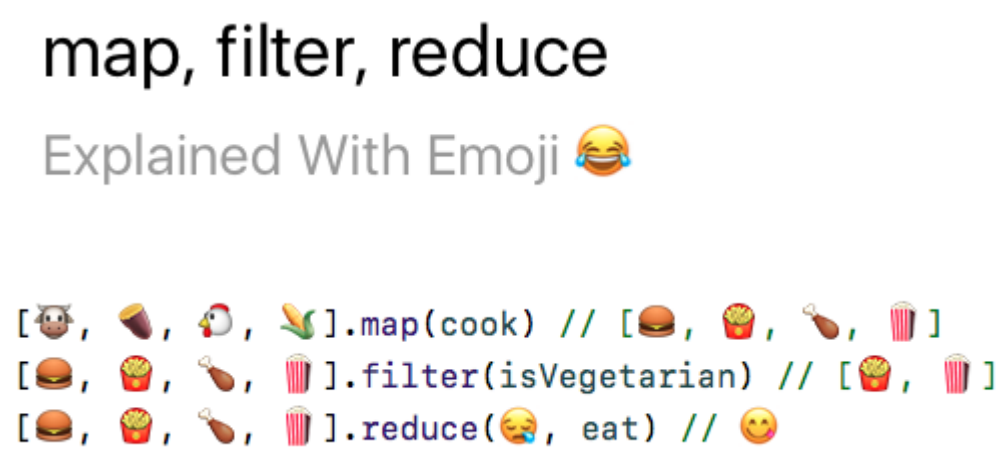
Введение в функциональное программирование (ч. 2)

