Анонимные (lambda) функции

- Помимо инструкции **def** в языке Python имеется возможность создавать объекты функций в форме выражений.
- Подобно инструкции **def** это выражение создает функцию, которая будет вызвана позднее, но в отличие от инструкции **def**, выражение возвращает функцию, а не связывает ее с именем.
- Именно поэтому **lambda**-выражения иногда называют анонимными (т.е. безымянными) функциями.
- На практике они часто используются как способ получить встроенную функцию или отложить выполнение фрагмента программного кода.
- Общая форма **lambda**-функций выглядит как ключевое слово **lambda**, за которым следует один или больше аргументов (очень похоже на список аргументов, заключенный в круглые скобки в заголовке **def**) и далее выражение после двоеточия:

```
lambda arg1, arg2, ... argN: expresion
```

Анонимные (lambda) функции

- Тело **lambda** является одиночным выражением, а не блоком операторов и похоже на то, что было бы указано в операторе **return** внутри тела **def**.
- Из-за ограничения только выражением, тело **lambda** менее универсально, чем **def** – можно помещать в тело **lambda** лишь определенную логику, не использующую операторы вроде **if**.
- Так было задумано, чтобы ограничить вложенность в программе: выражение **lambda** предназначено для записи простых функций, а оператор **def** можно использовать для решения более крупных задач.

```
>>> def f(x, y):
...     return x + y
>>> f(1, 2)
3
>>>
```

```
>>> f = lambda x, y: x + y
>>> f(1, 2)
3
>>>
```

Использование lambda-выражений

- Выражение **lambda** полезно как своего рода краткое условное обозначение функции, которое позволяет встраивать определение функции внутрь кода, где оно применяется.
- В сценариях, где нужно всего лишь встраивать небольшие порции кода, выражения **lambda** окажутся более простыми кодовыми конструкциями, чем операторы **def**.

```
>>> from random import randint
>>> x = [randint(-5, 5) for _ in range(10)]
>>> x
[-4, -2, -4, 2, -5, 0, -4, 4, -5, -3]
>>> min(x, key=lambda x: x ** 2 - 9 * x + 5)
4
```

- Фактически, поиск минимума происходил в этом списке, но результат – соответсвующий элемент из списка x:

```
>>> y = [num ** 2 - 9 * num + 5 for num in x]
>>> y
[57, 27, 57, -9, 75, 5, 57, -15, 75, 41]
>>>
```

Использование `lambda`-выражений

- Выражения **`lambda`** широко используются при реализации таблиц переходов, которые представляют собой списки или словари действий, подлежащих выполнению по запросу.

```
>>> function_list = [
...     lambda x: x ** 2,  # Список из трех вызываемых функций
...     lambda x: x ** 3,
...     lambda x: x ** 4
... ]
>>> for f in function_list:
...     print(f(3))
...
9
27
81
>>> function_list[0](5)
25
>>>
```

Использование `lambda`-выражений

- Эквивалентный код с `def` потребовал бы временных имен функций (возможны конфликты с остальными именами) и определений функций за пределами контекста их планируемого применения (который может находиться на сотни строк дальше):

```
>>> def f1(x):
...     return x ** 2
...
>>> def f2(x):
...     return x ** 3
...
>>> def f3(x):
...     return x ** 4
...
>>> function_list = [f1, f2, f3]
>>> for f in function_list:
...     print(f(3))
...
9
27
81
>>> print(function_list[0](5))
25
```

Использование lambda-выражений

- В действительности можно использовать словари и другие структуры данных в Python для построения более универсальных разновидностей таблиц действий:

```
>>> key = 'got'
>>> actions = {
...     'already': lambda: 2 + 2,
...     'got': lambda: 5 * 5,
...     'one': lambda: 2 ** 6
... }
>>> actions[key]()
25
>>>
```

- Когда Python создает словарь `actions`, каждое вложенное выражение **`lambda`** генерирует функцию, подлежащую вызову в будущем.
- Индексация по ключу извлекает одну из функций, а круглые скобки приводят к вызову извлеченной функции.
- При такой реализации словарь становится более универсальным инструментом для множественного ветвления.

