

Основы Python

# Урок 5. Генераторы и comprehensions. Множества



# На этом уроке

1. Рассмотрим одну из важнейших тем — генераторы.
2. Узнаем о новом способе создания списков.
3. Вернёмся к хеш-таблицам — поговорим о множествах

## Оглавление

[Что такое генератор?](#)

[Генератор своими руками](#)

[List Comprehensions: не генераторы!](#)

[Развиваем успех: Dict Comprehensions](#)

[Множества в Python](#)

[Снова множества — frozenset](#)

[Set Comprehensions](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

## Что такое генератор?

Представим, что нам в алгоритме необходима последовательность квадратов нечётных чисел до миллиона. Наивное решение будет выглядеть так:

```
import sys

nums = []
for num in range(1, 10 ** 6 + 1, 2):
    nums.append(num ** 2)
print(type(nums), sys.getsizeof(nums))  # <class 'list'> 4290008
```

Заполнили список в цикле. При помощи функции `getsizeof()` из модуля `sys` узнали, сколько байтов памяти занимает этот список. Если у вас 32-битная система, число будет в 2 раза меньше. Как можно раскритиковать такое решение? Мы храним ВСЕ числа в памяти. Вполне может быть, что этого не нужно делать, если в алгоритме необходимо итерироваться по ним — брать одно за другим. Именно для этих целей и придумали генераторы в Python:

```
...
nums_gen = (num ** 2 for num in range(1, 10 ** 6 + 1, 2))
print(type(nums_gen), sys.getsizeof(nums_gen)) # <class 'generator'> 112
```

Мы записали тот же цикл, но в одну строку и обернули в КРУГЛЫЕ скобки. Теперь вместо 4 Мбайт наш объект занимает в памяти 112 байт. Экономия серьезная, но не в ущерб ли функциональности? Попробуем посчитать сумму:

```
...
from time import perf_counter

start = perf_counter()
nums_sum = sum(nums)
print(nums_sum, perf_counter() - start)
# 166666666666500000 0.022040908999999997

start = perf_counter()
nums_gen_sum = sum(nums_gen)
print(nums_gen_sum, perf_counter() - start)
# 166666666666500000 0.19718228200000004
```

Первый вывод — генератор работает в 10 раз медленнее. Причина есть: при суммировании элементов списка нужно только брать готовые значения по соответствующим адресам памяти и суммировать. Это работает быстро. В случае с генератором необходимо каждое значение сгенерировать (по факту вычислить) — это требует некоторого времени. Давайте теперь посмотрим глубже:

```
from time import perf_counter

start = perf_counter()
nums = []
for num in range(1, 10 ** 6 + 1, 2):
    nums.append(num ** 2)
nums_sum = sum(nums)
print(nums_sum, perf_counter() - start)
# 166666666666500000 0.24574289

start = perf_counter()
nums_gen = (num ** 2 for num in range(1, 10 ** 6 + 1, 2))
nums_gen_sum = sum(nums_gen)
print(nums_gen_sum, perf_counter() - start)
# 166666666666500000 0.19840974100000003
```

Получили совершенно другой результат. Догадались, почему? Теперь мы берём «честное» время — «под ключ». Ведь элементы списка тоже необходимо создать. В предыдущем примере они уже были готовы к моменту суммирования, в этом учитываем время, которое уходит на заполнение списка в цикле. И тут генератор вырывается вперёд. Это очень важный пример так называемых [«ленивых вычислений»](#). То есть генератор не вычисляет всю последовательность сразу. Он просто выдаёт нам очередной элемент и сохраняет своё состояние, как бы «спит»:

```
nums = []
for num in range(1, 10 ** 6 + 1, 2):
    nums.append(num ** 2)

nums_gen = (num ** 2 for num in range(1, 10 ** 6 + 1, 2))

print(nums[:3])
# [1, 9, 25]
print(next(nums_gen), next(nums_gen), next(nums_gen), sep=', ')
# 1, 9, 25
```

