

Занятие 14

Вложенность исключений, пользовательские исключения

Функция-генератор, оператор `yield from`

Рекурсия

Что напечатает?

```
lst = [1, 2, 3, 4, 5]
```

```
for x in lst:
```

```
    lst.pop()
```

```
print(lst)
```

```
lst = [1, 2, 3, 4, 5]
```

```
for x in lst:
```

```
    lst.pop(0)
```

```
print(lst)
```

Не трогайте объекты, по которым идет цикл for!!! (внутри работает next())

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

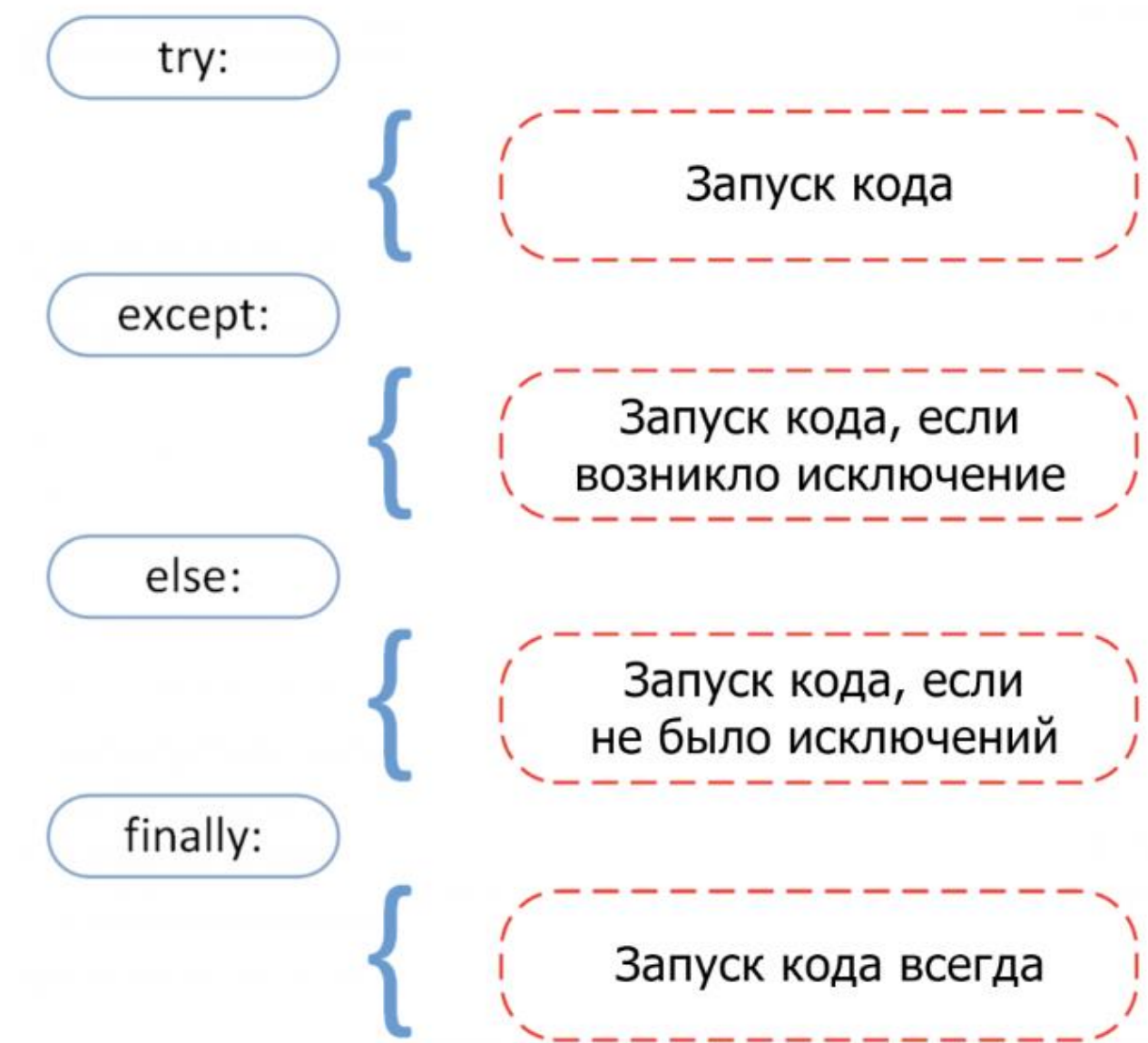
Создайте функцию-генератор, который принимает в качестве аргумента список целых чисел, а в качестве результатов формирует последовательность нечетных чисел из этого списка.