Использование lambda-выражений

- Версия без **lambda**, потребует три оператора **def** где-то в файле за пределами словаря, в котором функции будут применяться, и ссылаться на функции по их именам:

```
>>> def f1():
...     return 2 + 2
...
>>> def f2():
...     return 5 * 5
...
>>> def f3():
...     return 2 ** 6
...
>>> key = 'one'
>>> actions = {'already': f1, 'got': f2, 'one': f3}
>>> actions[key]()
64
>>>
```

- Выражение **lambda** создано для небольших частей внутристрочного кода; при необходимости написания более сложной логики используйте оператор **def**.

Функция `map()`

- Одним из наиболее часто встречающихся действий, выполняемых в программах со списками и другими последовательностями, является применение какой-то операции к каждому элементу и накопление результатов.
- В Python есть много инструментов, которые облегчают написание кода таких операций в масштабах целой коллекции. Например, обновление всех счетчиков в списке несложно обеспечить в цикле **for**:

```
>>> counters = [1, 2, 3, 4, 5]
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)
...
>>> updated
[11, 12, 13, 14, 15]
>>>
```

Функция `map()`

- Встроенная функция `map()` применяет переданную функцию к каждому элементу в итерируемом объекте и возвращает список, содержащий все результаты вызовов функции.

```
>>> def inc(x):  # Функция, подлежащая выполнению
...     return x + 10
...
>>> list(map(inc, counters))  # Накапливание результатов
[11, 12, 13, 14, 15]
>>>
```

- Здесь функция `map()` вызывает функцию `inc()` для каждого элемента списка и собирает все возвращаемые значения в новый список.
- Поскольку встроенная функция `map()` ожидает передачи функции, применяемой к элементам, она также относится к тем местам, где обычно появляется выражение **`lambda`**:

```
>>> list(map(lambda x: x * 2, counters))
[2, 4, 6, 8, 10]
>>>
```

- Подлежащая применению функция умножает на 2 каждый элемент в списке `counters`; так как эта небольшая функция далее нигде не нужна, она была записана как внутристрочная посредством выражения **`lambda`**.

Функция `map()`

- Функция `map()` – встроенная функция, она всегда доступна, всегда работает одинаково и дает преимущества в плане производительности (в некоторых режимах применения она быстрее вручную написанного цикла **`for`**).
- Функцию `map()` можно использовать более развитыми способами. Например, получив в качестве аргументов несколько последовательностей, `map()` передает извлеченные из последовательностей элементы как индивидуальные аргументы функции `pow()`:

```
>>> pow(2, 3)  # 2 ** 3
8
>>> list(map(pow, [1, 2, 3], [4, 5, 6]))  # 1 ** 4, 2 ** 5, 3 ** 6
[1, 32, 729]
>>>
```

- В случае множества последовательностей `map` ожидает функцию с n аргументами для n последовательностей. Функция `pow` принимает в каждом вызове два аргумента – по одному из каждой последовательности, переданной `map()`.

Функция `map()`

- Вызов `map` похож на выражения генераторов списков:

```
>>> list(map(inc, [10, 20, 30, 40]))
[20, 30, 40, 50]
>>> [inc(x) for x in [10, 20, 30, 40]]
[20, 30, 40, 50]
>>>
```

- Поскольку `map()` применяет к каждому элементу вызов функции, а не произвольное выражение, она является менее универсальным инструментом и часто требует добавочных вспомогательных функций или **`lambda`**-выражений.

Функция `filter()`

- Встроенная функция `filter()` выбирает элементы из итерируемого объекта на основе проверочной функции.
- Из-за возвращения итерируемого объекта функция `filter()` (подобно `range()`) требует вызова `list()` при отображении всех ее результатов.
- Например, следующий вызов `filter()` выбирает из последовательности элементы больше нуля.