Map / Filter / Reduce

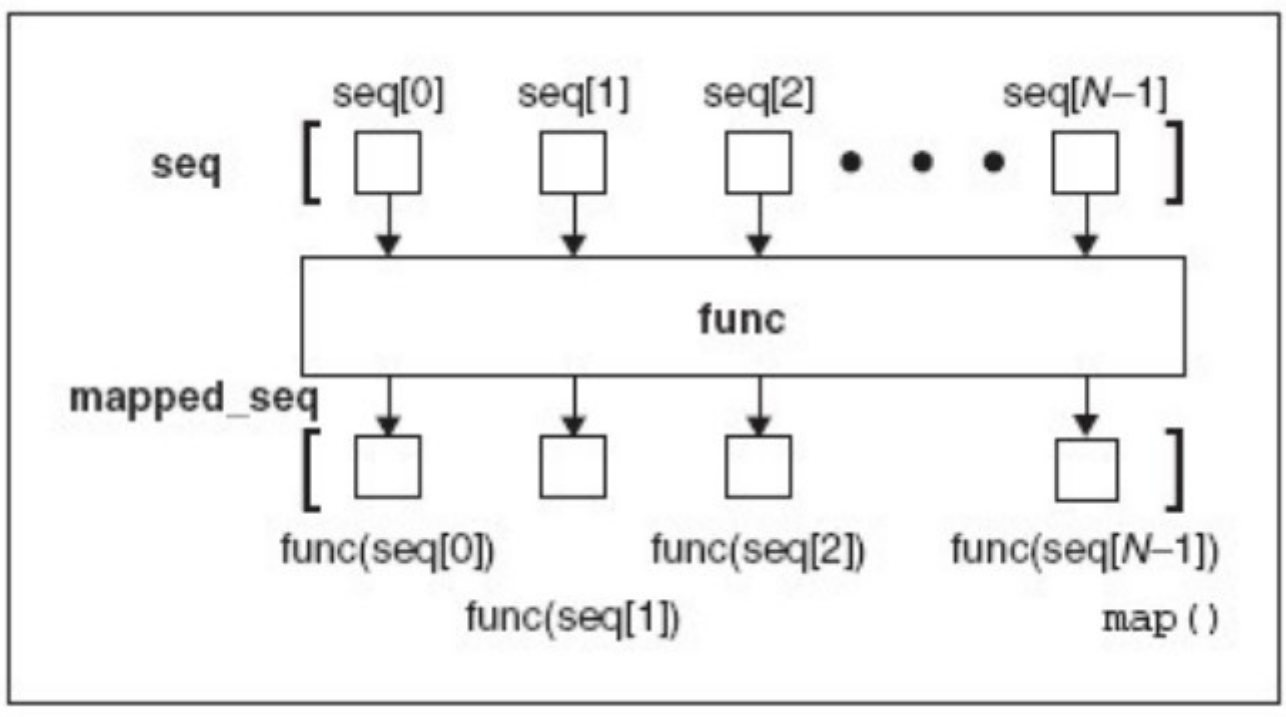


Map

Встроенная функция map() позволяет применить функцию к каждому элементу последовательности. Функция имеет следующий формат:

map(<Функция>, <Последовательность1>[, ... , <ПоследовательностьN>])

Функция возвращает объект, поддерживающий итерации, а не список.



In [1]:
```
squared = lambda x: x**2
```

In [5]:
```
list(range(10))
```

Out[5]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [7]:
```
m1 = map(squared, range(10))
m1
```

Out[7]:
```
<map at 0x1e381e88940>
```

In [8]:
```
list(m1)
```

Out[8]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [9]:
```
ls0 = list(zip(range(10), range(0, 100, 10)))
ls0
```

Out[9]:
```
[(0, 0),
 (1, 10),
 (2, 20),
 (3, 30),
 (4, 40),
 (5, 50),
 (6, 60),
 (7, 70),
 (8, 80),
 (9, 90)]
```

