Занятие 13

Обработка исключений: try…except

Функция-генератор, оператор yield

Найдите и исправьте три ошибки

Программист должен был написать программу подсчета количества строк в файле.

Он ее написал, но сделал три ошибки, каждая ошибка в отдельной строке.

```
total = 0
with open(“data.txt”, encoding='utf-8') as file:
    for _ in file.readlines():
        total = total + 1
print(total)
```

Задача 12-1

- Создайте функцию, которая принимает на вход список X и возвращает в качестве результата два списка:
- Список индексов элементов равных минимальному значению списка X
- Список индексов элементов равных максимальному значению списка X
- Например:
- Ввод: X = [1, 2, 3, 4, 1, 2, 3, 4, 1, 4]
- Вывод: [0, 4, 8], [3, 7, 9]

Задача 12-2

- Создайте списковое включение, которое генерирует следующую последовательность:
- 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, и т.д. до 10

Задача 12-3

- Напишите функцию, которая на вход принимает строку диапазонов натуральных чисел,

например: ‘1-2,4-4,3-6’.

- На выходе функция должна сформировать список натуральных чисел, которые попадают в один из этих диапазонов,

например: [1,2,4,3,4,5,6].

Тернарный оператор

- x = 1
- y = 2
- maximum = x if **x > y** else y

Например, можно так, но очень громоздко:

```
def abs(number):
    if number >= 0:
        return number
    return –number
```

А можно так, более лаконично:

```
def abs(number):
    return number if number >= 0 else -number
```

List comprehension

Do this	For this collection	In this situation
`[x**2`	`for x in range(0, 50)`	`if x % 3 == 0]`

1. При помощи цикла **for**

- s = []
- **for** i in range(10):
- s.append(i ** 3) # Добавляем к списку куб каждого числа
- print(s)
- # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

2. При помощи функции **map()**

- list(**map**(lambda x: x ** 3, range(0,10)))
- # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

3. При помощи конструкции **list comprehension**

- [x**3 for x in range(10)]
- # [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

Условие в конце включения

- **[**«значение» for «элемент списка» in «список» **if** «условие»**]**

- #Получить все нечетные цифры в диапазоне от 0 до 9
- [x for x in range(10) **if** x%2 == 1]
- #[1, 3, 5, 7, 9]

Условие в начале включения

- # Замена отрицального диапазона нулем
- >>> original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
- >>> prices = [i if i > 0 else 0 **for** i in original_prices]
- >>> prices
- [1.25, 0, 10.22, 3.78, 0, 1.16]

Вызов функции в выражении

```
# Замена отрицального числа нулем
original_prices = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]

def get_price(price):
    return price if price > 0 else 0

prices = [get_price(i) for i in original_prices]
```

Вложенные if else

- Постройте функцию FizzBuzz с помощью вложенных if else, функции, списковых включений (делится на 15-> FB, на 3->F, на 5->B)

```
def fu(x):
  return 'FizzBuzz' if x % 15 == 0 else 'Fizz' if x % 3 == 0 else 'Buzz' if x % 5 == 0 else x


a = [fu(x) for x in range(20)]
print(*a)
```

Вложенная генерация

- >>> matrix = [**[**i for i in range(5)**]** for j in range(6)]
- >>> matrix
- [
- [0, 1, 2, 3, 4],
- [0, 1, 2, 3, 4],
- [0, 1, 2, 3, 4],
- [0, 1, 2, 3, 4],
- [0, 1, 2, 3, 4],
- [0, 1, 2, 3, 4]
-]
- Внешний генератор [... for _ in range(6)] создает 6 строк в то время как внутренний генератор[i for i in range(5)]заполняет каждую строку значениями.

“Генераторы” словарей – dictionary comprehension

- Генерация словаря похожа на генерацию списка (list comprehension) и предназначена для создания словаря.
- d = {}
- for num in range(1, 10):
- d[num] = num**2
- print(d)
- {1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}
- d = **{** num:num ** 2 for num in range(1, 10)**}**
- >>>d
- {1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}

Задание

- Составьте список, основываясь на интервале от 1 до 10.
- Если число делится на 2, то включите его квадрат, если не делится, то куб.
- Т.е. на выходе должно получиться:
- [1, 4, 27, 16, 125, …, 100]

Обработка исключений

Обработка исключений в Python

Программа, написанная на языке Python, останавливается сразу как обнаружит ошибку.