В отличие от списков, мы не можем обратиться к произвольному элементу генератора. Тут нет «параллельного» доступа — только последовательный. Вызываем функцию [next\(\)](#), получаем очередное значение. Это как «долистать» до страницы книги, а не открыть её сразу на нужной странице. Поэтому мы не можем делать обычные срезы для генераторов. Но можно использовать функцию `islice()` из модуля [itertools](#):

```
...
from itertools import islice

print(*islice(nums_gen, 3)) # 49 81 121
```

Тут мы убедились, что генератор «помнит своё состояние»: уже 3 раза вызывали функцию `next()` до этого шага. Поэтому «срез» честно вернул 3 первых элемента из оставшейся последовательности. Причём `islice()` вернёт не список, а [итератор](#) (по сути тоже генератор). Мы из него вытянули все элементы при помощи `*`. У вас уйдёт немало времени на глубокое осознание нюансов, но пока можем думать, что итератор и генератор — по сути одно и то же.

Продолжим пример:

```
...
nums_gen_sum = sum(nums_gen)
print(nums_gen_sum) # 1666666666666499714

nums_gen_sum = sum(nums_gen)
print(nums_gen_sum) # 0
```

Ещё одна особенность генераторов — они **ОДНОРАЗОВЫЕ**. То есть после вызова функции `sum()` мы получили сумму **ОСТАВШИХСЯ** элементов последовательности и всё. Больше генерировать нечего — поэтому при повторной попытке посчитать сумму получили ноль.

**Вывод:** если последовательность нужна в алгоритме более одного раза, использование генератора может быть неэффективным или приводить к ошибкам.

## Генератор своими руками

Давайте ещё раз разберем синтаксис создания генератора:

```
nums_gen = (num ** 2 for num in range(1, 10 ** 6 + 1, 2))
```

Эта запись называется [Generator Expression](#) (генераторное выражение). В первой позиции мы описываем очередное возвращаемое генератором значение — в нашем случае квадрат числа. Дальше пишем цикл, в котором создаются исходные данные для этих значений. У нас это нечётные числа до миллиона включительно. А если нужен генератор английских букв? Попробуем:

```
eng_letters_gen = (chr(code) for code in range(ord('a'), ord('z') + 1))
print(*eng_letters_gen, sep='') # abcdefghijklmnopqrstuvwxyz
```

Бывают более сложные алгоритмы генерации последовательностей. Например, генерация чисел Фибоначчи. Поэтому часто мы пишем **функции, возвращающие генераторы**:

```
import sys

def nums_generator(max_num):
    for num in range(1, max_num + 1, 2):
        yield num ** 2

nums_gen = nums_generator(10 ** 6)
print(type(nums_gen), sys.getsizeof(nums_gen)) # <class 'generator'> 112
print(sum(nums_gen)) # 1666666666666500000
```

Первое — мы получили точно такой же генератор квадратов нечётных чисел, что и раньше. Второе — мы теперь можем многократно создавать такие генераторы для разных значений границы:

```

nums_1000_gen = nums_generator(10 ** 3)
print(sum(nums_1000_gen)) # 166666500
nums_100000_gen = nums_generator(10 ** 5)
print(sum(nums_100000_gen)) # 166666666650000

```

Рекомендуем поразмышлять над ключевым словом [yield](#). Необходимо понять его роль и осмыслить разницу с `return`. Что делает это слово? Возвращает генератор. При этом все переменные в теле функции **сохраняют** свои значения до генерации следующего элемента — неважно, будет это в цикле `for in` или просто в результате вызова функции `next()`. Именно это хотят от вас услышать на собеседовании: генератор хранит своё состояние. А `return` возвращает результат вычислений, выполненных в функции. При этом значения всех переменных, которые были объявлены в теле функции, **исчезают**. При следующем вызове функции все будет с чистого листа.