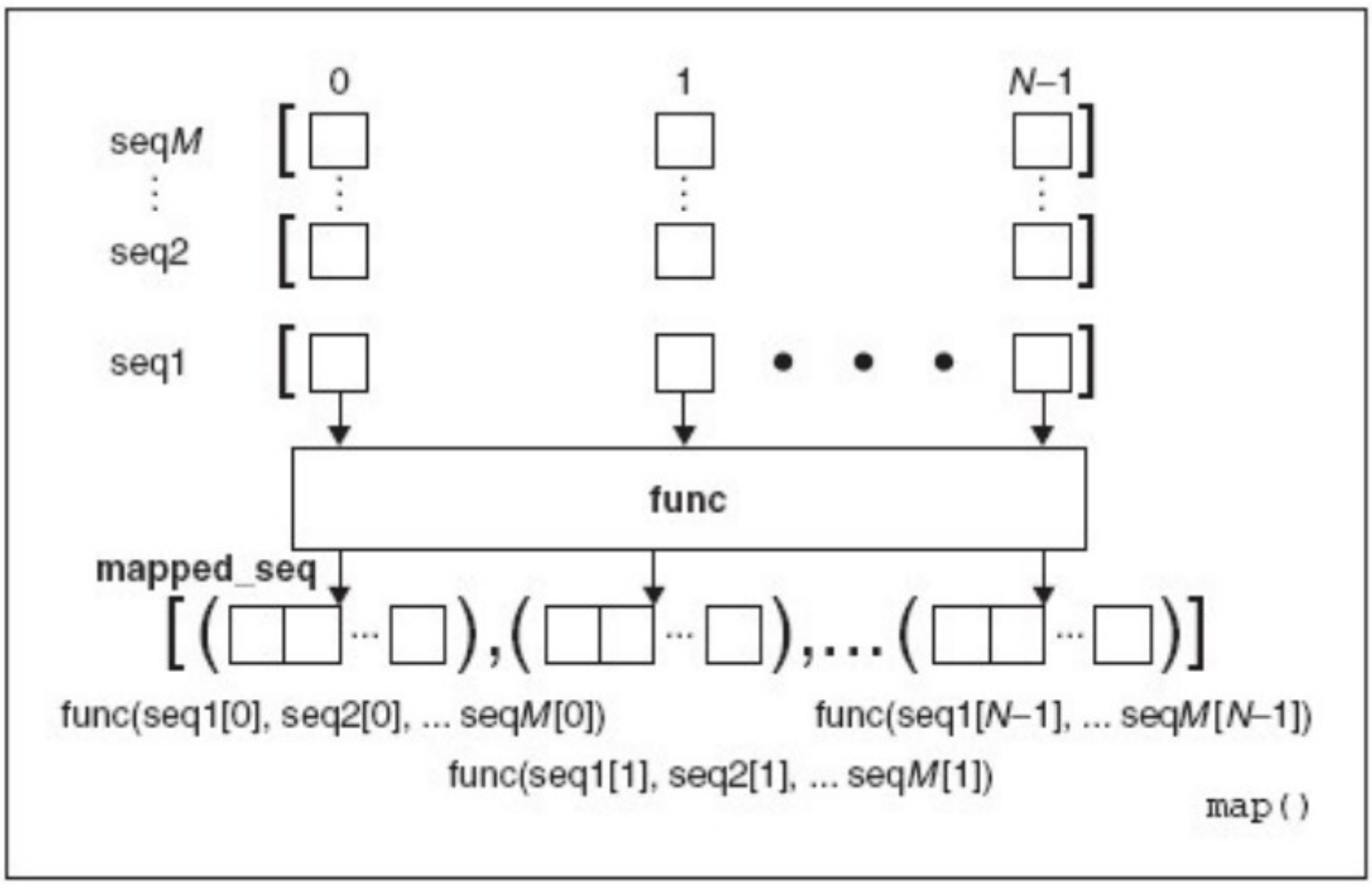
In [11]:
```
list(map(sum, ls0))
```

Out[11]:
```
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

In [12]:
```
# аналог генератор списков:
[sum(i) for i in ls0]
```

Out[12]:
```
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

Функции map() можно передать несколько последовательностей. В этом случае в функцию обратного вызова будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковом смещении.



In [13]:
```
ls1 = list(range(10))
ls2 = list(range(0, 100, 10))
ls3 = list(range(0, 1000, 100))
```

In [14]:
```
ls1, ls2, ls3
```

Out[14]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
 [0, 100, 200, 300, 400, 500, 600, 700, 800, 900])
```

In [15]:
```
list(map(lambda x, y: x+y, ls1, ls2))
```

Out[15]:
```
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

In [16]:
```
# аналог генератор списков:
[x+y for x, y in zip(ls1, ls2)]
```

Out[16]:
```
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

In [6]:
```
import operator as op
```

In [18]:
```
list(map(op.add, ls1, ls2))
```

Out[18]:
```
[0, 11, 22, 33, 44, 55, 66, 77, 88, 99]
```

In [20]:
```
list(map(lambda x, y, z: x+y+z, ls1, ls2, ls3))
```

Out[20]:
```
[0, 111, 222, 333, 444, 555, 666, 777, 888, 999]
```

In [21]:
```
list(map(sum, ls1, ls2, ls3))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-ed36c1442914> in <module>()
----> 1 list(map(sum, ls1, ls2, ls3))

TypeError: sum expected at most 2 arguments, got 3
```

In [22]:
```
sum(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-45c56698aff6> in <module>()
----> 1 sum(1, 2, 3)