Ошибки могут быть:

- **Синтаксические ошибки** — возникают, когда написанное выражение не соответствует правилам языка (например, написана лишняя скобка);

- **Логические ошибки** — это ошибки, когда синтаксис действительно правильный, но логика не та, какую вы предполагали. Программа работает успешно, но даёт неверные результаты.

- **Исключения** — возникают во время выполнения программы (например, при делении на ноль).

Перехват ошибок во время выполнения

- 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 ловушки **try/except**.

В каких случаях нужно предусматривать обработку ошибок ?

- **Работа с файлами** (нет прав доступа , диск переполнен и т.д)
- **Работа с СУБД** (сервер не доступен, ошибка выполнения запроса и т.д)
- **Работа с сетью** (сеть недоступна, ошибка соединения, обработка кода возврата и т.д)
- **Работа с различными библиотеками** которые бросают exception в случае обнаружения ошибки.

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.
- В случае когда ошибка не обрабатывается, программа прекратит свою работу и в консоли мы увидим Traceback с подробным описанием ошибки.

Иерархия классов исключений Python

- https://docs.python.org/3/library/exceptions.html

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
```

Некоторые исключения Python

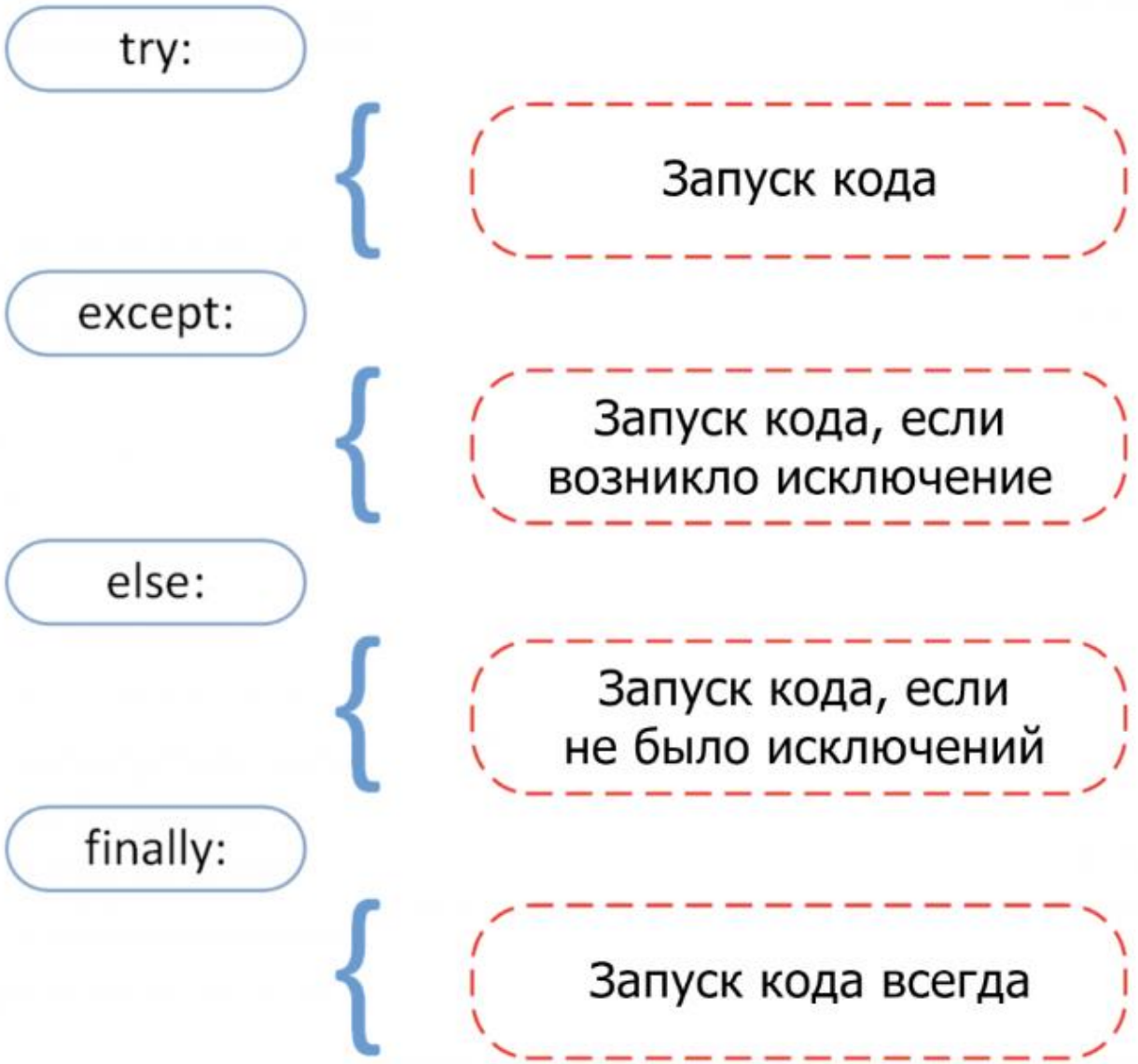
- **BaseException** - базовое исключение, от которого берут начало все остальные.
- **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
- **KeyboardInterrupt** - порождается при прерывании программы пользователем
- **GeneratorExit** - порождается при вызове метода close объекта generator.

- **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать, например:

- **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
- **ArithmeticError** - арифметическая ошибка.
- **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления.
- **ZeroDivisionError** - деление на ноль.

