из вышеперечисленного не случилось, сделай такое-то действие». В обработке исключений он ведёт себя точно так же.

С его помощью программе говорят, что делать, если исключение не перехвачено: «Если никаких исключений не перехватили, сделай такое-то действие».

```
def get_some_sum(some_arg):
    try: # пробуем получить первый элемент полученного на вход аргумента
        element = some_arg[0]
    except Exception: # перехватываем исключения, если они есть
        result = 'Не удалось выполнить действие с элементом!'
    else: # если нет исключений, выполняем это действие
        result = element + 2
    return result

print(get_some_sum(2)) # Не удалось выполнить действие с элементом!
print(get_some_sum([2]))  # 4
print(get_some_sum({'2': 1})) # Не удалось выполнить действие с элементом!
```

Если написать обработку исключения без `else`, ничего страшного не случится. Python — интерпретируемый язык. Это значит, что действия программы выполняются построчно. Не во всех языках так: в некоторых нужны более чёткие инструкции. Там блок `else` был бы обязательным.

> 📎 Писать блок `else` считается правильным, потому что с ним код надёжнее. Но ради удобства и быстроты его «опускают»: обычно всё пишут в блоке `try`.

Оператор `finally`

> 📍 Помогает, если обязательно нужны какие-то финальные действия. Этот блок выполняется всегда: не важно, перехватили исключение или нет.

Пример. Допустим, нужно понимать, когда функция начала действие и когда закончила. Как это сделать: зафиксировать время в самом начале работы функции и в конце, вычесть из одного другое. Для конца как раз и пригодится `finally`. Перехватили исключения или нет — блок выведет время.

Чтобы узнать текущее время, используем библиотеку `datetime`.

```
import datetime as dt


def get_some_sum(some_arg):
    start_time = dt.datetime.now()
    try:
        result = some_arg[0] + 2
    except TypeError:
        result = len(some_arg[0]) + 2
    except KeyError:
        result = 'Не удалось выполнить действие с элементом!'
    finally:
        stop_time = dt.datetime.now()
        print(f'Программа выполнилась за {stop_time - start_time}')
    return result

print(get_some_sum('это строка'))
print(get_some_sum(['это список']))
print(get_some_sum({'2': 1}))

# Получим примерно такой вывод
# Программа выполнилась за 0:00:00.000816
# 3
# Программа выполнилась за 0:00:00.000941
# 5
# Программа выполнилась за 0:00:00.001040
# Не удалось выполнить действие с элементом!
```

Из вывода видно: на список и словарь программа потратит разное количество времени. Иногда от скорости выполнения программы много зависит. Например, действие нужно успеть сделать строго за 30 секунд, иначе дальше всё собьётся. Вот тут и будет важно отследить тайминг.

Пример 2. Допустим, в коде нужно открыть файл, а в конце программы обязательно его закрыть. Блок `finally` поможет закрыть файл, даже если программа поймала исключение.

```
def do_something_with_file():
    try:
        # Открытие файла в режиме записи
        file = open("file.txt", "w")
        print("Successfully opened the file")
        # записываем в файл строку 'Hello!'
        file.write('Hello!')
    except FileNotFoundError:
        # Обработка исключения, возникающего в том случае, если файл не найден
        print("File Not Found Error: No such file or directory")
        exit()
```

```
    except PermissionError:
        # Обработка ошибок, связанных с разрешением на доступ к файлу
        print("Permission Denied Error: Access is denied")
    # закрываем файл даже в том случае, если выше возникло исключение
    finally:
        file.close()
```

Что тут происходит:

- Программа пытается открыть файл `file.txt` для записи.
- Если открыть файл не получается и возникают ошибки `FileNotFoundError` или `PermissionError`, выполнятся блоки `except`.
- Если в блоке `try` исключений не возникло, программа спокойно обрабатывает содержимое файла.
- В блоке `finally` программа закрывает файл в любом случае. Не важно, возникли исключения или всё прошло гладко.

✍️ **Блок** `finally` выполняет действия в любом случае. Были исключения или нет — не важно.

Как вызвать исключение

📌 Понадобится оператор `raise`. Он ставит программу на паузу и «выбрасывает» исключение. Например, `raise Exception`

Бывает так, что какую-то ситуацию нужно проектировать как исключение. То есть программа могла бы без проблем работать дальше, но разработчику нужно принудительно попросить её сделать паузу и что-то вывести.

Пример. Функции `get_some_sum()` передают аргумент в виде списка. Она должна взять первый элемент и прибавить к нему число 2. Есть особенность: нужно проверить длину элемента. Если она равна двум, программа должна об этом сообщить как об ошибке.

То есть вообще это никакая не ошибка. Но почему-то программа спроектирована так, что должна сообщить о длине, равной 2.

Вот как выглядит пример с `get_some_sum()`:

```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
        # проверяем, чему равна длина элемента
        if len(some_arg) == 2:
            # вызываем исключение
            raise Exception
```

Теперь, как только длина элемента будет равна двум, программа остановится и сообщит об этом.