TypeError: sum expected at most 2 arguments, got 3
```

In [23]:
```
sum((1, 2, 3))
```

Out[23]: 6

In [ ]:
```
def f(*a):
    return sum(a)
```

In [24]:
```
sum2 = lambda *a: sum(a)
```

In [25]:
```
sum2(1, 2, 3)
```

Out[25]: 6

In [27]:
```
list(map(sum2, ls1, ls2, ls3, ls1))
```

Out[27]: [0, 112, 224, 336, 448, 560, 672, 784, 896, 1008]

In [29]:
```
list(map(sum, [ls1, ls2, ls3]))
```

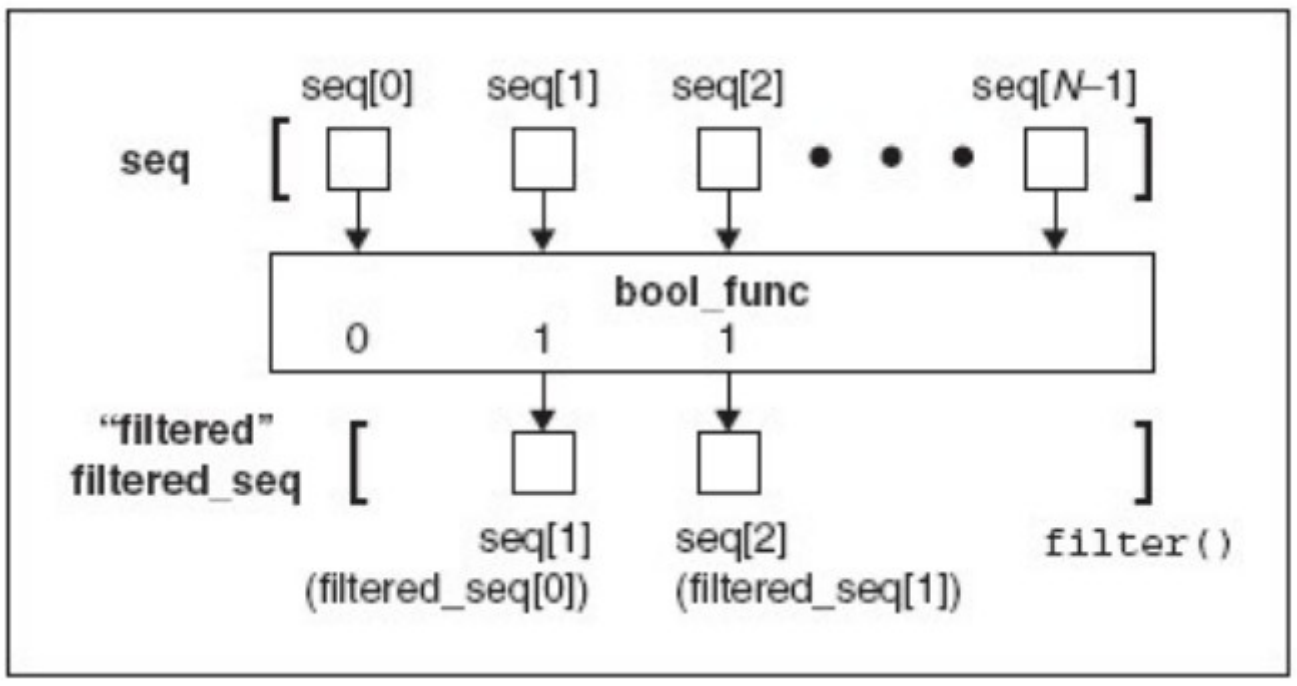
Out[29]: [45, 450, 4500]

Filter

Функция filter() позволяет выполнить проверку элементов последовательности. Формат функции:

filter(<Функция>, <Последовательность>)

Если в первом параметре вместо названия функции указать значение None, то каждый элемент последонательности будет проверен на соответствие значению True. Если элемент в логическом контексте возвращает значение False, то он не будет добавлен в возвращаемый результат. Функция возвращает объект, поддерживающий итерации, а не список.



In [30]:
```
import random
random.seed(42)
```

In [31]:
```
lr1 = [random.randint(-100, 100) for i in range(20)]
lr1
```

Out[31]:
```
[63,
 -72,
 -94,
 89,
 -30,
 -38,
 -43,
 -65,
 88,
 -74,
 73,
 89,
 39,
 -78,
 51,
 8,
 -92,
 -93,
 -77,
 -45]
```

In [32]:
```
list(filter(lambda x: x%3 == 0, lr1))
```

Out[32]:
```
[63, -72, -30, 39, -78, 51, -93, -45]
```

In [33]:
```
# аналог генератор списков:
[i for i in lr1 if i%3==0]
```

Out[33]:
```
[63, -72, -30, 39, -78, 51, -93, -45]
```

In [34]:
```
lr2 = [random.randint(-1, 1) for i in range(20)]
lr2
```

Out[34]:
```
[-1, 1, 1, -1, 1, -1, 1, 1, 1, 1, 0, -1, 0, 1, 0, -1, -1, 1, 0, 0]
```

In [38]:
```
# первый параметр None имеет особую семантику:
list(filter(None, lr2))
```

Out[38]:
```
[-1, 1, 1, -1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, 1]
```

In [42]:
```
# последовательное примененеие преобразований:
list(map(op.abs, filter(lambda x: x%3 == 0, lr1)))
```

Out[42]:
```
[63, 72, 30, 39, 78, 51, 93, 45]
```

In [39]:
```
%%timeit
[i for i in lr1 if i%3==0]
```

