Списки и итераторы с разных ракурсов

Python

1. Итераторы;
2. Списочные встраивания;
3. Генераторные выражения;
4. Генераторы;
5. Модуль Itertools.

Итератор

Итератор — сущность, указывающая на единицу данных в потоке данных. От этой сущности можно вызвать метод **next()**, который возвращает следующий элемент в потоке.

В качестве потока данных могут выступать итерируемые структуры данных (списки, кортежи, словари, строки), файлы и пр. Все функции, принимающие структуры данных, неявно получают итератор для них.

В них всегда можно передать итератор.

Итератор

```
>>> L= [1, 2]
>>> it = iter(L)
>>> print(it)
<list_iterator object at 0x000000000346F390>
>>> next(it)
1
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
StopIteration
```

Итератор

```
>>> for i in L:
...     print(i)
>>> for i in iter(L):
...     print(i)

>>> iterator = iter(L)
>>> t = tuple(iterator) #list
>>> t
(1, 2)
```

Итератор

```
>>> iterator = iter(L)
>>> a, b = iterator
>>> a, b
(1, 2)

>>> L = [('a', 1), (5, True), (7, 'abc')]
>>> d = dict(iter(L))
>>> d
{'a': 1, 5: True, 7: 'abc'}
```

Итератор

```
>>> d = {'Mon': 1, 'Tue': 2, ... , 'Sun': 7}
>>> for key in d:
...     print (key, d[key])
Wed 3
Sun 7
Fri 5
Tue 2
Sat 6
Mon 1
Thu 4
```

Итератор

У словарей есть методы **keys, values, items,** возвращающие “**view**” на ключи, значения, кортежи (ключ, значение). Например, **for key, value in d.items():** будет последовательно записывать в key ключи, а в value — соответствующие им значения (распаковка кортежа).

View – это не список! Но и не итератор! Он имеет непосредственную связь с объектом (словарем)

Итератор vs View

```
>>> d = {1: "a", 2: "b"}
>>> i1 = iter(d)
>>> 1 in i1
True
>>> 2 in i1
True
>>> 1 in i1
False
>>> i2 = d.keys()
>>> 1 in i2
True
>>> 2 in i2
True
>>> 1 in i2
True
```

Итератор vs View

```
>>> d = {1: "a", 2: "b"}
>>> i1 = iter(d)
>>> i2 = d.keys()
>>> d[3] = "c"
>>> for x in i1:
        print(x, end=" ")
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    for x in i1:
RuntimeError: dictionary changed size during iteration
>>> for x in i2:
        print(x, end=" ")
1 2 3
```

Итератор

Важно то, что итератор «запоминает» именно позицию единицы данных в изменяющемся списке (в случае со словарём будет сгенерировано исключение):

```
>>> L=[1,2,3]
>>> i1 = iter(L)
>>> next(i1)
1
>>> L.insert(0, 0)
>>> L
[0, 1, 2, 3]
>>> next(i1)
1  #а не 2, как можно было бы предположить
```

Итератор по файлам

```
f = file("text.txt", "r")
for line in f:
    print line
```

```
В результате получим:
string #1
string #2
```

Слайд для привлечения внимания

Коротко про классы в Python

```
class A:
    def __init__(self, a, b):
        # конструктор с двумя параметрами
        self.foo = a
    def f1(self, x):
        # метод с одним параметром
        return self.foo + x
    def f2(self):
        # метод без параметров

a = A(2, 5)
a.f1(5)
```

Пользовательские итераторы

Итераторы могут быть бесконечными. Понятно, что такие итераторы не указывают на структуру данных или файл, результат их работы генерируется «на ходу». Для этого обычно используются функции-генераторы (см. далее).

При работе с бесконечными итераторами некоторые встроенные функции зацикливаются.

Например, max(), min(), list(), tuple(), операторы in и not in, а также цикл for.