У этого кода есть недостаток: исключение вызывается, но программа не сообщает почему. Хорошим тоном будет спроектировать вызов исключения так, чтобы вывести понятное сообщение об ошибке. Например, `Нельзя обрабатывать список с размером 2!`.

Для этого после `raise Exception` нужно написать сообщение в скобках:

```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
        # проверяем, чему равна длина элемента
        if len(some_arg) == 2:
            # вызываем исключение
            raise Exception('Нельзя обрабатывать список с размером 2!')
```

Обрати внимание: после вызова исключения можно добавить обычную обработку для других случаев. Условно говоря, проверили: длина не равна двум. Дальше ловим другие исключения как обычно. Например, `TypeError` и `KeyError`:

```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
        # проверяем, чему равна длина элемента
        if len(some_arg) == 2:
            # вызываем исключение
            raise Exception('Нельзя обрабатывать список с размером 2!')
    except TypeError:
        result = len(some_arg[0]) + 2
    except KeyError:
        result = 'Не удалось выполнить действие с элементом!'
    return result
```

Как вызвать определённый тип исключения

📌 Понадобится оператор `raise`: вместо общего `Exception` пишешь конкретный подкласс. Например, `raise ValueError('Нельзя обрабатывать список с размером 2!')`

Можно вызывать не общее исключение `Exception`, а конкретный тип. Например, `ValueError` для списка. Тут всё как и с обычной обработкой — вместо общего `Exception` пишешь конкретный подкласс:

```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
        # проверяем, чему равна длина элемента
        if len(some_arg) == 2:
            # Указываем конкретное исключение, которое хотим получить
            raise ValueError('Нельзя обрабатывать список с размером 2!')
```

Ну и потом всё ещё можно перехватывать другие исключения:

```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
        # проверяем, чему равна длина элемента
        if len(some_arg) == 2:
            # Указываем конкретное исключение, которое хотим получить
            raise ValueError('Нельзя обрабатывать список с размером 2!')
    except TypeError:
        result = len(some_arg[0]) + 2
    except KeyError:
        result = 'Не удалось выполнить действие с элементом!'
    return result

# если передать строку, вызовется исключение TypeError
print(get_some_sum('это строка')) # 3
# В случае передачи словаря будет вызвано исключение KeyError
print(get_some_sum({'это': 'словарь'})) # Не удалось выполнить действие с элементом!
# В случае передачи списка с цифрами никакого исключения не должно быть вызвано
# и программа выполнится
print(get_some_sum([2, 3, 4]))
# а здесь мы получим исключение и программа прекратит свою работу, потому что
# решили принудительно вызывать исключение, если длина списка равна двум
print(get_some_sum([2, 3]))
```

Вызвать исключение несколько раз

Иногда нужно вызвать исключение несколько раз. Первый — чтобы оставить запись об ошибке в логах, а второй — чтобы выдать статус-код и сообщение об ошибке.

Представь: есть большой проект, который получает запросы и отдаёт ответы, то есть поддерживает API. У такого сервиса внутри будет много вспомогательных функций. Например, для расчётов чего-либо.

На каком-то этапе расчётов вызвано исключение. Нужно получить его статус-код и сообщение об ошибке. Так будет сразу понятно, что случай необычный: нужно обратить на него внимание и разобраться. Неправильные расчёты могут привести к большим убыткам.

Такая обработка исключений может выглядеть так:

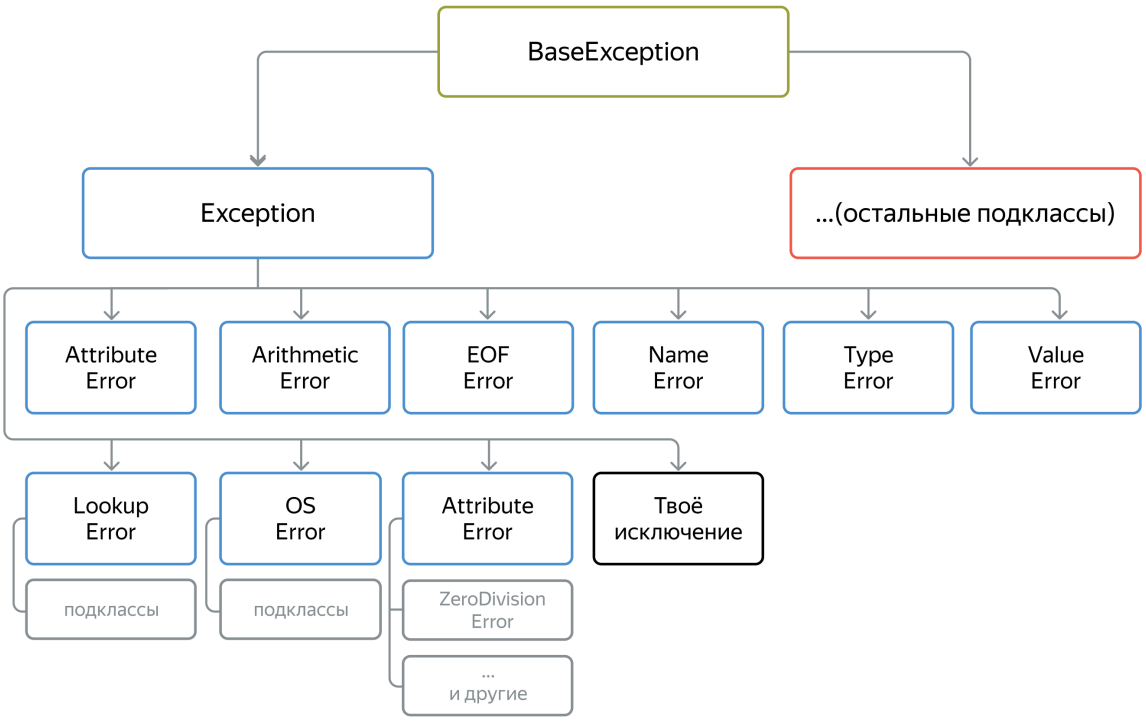
```
def get_some_sum(some_arg):
    try:
        result = some_arg[0] + 2
    # перехватываем исключение и сохраняем его в переменную
    except TypeError as e:
        # выводим результат обработки на экран. Вместо этого можно
        # производить логирование
        print(len(some_arg[0]) + 2)
        # принудительно вызываем ту же ошибку, что получили ранее
        raise e

# В случае передачи строки будет вызвано исключение TypeError.
# Мы обработаем его и выведем результат на экран, однако после этого снова
# вызовем то же самое исключение, и программа завершит свою работу
print(get_some_sum('это строка'))
```

Создать своё исключение

Иногда тебе может понадобиться создать своё исключение. Например, метод должен записать строку в переменную. Строка не должна превышать 255 символов. Нужно проверить длину: если она больше — выбросить исключение. Для этого случая можно создать исключение и назвать его, например, `TooLongArgument`.

Все типы исключений наследуются от класса `Exception`. Соответственно, чтобы создать своё исключение, понадобится добавить `Exception` ещё одного наследника.



Алгоритм такой:

1. **Создать класс-исключение.** Например, `class MyCustomException`.
2. **Сделать исключение наследником класса `Exception`.** Всё как обычно: `class MyCustomException(Exception)`.
3. **Определить в новом классе метод `__str__()`.** Про него уже заходила речь: он нужен, чтобы исключение выводило информацию о себе в читабельном виде.

> ⚠️ Писать метод `__str__` обязательно: если этого не сделать, программа не выведет сообщение об ошибке.

В родительском классе `Exception` уже есть метод `__init__`. Поэтому можно обойтись без него, если не нужно выводить разный текст для разных случаев:

```
# создаём собственный класс-исключение и наследуем его от класса Exception
class MyCustomException(Exception):
```

```
    # задаём метод для вывода информации на экран
    def __str__(self):
        return 'Произошло исключение MyCustomException'
```

Создали исключение, теперь можно вызвать его:

```
# создаём собственный класс-исключение и наследуем его от класса Exception
class MyCustomException(Exception):

    # задаём метод для вывода информации на экран
    def __str__(self):
        return 'Произошло исключение MyCustomException'


def test_my_custom_exception():
    try:
        raise MyCustomException
    except MyCustomException as e:
        print(e)

test_my_custom_exception() # Произошло исключение MyCustomException
```

Теперь сделаем исключение информативнее: напишем метод `__init__`. Так можно передать разный текст в разных ситуациях. Например, если это будет связано с заказами — передавать номер заказа. Аргументом конструктора будет сообщение, которое нужно выводить на экран:

```
# создаём собственный класс-исключение и наследуем его от класса Exception
class MyCustomException(Exception):

    # задаём метод инициации, который на вход может принимать аргумент
    def __init__(self, message=None):
        self.message = message

    # задаём метод для вывода информации на экран
    def __str__(self):
        # если при вызове нашего исключения было передано сообщение
        if self.message:
            # добавляем его в строку, которую вернёт и выведет на экран
            # исключение при вызове
            return f'MyCustomException: {self.message}'
        # если условие не выполнилось, то есть никакое сообщение при вызове
        # исключения не передавали, вернём какое-то дефолтное сообщение
        return 'Произошло исключение MyCustomException'


def test_my_custom_exception():
    try:
```

```
        raise MyCustomException('Вызвано пользовательское исключение!')
    except MyCustomException as e:
        print(e)
        print(e.message) # имеем доступ к полям, как и в обычном классе!

test_my_custom_exception() # MyCustomException: Вызвано новое исключение!
```

Теперь при вызове исключения без аргументов будет выведено дефолтное сообщение. А если передать аргумент, будет выводиться именно он.