Иерархия классов исключений Python

<https://docs.python.org/3/library/exceptions.html>

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
├── Exception
│   ├── ArithmeticError
│   │   ├── FloatingPointError
│   │   ├── OverflowError
│   │   └── ZeroDivisionError
│   ├── AssertionError
│   ├── AttributeError
│   ├── BufferError
│   ├── EOFError
│   ├── ExceptionGroup [BaseExceptionGroup]
│   ├── ImportError
│   │   └── ModuleNotFoundError
│   ├── LookupError
│   │   ├── IndexError
│   │   └── KeyError
│   ├── MemoryError
│   ├── NameError
│   │   └── UnboundLocalError
│   ├── OSError
│   │   ├── BlockingIOError
│   │   ├── ChildProcessError
│   │   ├── ConnectionError
│   │   │   ├── BrokenPipeError
│   │   │   ├── ConnectionAbortedError
│   │   │   ├── ConnectionRefusedError
│   │   │   └── ConnectionResetError
│   │   ├── FileExistsError
│   │   ├── FileNotFoundError
│   │   ├── InterruptedError
│   │   └── IsADirectoryError
```

Синтаксис конструкции



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 удастся 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

Ловушка для двух исключений

```
def div(x, y):
```

```
    return x / y
```

```
try:
```

```
    result = div(100, 0)
```

```
    print("Расчёт проведён успешно")
```

```
except (ZeroDivisionError, KeyError) as e:
```

```
    print("Ошибка деления или ошибка обращения по ключу. Вот она:", e)
```

Несколько ловушек

```
def div(x, y):  
    return x / y
```

- try:
- result = div(100, 0)
- print("Расчёт проведён успешно")
- except **ZeroDivisionError** as e:
- print("Ошибка деления произошла", e)
- except **KeyError** as e:
- print("Ошибка обращения по ключу произошла:", e)

Порядок следования обработки

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

Какой блок поймает исключение если файл не обнаружен ?

А если поменять местами `except`?

Каскадное включение перехватчиков

```
def divide(x, y):
```

```
    try:
```

```
        out = x/y
```

```
    except:
```

```
        try:
```

```
            import math
```

```
            out = math.inf * x / abs(x)
```

```
        except:
```

```
            out = None
```

```
    finally:
```

```
        return out
```

```
print(divide(15, 3)) # 5.0
```

```
print(divide(15, 0)) # inf
```

```
print(divide(-15, 0)) # -inf
```

```
print(divide(0, 0)) # None
```

Генерация исключений в Python

- Для принудительной генерации исключения используется инструкция **raise**.
- try:
- **raise** Exception("Что то пошло не так")
- except Exception as e:
- print("Message:" + str(e))

Выполните этот код

Валидатор входного строкового значения на имя человека.

- `def validate(name):`
- `if len(name) < 10:`
- `raise ValueError`
- `try:`
- `name = input("Введите имя:")`
- `validate(name)`
- `except ValueError:`
- `print("Имя слишком короткое:")`
-

Пользовательские исключения

- В Python можно создавать собственные исключения.
- Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.
- ```
class NameTooShortError (ValueError) :
 pass

def validate(name) :
 if len(name) < 10 :
 raise NameTooShortError
```
- Выполните этот код

# Пользовательские исключения в Python

- Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений.
- `class NegValException(Exception):`
- `pass`
- `try:`
- `val = int(input("input positive number: "))`
- `if val < 0:`
- `raise NegValException("Neg val: " + str(val))`
- `print(val + 10)`
- `except NegValException as e:`
- `print(e)`



# Задание

Напишите функцию, которая принимает список чисел как аргумент и анализирует его.

Если число из списка отрицательное, то печатает «Отрицательное», если положительное, то – «Положительное», если 0, то печатает 0.

Создайте два пользовательских исключения: Positive и Negative, унаследованных от ValueError, которые печатают соответствующие слова.

```
class Positive(ValueError):
 pass
```

Функция генератор

# 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`

# 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
```

# Пример - факториал

```
def factorial():
```

```
 f, k = 1, 1
```

```
 while True:
```

```
 yield f
```

```
 k += 1
```

```
 f *= k
```

```
gf = factorial()
```

```
for m in gf:
```

# Первый вариант использования

```
 print(m)
```

```
while True:
```

# Второй вариант использования

```
 print(next(gf))
```

# Как еще прекратить генерацию?

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

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

for x in bouncer_repeater("Раз", 2)
 print(x)
```

# Что еще можно делать с функцией генератором?