Пример №1

```
class infinite_iterator:
    def __init__(self):
        self.counter = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.counter += 1
        return self.counter

>>> it = infinite_iterator()
>>> next(it)
0
```

Пример №2

```
class counter:
    def __init__(self, maximum):
        self.counter = 1
        self.maximum = maximum
    def __iter__(self):
        self.counter
        return self
    def __next__(self):
        self.counter += 1
        if self.counter < self.maximum:
            return self.counter
        raise StopIteration()
```

Пример №2

```
>>> it = counter(4)
>>> for x in it:
...     print(x, end=“ “)
0 1 2 3
>>> next(it)
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
it.next()
File "example.py", line 9, in next
raise StopIteration()
```

Пример №2-а

```
def send(self, value):
    self.counter = value - 1
    return next(self)

>>> c = counter(20)
>>> next(c)
0
>>> c.send(18)
18
>>> next(c)
19
>>> next(c)
StopIteration
```

Пример №3

```
class fibnum:
    def __init__(self):
        self.fn1 = 1
        self.fn2 = 1
    def __iter__(self):
        return self
    def __next__(self):
        oldfn2 = self.fn2
        self.fn2 = self.fn1
        self.fn1 += oldfn2
        return oldfn2
for i in fibnum():
    print(i, end=“ “)
    if i > 50:
        break
1 1 2 3 5 8 13 21 34 55
```

Пример №4

```
class simple_ints:
    def __init__(self):
        self.num = 1
    def __iter__(self):
        return self
    def __next__(self):
        while True:
            self.num += 1
            for x in range(2,self.num//2+1):
                if self.num % x == 0:
                    break
            else:
                return self.num
```

Вопрос №1

```
class cycle_iter:
    def __init__(self, L):
        self.L = L
        self.index = 1
    def __iter__(self):
        return self
    def __next__(self):
        if len(self.L) == 0:
            raise StopIteration()
        self.index += 1
        if self.index >= len(self.L):
            self.index = 0
        return self.L[self.index]
```

Вопрос №1

```
>>> L = ["A", "B", "C"]
>>> it = cycle_iter(L)
>>> it.next()
'A'
>>> L.insert(0, "Z")
>>> L
['Z', 'A', 'B', 'C']
>>> it.next()
# ???
...
>>> it.next()
'Z‘
>>> del L
>>> it.next()
# ???
```

Ответ

```
>>> L = ["A", "B", "C"]
>>> it = cycle_iter(L)
>>> it.next()
'A'
>>> L.insert(0, "Z")
>>> L
['Z', 'A', 'B', 'C']
>>> it.next()
>>> 'A' #итератор помнит позицию в списке
...
>>> it.next()
'Z'
>>> del L
>>> it.next()
>>> 'A' #пока хоть одна ссылка жива...
```

Списочные встраивания и генераторные выражения

Списочные встраивания (List comprehensions)

```
>>> L = []
>>> for x in range(10):
...     L.append(x)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


>>> [x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Списочные встраивания

```
>>> L = []
>>> for x in range(10):
...     if x % 2 == 0:
...     L.append(x)
>>> L
[0, 2, 4, 6, 8]
```

```
>>> [x for x in range(10)
        if x % 2 == 0]
[0, 2, 4, 6, 8]
```

Списочные встраивания

```
>>> L = []
>>> for i in range(3):
...     for j in range(3):
...         L.append(10*i+j)
>>> L
[0, 1, 2, 10, 11, 12, 20, 21, 22]


>>> [10*i+j for i in range(3)
        for j in range(3)]
[0, 1, 2, 10, 11, 12, 20, 21, 22]
```

Списочные встраивания

```
>>> L = []
>>> for i in range(4):
...     for j in range(4):
...         if i < j:
...             L.append((i, j))
>>> L
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]

>>> [(i, j) for i in range(4)
        for j in xrange(4) if i < j]
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

Списочные встраивания

```
[ expression for item1 in sequence1 if condition1
        for item2 in sequence2 if condition2 … ]
```

Такое выражение действует как вложенные циклы (с необязательными условиями), к результатам которого применяется выражение expression. При этом весь список сохраняется в памяти. Например,

```
>>> t = 100
>>> [x**2 + t for x in range(10) if x%2 == 0]
[100, 104, 116, 136, 164]
```

Списочные встраивания

Извлечём первые элементы из списков…

```
>>> list_of_lists =    [[1,2,3],
                        [4,5,6],
                        [7,8,9]]
>>> for list in list_of_lists:
...     list[0] #кладем его в список

>>> [list[0] for list in list_of_lists]
[1, 4, 7]
```

