

# Погружение в Python

Урок 5

Итераторы и генераторы



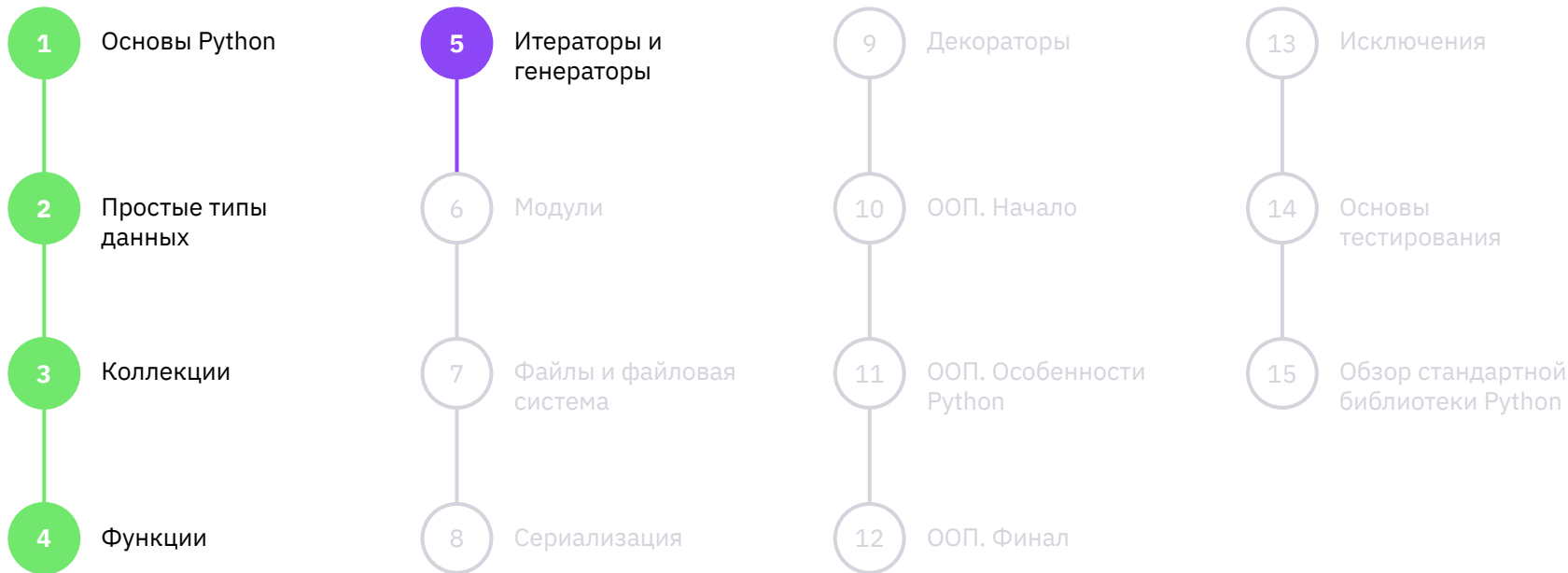


# Содержание урока









## План курса



## Что будет на уроке сегодня

-  Разберём решения задач в одну строку.
-  Изучим итераторы и особенности их работы.
-  Узнаем о генераторных выражениях и генераторах списков, словарей, множеств.
-  Разберём создание собственных функций генераторов.





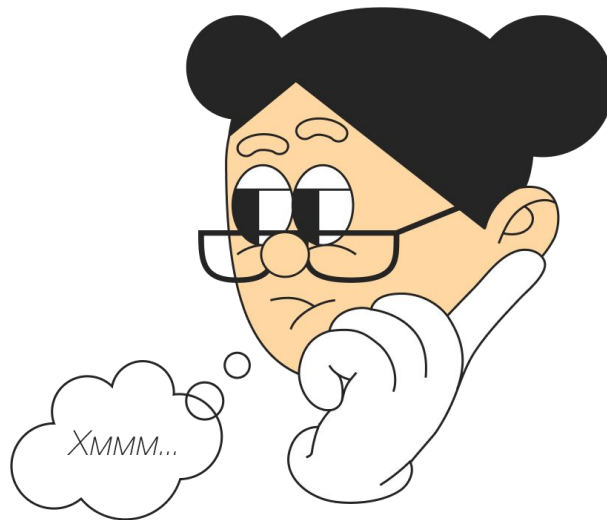
# Однострочники



## Обмен значения переменных

Классика Python

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





# Распаковка

Варианты распаковки значений



## Обычная распаковка

```
a, b, c =  
последовательность
```



## Распаковка с упаковкой

```
a, *b, c =  
последовательность
```



## Распаковка со звёздочкой

```
*последовательность
```



# Множественное присваивание и сравнение

Объединяем несколько операций в одну

1

## Присваивание

```
a = b = c = 0  
a, b, c = 1, 2, 3
```

2

## Сравнение

```
a == b == c  
a < b < c
```



Множественное присваивание допустимо только для неизменяемых типов данных!