- Вызовите несколько ошибок. Деление на ноль, int от строки с буквами, неверный индекс списка.

Синтаксис конструкции



Пример

- **try:**
  a = 1/0

- **except ZeroDivisionError**:
  print("Возникло исключение:ошибка деления на ноль! ")
  a = 0

- **else:**
  print("Ветка else вызывается, если не возникло исключения")

- **finally:**
  print(«Идем дальше...")

Задание

- Напишите программу ввода целого числа.
- Если вводится “неправильная” строка, содержащая цифры с буквой или со знаком препинания или что-то другое, то программа должна сообщить, что это не целое число и предложить повторить ввод.
- Если вводится «правильная» строка, то напечатайте введенное число.
- Используйте исключение ValueError в бесконечном цикле.

Часто встречающиеся типы исключений

- Что касается типов исключений, наибольший интерес с практической точки зрения представляют следующие:

- IndexError: возникает, когда индекс (например, для элемента списка) указан неправильно (выходит за границы допустимого диапазона)
- KeyError: возникает при неверно указанном ключе словаря
- NameError: возникает, если не удается найти переменную с некоторым названием
- SyntaxError: возникает при наличии в исходном коде синтаксических ошибок
- TypeError: возникает при несоответствии типов, когда для обработки требуется значение определенного типа, а передается значение другого типа
- FileNotFoundError: возникает при открытии несуществующего файла
- ValueError: возникает, когда в функцию передается аргумент с неподдерживаемым значением
- ZeroDivisionError: возникает при попытке выполнить деление на ноль

Функция генератор

Заменим return на yield.Что получится?

```
def fun(n):
        for x in range(n):
                return x * x
g = fun(3)
print(g) # -> 0 и выполнение функции заканчивается.

# Заменяем return на yield
def fun(n):
        for x in range(n):
                yield x * x
g = fun(3)
print(g) # generator
for k in g:
        print(k)
```

Посмотрим внимательней, как это работает