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter(lambda x: x > 0, range(-5, 5)))
[1, 2, 3, 4]
>>>
```

- Элементы из последовательности либо итерируемого объекта, для которых проверочная функция возвращает истинное значение, добавляются в результирующий список.

Функция `filter()`

- Как и `map()`, функция `filter()` приблизительно эквивалентна циклу `for`, но является встроенной, лаконичной и часто более быстрой:

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
>>>
```

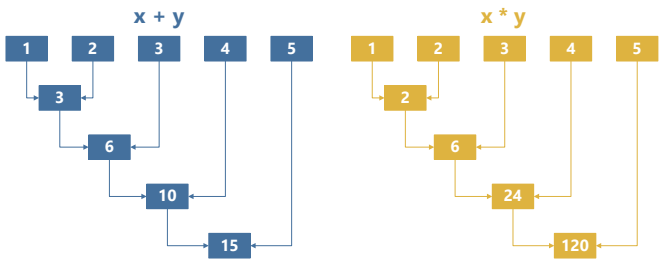
- Также подобно `map()` функцию `filter()` можно эмулировать посредством синтаксиса генератора списка с часто более простыми результатами (особенно если удается избежать создания новой функции):

```
>>> [x for x in range(-5, 5) if x > 0]
[1, 2, 3, 4]
>>>
```

Функция reduce()

- Функция reduce(), которая находится в модуле functools, принимает итерируемый объект, подлежащий обработке, но сама возвращает одиночный результат.

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
120
>>>
```



Функция `reduce()`

- На каждом шаге `reduce()` передает текущую сумму или произведение вместе с очередным элементом из списка указанной функции **`lambda`**.
- По умолчанию начальное значение инициализируется первым элементом последовательности.
- В целях демонстрации рассмотрим эквивалент первого из двух вызовов в виде цикла **`for`** с жестко закодированным сложением внутри цикла:

```
>>> arr = [1, 2, 3, 4, 5]
>>> res = arr[0]
>>> for x in arr[1:]:
...     res += x
...
>>> res
15
>>>
```

- Функция `reduce()` разрешает передавать необязательный третий аргумент, который помещается перед элементами последовательности и служит начальным значением, а также стандартным результатом, когда последовательность пуста.

Итерируемые объекты

- **Итерация** – это общий термин, который описывает процедуру взятия элементов чего-то по очереди.
- В более общем смысле, это последовательность инструкций, которая повторяется определенное количество раз или до выполнения указанного условия.
- **Итерируемый объект** (iterable) – это объект, который способен возвращать элементы по одному.

Примеры итерируемых объектов

- Списки
- Строки
- Кортежи
- Словари
- Множества

Итераторы

- **Итератор** (iterator) – это объект, который возвращает свои элементы по одному за раз.
- С точки зрения Python – это любой объект, у которого есть метод `__next__()`. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение `StopIteration`, когда элементы закончились.
- Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию.
- В Python у каждого итератора присутствует метод `__iter__()` – то есть, любой итератор является **итерируемым объектом**. Этот метод просто возвращает сам итератор.
- В Python за получение итератора отвечает функция `iter()`:

```
>>> some_numbers = [1, 2, 3]
>>> iter(some_numbers)
<list_iterator at 0xb4ede28c>
>>>
```

- Функция `iter()` работает с любым объектом, у которого есть метод `__iter__()` или метод `__getitem__()`.
- Метод `__iter__()` возвращает итератор. Если этого метода нет, функция `iter()` проверяет, нет ли метода `__getitem__()` – метода, который позволяет получать элементы по индексу.

Итераторы

- Пример создания итератора из списка:

```
>>> numbers = [1, 2, 3]
>>> i = iter(numbers)
```

- Теперь можно использовать функцию `next()`, которая вызывает метод `__next__()`, чтобы взять следующий элемент:

```
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

- После того, как элементы закончились, возвращается исключение **`StopIteration`**.
- Для того, чтобы итератор снова начал возвращать элементы, его нужно создать заново.

