1. Использование генератора дважды

```
>>> numbers = [1,2,3,4,5]

>>> squared_numbers = (number**2 for number in numbers)

>>> list(squared_numbers)
[1, 4, 9, 16, 25]

>>> list(squared_numbers)
[]
```

Как мы видим в этом примере, использование переменной squared_numbers дважды, дало ожидаемый результат в первом случае, и, для людей незнакомых с Python в достаточной мере, неожиданный результат во втором.

2. Проверка вхождения элемента в генератор

Возьмём всё те же переменные:

```
>>> numbers = [1,2,3,4,5]
>>> squared_numbers = (number**2 for number in numbers)
```

А теперь, дважды проверим, входит ли элемент в последовательность:

```
>>> 4 in squared_numbers
True
>>> 4 in squared_numbers
False
```

Последовательности и итерируемые объекты

По-сути, вся разница, между последовательностями и итерируемымыи объектами, заключается в том, что в последовательностях элементы упорядочены.

Так, последовательностями являются: списки, кортежи и даже строки.

```
>>> numbers = [1,2,3,4,5]
>>> letters = ('a','b','c')
>>> characters = 'habristhebestsiteever'
>>> numbers[1]
```

```
2
>>> letters[2]
'c'
>>> characters[11]
's'
>>> characters[0:4]
'habr'
```

Итерируемые объекты же, напротив, не упорядочены, но, тем не менее, могут быть использованы там, где требуется итерация: цикл for, генераторные выражения, списковые включения — как примеры.
Итерируемые объекты же, напротив, не упорядочены, но, тем не менее, могут быть использованы там, где требуется итерация: цикл for, генераторные выражения, списковые включения — как примеры.

```
# Can't be indexed
>>> unordered_numbers = {1,2,3}
>>> unordered_numbers[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable

>>> users = {'males': 23, 'females': 32}
>>> users[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1

# Can be used as sequence
>>> [number**2 for number in unordered_numbers]
[1, 4, 9]
>>>
>>> for user in users:
...     print(user)
...
males
females
```

Отличия цикла for в Python от других языков

Стоит отдельно остановиться на том, что цикл for, в Python, устроен несколько иначе, чем в большинстве других языков. Он больше похож на for...each, или же for...of.

Если же, мы перепишем цикл for с помощью цикла while, используя индексы, то работать такой подход будет только с последовательностями:

```
>>> list_of_numbers = [1,2,3]
>>> index = 0
>>> while index < len(list_of_numbers):
...     print(list_of_numbers[index])
...     index += 1
...
1
2
3
```

А с итерируемыми объектами, последовательностями не являющимися, не будет:

```
>>> set_of_numbers = {1,2,3}
>>> index = 0
>>> while index < len(set_of_numbers):
...     print(set_of_numbers[index])
...     index += 1
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'set' object is not subscriptable
```

Если же вам нужен index, то следует использовать встроенную функцию enumerate:

```
>>> set_of_numbers = {1,2,3}
>>> for index, number in enumerate(set_of_numbers):
...     print(number, index)
...
1 0
2 1
3 2
```

Как мы могли убедиться, цикл for не использует индексы. Вместо этого он использует так называемые **итераторы**.

Итераторы — это такие штуки, которые, очевидно, можно итерировать :)
Получить итератор мы можем из любого итерируемого объекта.

Для этого нужно передать итерируемый объект во встроенную функцию iter:

```
>>> set_of_numbers = {1,2,3}
>>> list_of_numbers = [1,2,3]
>>> string_of_numbers = '123'
>>>
```

```
>>> iter(set_of_numbers)
<set_iterator object at 0x7fb192fa0480>
>>> iter(list_of_numbers)
<list_iterator object at 0x7fb193030780>
>>> iter(string_of_numbers)
<str_iterator object at 0x7fb19303d320>
```

После того, как мы получили итератор, мы можем передать его встроенной функции next.

```
>>> set_of_numbers = {1,2,3}
>>>
>>> numbers_iterator = iter(set_of_numbers)
>>> next(numbers_iterator)
1
>>> next(numbers_iterator)
2
```