Списочные встраивания

Извлечём элементы номер 0 и 2 из списков…

```
>>> list_of_lists =   [[1,2,3],
                        [4,5,6],
                        [7,8,9]]
>>> for list in list_of_lists:
...     list[0] #кладем его в список
...     list[2] #кладем его в список

>>> [x  for list in list_of_lists
        for x in (list[0], list[2])]
[1, 3, 4, 6, 7, 9]
```

Вопрос №2

Что будет выведено на консоль?

```
>>> list_of_lists =    [[1,2,3],
                         [4,5,6],
                         [7,8,9]]

>>> [func  for list in list_of_lists
           for first_x in (list[0],)
           for func in (first_x**2,)][1]
```

Ответ

16

```
>>> list_of_lists = [[1,2,3], [4,5,6],
[7,8,9]]
>>> [func  for list in list_of_lists
           for first_x in (list[0],)
           for func in (first_x ** 2,)]
[1, 16, 49]
>>> [list[0] ** 2
           for list in list_of_lists]
[1, 16, 49]
```

Списочные встраивания

Недостатки списочных встраиваний:

- Они расходуют ресурсы;
- Списки всегда конечны.

Хочется итерироваться по списку, который может быть бесконечным, не расходуя ресурсы. Т.е. хотелось бы иметь возможность получить итератор вместо списка.

Генераторные выражения

Действует аналогично генератору списков, но генерируется не весь список, а итератор на него.

Значения будут вычисляться по мере вызова у итератора метода next(). Т. о. генераторные выражения вобрали в себя всё лучшее от итераторов (экономия ресурсов, возможность работы с бесконечными последовательностями) и списочных встраиваний (простой и короткий синтаксис).

Выражение при этом пишется в круглых скобках.

Генераторные выражения

```
>>> line_list = [" abc ", "ab", "a b "]
>>> [line.strip() for line in line_list]
['abc', 'ab', 'a b']
>>> stripped_iter =
(line.strip() for line in line_list)
>>> next(stripped_iter)
'abc'
>>> next(stripped_iter)
'ab'
```

Списочные встраивания vs Генераторные выражения

Списочные встраивания нельзя использовать для генерации бесконечных списков, а генераторные выражения — можно.

Генераторные выражения всегда пишутся в (), но если мы передаем их в функцию, скобки от функции считаются (вторые не ставим):

```
>>> sum(len(line) for line in line_list)
13
```

Списочные встраивания vs Генераторные выражения

При работе с генераторными выражениями надо быть осторожным с переопределением глобальных переменных и функций:

```
>>> def f(x):
...     return x**2
>>> it = (f(x) for x in xrange(2, 10))
>>> it.next()
4
>>> def f(x):
...     return x**3
>>> it.next()
27
```

Вопрос

Make list of tuples from the list in the following way.
From [1, 3, 5, 2, 2, 3, 5, 7] to

[(1), (3, 5, 2), (5, 2, 2, 3, 5), (2, 2), (2, 3), (3, 5, 7), (5, 7), (7)]

Вопрос+Ответ

Make list of tuples from the list in the following way.
From [1, 3, 5, 2, 2, 3, 5, 7] to

[(1), (3, 5, 2), (5, 2, 2, 3, 5), (2, 2), (2, 3), (3, 5, 7), (5, 7), (7)]

```
L=[1, 3, 5, 2, 2, 3, 5, 7]
[tuple(x) for i in range(len(L)) for x in (L[i:i+L[i]], )]

[(1), (3,5,2), (5,2,2,3,5), (2,2), (2,3), (3,5,7), (5,7), (7)]
```