## Плохие однострочники

Запись нескольких строк в одну через ; и отсутствие перехода на новую строку после :

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





Перед вами несколько строк кода.  
Напишите в чат, что они вернут,  
не запуская программу.

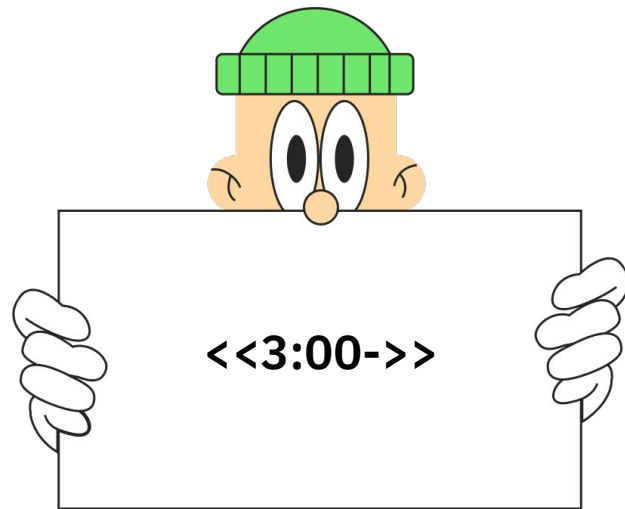
У вас 3 минуты.





## Однострочники

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





# Итераторы

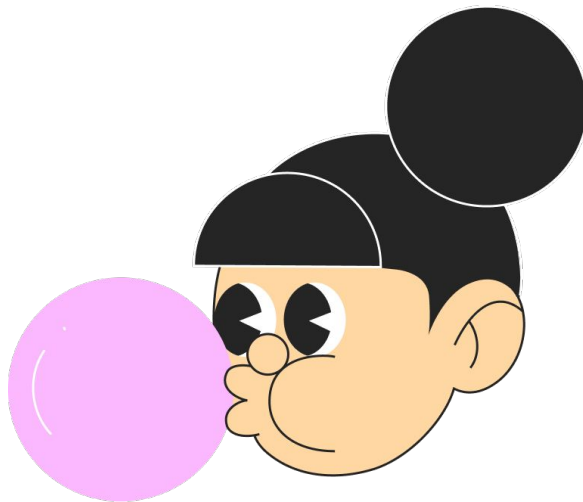




## Функция `iter()`

Разберём на примерах

- ✓ `iter(object[, sentinel])`  
Функция принимает на вход `object` поддерживающий итерацию. Второй параметр функции `iter` — `sentinel` передают для вызываемых объектов-итераторов.

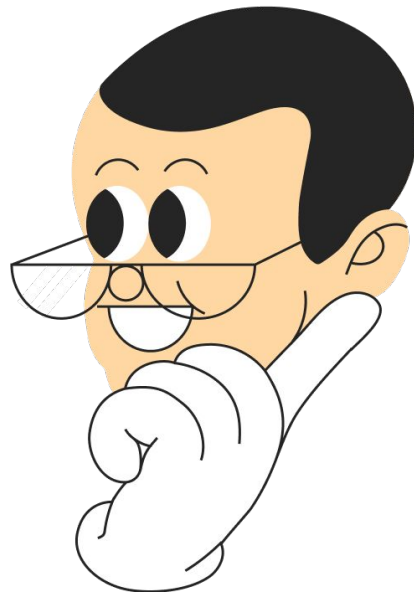




## Функция next()

Разберём на примерах

- ✓ `next(iterator[, default])`  
На вход функция принимает итератор, который вернула функция `iter`.  
  
Второй параметр функции `next` — `default` нужен для возврата значения по умолчанию вместо выброса исключения `StopIteration`.





Перед вами несколько строк кода.  
Напишите в чат, что они вернут,  
не запуская программу.