При каждом новом вызове, функция отдаёт один элемент. Если же в итераторе элементов больше не осталось, то функция next породит исключение StopIteration.

```
>>> next(numbers_iterator)
3
>>> next(numbers_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

По-сути, это единственное, что мы может сделать с итератором: передать его функции next.
Как только итератор становится пустым и порождается исключение StopIteration, он становится совершенно бесполезным.

Хотите познакомиться со внутренней работой итератора и узнать, как он создает следующую последовательность? Давайте создадим итератор, который возвращает серию чисел. К примеру, это можно сделать так:

```
class Series(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high
    def __iter__(self):
        return self
    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
```

```
        self.current += 1
        return self.current - 1
n_list = Series(1,10)
print(list(n_list))
```

__iter__ возвращает сам объект итератора, а метод __next__ возвращает следующее значение из итератора. Если больше нет элементов для возврата, возникает исключение StopIteration.

Итерабельный объект представляет собой объект, элементы которого можно перебирать в цикле или иными доступными способами о которых мы поговорим ниже.

Итератор — это объект, который выполняет фактическую итерацию.

Есть еще одно правило об итераторах, которое делает все намного интереснее: итераторы также являются **итераторабельными объектами**, а их итератор — это они сами.

Зачем нужно создать итератор?

Итераторы позволяют создать итерабельный объект, который перебирает свои элементы по мере выполнения итерации. Это означает, что вы можете создавать ленивые итераторы, которые не определяют следующий элемент, пока вы не попросите их об этом.

Использование итератора вместо списка, множества или другой итерирабельной структуры данных иногда позволяет экономить память. Например, мы можем использовать `itertools.repeat` для создания итератора, который предоставит нам 100 миллионов четверок (`4`):

```
from itertools import repeat

lots_of_fours = repeat(4, times=100_000_000)
```

На моем компьютере этот итератор занимает 56 байт памяти:

```
import sys

print(sys.getsizeof(lots_of_fours)) # 56
```

Такой же список из 100 миллионов четверок созданный более примитивным способом занимает 762.94 Мб:

```
import sys

lots_of_fours = [4] * 100_000_000
print(sys.getsizeof(lots_of_fours)) # 800000064 байт
```

Хотя итераторы могут экономить память, они также могут экономить время. Например, если вы хотите вывести только первую строку из 10-гигабайтного файла с логами, вы можете сделать следующее:

```
first_line = next(open('giant_log_file.txt'))

print(first_line)

# Вывод: Это первая строка из гигантского файла
```

Файловые объекты в Python **реализованы как итераторы**. При **итерации по файлу** данные считываются в память по одной строке за раз. Если бы вместо этого мы использовали метод `readlines` для хранения всех строк в памяти, мы могли бы исчерпать всю системную память и убить процесс.

Кроме того, у итераторов есть возможности, которых нет у других итерабельных объектов. Например, их «лень» можно использовать для **создания итерабельных объектов неизвестной длины**. На самом деле, можно даже создавать бесконечно длинные итераторы.

Например, метод `itertools.count` создаст нам итератор, который будет выдавать каждое следующее число от `0` до «бесконечности» в зависимости когда вы завершите цикл:

```
from itertools import count

for n in count():
    print(n)
```

Результат:

```
0
1
2
(это будет продолжаться вечность)
```

Метод `itertools.count` по сути является **бесконечно длинным итерабельным объектом**. И он реализован как итератор.

Генераторы: простой способ создания итератора

Самый простой способ **создания собственных итераторов в Python** — это создание генератора.

В Python есть два способа создания генераторов.

Дан список чисел:

```
favorite_numbers = [6, 57, 4, 7, 68, 95]
```

Мы можем сделать генератор, который будет лениво выдавать все квадраты этих чисел следующим образом:

```
def square_all(numbers):
    for n in numbers:
        yield n**2

squares = square_all(favorite_numbers)
```

Или мы можем сделать такой же генератор следующим образом:

```
 squares = (n**2 for n in favorite_numbers)
```

Первый подход называется **функцией-генератором**, а второй — выражением-генератором.

Оба этих объекта-генератора работают одинаково. Они оба имеют тип `generator` и оба являются **итераторами**, которые предоставляют квадраты чисел из нашего списка чисел.

