



# Лекция 5. Итераторы и генераторы

Погружение в Python



# Оглавление

На этой лекции мы	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
<b>1. Однострочники</b>	<b>4</b>
Полезные однострочники	4
Обмен значения переменных	4
Распаковка коллекции	4
Распаковка коллекции с упаковкой “лишнего”, упаковка со звёздочкой	5
Распаковка со звёздочкой	6
Множественное присваивание	6
Множественное сравнение	7
Плохие однострочники	8
Задание	8
<b>2. Итераторы</b>	<b>9</b>
Функции <code>iter()</code> и <code>next()</code>	9
Функция <code>iter</code>	9
Функция <code>next</code>	11
Задание	12
<b>3. Генераторы</b>	<b>12</b>
Комбинации <code>for</code> и <code>if</code> в генераторах и выражениях	13
Допустимые размеры односточника	14
List comprehensions	15
Генераторные выражения или генерация списка	16
Set comprehensions	16
Dict comprehensions	17
Задание	18
<b>4. Создание функции генератора</b>	<b>18</b>

Команда yield	19
Функции iter и next для генераторов	20
Задание	20
Вывод	20

## На этой лекции мы

1. Разберем решения задач в одну строку
2. Изучим итераторы и особенности их работы
3. Узнаем о генераторных выражениях и генераторах списков, словарей, множеств
4. Разберем создание собственных функций генераторов.

## Краткая выжимка, о чём говорилось в предыдущей лекции

### На прошлой лекции мы:

1. Разобрались с созданием собственных функций в Python
2. Изучили работу встроенных функций “из коробки”

## Термины лекции

- **Итератор** — это объект, представляющий поток данных; объект возвращает данные по одному элементу за раз. Итератор Python должен поддерживать метод с именем `__next__()`, который не принимает аргументов и всегда возвращает следующий элемент потока.
- **Генератор** — это объект, который сразу при создании не вычисляет значения всех своих элементов. Он хранит в памяти только последний вычисленный элемент, правило перехода к следующему и условие, при котором выполнение прерывается. Вычисление следующего значения происходит

лишь при выполнении метода `next()`. Предыдущее значение при этом теряется.

- **Факториал** — функция, определённая на множестве неотрицательных целых чисел. Название происходит от лат. *factorialis* — действующий, производящий, умножающий; обозначается  $n!$ , произносится эн факториал. Факториал натурального числа  $n$  определяется как произведение всех натуральных чисел от 1 до  $n$  включительно.

## Подробный текст лекции

### 1. Однострочники

Python позволяет разработчикам решать задачи быстрее, чем с использованием других языков. Одна из причин — возможность писать меньше кода для получения результата.

### Полезные однострочники

Рассмотрим несколько примеров кода в одну строку, которые упрощают жизнь.

- Обмен значения переменных

Начнём с классического обмена значений переменных.

```
a, b = b, a
```

Мы поменяли местами содержимое переменных без создания дополнительных, в одну строку.

- Распаковка коллекции

Рассмотрим другие варианты упаковки и распаковки значений.

```
a, b, c = input("Три символа: ")
print(f'{a=} {b=} {c=}')

```

Как вы помните функция `input` возвращает строку `str`. Указав не одну переменную, а три мы распаковываем строку из 3-х символов в три отдельные переменные.

```
a, b, c = ("один", "два", "три",)
print(f'{a=} {b=} {c=}')
```

Аналогичным образом можно распаковать кортеж из трёх элементов в три переменные. Со списком `list`, множеством `set` и прочими коллекциями будет работать аналогично.

```
a, b, c = {"один", "два", "три", "четыре", "пять"}
print(f'{a=} {b=} {c=}')
```

*# ValueError: too many values to unpack (expected 3)*

Если количество переменных слева от равенства не совпадает с количеством элементов последовательности Python вернёт ошибку.