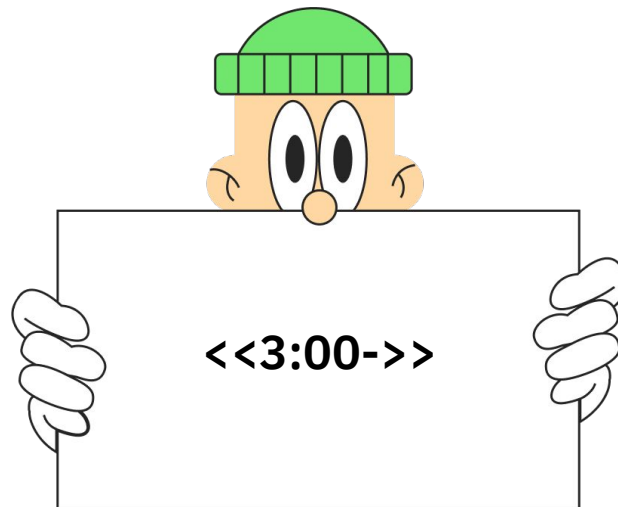
У вас 3 минуты.





## Генераторы

```
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)
```







# Генераторы





## Генераторные выражения

Генераторные выражения Python позволяют создать собственный генератор, перебирающий значения.

### Общий вид выражения:

```
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)
```

### Аналог на Python:

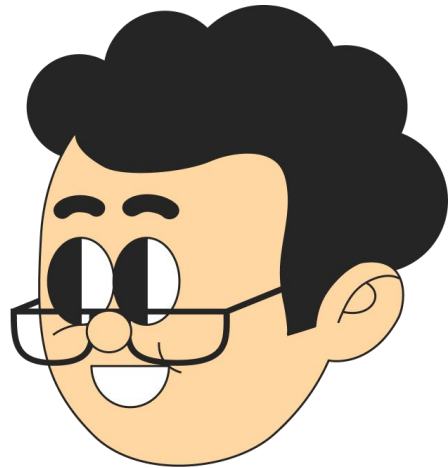
```
for expr in sequence1:
    if not condition1:
        continue
    for expr in sequence2:
        if not condition2:
            continue
    ...
    for expr in sequenceN:
        if not conditionN:
            continue
```



## List comprehensions

✓ `list_comp = [expression for expr in  
sequence1 if condition1 ...]`

Генератор списков формирует list заполненный  
данным и присваивает его переменной.





## Генераторные выражения или генерация списка?

? Задайте себе вопрос



**На выходе нужен готовый список?**

- ✓ list comprehensions
- ✓ [квадратные скобки]



**Элементы нужны последовательно?**

- ✓ генераторное выражение
- ✓ (круглые скобки)



## Set и dict comprehensions



### Set comprehensions

```
set_comp = {expression for expr in sequence1 if condition1 ...}
```



### Dict comprehensions

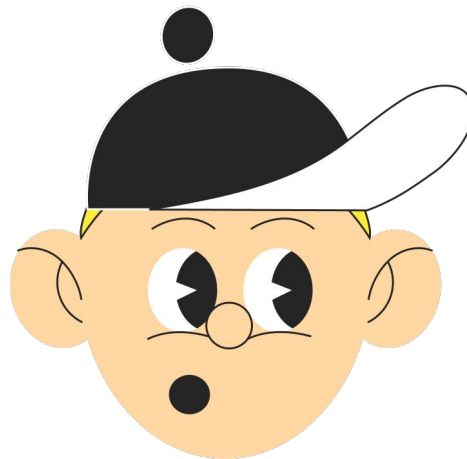
```
dict_comp = {key: value for expr in sequence1 if condition1 ...}
```



### Сходства и различия

{используются фигурные скобки для выражения}

словарь подставляет ключ и значение через двоеточие





Перед вами несколько строк кода.  
Напишите в чат, что они вернут,  
не запуская программу.