Для закрепления функция-генератор букв:

```

def letters_generator(start, end):
    for code in range(ord(start), ord(end) + 1):
        yield chr(code)

eng_uppercase_letters = letters_generator('A', 'Z')
print(*eng_uppercase_letters, sep='') # ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

В каких случаях реализуем генераторы через функции? Когда генераторное выражение получается слишком сложным или вообще не позволяет реализовать требуемую логику.

Какие встроенные функции работают как генераторы (итераторы)? Вы их уже знаете: `range()`, `reversed()`, `map()`, `filter()`, `zip()`.

## List Comprehensions: не генераторы!

Напишем генератор кубов чисел до 5 включительно:

```

nums_cube_gen = (num ** 3 for num in range(5 + 1))
print(type(nums_cube_gen), *nums_cube_gen)
# <class 'generator'> 0 1 8 27 64 125

```

А если нам в алгоритме нужен список? Можем выполнить преобразование:

```

nums_cube_gen = (num ** 3 for num in range(5 + 1))
nums_cube = list(nums_cube_gen)
print(type(nums_cube), *nums_cube)
# <class 'list'> 0 1 8 27 64 125

```

Но в Python есть «[синтаксический сахар](#)» для более лаконичной записи такого кода — [List Comprehensions](#):

```
nums_cube = [num ** 3 for num in range(5 + 1)]
print(type(nums_cube), *nums_cube)
# <class 'list'> 0 1 8 27 64 125
```

В чём разница с генераторным выражением? Всего лишь скобки.

**Важно: ЭТО НЕ ГЕНЕРАТОРЫ** списков! Не надо их так называть. Кто-то неудачно перевел на русский язык, и многие повторяют, не понимая сути. Это не генератор, потому что здесь нет ЛЕНИВЫХ вычислений. В данном примере мы получаем обычный список альтернативным циклу-итератору способом. Давайте перепишем этот код по-старому:

```
nums_cube = []
for num in range(5 + 1):
    nums_cube.append(num ** 3)
```

List Comprehensions, как правило, работают быстрее, чем код, записанный через цикл for in. Они лаконичнее как минимум на одну строку. Но в результате мы всегда получаем список, а не генератор.

Внутри List Comprehensions можно использовать List Comprehensions — получаем [Nested List Comprehensions](#). Например, нам необходимо [сложить две матрицы](#). Можем решить задачу наивным способом:

```
matrix_1 = [
    [1, 4, 3, 3],
    [15, 0, 8, 11],
]
matrix_2 = [
    [17, 3, 2, 8],
    [4, 3, 6, 4],
]

matrix_sum = []
for i in range(len(matrix_1)):
    row = []
    for j in range(len(matrix_1[0])):
        row.append(matrix_1[i][j] + matrix_2[i][j])
    matrix_sum.append(row)
print(*matrix_sum, sep='\n')
# [18, 7, 5, 11]
# [19, 3, 14, 15]
```

Можем [отрефакторить код](#) при помощи List Comprehensions:

```
matrix_sum = [
    [matrix_1[i][j] + matrix_2[i][j] for j in range(len(matrix_1[0]))]
    for i in range(len(matrix_1))
]
```

Сложно читается? Это со временем пройдет. Зато вместо шести строк кода получили одну. Да, это считается одной строкой, просто мы отформатировали в несколько для читабельности. А вы сможете переписать эти примеры через `zip()`? Если да — супер! Иначе повторяйте материал предыдущих уроков:

```
matrix_sum = [
    [cell_1 + cell_2 for cell_1, cell_2 in zip(row_1, row_2)]
    for row_1, row_2 in zip(matrix_1, matrix_2)
]
```

Это решение самое правильное с точки зрения языка Python. **Очень важно**, что тут **нет индексов**. Читабельность такого кода на порядок выше, чем предыдущих вариантов. В принципе любой математик его прочитает, даже если он не программист.

Решим задачу посложнее. Нам необходимо сделать данные плоскими (`flat`). Например, мы записывали данные о температуре воздуха в отдельную строку таблицы (списка) каждый день. Теперь нам нужен непрерывный ряд данных о температуре: не по дням, а в виде одномерного списка. Зачем? Например, чтобы построить график изменения температуры в течение недели (месяца, полугодия и т.д.) То есть от двумерных данных нужно перейти к одномерным (плоским). Решаем задачу через цикл:

```
weather_data = [
    [-17.5, -18.9, -21.0, -16.1],
    [-9.3, -11.7, -14.3, -15.8],
]

flat_weather_data = []
for row in weather_data:
    for el in row:
        flat_weather_data.append(el)
print(flat_weather_data)
# [-17.5, -18.9, -21.0, -16.1, -9.3, -11.7, -14.3, -15.8]
```



Мы могли бы решить задачу проще через метод `.extend()`, но иногда на собеседованиях «не разрешают» использовать таких помощников. Теперь решим задачу через List Comprehensions:

```
flat_weather_data = [el for row in weather_data for el in row]
```

Решение в одну строку. Надо только научиться «читать» такой код. Можно переформатировать:

```
flat_weather_data = [  
    el  
    for row in weather_data  
        for el in row  
]
```

А если нам нужно ещё отфильтровать данные? Например, берём только температуры до -20 градусов:

```
flat_weather_data_2 = [  
    el for row in weather_data for el in row  
    if el > -20  
]  
print(flat_weather_data_2)  # [-17.5, -18.9, -16.1, -9.3, -11.7, -14.3, -15.8]
```

Разумеется, можно записать это решение в одну строку.

**НАСТОЯТЕЛЬНО** рекомендуем как можно чаще заменять обычные циклы List Comprehensions. И не забывайте, что это не «генераторы»!

## Развиваем успех: Dict Comprehensions

Есть ли в Python для словарей синтаксис, аналогичный List Comprehensions? [Есть](#):

```
nums_cube = {num: num ** 3 for num in range(1, 5 + 1)}  
print(nums_cube)  # {1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```

Разница лишь в наличии символа `:`, разделяющего ключ и значение. Так выглядел бы обычный код для создания такого словаря:

```
nums_cube = {}  
for num in range(1, 5 + 1):  
    nums_cube[num] = num ** 3
```

Dict Comprehensions могут быть полезны для преобразования одного словаря в другой:

```
eng_ru = {'may': 'май', 'june': 'июнь', 'july': 'июль'}
ru_eng = {val: key for key, val in eng_ru.items()}
print(ru_eng)  # {'май': 'may', 'июнь': 'june', 'июль': 'july'}
```

В чём коварство такой манипуляции? Надо быть уверенным, что значения, как и ключи, уникальны. Иначе потеряете часть элементов исходного словаря:

```
eng_ru_nums = {'one': 'один', 'first': 'один', 'two': 'два'}
ru_eng_nums = {val: key for key, val in eng_ru_nums.items()}
print(ru_eng_nums)  # {'один': 'first', 'два': 'two'}
```

## Множества в Python

Помните [круги Эйлера](#) из школьного курса математики? Что дают нам множества в разработке? Предположим, нам нужно взять из списка только уникальные значения. Можно решить эту задачу поверхностно:

```
basket = ['apple', 'samsung', 'apple', 'huawei', 'asus', 'samsung']
unique_brands = [el for el in basket if basket.count(el) == 1]
print(unique_brands)  # ['huawei', 'asus']
```

Многие скажут: «А что здесь такого? Код ведь работает!» Если вы напишете такое на собеседовании, провалите его. Предположим, что в списке  $n=1000$  элементов (на самом деле их может быть намного больше). Сколько раз выполняется цикл внутри List Comprehension? Да,  $n$  раз. Метод `.count()` обходит весь список и накапливает сумму вхождений объекта. То есть каждый раз он перебирает  $n$  элементов — это ещё  $n$  шагов на **каждой** итерации цикла. Итого получим  $n * n = n^2 = 1\,000\,000$  шагов при выполнении этого кода. В реальной жизни это значит, что при росте нагрузки на ресурс **очень** быстро будет падать его производительность.

А что, если нам поискать такую структуру данных, в которой проверка наличия элемента не потребует полного перебора? К таким структурам относятся, например, [хеш-таблицы](#). Более того, мы уже работали с хеш-таблицами: один из вариантов их реализации в Python — словари (dict). А другой — [множества](#) (set):

```
...
unique_brands = set()
tmp = set()
for el in basket:
    if el not in tmp:
```

```
        unique_brands.add(el)
    else:
        unique_brands.discard(el)
    tmp.add(el)
print(unique_brands)  # {'asus', 'huawei'}
```

Мы подразумеваем, что вы уже умеете работать с функцией `dir()` и посмотрели [docstring](#) у всех методов множеств. Если нет — дальше нельзя.