```
print(type(squares))
# Вывод: <class 'generator'>

print(next(squares))
# Вывод: 36

print(next(squares))
# Вывод: 3249
```

Мы поговорим об обоих этих подходах к созданию генератора, но сначала давайте обсудим терминологию.

Слово «генератор» в Python используется в разных смыслах:

- **Генератор**, также называемый **объектом-генератором**, — это итератор, тип которого — `generator`;
- **Функция-генератор** — это специальный синтаксис, который позволяет нам создать функцию, возвращающую **объект-генератор** при вызове;
- **Выражение-генератор** — это синтаксис, напоминающий представление списков (list comprehension), которое позволяет создавать **объект-генератор** в одну линию кода.

Убрав эту терминологию, давайте рассмотрим каждую из этих вещей по отдельности. Сначала мы рассмотрим **функции-генераторы**.

Функции-генераторы

Функции-генераторы отличаются от обычных функций тем, что в них есть один или несколько операторов `yield`.

Обычно при вызове функции выполняется ее код:

```
def gimme4_please():
    return 4


num = gimme4_please()
print(num) # Результат: 4
```

Но если в теле функции есть оператор `yield`, то это уже не обычная функция. Теперь это **функция-генератор**, то есть при вызове она возвращает объект-генератор. Этот объект-генератор может выполняться в цикле до тех пор, пока не будет выполнен оператор `yield`:

```
def gimme4_later_please():
    yield 4

get4 = gimme4_later_please()
print(get4)
# <generator object gimme4_later_please at 0x7f78b2e7e2b0>

num = next(get4)
print(num)
# Результат: 4
```

Одно только присутствие оператора `yield` превращает функцию в функцию-генератор. Если вы видите функцию и в ней есть оператор `yield`, вы работаете с чем-то иным нежели с обычной функцией. Это немного странно, но именно так работают **функции-генераторы**.

Хорошо, давайте рассмотрим реальный пример функции-генератора. Мы создадим функцию-генератор, которая будет делать то же самое, что и класс-итератор `Count`.

```
class Count:

    """Итератор, который считает до бесконечности."""

    def __init__(self, start=0):
        self.num = start

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num += 1
        return num
```

```
def count(start=0):
    num = start
    while True:
        yield num
        num += 1
```

Подобно классу-итератору `Count`, мы можем вручную перебирать генератор, полученный в результате вызова функции `count`:

```
c = count()

print(next(c))  # вывод: 0
print(next(c))  # вывод: 1
```

И мы можем перебирать этот объект генератора с помощью цикла `for`, как и раньше:

```
for n in count():
    print(n)
```

Результат:

```
0
1
2
(это будет продолжаться вечно)
```

Согласитесь, что данная функция значительно короче и понятнее, чем класс `Count`, который мы создали ранее.

Выражения-генераторы

Выражения-генераторы — это синтаксис, похожий на синтаксис представления списка (list comprehension), который позволяет нам создать объект-генератор.

Допустим, у нас есть представление-списка, который фильтрует пустые строки из файла и удаляет переход на новую строку в конце `\n`:

```
lines = [
    line.rstrip('\n')
    for line in file('esenin-berioza.txt').readlines()
    if line != '\n'
]
```

Мы можем создать генератор вместо списка, превратив квадратные скобки в круглые скобки:

```
lines = (
    line.rstrip('\n')
    for line in file('esenin-berioza.txt').readlines()
    if line != '\n'
)
```

Точно так же, как представление списков (list comprehension) вернуло бы нам список, выражение-генератор вернет нам объект-генератор:

```
print(type(lines)) # <class 'generator'>

next_line = next(lines)
print(next_line) # Вывод: 'Белая береза'

next_line = next(lines)
print(next_line) # Вывод: 'Под моим окном'

next_line = next(lines)
print(next_line) # Вывод: 'Принакрылась снегом,'

next_line = next(lines)
print(next_line) # Вывод: 'Точно серебром.'
```

Выражения-генераторы используют более короткий синтаксис кода по сравнению с **функциями-генераторами**. Однако они не такие мощные.

Вы можете написать свою функцию-генератор в такой форме:

```
def get_a_generator(some_iterable):
    for item in some_iterable:
        if some_condition(item):
            yield item
```

Затем вы можете заменить тело функции на выражение-генератор:

```
def get_a_generator(some_iterable):
    return (
        item
        for item in some_iterable
        if some_condition(item)
    )
```

Если вы не можете написать свою функцию-генератор в такой форме, то вы не сможете создать выражение-генератор для её замены.

Обратите внимание, что мы изменили используемый пример, потому что мы не можем использовать выражение-генератор для предыдущего примера, который реализует `itertools.count` который по сути является вечным циклом.

Выражения-генераторы или функции-генераторы?

Выражения-генераторы можно рассматривать как представление-списков (list comprehensions) в мире генераторов.

Выражения-генераторы являются функциями-генераторами так же, как представление-списков являются простым циклом `for` с добавлением и условием.

Выражения-генераторы очень похожи на представление-списков, их даже можно называть **представление-генераторов**. Технически это не совсем правильное название, но если вы его произнесете, все поймут, о чем вы говорите.

Лучший способ создания итератора

Чтобы создать итератор, можно создать класс-итератор, функцию-генератор или выражение-генератор. Но какой способ лучше?

Выражения-генераторы очень лаконичны, но они не такие гибкие, как функции-генераторы. Функции-генераторы гибкие, но если вам нужно добавить дополнительные методы или атрибуты к объекту-итератору, то, скорее всего, придется перейти на использование класса-итератора.

Я бы рекомендовал смотреть в сторону к **выражениям-генераторам** так же и представление-списков (list comprehensions). Если вы выполняете простую операцию

вывода или фильтрации, выражение-генератор — отличное решение. Если вы делаете что-то более сложное, вам, скорее всего, понадобится **функция-генератор**.

Я бы рекомендовал использовать функции-генераторы так же, как использование цикла `for` для добавления данных в список. Везде, где требуется метод `append`, вы зачастую увидите оператор `yield` вместо него.

А теперь вернёмся к тем особенностям, которые были изложены в начале

1. Использование генератора дважды

```
>>> numbers = [1,2,3,4,5]

>>> squared_numbers = (number**2 for number in numbers)

>>> list(squared_numbers)
[1, 4, 9, 16, 25]

>>> list(squared_numbers)
[]
```

В данном примере, список будет содержать элементы только в первом случае, потому что генераторное выражение — это итератор, а итераторы, как мы уже знаем — сущности одноразовые. И при повторном использовании не будут отдавать никаких элементов.

2. Проверка вхождения элемента в генератор

```
>>> numbers = [1,2,3,4,5]
>>> squared_numbers = (number**2 for number in numbers)
```

А теперь дважды проверим, входит ли элемент в последовательность:

```
>>> 4 in squared_numbers
True
>>> 4 in squared_numbers
False
```

В данном примере, элемент будет входить в последовательность только 1 раз, по причине того, что проверка на вхождение проверяется путем перебора всех

элементов последовательности последовательно, и как только элемент обнаружен, поиск прекращается. Для наглядности приведу пример:

```
>>> 4 in squared_numbers
True
>>> list(squared_numbers)
[9, 16, 25]
>>> list(squared_numbers)
[]
```

Как мы видим, при создании списка из генераторного выражения, в нём оказываются все элементы, после искомого. При повторном же создании, вполне ожидаемо, список оказывается пуст.