- Распаковка коллекции с упаковкой “лишнего”, упаковка со звёздочкой

Для упаковки может применяться символ “звёздочка” перед именем переменной. Такая переменная превратится в список и соберёт в себя все значения, не поместившиеся в остальные переменные.

```
data = ["один", "два", "три", "четыре", "пять", "шесть", "семь",
]
a, b, c, *d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
a, b, *c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
a, *b, c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
*a, b, c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

Элементы коллекции попадают в переменные в зависимости от того, какая из переменных отмечена звёздочкой.



**Важно!** Звёздочкой можно отметить только одну переменную из перечня.

Если нам нужна часть данных в переменных, а упакованный список в дальнейших расчётах не участвует, в качестве переменной используют подчеркивание.

```
link = 'https://docs.python.org/3/faq/programming.html#how-can-i-pass-optional-or-keyword-parameters-from-one-function-to-another'
prefix, *_ , suffix = link.split('/')

```

- Распаковка со звёздочкой

Ещё один способ применения звёздочки — распаковка элементов коллекции.

Длинный вариант вывода элементов последовательности в одну строку с разделителем табуляцией:

```
data = [2, 4, 6, 8, 10, ]
for item in data:
    print(item, end='\t')

```

И аналогичная операция в одну строку с распаковкой:

```
data = [2, 4, 6, 8, 10, ]
print(*data, sep='\t')

```

- Множественное присваивание

Если несколько переменных должны получить одинаковые значение, можно объединить несколько строк в одну.

```
a = b = c = 0
a += 42
print(f'{a=} {b=} {c=}')

```

Подобная запись допустима только с неизменяемыми типами данных. В противном случае изменение одной переменной приведёт к изменению и других.

```
a = b = c = {1, 2, 3}
a.add(42)
print(f'{a=} {b=} {c=}')

```


Другой вариант множественного присваивания похож на обмен переменных местами.

```
a, b, c = 1, 2, 3
print(f'{a=} {b=} {c=}')
```

Число элементов в левой части должно совпадать с числом элементов справа от равно.

А если в левой части указать лишь одну переменную, получим кортеж.

```
t = 1, 2, 3
print(f'{t=}, {type(t)}')
```

 **Важно!** Тип объектов может отличаться. Не только целые числа, как в примерах. Строки, любые коллекции. Ошибки это не вызовет. Но для повышения читаемости рекомендуется не смешивать разные типы данных при присваивании одной строкой.

- Множественное сравнение

Аналогично присваиванию можно сравнить несколько переменных внутри конструкции if.

```
a = b = c = 42
# if a == b and b == c:
if a == b == c:
    print('Полное совпадение')
```

Запись становится короче, т.к. исключается команда and внутри сравнения. Работает подобная запись не только с проверкой на равенство, но и с другими операциями.

```
if a < b < c:
    print('b больше a и меньше c')
```

Проверяем, что b больше a и b меньше c.

# Плохие однострочники

А теперь несколько примеров плохого кода. Да, он будет работать, т.к. Python прощает ошибки. Но читать такой код будет сложно. И есть большая вероятность, что коллеги по проекту не простят подобный код.

```
a = 12; b = 42; c = 73
if a < b < c: b = None; print('Ужасный код')
```

Перечислили несколько операций присваивания через точку с запятой в одной строке. Такая запись противоречит PEP-8. Скорее это защита от поломки, если разработчик пришёл из другого языка и привык ставить точку с запятой.

Во второй строке не перешли на новую строку с отступами после двоеточия. Так же идёт против PEP-8. Кроме того подобные условия, циклы или функции не смогут содержать более одной вложенной строки. Если конечно не начать добавлять точки с запятой.



**Очень важно!** Отсутствие перехода на новую строку после двоеточия и запись нескольких строк кода в одну через точку с запятой — плохой стиль программирования. Будьте готовы получить “неудовлетворительно” за подобные антипаттерны во время учёбы и отказ в трудоустройстве во время собеседования!

## Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет print, не запуская код. У вас 3 минуты.