Итераторы

- Аналогичные действия выполняются, когда цикл **for** проходит по списку:

```
>>> for item in numbers:
...     print(item)
...
1
2
3
>>>
```

- Когда мы перебираем элементы списка, к списку сначала применяется функция `iter()`, чтобы создать итератор, а затем вызывается его метод `__next__()` до тех пор, пока не возникнет исключение **StopIteration**.
- Итераторы полезны тем, что они отдают элементы по одному.
- Например, при работе с большими коллекциями в памяти будет находиться не все ее элементы, а только один.

Функция `sorted()`

- Функция `sorted()` возвращает **новый отсортированный список**, который получен из **итерируемого** объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.
- Если сортировать **список** элементов, то возвращается **новый список**:

```
>>> list_of_words = ['one', 'two', 'list', '', 'dict']
>>> sorted(list_of_words)
['', 'dict', 'list', 'one', 'two']
```

- При сортировке **кортежа** также возвращается список:

```
>>> tuple_of_words = ('one', 'two', 'list', '', 'dict')
>>> sorted(tuple_of_words)
['', 'dict', 'list', 'one', 'two']
>>>
```

Функция sorted()

- Сортировка **множества**:

```
>>> set_of_words = {'one', 'two', 'list', '', 'dict'}
>>> sorted(set_of_words)
['', 'dict', 'list', 'one', 'two']
>>>
```

- Сортировка **строки**:

```
>>> string_to_sort = 'long string'
>>> sorted(string_to_sort)
>>> [' ', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 's', 't']
>>>
```

Функция sorted()

- Если передать sorted() словарь, функция вернет отсортированный список ключей:

```
>>> dict_for_sort = {
...     'id': 1,
...     'name': 'London',
...     'IT_VLAN': 320,
...     'User_VLAN': 1010,
...     'Mngmt_VLAN': 99,
...     'to_name': None,
...     'to_id': None,
...     'port': 'G1/0/11'
...  }
>>> sorted(dict_for_sort)
['IT_VLAN',
'Mngmt_VLAN',
'User_VLAN',
'id',
'name',
'port',
'to_id',
'to_name']
>>>
```

Дополнительные аргументы функции `sorted()`

Параметр `reverse`

- Флаг `reverse` позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.
- Указав флаг `reverse`, можно поменять порядок:

```
>>> list_of_words = ['one', 'two', 'list', '', 'dict']
>>> sorted(list_of_words)
['', 'dict', 'list', 'one', 'two']
>>> sorted(list_of_words, reverse=True)
['two', 'one', 'list', 'dict', '']
>>>
```

Дополнительные аргументы функции `sorted()`

Параметр key

- С помощью параметра key можно указывать, как именно выполнять сортировку. Параметр key ожидает функцию с одним параметром, с помощью которой должно быть выполнено сравнение.
- Например, таким образом можно отсортировать список строк по длине строки:

```
>>> list_of_words = ['one', 'two', 'list', '', 'dict']
>>> sorted(list_of_words, key=len)
['', 'one', 'two', 'list', 'dict']
>>>
```

Дополнительные аргументы функции `sorted()`

- Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
>>> dict_for_sort = {
...     'id': 1,
...     'name': 'London',
...     'IT_VLAN': 320,
...     'User_VLAN': 1010,
...     'Mngmt_VLAN': 99,
...     'to_name': None,
...     'to_id': None,
...     'port': 'G1/0/11'
... }
>>> sorted(dict_for_sort, key=str.lower)
['id',
'IT_VLAN',
'Mngmt_VLAN',
'name',
'port',
'to_id',
'to_name',
'User_VLAN']
>>>
```

Дополнительные аргументы функции `sorted()`

- Параметру key можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию **`lambda`**.
- С помощью параметра key можно сортировать объекты не по первому элементу, а по любому другому.
- Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
>>> list_of_tuples = [('IT_VLAN', 320),
...   ('Mngmt_VLAN', 99),
...   ('User_VLAN', 1010),
...   ('DB_VLAN', 11)]
>>> sorted(list_of_tuples, key=lambda x: x[1])
[('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320),
 ('User_VLAN', 1010)]
>>>
```