```
def fun(n):
    for x in range(n):
        print("До yield”, x)
        yield x * x
        print("После yield") # Сюда возвращается выполнение программы!
g = fun(3)

for k in g:
    print("Перед печатью x * x")
    print(f"k = {k}")
    print("После печати x * x")
```

Что произошло?

- Мы создали функцию – генератор
- Она выдает не одно значение по return, а много по yield
- После выдачи значения, она сохраняет все локальные переменные
- Мы снова входим в эту функцию в оператор следующий за yield и продолжаем работу функции до следующего yield
- Мы можем поместить функции в цикл for, который с ней прекрасно работает
- Мы можем использовать этот генератор не только с for
- Но тогда мы можем столкнуться с исключением StopIteration

Задание

- Сделайте функцию генератор, которая выдает квадрат числа, если оно четное, и куб, если оно нечетное

Сравнение Return и Yield

RETURN	YIELD
Оператор return возвращает только одно значение.	Оператор yield может возвращать серию результатов в виде объекта-генератора.
Return выходит из функции, а в случае цикла он закрывает цикл. Это последний оператор, который нужно разместить внутри функции.	Не уничтожает локальные переменные функции. Выполнение программы приостанавливается, значение отправляется вызывающей стороне, после чего выполнение программы продолжается с последнего оператора yield.
Логически, функция должна иметь только один return.	Внутри функции может быть более одного оператора yield.
Оператор return может выполняться только один раз.	Оператор yield может выполняться несколько раз.
Return помещается внутри обычной функции Python.	Оператор yield преобразует обычную функцию в функцию-генератор.

Используем функцию-генератор по другому – next()

```
def fun(n):
        for x in range(n):
                yield x * x
 g = fun(3)
 print(next(g))
 print(next(g))
 print(next(g))
 print(next(g))  #  StopIteration
# Как это преодолеть? Например, так.
 g = fun(3)
 for k in range(10):
        try:
                print(next(g))
        except StopIteration:
                break
```

Выполните эту программу

Пример - факториал

```
def factorial():
        f, k = 1, 1
        while True:
                yield f
                k += 1
                f *= k
gf = factorial()
for m in gf:
        print(m)
        input()
```

Выполните эту программу, вставьте промежуточные печати.

Что сделать, чтобы эта программа не работала бесконечно?

Пример – факториал (другой способ)

```
def factorial():
        f, k = 1, 1
        while True:
                yield f
                k += 1
                f *= k
gf = factorial()

while True:   # for k in range(int(input("Введите число:")))
        print(next(gf))
        input()
```

Задание

Дан список чисел

Написать функцию-генератор, которая выдает последовательность сумм первых чисел списка.

Например:

[1, 10, 100, 2, 20, 200]

На выходе должно получиться 1, 11, 111, 113, 133, 333

Итак:

- Генераторы похожи на нормальные функции , но их поведение различаются.

Вызов функции-генератора вообще не выполняет функцию а просто создает и возвращает объект-генератор

- def repeater(value):
- while True
- **yield value**

- generator_obj = repeater(“Hello”)

- Программный код выполняется тогда когда функция next вызывается с объектом генератора

- next(generator_obj)

Как прекратить генерацию?

- Генераторы прекращают порождать значения, как только поток управления возвращается из функции-генератора каким-либо иным способом, кроме инструкции yield, например встречает **return**

```
def bounder_repeater(value, max_repeats):
        count = 0
        while True:
                if count >= max_repeats:
                        return
        count += 1
        yield value

for x in bounder_repeater(“Раз”,  2)
    print(x)
```

Или срабатывает StopIteration

- def repeater_tree_value():
    yield 1
    yield 2
    yield 3

  for i in repeater_tree_value():
    print(i)
- 1
- 2
- 3

- it = repeater_tree_value()
  print(next(it))
  print(next(it))
  print(next(it))
  print(next(it)) → **StopIteration**

Задание

Создайте функцию-генератор, которая создает последовательность:

1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, …

Задача 13-1

Создайте функцию-генератор, которая создает бесконечную последовательность:

1, -2, 3, -4, 5, -6, …

Задача 13-2

Создайте функцию-генератор, которая создает последовательность числовых палиндромов:

1 2 3 4 5 6 7 8 9 11 22 33 44 55 66 77 88 99 101 111 121 131 141 151 161 171 181 191 202 212…

Задача 13-3

Создайте функцию-генератор, который принимает в качестве аргумента список целых чисел, а в качестве результатов формирует последовательность нечетных чисел из этого списка.