```
data = {10, 9, 8, 1, 6, 3}
a, b, c, *d, e = data
print(a, b, c, d, e)
```



## 2. Итераторы

С итераторами мы уже знакомы. Любая Python коллекция будь то список, словарь, строка и т.п. предоставляют интерфейс итератора. Если коллекцию можно передать в цикл `for in` для последовательного перебора элементов, значит коллекция итерируемая, поддерживает интерфейс итерации. При этом у каждой коллекции может быть свой интерфейс. Списки, строки, кортежи возвращают элементы слева направо, от нулевого индекса к последнему. Множества возвращают элементы в случайном порядке.



**Секрет!** На самом деле порядок вывода элементов множества не случаен. Он зависит от результата работы встроенного хэша. Хэш функция определяет в какое место множества будет помещён элемент и возвращает их в порядке возрастания хэша.

Просто этот порядок может не совпадать со значением элементов.

Что касается словарей, они поддерживают сразу три интерфейса итерации: по ключам, по значениям и по парам ключ-значение. Вспомните методы `keys`, `values` и `items`.

## Функции `iter()` и `next()`

В Python объект является итерируемым, если поддерживает работу дандер методов `__iter__` (или `__getitem__`) и `__next__`. Первый метод должен возвращать объект итератор. Второй, `next` — возвращает очередной элемент коллекции или исключение `StopIteration`. Рассмотрим подробнее.

- Функция `iter`

Функция `iter` имеет формат `iter(object[, sentinel])`. `object` является обязательным аргументом. Если объект не реализует интерфейс итерации через методы `__iter__` или `__getitem__`, получим ошибку `TypeError`.


```
a = 42
iter(a)  # TypeError: 'int' object is not iterable
```

Получим итератор списка

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(list_iter)
```

Напрямую извлечь данные из итератора не получится. Python сообщает, что переменная `list_iter` указывает на `<list_iterator object at 0x0000025383D29400>`, т.е. объект итератор списка. Для кортежа функция `iter` вернёт `tuple_iterator`, для множеств `set_iterator`, а например для `dict.items()` вернётся `dict_itemiterator`. Один из простейших способов получить элементы - распаковать итератор через звёздочку.

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(*list_iter)
print(*list_iter)
```

 **Внимание!** Обратите внимание, что итератор является одноразовым объектом. Получив все элементы коллекции один раз он перестает работать. Для повторного извлечения элементов необходимо создать новый итератор.

Второй параметр функции `iter` — `sentinel` передают для вызываемых объектов-итераторов. Параметр указывает в какой момент должна быть завершена итерация, при совпадении возвращаемого значения со значением `sentinel`.

```
data = [2, 4, 6, 8]
list_iter = iter(data, 6)  # TypeError: iter(v, w): v must be callable
```

Список не является функцией, его нельзя вызвать. Получили ошибку `TypeError`. Один из вариантов работы функции `iter` с двумя параметрами — чтение бинарного файла блоками фиксированного размера до тех пор, пока не будет достигнут конец файла.

```
import functools

f = open('mydata.bin', 'rb')
for block in iter(functools.partial(f.read, 16), b''):
    print(block)
f.close()
```

Подробнее работу с файлами разберём на одной из следующих лекций. А сейчас коротко. Открыли бинарный файл на чтение байт. В цикле считываем блоки по 16 байт и выводим их на печать. Чтение прекратится после считывание пустого байта b'', окончания файла. После этого выходим из цикла и закрываем файл.

- Функция next

Функция next имеет формат next(iterator[, default]). На вход функция принимает итератор, который вернула функция iter. Каждый вызов функции возвращает очередной элемент итератора.

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter)) # StopIteration
```

При завершении элементов выбрасывается исключение StopIteration. Данное исключение служит указанием циклу for in о завершении работы. Каждый раз когда происходит перебор коллекции в цикле создаётся исключение. Но цикл обрабатывает его как сигнал для завершения итерации и перехода к следующему блоку кода. Исключение не останавливает программу.

Второй параметр функции next нужен для возврата значения по умолчанию вместо выброса исключения StopIteration

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
```

Промежуточный итог. Любая коллекция в Python поддерживает интерфейс итерации, т.е. перебор элементов в цикле for in. Для этого она возвращает итератор, который последовательно возвращает следующий, next элемент. Отдельно управлять итерацией позволяют функции iter и next. Кроме того при ООП можно создавать свои классы, поддерживающие итерации. Подробнее об этом в рамках соответствующей лекции.

# Задание

Перед вами несколько строк кода. Напишите, что выведет каждая из строк, не запуская код. У вас 3 минуты.