```
1000000 loops, best of 3: 1.7 µs per loop
```

In [40]:
```
%%timeit
list(filter(lambda x: x%3 == 0, lr1))
```

```
100000 loops, best of 3: 3.76 µs per loop
```

Reduce

In [1]:
```
import functools
```

In [2]:
```
from functools import reduce
```

functools.reduce(function, iterable[, initializer])

Вычисляет функцию от двух элементов последовательно, для элементов последовательности с права на лево таким образом, что результатом вычисления становится единственное число. Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the sequence. If the optional initializer is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If initializer is not given and sequence contains only one item, the first item is returned.

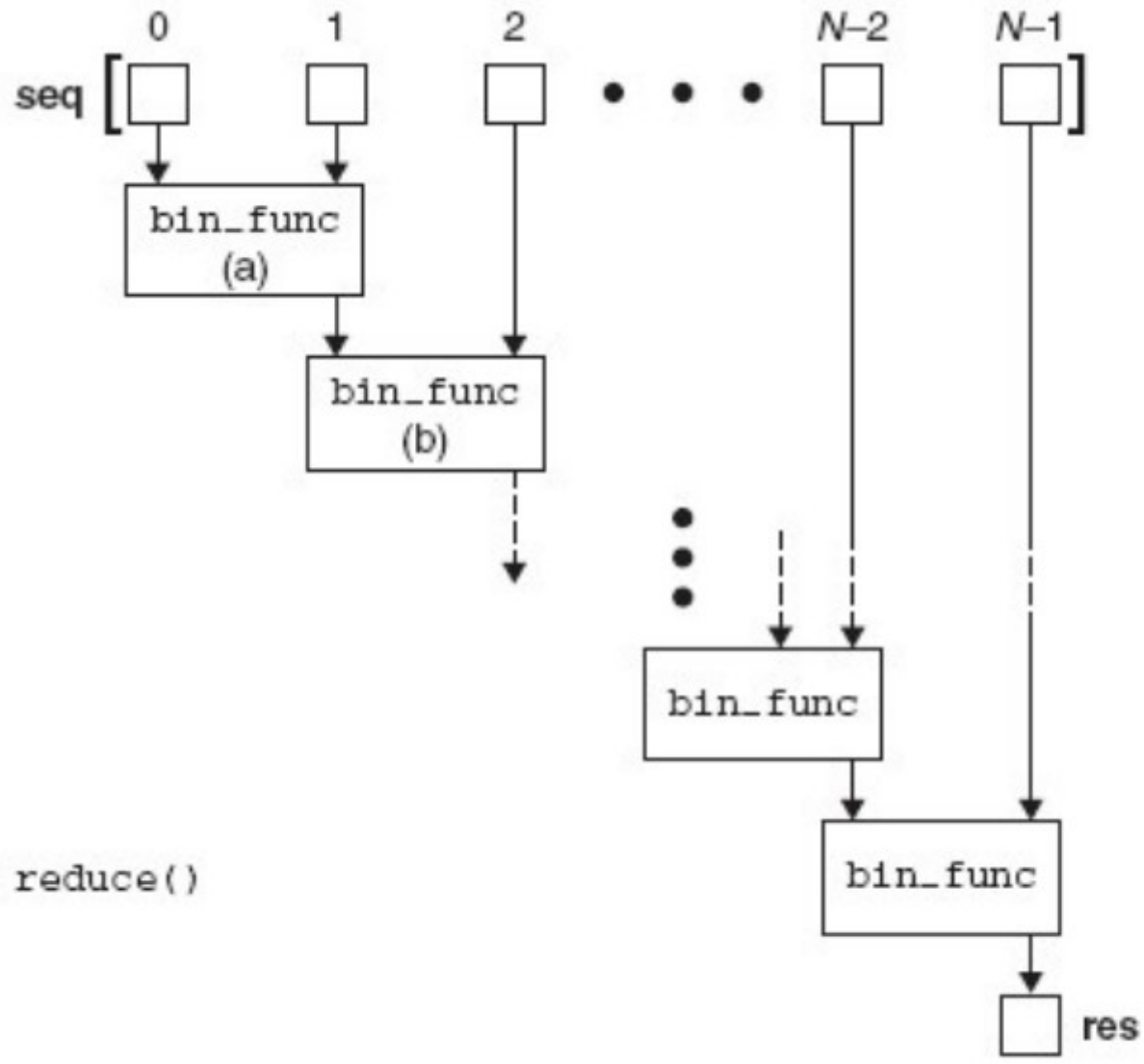
In [43]:
```
# Пример:
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) # вычисляется как ((((1+2)+3)+4)+5)
```

Out[43]:
```
15
```

Левый аргумент функции (аргумента reduce) это аккумулированное значение, правый аргумент - очередное значение из списка.

Если передан необязательный аргумент initializer, то он используется в качестве левого аргумента при первом применении функции (исходного аккумулированного значения).

Если initializer не пеередан, а последовательность имеет только одно значение, то возвращается это значенние.



(a) The value of this result is `bin_func(seq[0], seq[1])`
(b) The value of this result is `bin_func(bin_func(seq[0], seq[1]), seq[2])`, etc.

In [3]:
```
ls4 = list(range(10, 20))
ls4
```

Out[3]:
```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [7]:
```
reduce(op.add, ls4)
```