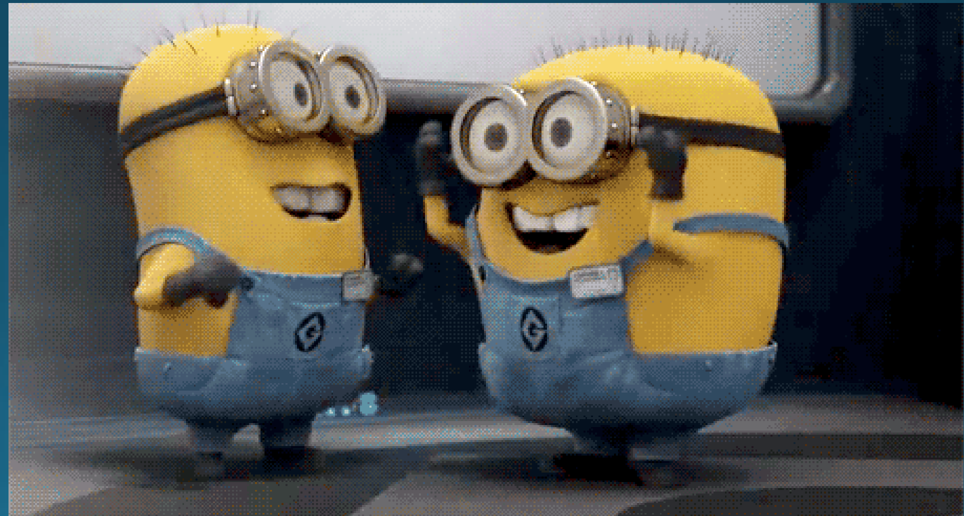
Вопрос

```
print(", ".join(["ha" if i else "Ha" for i in range(3)])+"!")
```

Вопрос+Ответ

```
print(", ".join(["ha" if i else "Ha" for i in range(3)])+"!")
```

```
Ha, ha, ha!
```



Генераторы

```
def not_a_generator():
    result = []
    for i in range(2000):
        result.append(expensive_computation(i))
    return result
```

Плохая производительность!

```
def my_generator():
    for i in range(2000):
        yield expensive_computation(i)

for i in my_generator():
        print(i)
```

Команда yield «замораживает» выполнение функции в указанном месте!

Генератор

Генераторы – это функции, возвращающие генераторные объекты, реализующие интерфейс итераторов (и не только).

```
>>> def infinite_generator():
...     i = 0
...     while (True):
...         yield i
...         i += 1
>>> it = infinite_generator();
>>> it.next()
0
>>> it.next()
1
>>> max(it) #выполнения этой строки вы не дождетесь
```

Генератор

```
>>> def counter(maximum):
...       for i in range(maximum):
...             yield i
>>> gen_obj = counter(10)
>>> gen_obj.next()
0
...
>>> gen_obj.next()
9
>>> gen_obj.next()
Traceback (most recent call last):
File "<pyshell#81>", line 1, in <module>
gen_obj.next()
StopIteration
```

```
>>> for i in counter(10):
...     print i
>>> a, b, c = counter(3)
```

Окончание генератора

Генераторы выбрасывают исключение StopIteration и прекращают итерирование генераторного объекта, когда:

- выполняется инструкция return, в генераторах она не может содержать возвращаемое значение (например, return 5);
- выполнение фукнции оказывается «за последней строчкой» генератора, т.е. инструкции закончились;
- в генераторе сгенерировано исключение StopIteration.

Генераторный объект

Генераторный объект — это больше, чем итератор. Он содержит методы:

- __next__();
- send(value) задать значение генераторному объекту;
- throw(ex_type) выбросить исключение в генераторе (на месте инструкции yield);
- close() генерирует исключение GeneratorExit внутри генератора. Генератор может отреагировать на это, сгенерировав исключение StopIteration или GeneratorExit. Отлавливание исключения и генерация чего-либо другого приведет к RuntimeError! Этот метод вызывается сборщиком мусора перед уничтожением генераторного объекта.

Пример

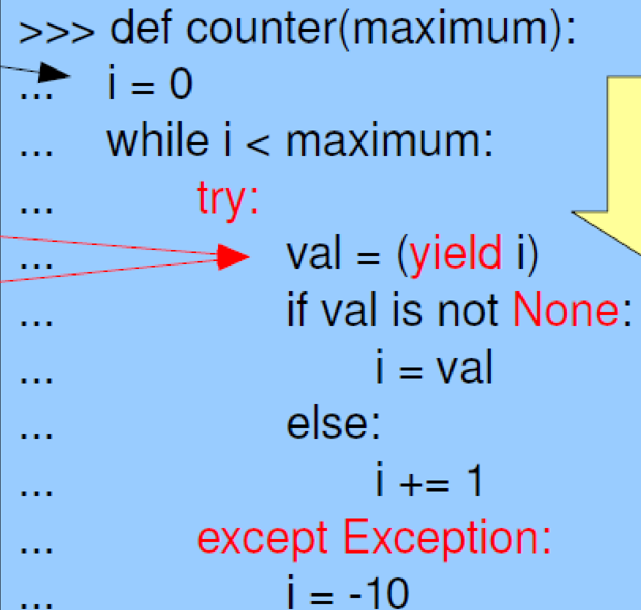
```
>>> def counter(maximum):
...  i = 0
...  while i < maximum:
...       try:
...            val = (yield i)
...            if val is not None:
...                  i = val
...            else:
...                 i += 1
...       except Exception:
...            i = -10
```

```
c = counter(5)
>>> c.next()
0
>>> c.next()
1
>>> c.send(4)
4
>>> c.throw(Exception)
10
>>> c.next()
9
>>> c.close()
>>> c.next() #StopIteration
```