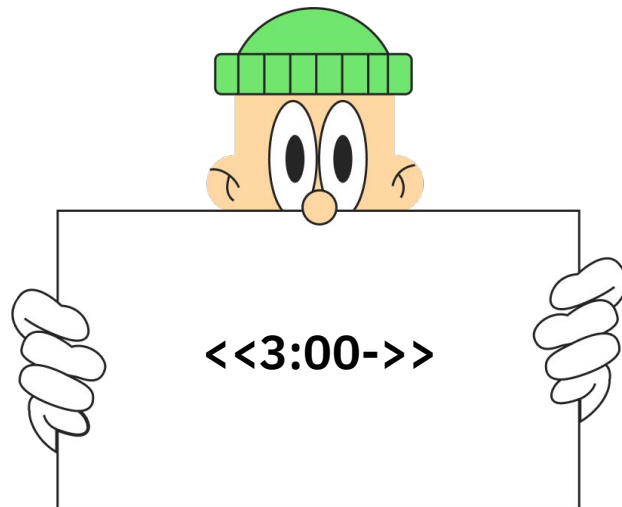
У вас 3 минуты.





## Итераторы

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





# Создание функции генератора







## Команда `yield`

Команда `yield` работает аналогично `return`.  
Но вместо завершения функции запоминает её состояние.  
Повторный вызов продолжает код после `yield`.

```
def gen(*args, **kwargs):  
    ...  
    yield result
```





Перед вами несколько строк кода.  
Напишите в чат, что они вернут,  
не запуская программу.

У вас 3 минуты.

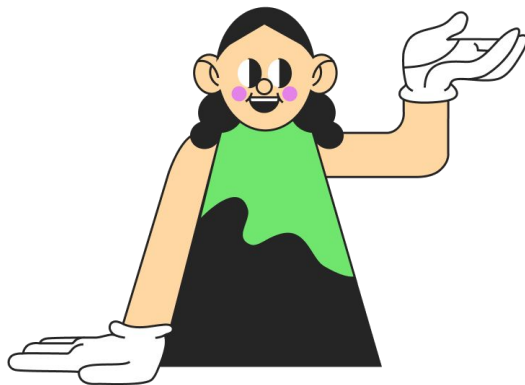




## Функции `iter` и `next` для генераторов



Функции `iter` и `next` одинаково работают с генераторами «из коробки» и с созданными самостоятельно.

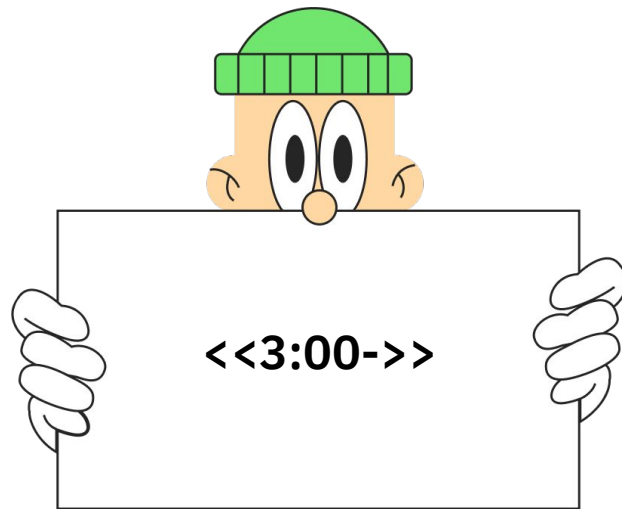




## Функция генератор

```
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 = }')
```





# Итоги занятия



## На этой лекции мы

- 📌 Разобрали решения задач в одну строку.
- 📌 Изучили итераторы и особенности их работы.
- 📌 Узнали о генераторных выражениях и генераторах списков, словарей, множеств.
- 📌 Разобрали создание собственных функций генераторов.

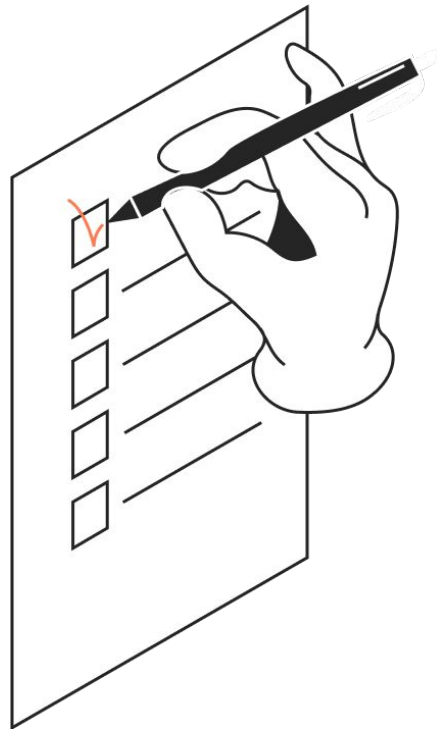




## Задание

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.





Спасибо за внимание