Out[7]:
```
145
```

In [8]:
```
reduce(op.add, ls4, 1000)
```

Out[8]:
```
1145
```

In [9]:
```
def add_verbose(x, y):
    print(f"add(x={x}, y={y}) -> {x+y}")
    return x + y
```

In [10]:
```
reduce(add_verbose, ls4)
```

```
add(x=10, y=11) -> 21
add(x=21, y=12) -> 33
add(x=33, y=13) -> 46
add(x=46, y=14) -> 60
add(x=60, y=15) -> 75
add(x=75, y=16) -> 91
add(x=91, y=17) -> 108
add(x=108, y=18) -> 126
add(x=126, y=19) -> 145
```

Out[10]:
```
145
```

In [11]:
```
reduce(add_verbose, ls4, 1000)
```

```
add(x=1000, y=10) -> 1010
add(x=1010, y=11) -> 1021
add(x=1021, y=12) -> 1033
add(x=1033, y=13) -> 1046
add(x=1046, y=14) -> 1060
add(x=1060, y=15) -> 1075
add(x=1075, y=16) -> 1091
add(x=1091, y=17) -> 1108
add(x=1108, y=18) -> 1126
add(x=1126, y=19) -> 1145
```

Out[11]:
```
1145
```

In [12]:
```
st = "This is a test.".split()
st
```

Out[12]:
```
['This', 'is', 'a', 'test.']
```

In [13]:
```
def f2(n, s):
    print(f'n: {n}, s: {s}, len(s): {len(s)}')
    return n + len(s)
```