В примере мы итерируемся по элементам исходного списка и добавляем каждый элемент в множество `tmp` при помощи метода `.add()`. Подумайте, почему метод называется именно так, а не `.append()`, как у списков? Одно из **важнейших** особенностей множеств — отсутствие какого-либо порядка элементов внутри. Это значит, что мы можем проверить факт вхождения элемента в множество (оператор `in`), но у этого элемента нет никакого номера или индекса, как у тех же списков. При выводе множества через функцию `print()` вы можете каждый раз видеть разный порядок элементов — это нормально. В некоторых случаях, например если множество содержит одни числа, может наблюдаться один и тот же порядок элементов (по возрастанию), но это девиации от стандартного поведения.

Какой смысл у метода `.append()`? Добавить в конец списка. Есть ли у множества конец? Нет. И у метода `.append()` — тоже нет.

В начале каждой итерации цикла мы проверяем, встречался ли текущий элемент ранее. Эта манипуляция для множества условно [стоит нам один](#) шаг, а не `n` шагов, как было бы со списком. В этом и есть самая важная роль множеств в алгоритмах! Если текущий элемент не встречался ранее, добавляем его в результат, иначе — удаляем.

**Важно:** для удаления элемента из множества есть два метода:

- `.remove()` — если элемента нет, поднимет исключение [KeyError](#);
- `.discard()` — если элемента нет, ничего не происходит.

Подумайте: можно ли в нашем примере использовать `.remove()`?

Итак, при размере входного списка 1000 элементов решили ту же задачу за 1000 шагов вместо 1 000 000. Но что ещё можно тут улучшить? Если потребуется вывод элементов в той же последовательности, как они встречаются в исходном списке, нужно будет написать ещё один цикл или List Comprehension:

```
...
unique_brands_ord = [el for el in basket if el in unique_brands]
print(unique_brands_ord)  # ['huawei', 'asus']
```

Как вы уже видели, множество можно создавать через имя класса. Есть альтернативный синтаксис:

```
pencil_colors = {'red', 'pink', 'green'}
```

Очень похоже на создание словарей. Разница — нет пар, только значения через запятую. Тот же список, но скобки другие.

Что будет, если несколько раз добавить элемент в множество? Ничего, если он же там уже есть. Это как включить уже включённый светильник. Иногда этим пользуются, чтобы убрать повторы объектов. Но есть опасность — вы **всегда** теряете исходный порядок элементов.

Множества позволяют отвечать на вопросы вида: «найдем всех пользователей, которые одновременно участвуют в этих двух чатах» или «найдем всех пользователей этого чата, которые не участвуют в другом чате», «найдем всех пользователей, которые участвуют в первом или втором чате».

```
chat_1 = {'user_1', 'user_5', 'user_7', 'user_8', 'user_11'}
chat_2 = {'user_1', 'user_2', 'user_2', 'user_7', 'user_9', 'user_10'}

chats_common = chat_1.intersection(chat_2)
print(chats_common)  # {'user_1', 'user_7'}

chat_1_only = chat_1 - chat_2
chat_2_only = chat_2 - chat_1
print(chat_1_only)  # {'user_11', 'user_5', 'user_8'}
print(chat_2_only)  # {'user_9', 'user_10', 'user_2'}

both_chats = chat_1.union(chat_2)
print(both_chats)
# {'user_11', 'user_7', 'user_2', 'user_5', 'user_10', 'user_9', 'user_1',
#  'user_8'}
```

Пересечение множеств находим через метод [.intersection\(\)](#). Обратите внимание, что вместо вызова метода можно использовать оператор `&`. Аналогично, разницу находим при помощи оператора `-` или метода [.difference\(\)](#). Для объединения мы вызвали метод [.union\(\)](#), но можно было использовать оператор `|`.

## Снова множества — frozenset

Помните разницу между `list` и `tuple`? Изменяемость первых и неизменяемость вторых. Можно ли списки использовать в качестве ключей словаря? Нет. Можно ли списки добавлять в множества? Нет. А кортежи можно — они неизменяемы.

Можно ли множество использовать в качестве ключа словаря? Нет. Причина — множества изменяемы: мы можем добавлять или удалять из них элементы. А если в алгоритме нужно именно множество использовать в качестве ключа? Не преобразовывать же его в кортеж! Ведь тогда придётся ещё добавлять сортировку элементов, иначе из одного и то же множества получим разные кортежи. Вы же помните, что порядок элементов в множестве может быть любой? Значит, и в кортеже получится произвольный порядок, поэтому — только сортировать, но это будет костыль.

