**Оглавление**

[**TODO** 2](#_Toc99382452)

[**Конспекты** 3](#_Toc99382453)

[**Иммутабельность переменных** 3](#_Toc99382454)

[**Списковое включение/представление списков (*list comprehension*)** 3](#_Toc99382455)

[**Контекстный менеджер** 3](#_Toc99382456)

[**Отличия кортежа от списка** 3](#_Toc99382457)

[**Функция *map*** 3](#_Toc99382458)

[**Функция *filter*** 4](#_Toc99382459)

[**Различия методов *type()* и *isinstance()*** 4](#_Toc99382460)

[**Итераторы** 5](#_Toc99382461)

[**Генераторы** 6](#_Toc99382462)

[**Отличия генераторов от итераторов** 7](#_Toc99382463)

[**\*args и \*\*kwargs** 8](#_Toc99382464)

**TODO:**

1. docker
2. poetry
3. django + debug toolbar
4. parser
5. магические методы
6. **прогуглить вопросы и задачи на собесах**
7. GIL (async/await/asyncio)
8. SOLID
9. Тестирование (unittest), mock-объекты

**Конспекты**

## **Иммутабельность переменных**

Тип переменной *a* **иммутабелен**, если после перезаписи значения переменной изменился ее идентификатор (*id(a)*). Изменение идентификатора указывает на то, что после перезаписи значения создался новый объект, следовательно, старый объект нельзя было изменить.

## **Списковое включение/представление списков (*list comprehension*)**

***List comprehension*** – короткий способ генерирования списков.

**Пример**

squares = [i \* i for i in range(10) if i % 2 == 0]

Представление списков состоит из трех элементов:

1. *expression* – выражение, возвращающее значение. В приведенном выше примере выражение *i \* i* выражением.
2. *member* – является объектом или значением в списке или итерируемым объектом (*iterable*). В приведенном выше примере значением элемента является *i*.
3. *iterable* – объект, возвращающий элементы. В примере выше этим объектом является *range*.

## **Контекстный менеджер**

<https://devpractice.ru/python-lesson-21-context-manager/>

## **Отличия кортежа от списка**

Элементы списка можно редактировать, а кортежа – нет.

## **Функция *map***

**Функция** *map* применяет функцию к каждому элементу последовательности и **возвращает итератор** с результатами.

**Пример**

list\_of\_str = ['1', '2', '5', '10']

list = list(map(int, list\_of\_str))

print(list)

> [1, 2, 5, 10]

Вместо *map* можно использовать *list comprehension*, но следует учитывать, что *map* **возвращает итератор**, а *list comprehension* – **список**. Если нужно сгенерировать большое количество элементов, то использование *map* выглядит уместнее в целях экономии памяти.

## **Функция *filter***

Функция *filter* применяет функцию ко всем элементам последовательности и **возвращает итератор** с теми объектами, для которых функция вернула *True*. Например, вернуть только те строки, в которых находятся числа:

list\_of\_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

list(filter(str.isdigit, list\_of\_strings))

> ['100', '1', '50']

Из списка чисел оставить только нечетные:

list(filter(lambda x: x % 2 == 1, [10, 111, 102, 213, 314, 515]))

> [111, 213, 515]

## **Различия методов *type()* и *isinstance()***

В отличие от *type*, функция *isinstance* **возвращает** не тип данных аргумента, а **булево значение**, говорящее о том, принадлежит объект к определенному классу или нет:

num = 4.44

print(isinstance(num, float))

> True

Также отличие от *type* состоит в том, что *isinstance* **"знает" о наследовании**. Функция воспринимает объект производного класса, как объект базового. Поэтому нельзя использовать *type* для проверок наследующихся типов.

**Пример**

class BaseExample:

pass

class DerivedExample(BaseExample):

pass

test = DerivedExample()

print(isinstance(test, BaseExample))

> True

## **Итераторы**

**Итератор** – это объект, который может возвращать элементы последовательности по одному.

Технически же это любой объект, **реализующий метод** *\_\_next\_\_*, который должен вернуть **следующий элемент** или исключение *StopIteration,* если перечислены все элементы.

Также итератор **реализует метод** *\_\_iter\_\_*, **возвращающий итератор**. Если этого метода нет, функция *iter()* проверяет, нет ли метода *\_\_getitem\_\_* – метода, который позволяет получать элементы по индексу. Если метод *\_\_getitem\_\_* есть, **возвращается итератор**, который проходится по элементам, используя индекс (начиная с 0).  Если не реализован ни один из этих методов, тогда будет **вызвано исключение** *TypeError*.

**Пример**

numbers = [1, 2]

i = iter(numbers)

next(i)

> 1

next(i)

> 2

next(i)

> StopIteration Traceback (most recent call last) in () ----> 1 next(i) StopIteration:

Для того, чтобы итератор снова начал возвращать элементы, его надо **заново создать**.

**Итераторы полезны** тем, что они **отдают элементы по одному**. Например, при работе с файлом или любой последовательностью это полезно тем, что **в памяти будет находиться** не вся последовательность, а **только одно текущее значение** итератора.

## **Генераторы**

В языках программирования есть такие понятия, как ленивые/отложенные вычисления (*lazy evaluation*). Генераторы можно считать реализацией механизма отложенного вычисления.

Генератор генерирует значения, возвращающиеся по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами **генератор сохраняет свое состояние**.

Python позволяет **создавать** генераторы **двумя способами**:

1. **генераторное выражение**;
2. **функция-генератор**.

**Генераторное выражение** использует такой же синтаксис, как *list comprehension* (но используются скобки круглые), но **возвращает итератор**, а не список:

genexpr = (x\*\*2 for x in range(1,5))

next(genexpr)

> 1

next(genexpr)

> 4

next(genexpr)

> 9

**Генераторные функции** – это функции, где есть хотя бы одно выражение *yield*. При запуске генератора, функция выполняется до первого выражения *yield*. Генератор при этом встанет «на паузу» до следующей итерации. При следующей итерации выполнение генератора продолжится до очередного *yield*:

def f\_gen(m):

s = 1

for n in range(1,m):

yield n\*\*2 + s

s += 1

a = f\_gen(5)

for i in a:

print(i)

> 2

> 6

> 12

> 20

## **Отличия генераторов от итераторов**

**Итератор** – более общая концепция, чем генератор. Можно сказать, что генератор является итератором, но не наоборот.

Главной целью **итератора** является предоставление интерфейса для поочередного доступа к элементам коллекций и потоков данных.

Главная задача **генератора** – хранить закономерность, по которой генерируется последовательность.

## **\*args и \*\*kwargs**

*\*args* – кортеж параметров (является типом *tuple*), позволяющий передавать в функцию неопределенное количество аргументов:

def getsum(\*numbers):

sum = 0

for num in numbers:

sum += num

return sum

print(add(2, 3)) # 5

print(add(2, 3, 5)) # 10

*\*\*kwargs* – то же самое, что и \*args, но обязывает проставлять имена параметров и является типом *dictionary*. Позволяет передавать параметры в произвольном порядке:

def total\_fruits(\*\*fruits):

total = 0

for amount in fruits.values():

total += amount

return total

print(total\_fruits(banana=5, mango=7, apple=8))

print(total\_fruits(banana=5, mango=7, apple=8, oranges=10))

555