In [14]:
```
reduce(f2, ['This', 'is', 'a', 'test.'], 0)
```

```
n: 0, s: This, len(s): 4
n: 4, s: is, len(s): 2
n: 6, s: a, len(s): 1
n: 7, s: test., len(s): 5
```

Out[14]:
```
12
```

In [15]:
```
reduce(lambda n, s: n + len(s), "This is a test.".split(), 0)
```

Out[15]:
```
12
```

In [16]:
```
reduce(lambda n, s: n + s, "This is a test.".split(), "")
```

Out[16]:
```
'Thisisatest.'
```

Итераторы

Итерируемый тип данных - это такой тип, который может возвращать свои элементы по одному. Любой объект, имеющий метод __iter__(), или любая последовательность (то есть объект, имеющий метод __getitem__(), принимающий целочисленный аргумент со значением от 0 и выше), является итерируемым и может предоставлять итератор.

Итератор - это объект, имеющий метод __next__(), который при каждом вызове возвращает очередной элемент и возбуждает исключение StopIteration после исчерпания всех элементов.

In [18]:
```
ls5 = [1, 2, 5, 3, 4, 8]
```

In [19]:
```
product = 1
it = iter(ls5) # iter() вызывает l5.__iter__() (ls5 - значение итерируемого типа, i - итератор )
while True:
    try:
        i = next(it) # next вызывает i.__next__()
        product *= i
    except StopIteration:
        break
product
```

Out[19]:
```
960
```

In [20]:
```
product = 1
for i in ls5:
    product *= i
print(product)
```

```
960
```

In [61]:
```
# другой способ:
reduce(op.mul, ls5)
```

Out[61]:
```
960
```

К итерируемым типам могут применяться: all(), any(), len(), min(), max(), sum():

In [62]:
```
len(ls5), min(ls5), max(ls5), sum(ls5)
```

Out[62]:
```
(6, 1, 8, 23)
```

In [63]:
```
ls6 = [i%2 ==0 for i in ls5]
ls6
```

Out[63]:
```
[False, True, False, False, True, True]
```

In [64]:
```
all(ls6), any(ls6)
```

Out[64]:
```
(False, True)
```

In [65]:
```
reduce(op.and_, ls6)
```

Out[65]:
```
False
```

Модуль itertools

In [66]:
```
import itertools as itl
```

https://docs.python.org/3.4/library/itertools.html (https://docs.python.org/3.4/library/itertools.html)

http://www.ilnurgi1.ru/docs/python/modules/itertools.html (http://www.ilnurgi1.ru/docs/python/modules/itertools.html)

itertools.islice(iterable, [start,]stop[, step])

Создает итератор, воспроизводящий элементы, которые вернула бы операция извлечения среза iterable[start:stop:step]. Первые start элементов пропускаются и итерации прекращаются по достижении позиции, указанной в аргументе stop. В необязательном аргументе step передается шаг выборки элементов. В отличие от срезов, в аргументах start, stop и step не допускается использовать отрицательные значения. Если аргумент start опущен, итерации начинаются с 0. Если аргумент step опущен, по умолчанию используется шаг 1.

In [67]:
```
itl.islice('ABCDEFG', 5)
```

Out[67]:
```
<itertools.islice at 0x1e3822bf908>
```

In [68]:
```
list(itl.islice('ABCDEFG', 5))
```

Out[68]:
```
['A', 'B', 'C', 'D', 'E']
```

In [69]:
```
list(itl.islice('ABCDEFG', 2,5))
```

Out[69]:
```
['C', 'D', 'E']
```

In [70]:
```
list(itl.islice('ABCDEFG', 2,8,2))
```

Out[70]:
```
['C', 'E', 'G']
```

Бесконечные итераторы

itertools.cycle(iterable)

Создает итератор, который в цикле многократно выполняет обход элементов в объекте iterable. За кулисами создает копию элементов в объекте iterable. Эта копия затем используется для многократного обхода элементов в цикле.