```
data = {"один": 1, "два": 2, "три": 3}
x = iter(data.items())
print(x)
y = next(x)
print(y)
z = next(iter(y))
print(z)
```

## 3. Генераторы

Возвращаемся к однострочникам и рассмотрим генераторы в Python.



**Важно!** Генератор не обяз быть однострочником.

Генераторы похожи на итераторы тем, что возвращают некую последовательность значений. Отличие в том, что итераторы чаще всего возвращают коллекции. Т.е. коллекция хранит все данные и тратит на хранение память. Далее коллекция возвращает хранимые элементы через интерфейс итерации. Генераторы не хранят данные в памяти, а вычисляют их по мере необходимости, чтобы вернуть очередное значение.

С одним из генераторов мы уже знакомы. Это объект, возвращаемый функцией `range`. При создании генератора мы указываем диапазон перебираемых целых чисел, но не сохраняем их в памяти. Каждое из значений генерируется на очередном витке цикла.

```
a = range(0, 10, 2)
print(f'{a=}, {type(a)=}, {a.__sizeof__()=}, {len(a)}')
b = range(-1_000_000, 1_000_000, 2)
print(f'{b=}, {type(b)=}, {b.__sizeof__()=}, {len(b)}')
```

Генератор `a` на пять значений и генератор `b` на 1 млн. значений занимают одинаковое место в памяти.

# Генераторные выражения

Генераторные выражения Python позволяют создать собственный генератор, перебирающий значения.

```
my_gen = (chr(i) for i in range(97, 123))
print(my_gen)          # <generator object <genexpr> at
                        # 0x000001ED58DD7D60>
for char in my_gen:
    print(char)
```

Для создания генераторного выражения используют круглые скобки, внутри которых прописывается логика выражения. В нашем примере циклический перебор целых чисел от 97 до 122 и возврат символов из таблицы ASCII с соответствующими кодами.

## Комбинации for и if в генераторах и выражениях

В общем виде генератор можно записать как:

```
gen = (expression for expr in sequence1 if condition1
        for expr in sequence2 if condition2
        for expr in sequence3 if condition3
        ...
        for expr in sequenceN if conditionN)
```

Если расписать выражение в обычном коде, получим следующий код:

```
for expr in sequence1:
    if not condition1:
        continue
    for expr in sequence2:
        if not condition2:
            continue
        ...
        for expr in sequenceN:
            if not conditionN:
                continue
```

Каждый следующий цикл вложен в предыдущий. Логическая проверка определяет добавлять элемент в вывод или опустить его — `continue`. При этом логические проверки могут отсутствовать для любых циклов в любом порядке.

При работе с генераторами стоит помнить, что для каждого витка внешнего цикла внутренний перебирает все элементы от начала до конца. Например:

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
mult = (i + j for i in x if i % 2 != 0 for j in y if j != 1)
res = list(mult)
print(f'{len(res)=}\n{res}')
```

Создаём генератор на основе двух списков `x` и `y`. 7 элементов в первом списке и 6 во втором. Всего 13. Генератор считает сумму пар элементов.

Генератор перебирает все значения списка `x` и оставляет только нечётные. Исключаем 2 и 8, т.е. оставляем 5 из 7 элементов списка для вычисления суммы. В списке `y` исключаем единицу, т.е. оставляем 5 из 6 элементов. Новичок может подумать, что на выходе получим 10 элементов - 5 из `x` и 5 из `y`. Но циклы вложены друг в друга, следовательно количество элементов на выходе  $5 \times 5 = 25$ . А асимптотика данного генератора квадратичная,  $O(N \times M)$ , где `N` и `M` - длина списков `x` и `y`.



**Важно!** На асимптотическую сложность генератора влияют только количество циклов. Наличие `if` проверок конечно же замедляет генерацию значений. Но `if` воспринимается как константа в вычислении асимптотики. 4 вложенных цикла без проверок будут иметь асимптотику 4 степени, а 3 цикла с 3 проверками — асимптотику 3-й степени. Не стоит злоупотреблять количеством вложенных циклов.