Функция `reversed()`

- Функция `reversed()` возвращает обратный итератор, то есть возвращает итератор, который перебирает элементы оригинала в обратном порядке.
- Функция `reversed()` не создает копию и не изменяет оригинал последовательности.

```
>>> seq_string = 'Python'  # для строки
>>> list(reversed(seq_string))
['n', 'o', 'h', 't', 'y', 'P']
>>> seq_tuple = ('P', 'y', 't', 'h', 'o', 'n')  # для кортежа
>>> list(reversed(seq_tuple))
['n', 'o', 'h', 't', 'y', 'P']
>>> seq_list = [1, 2, 4, 3, 5] # для списка
>>> list(reversed(seq_list))
[5, 3, 4, 2, 1]
>>> seq_range = range(5, 9)  # для итерируемого объекта
>>> list(reversed(seq_range))
[8, 7, 6, 5]
>>>
```

- Так как функция `reversed()` возвращает итерируемый объект, для получения всех элементов использовалась функция `list()`.

Функция `zip()`

- Функция `zip()` принимает последовательности, а возвращает итератор с кортежами, в котором n-ый кортеж состоит из n-ых элементов последовательностей, которые были переданы как аргументы.
- Например, пятый кортеж будет содержать пятый элемент каждой из переданных последовательностей.
- Если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности.
- Порядок элементов сохраняется.

Примечение
Так как `zip()` – это итератор, для отображения его содержимого используется `list()`

```
>>> a = [1, 2, 3]
>>> b = [100, 200, 300]
>>> list(zip(a, b))
[(1, 100), (2, 200), (3, 300)]
>>>
```

Функция `zip()`

- Использование `zip()` со списками разной длины:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [10, 20, 30, 40, 50]
>>> c = [100, 200, 300]
>>> list(zip(a, b, c))
[(1, 10, 100), (2, 20, 200), (3, 30, 300)]
>>>
```

Функция zip()

- Использование zip() для создания словаря:

```
>>> d_keys = ['hostname', 'location', 'vendor', 'model',
...           'IOS', 'IP']
>>> d_values = ['london_r1', '21 New Globe Walk', 'Cisco',
...             '4451', '15.4', '10.255.0.1']
>>> dict(zip(d_keys, d_values))
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
>>>
```

Функция `enumerate()`

- Иногда, при переборе объектов в цикле **`for`**, нужно не только получить сам объект, но и его порядковый номер.
- Это можно сделать, создав дополнительную переменную, которая будет расти на единицу с каждым прохождением цикла.
- Однако, гораздо удобнее это делать с помощью итератора `enumerate()`.

```
>>> list1 = ['str1', 'str2', 'str3']
>>> for position, string in enumerate(list1):
...     print(position, string)
...
0 str1
1 str2
2 str3
>>>
```

Функция `enumerate()`

- Функция `enumerate()` умеет считать не только с нуля, но и с любого значения, которое ему указали после объекта:

```
>>> list1 = ['str1', 'str2', 'str3']
>>> for position, string in enumerate(list1, 100):
...     print(position, string)
...
100 str1
101 str2
102 str3
>>>
```

- Если необходимо увидеть содержимое, которое сгенерирует итератор, полностью, можно воспользоваться функцией `list()`:

```
>>> list1 = ['str1', 'str2', 'str3']
>>> list(enumerate(list1, 100))
[(100, 'str1'), (101, 'str2'), (102, 'str3')]
>>>
```

Функции-генераторы

- **Функция-генератор** – это функция, которая может возвращать значение, а позднее продолжить свою работу с того места, где она была приостановлена.
- Такая функция генерирует последовательность значений с течением времени, а также автоматически поддерживает протокол итераций и может использоваться в контексте итераций.
- Функции-генераторы записываются как обычные операторы `def`, но в них применяются операторы `yield`, вместо операторов `return`.
- Главное отличие функций-генераторов от обычных функций состоит в том, что они **поставляют значение**, а не возвращают его – оператор `yield` приостанавливает работу функции и передает значение вызывающей программе, при этом сохраняется информация о состоянии, необходимая для возобновления работы с того места, где она была приостановлена.
- Функция-генератор возобновляет работу, ее выполнение продолжается с первой инструкции, следующей после оператора `yield`. Это позволяет функциям воспроизводить последовательности значений в течение долгого времени, вместо того, чтобы создавать всю последовательность сразу и возвращать ее в виде некоторой конструкции, такой как список.

Функции-генераторы

- Определим функцию-генератор, которую можно применять для получения квадратов серии чисел с течением времени:

```
>>> def generate_squares(n):
...     for x in range(n):
...         yield x ** 2
...
```

- Функция `generate_squares` выдает значение и потому возвращает управление вызывающему коду на каждой итерации цикла; при возобновлении ее выполнения восстанавливается предыдущее состояние, включая последние значения переменных x и n, а управление снова подхватывается непосредственно после оператора **`yield`**.

```
>>> for i in generate_squares(5):
...     print(i, end=' | ')
...
0 | 1 | 4 | 9 | 16 |
```

- При вызове функции-генератора напрямую будет получен объект генератора:

```
>>> fg = generate_squares(5)
>>> fg
<generator object generate_squares at 0x00000217B6B79E40>
```

Функции-генераторы

- Возвращенный объект генератора имеет метод `__next__()`, который запускает функцию или возобновляет ее выполнение с места, откуда она последний раз выдала значение, и инициирует исключение `StopIteration`, когда достигнут конец серии значений.
- Встроенная функция `next()` вызывает метод `__next__()` итерируемого объекта:

```
>>> next(fg)
0
>>> next(fg)
1
>>> next(fg)
4
>>> next(fg)
9
>>> next(fg)
16
>>> next(fg)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

- После использования итерации по генератору, он **останется пустым**. Для повторной итерации придется создать новый объект генератора.

Генераторные выражения

- Синтаксически генераторные выражения похожи на генераторы списков и поддерживают весь их синтаксис, в том числе фильтры **if** и вложение циклов, но они помещаются в круглые скобки.

```
>>> [x ** 2 for x in range(5)]  # Генератор списка
[0, 1, 4, 9, 16]
>>> (x ** 2 for x in range(5))  # Генраторное выражение
<generator object <genexpr> at 0x00000217B6E4CCF0>
>>>
```

- Генераторные выражения возвращают объект генератора – автоматически созданный итерируемый объект.
- Этот итерируемый объект поддерживает протокол итерации, чтобы выдавать по одной порции результирующего списка за раз в любом итерационном контексте.
- Итерируемый объект также сохраняет состояние генератора, пока он действует – переменную x в предшествующих выражениях наряду с местоположением в коде генератора.

Генераторные выражения

- Совокупный эффект во многом подобен функциям-генераторам, но в контексте выражения: мы получаем обратно объект, который запоминает оставленное им место после возвращения каждой части результата.

```
>>> g = (x ** 2 for x in range(5))
>>> iter(g) is g
True
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Генераторные выражения

- Внутренне генераторные выражения имеют механизм итерации, в большинстве случаев скрытый от пользователей, т.к. циклы **for** запускают его автоматически:

```
>>> for number in (x ** 2 for x in range(5)):
...     print(f'{number:>4}, {number / 2.0:>4.1f}')
...
   0,  0.0
   1,  0.5
   4,  2.0
   9,  4.5
  16,  8.0
>>>
```

- В точности как функции-генераторы генераторные выражения обеспечивают оптимизацию **расхода памяти** – они не требуют создания сразу всего результирующего списка, что происходит в случае генератора списка.
- Также подобно функциям-генераторам они разделяют работу по выпуску результатов на небольшие временные интервалы – результаты выдаются постепенно вместо того, чтобы заставлять вызывающий код ожидать создания полного набора в единственном вызове.
- С другой стороны, на практике генераторные выражения могут выполняться несколько медленнее генераторов списков, а потому их лучше всего применять для очень крупных результирующих наборов или в приложениях, которые не могут ожидать генерации полных результатов.