Вместо этого просто создаем [frozenset](#). Название — говорящее.

```
chat_1 = frozenset(('user_1', 'user_5', 'user_7', 'user_8', 'user_11'))
chat_2 = frozenset(('user_1', 'user_2', 'user_2', 'user_7', 'user_9'))

chats_common = chat_1.intersection(chat_2)
print(chats_common)  # frozenset({'user_7', 'user_1'})
```

Получили неизменяемое множество. То есть можем делать всё то же самое, что и с обычными множествами, но не можем менять состояние. Вопрос: есть ли у `frozenset` метод `.add()`? Если вы ответили «да», тему изменяемости объектов надо повторить: не разобрался в ней до конца. Конечно, этого метода нет, как и методов `.discard()` и `.remove()`.

## Set Comprehensions

Давайте рассмотрим ещё один способ сделать код лучше: [set comprehensions](#).

```
import random

random_nums = {random.randint(1, 100) for _ in range(10)}
print(len(random_nums), random_nums)  # 9 {97, 45, 77, 78, 15, 19, 86, 91, 93}
```

Вы уже заметили, что у генераторных выражений и `comprehensions` очень похожий синтаксис. Однако не забывайте, что между ними принципиальная разница: генераторы хранят своё состояние и отдают значения по одному, например через функцию `next()`. В то время как `comprehensions` создают объекты соответствующих типов целиком. Разумеется, для них бесполезно вызвать функцию `next()`.

На сегодня всё.

# Практическое задание

1. Написать генератор нечётных чисел от 1 до n (включительно), используя ключевое слово `yield`, например:

```
>>> odd_to_15 = odd_nums(15)
>>> next(odd_to_15)
1
>>> next(odd_to_15)
3
...
>>> next(odd_to_15)
15
>>> next(odd_to_15)
...StopIteration...
```

2. \*(вместо 1) Решить задачу генерации нечётных чисел от 1 до n (включительно), не используя ключевое слово `yield`.
3. Есть два списка:

```
tutors = [
    'Иван', 'Анастасия', 'Петр', 'Сергей',
    'Дмитрий', 'Борис', 'Елена'
]
klassess = [
    '9А', '7В', '9В', '9В', '8Б', '10А', '10Б', '9А'
]
```

Необходимо реализовать генератор, возвращающий кортежи вида (<tutor>, <klass>), например:

```
('Иван', '9А')
('Анастасия', '7В')
...
```

Количество генерируемых кортежей не должно быть больше длины списка `tutors`. Если в списке `klassess` меньше элементов, чем в списке `tutors`, необходимо вывести последние кортежи в виде: (<tutor>, `None`), например:

```
('Станислав', None)
```

Доказать, что вы создали именно генератор. Проверить его работу вплоть до истощения. Подумать, в каких ситуациях генератор даст эффект.

4. Представлен список чисел. Необходимо вывести те его элементы, значения которых больше предыдущего, например:

```
src = [300, 2, 12, 44, 1, 1, 4, 10, 7, 1, 78, 123, 55]
result = [12, 44, 4, 10, 78, 123]
```

Подсказка: использовать возможности python, изученные на уроке. Подумайте, как можно сделать оптимизацию кода по памяти, по скорости.

5. Представлен список чисел. Определить элементы списка, не имеющие повторений. Сформировать из этих элементов список с сохранением порядка их следования в исходном списке, например:

```
src = [2, 2, 2, 7, 23, 1, 44, 44, 3, 2, 10, 7, 4, 11]
result = [23, 1, 3, 10, 4, 11]
```

Подсказка: напишите сначала решение «в лоб». Потом подумайте об оптимизации.

Задачи со \* предназначены для продвинутых учеников, которым мало сделать обычное задание.

## Дополнительные материалы

1. [Лутц Марк. Изучаем Python.](#)
2. [Модуль datetime.](#)

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://realpython.com/python-sets/>.
2. <https://wiki.python.org/moin/TimeComplexity>.
3. <https://docs.python.org/3/>.