## Допустимые размеры односточника

**РЕР-8!** Ограничение в 80 (120) символов на строку касается и генераторов. Если ваш код выходит за границу, попробуйте его упростить. Если упрощение невозможно, стоит перейти от однострочного генераторного выражения к функции генератору. Их мы рассмотрим далее на уроке.

Аналогичные ограничения касаются и рассматриваемых далее генераторов списков, множеств и словарей.

# List comprehensions

Что будет, если генераторное выражение записать не в круглых скобках, а в квадратных? Получим list comprehensions. Другие названия: list comp, генератор списков, списковое включение. И нет, это не генераторное выражение. Генератор списков полностью формирует список с элементами до его присваивания переменной слева от знака равно.

```
my_listcomp = [chr(i) for i in range(97, 123)]
print(my_listcomp)  # ['a', 'b', 'c', 'd', ..., z]
for char in my_listcomp:
    print(char)
```

Как и генераторные выражения списковые включения поддерживают несколько циклов и логические проверки для каждого из циклов. Можно воспринимать их как синтаксический сахар, более короткую запись. Например выбираем все чётные числа из исходного списка и складываем их в результирующий.

Длинный код:

```
data = [2, 5, 1, 42, 65, 76, 24, 77]
res = []
for item in data:
    if item % 2 == 0:
        res.append(item)
print(f'{res = }')
```

Аналогичное решение, но с использованием синтаксического сахара listcomp:

```
data = [2, 5, 1, 42, 65, 76, 24, 77]
res = [item for item in data if item % 2 == 0]
print(f'{res = }')
```

1. Не создаём пустой список в начале.
2. Не пишем двоеточия после цикла и логической проверки.
3. Исключаем метод append.

Итого вместо 4 строк кода получаем одну.

# Генераторные выражения или генерация списка

В примере из раздела “генераторные выражения” мы обернули генератор функцией `list`, чтобы сохранить значения в список. Можно воспринимать это как антипример кода. Какой же пример является верным? Если на выходе нужен готовый список, оптимальным будет следующий код:

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
res = [i + j for i in x if i % 2 != 0 for j in y if j != 1]
print(f'{len(res)=}\n{res}')
```

А если нам не нужны все элементы разом. Например мы в дальнейшем хотим перебирать значения по одному в цикле. В этом случае подойдет генераторное выражение без преобразования в список.

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
mult = (i + j for i in x if i % 2 != 0 for j in y if j != 1)
for item in mult:
    print(f'{item = }')
```



**Важно!** При написании кода заранее решите нужна вам сгенерированная коллекция целиком или нет. Не стоит тратить память на хранение всех элементов, если вы ими не пользуетесь одновременно.

## Set comprehensions

Кроме синтаксического сахара для генерации списков можно создавать множества в одну строку. Синтаксис аналогичен примерам выше. Изменяются лишь скобки. Для множеств используются фигурные.

```
my_setcomp = {chr(i) for i in range(97, 123)}
print(my_setcomp)  # {'f', 'g', 'b', 'j', 'e', ... }
for char in my_setcomp:
```



```
print(char)
```

Мы также перебираем элементы в цикле. Также можно использовать вложенные циклы. Также для каждого цикла может быть проверка на включение элемента в множество.

Стоит обратить внимание на следующие особенности:

- порядок элементов внутри множества может не совпадать с порядком добавления элементов.
- множество хранит только уникальные значения

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
res = {i + j for i in x if i % 2 != 0 for j in y if j != 1}
print(f'{len(res)=}\n{res}')
```

Как и в примерах выше для генерации множества перебрали все возможные комбинации пар *x* и *y* списков. Но в итоге осталось не 25 элементов, а 19 уникальных. 6 дублирующих элементов не были добавлены в множество, но время на их обработку было затрачено. Асимптотика не улучшилась.

## Dict comprehensions

Ещё один вариант синтаксического сахара — генерация словаря.