Программа

```
>>> c = counter(5)
>>> c.next()
0
>>> c.next()
1
>>> c.send(4)
4
>>> c.throw(Exception)
-10
>>> c.next()
-9
>>> c.close()
>>> c.next()
~~~some info~~~
StopIteration
```

«Сопрограмма»-генератор



Пример

```
def fibnum():
    fn1 = 1
    fn2 = 1
    while True:
        yield fn2
        (fn1, fn2) = (fn1 + fn2, fn1)

for x in fibnum():
    print(x, end=“ “)
    if x > 50:
        break
1 1 2 3 5 8 13 21 34 55
```

Пример

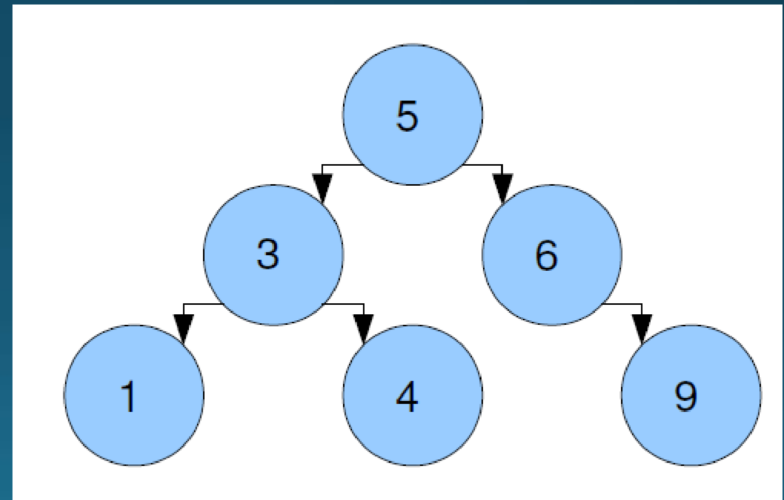
```
def simple_ints():
    num = 1
    while True:
        num += 1
        for x in range(2, num/2+1):
            if num % x == 0:
                break
        else:
            yield num

for x in simple_ints():
    print(x, end=“ “)
    if x > 60:
        break
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
```

Пример

```
class node:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
    def __iter__(self):
        return inorder(self)

n1 = node(1)
n4 = node(4)
n3 = node(3, n1, n4)
n9 = node(9)
n6 = node(6, right=n9)
root = node(5, n3, n6)
```



Пример

```
>>> list(root)
[1, 3, 4, 5, 6, 9]

>>>def inorder(t):
...     if t:
...         for x in inorder(t.left):
...             yield x
...         yield t.value
...         for x in inorder(t.right):
...             yield x
```

Пример

```
>>> list(root)
[1, 3, 4, 5, 6, 9]

>>> def inorder(t):
        if t:
            yield from inorder(t.left)
            yield t.value
            yield from inorder(t.right)
```

Itertools

Модуль itertools

Содержит функции для:

1. создания итераторов (в том числе на основе имеющихся итераторов);
2. обработки итерируемых элементов;
3. фильтрования возвращаемых итератором значений;
4. группировки возвращаемых итератором значений.

Itertools

```
>>> import itertools
>>> it = itertools.count(10)
>>> it.next()
10
>>> it.next()
11