```
def repeater123():
 yield 1
 yield 2
 yield 3
```

```
print(*repeater123())
```

# Распаковать значения

1 2 3

```
print(2 in repeater123())
```

# Использовать функцию in

True

```
n1, n2, n3 = repeater123()
```

# Множественное присвоение значений

```
print(n1, n2, n3)
```

1 2 3



# Что нельзя

- нельзя получить длину генератора функцией `len()`
- нельзя распечатать элементы генератора функцией `print()` без предварительной распаковки
- у генератора нельзя получить элемент по индексу
- после итерации по генератору он становится пустым

# yield from <iterable>

```
def fun_gen(n):
 for x in range(n):
 yield x
```

# Можно упростить

```
def fun_gen(n):
 yield from range(n)
```

# Например:

```
yield from 'abcdef'
```

```
yield from [11, 22, 33, 'abc', {1:111}]
```

```
yield from {1,22,333,4444}
```

# В каком порядке напечатаются эти числа?

# Вложенные генераторы

```
def fun_gen1():
 yield "Красный"
 yield "Зеленый"
 yield "Синий"

def fun_gen2():
 yield "Круглый"
 yield from fun_gen1()
 yield "Квадратный"

print(*fun_gen2())
```

# Что напечатает эта программа?

# Генераторные выражения

list comprehension:

```
list_comp = [x for x in range(10)] #
```

Список

- Генераторное выражение:

```
gen_expr = (x * x for x in range(10)) #
```

Генератор

```
for x in gen_expr: #
```

Можно обращаться как с функцией-генератором

```
 print(x)
```

```
print(next(gen_expr))
```

```
print(*gen_expr)
```

# Создайте “почти” одинаковые list compr. и ген.выражение и напечатайте содержимое того и другого

# Что нельзя делать с генераторными выражениями?

- Нельзя получить длину генераторного выражения с помощью встроенной функции `len()`.
- Нельзя распечатать элементы генераторного выражения с помощью функции `print()`, без предварительной распаковки.
- Генераторные выражения не поддерживают получение элемента по индексу.
- К генераторному выражению нельзя применить обычные операции среза.
- После использования генераторного выражения, оно остается пустым.

# Конвейеры генераторов

```
def integers(n):
 for i in range(1, n + 1):
 yield i

def evens(iterable):
 for i in iterable:
 if not i % 2:
 yield i

def squared(iterable):
 for k in iterable:
 yield k * k

chain = squared(evens(integers(10)))
print(*chain) # Что будет напечатано?
```

# Задание

Создайте конвейер генераторов,

Первый создает последовательность чисел от 65 до ... (подберите сами),

Второй – к каждому этому числу применяет функцию `chr()`

На выходе получается последовательность больших латинских букв: A B C D ... X Y Z





# Может ли функция вызвать себя?

```
def func():
 print("func")
 input()
 func()
```

```
func()
```

```
Что будет напечатано?
```

# А теперь?

```
def func(n):
 print("func", n)
 input()
 n -= 1
 if n > 0:
 func(n)
```

```
func(5)
```

```
Что будет напечатано?
```

# А теперь?

```
def func(n):
 print("func1", n)
 input()
 n -= 1
 if n > 0:
 func(n)
 print("func2", n)
 input()
 return
```

func(5)

# Что будет напечатано?

# Рекурсия – расчет факториала

```
def fact(n):
```

```
 if n == 1:
```

```
 return 1
```

# **базовый** случай – вариант решения

# без рекурсии

```
 else:
```

```
 return n * fact(n - 1)
```

# **рекурсивный** случай – сведение задачи

# к более простой

```
print(fact(1))
```

```
print(fact(2))
```

```
print(fact(3))
```

# Задание

Напишите рекурсивную функцию `triangle(n)` печати треугольника из звездочек.

Например, для `n = 5`

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

# Задание

- Напишите рекурсивную функцию `sumn(n)`, которая вычисляет и печатает сумму чисел от 0 до  $n$

# Задание

- Напишите рекурсивную функцию расчета чисел Фибоначчи:
- 0, 1, 1, 2, 3, 5, 8, 13, и т.д.

## Задача 14-1

- Напишите рекурсивную функцию, которая вычисляет количество цифр введенного натурального числа



## Задача 14-2

- Напишите рекурсивную функцию, которая вычисляет сумму цифр натурального числа

# Задача 14-3

- Напишите рекурсивную функцию `tri_2(n)`, которая печатает два треугольника.
- Например, для  $n = 5$ :
  - \*\*\*\*\*
  - \*\*\*\*\*
  - \*\*\*
  - \*\*
  - \*
  - \*\*
  - \*\*\*
  - \*\*\*\*
  - \*\*\*\*\*
- Подсказка: одна строка печатается до вызова функции, а вторая после вызова