```
my_dictcomp = {i: chr(i) for i in range(97, 123)}
print(my_dictcomp)  # {97: 'a', 98: 'b', 99: 'c', ... }
for number, char in my_dictcomp.items():
    print(f'dict[{number}] = {char}')
```

Запись похожа на создание множества, но в качестве выражения для добавления указываются две переменные через двоеточие: *key: value*. Благодаря такой записи Python понимает, что надо создать словарь.



**Важно!** Стоит помнить, что ключи словаря должны быть объектами неизменяемого типа.

Во всём остальном для генерации словарей в одну строку действуют те же правила, что и для других типов данных.

## Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет print, не запуская код. У вас 3 минуты.

```
data = {2, 4, 4, 6, 8, 10, 12}
res1 = {None: item for item in data if item > 4}
res2 = (item for item in data if item > 4)
res3 = [[item] for item in data if item > 4]
print(res1, res2, res3)
```

## 4. Создание функции генератора

Рассмотрим создание генератора не в одну строку, а как отдельную функцию. Например нам надо посчитать факториал чисел от одного до n.

Прежде чем создавать генератор, создадим обычную функцию, которая вернёт список чисел.

```
def factorial(n):
    number = 1
    result = []
    for i in range(1, n + 1):
        number *= i
        result.append(number)
    return result

for i, num in enumerate(factorial(10), start=1):
    print(f'{i}! = {num}')
```

Внутри функции создали переменную для хранения очередного числа и результирующий список. Далее в цикле перебираем числа от одного до n включительно. Число number умножается на очередное число, вычисляется следующий по порядку факториал. Результат помещается в список. По завершении

цикла возвращаем список ответов.

Получив нужное количество значений в цикле выводим факториалы и их значения. Код отлично работает, но есть но. Мы не используем все факториалы сразу, а последовательно выводим их на печать. Если бы у нас был однострочный listcomp, достаточно было бы поменять квадратные скобки на круглые и получить генераторное выражение. В нашем примере также заменим функцию на генератор.

## Команда yield

Как вы помните команда return возвращает готовый результат работы функции и завершает её работу. Зарезервированное слово yield превращает функцию в генератор. Значение после yield возвращается из функции. Сама функция запоминает своё состояние: строку, на которой остановилось выполнение, значения локальных переменных. Повторный вызов функции продолжает работу с момента остановки.

Изменим функцию для получения факториала чисел, превратив её в генератор.

```
def factorial(n):
    number = 1
    for i in range(1, n + 1):
        number *= i
        yield number

for i, num in enumerate(factorial(10), start=1):
    print(f'{i}! = {num}')
```

Теперь внутри функции не создаётся пустой список для результатов. В цикле вычисляется факториал очередного числа. Далее команда yield возвращает значение. Следующий вызов вернёт функцию к циклу for для вычисления очередного числа.

Как вы помните, если в функции отсутствует команда return Python в конце тела функции добавляет return None. Явная или неявная, как в нашем примере, команда return завершает работу генератора вызовом исключения StopIteration.

# Функции `iter` и `next` для генераторов

Уже знакомые по сегодняшнему уроку функции `iter` и `next` могут работать с созданными генераторами. Например так:

```
my_iter = iter(factorial(4))
print(my_iter)
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
```

## Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет `print`, не запуская код. У вас 3 минуты.

```
def gen(a: int, b: int) -> str:
    if a > b:
        a, b = b, a
    for i in range(a, b + 1):
        yield str(i)

for item in gen(10, 1):
    print(f'{item} = ')
```

## Вывод

На этой лекции мы:

1. Разобрали решения задач в одну строку
2. Изучили итераторы и особенности их работы

3. Узнали о генераторных выражениях и генераторах списков, словарей, множеств
4. Разобрали создание собственных функций генераторов.

## Краткий анонс следующей лекции

- 1.

## Домашнее задание

Возьмите несколько задач из прошлых уроков, где вы создавали функции и сделайте из них генераторы. Внесите правки в код, чтобы он работал без ошибок в новой реализации.



**Подсказка:** замените `return` на `yield` в теле функций.