>>> it = itertools.repeat("abc") #n
>>> it.next()
'abc'
>>> it.next()
'abc'
```

Itertools

itertools.count(start=0, step=1) - бесконечная арифметическая прогрессия с первым членом start и шагом step.

itertools.cycle(iterable) - возвращает по одному значению из последовательности, повторенной бесконечное число раз.

itertools.accumulate(iterable) - аккумулирует суммы.

accumulate([1,2,3,4,5]) --> 1 3 6 10 15

itertools.chain(*iterables) - возвращает по одному элементу из первого итератора, потом из второго, до тех пор, пока итераторы не кончатся.

Itertools

itertools.combinations(iterable, [r]) - комбинации длиной r из iterable без повторяющихся элементов.

combinations('ABCD', 2) --> AB AC AD BC BD CD

itertools.combinations_with_replacement(iterable, r) - комбинации длиной r из iterable с повторяющимися элементами.

combinations_with_replacement('ABCD', 2) --> AA AB AC AD BB BC BD CC CD DD

itertools.compress(data, selectors) - (d[0] if s[0]), (d[1] if s[1]), ...

compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F

Itertools

itertools.permutations(iterable, r=None) - перестановки длиной r из iterable.

itertools.product(*iterables, repeat=1) - аналог вложенных циклов.

product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy

itertools.starmap(function, iterable) - применяет функцию к каждому элементу последовательности (каждый элемент распаковывается).

starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000

itertools.tee(iterable, n=2) - кортеж из n итераторов.

itertools.zip_longest(*iterables, fill=None) - как встроенная функция zip, но берет самый длинный итератор, а более короткие дополняет fill.

zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

Пример

```
[list(f) for r in range(len(L)+1) for f
in itertools.combinations(L, r )]
```

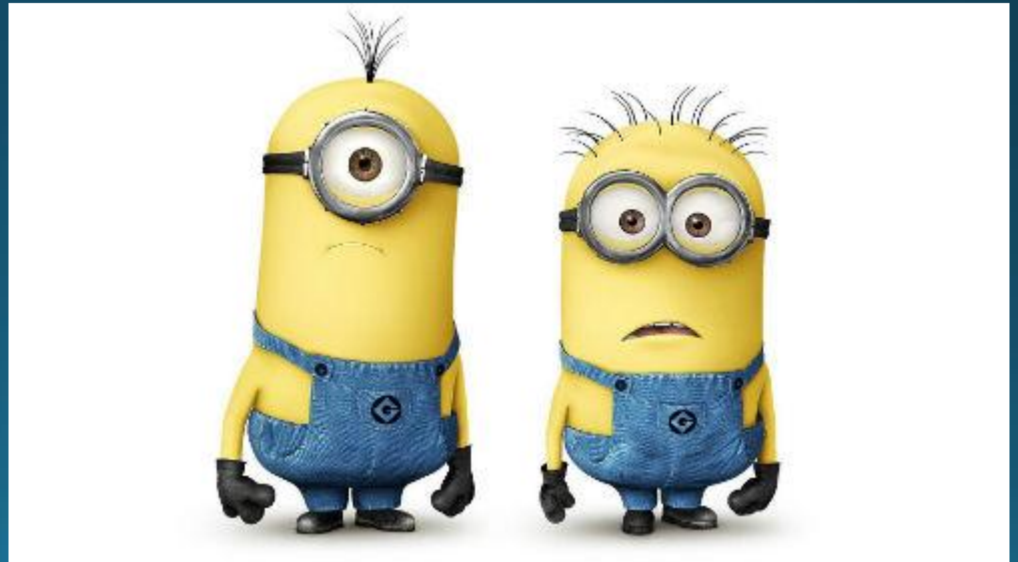
Пример

```
[list(f) for r in range(len(L)+1) for f
in itertools.combinations(L, r )]
```

```
L=[1, 2, 3]
[[], [1], [2], [3], [1, 2], [1, 3], [2,
3], [1, 2, 3]]
```

Пример

```
[ [a*b for a,b in x if a]
    for x in [ zip(x,L)
        for x in itertools.product(
        [1,0],repeat = len(L) ) ] ]
```



Пример

```
[ [a*b for a,b in x if a]
    for x in [ zip(x,L)
        for x in itertools.product(
        [1,0],repeat = len(L) ) ] ]
```

```
[[1, 2, 3], [1, 2], [1, 3], [1], [2, 3],
[2], [3], []]
```