In [71]:
```
list(itl.islice(itl.cycle('ABCD'), 10))
```

Out[71]:
```
['A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B']
```

itertools.count(start=0, step=1)

Создает итератор, который воспроизводит упорядоченную и непрерывную последовательность целых чисел, начиная с n. Если аргумент n опущен, в качестве первого значения возвращается число 0. (Обратите внимание, что этот итератор не поддерживает длинные целые числа. По достижении значения sys.maxint счетчик переполнится и итератор продолжит воспроизводить значения, начиная с -sys.maxint - 1.)

In [73]:
```
list(itl.islice(itl.count(7), 10))
```

Out[73]:
```
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

itertools.repeat(object[, times])

Создает итератор, который многократно воспроизводит объект object. В необязательном аргументе times передается количество повторений. Если этот аргумент не задан, количество повторений будет бесконечным.

In [74]:
```
list(itl.islice(itl.repeat('a'), 10))
```

Out[74]:
```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

In [75]:
```
list(itl.repeat('a', 10))
```

Out[75]:
```
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```

Примеры других итераторов

In [76]:
```
list(itl.accumulate([1, 2, 3, 4, 5, 6]))
```

Out[76]:
```
[1, 3, 6, 10, 15, 21]
```

In [77]:
```
list(itl.accumulate(itl.repeat(3, 10)))
```

Out[77]:
```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

In [78]:
```
list(itl.chain([1,2,3], [4,5,6], [7,8,9]))
```

Out[78]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [79]: list(itl.chain.from_iterable(['abc', 'def']))
Out[79]: ['a', 'b', 'c', 'd', 'e', 'f']

In [81]: list(itl.zip_longest('abcd', 'def'))
Out[81]: [('a', 'd'), ('b', 'e'), ('c', 'f'), ('d', None)]

In [82]: list(itl.compress('ABCDEF', [1,0,1,0,1,1]))
Out[82]: ['A', 'C', 'E', 'F']

In [83]: list(itl.dropwhile(lambda x: x<5, [1,4,6,4,1]))
Out[83]: [6, 4, 1]
```

Функции генераторы

Функция-генератор, или метод-генератор - это функция, или метод, содержащая выражение yield. В результате обращения к функции-генератору возвращается итератор. Значения из итератора извлекаются по одному, с помощью его метода __next__(). При каждом вызове метода __next__() он возвращает результат вычисления выражения yield. (Если выражение отсутствует, возвращается значение None.) Когда функция-генератор завершается или выполняет инструкцию return, возбуждается исключение StopIteration.

На практике очень редко приходится вызывать метод __next__() или обрабатывать исключение StopIteration. Обычно функция-генератор используется в качестве итерируемого объекта.

```
In [85]: # Создает и возвращает список
def letter_range_l(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result

In [86]: letter_range_l('a','o')
Out[86]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']

In [87]: for l in letter_range_l('a', 'o'):
    print(l, end=', ')

a, b, c, d, e, f, g, h, i, j, k, l, m, n,

In [88]: # Возвращает каждое значение по требованию
def letter_range_g(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)

In [89]: for l in letter_range_g('a', 'o'):
    print(l, end=', ')

a, b, c, d, e, f, g, h, i, j, k, l, m, n,

In [86]: list(letter_range_g('a', 'o'))
Out[86]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']

In [90]: g1 = letter_range_g('a', 'c')
g1
Out[90]: <generator object letter_range_g at 0x000001E3822C5620>

In [91]: next(g1)
Out[91]: 'a'

In [92]: next(g1)
Out[92]: 'b'

In [93]: next(g1)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-93-9abcc56105c0> in <module>()
----> 1 next(g1)

StopIteration:
```

In [94]:
```
list(letter_range_g('a', 'o'))
```

Out[94]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']
```

Кроме функций-генераторов уществует возможность создавать еще и выражения-генераторы. Синтаксически они очень похожи на генераторы списков, единственное отличие состоит в том, что они заключаются не в квадратные скобки, а в круглые.

(expression for item in iterable)

(expression for item in iterable if condition)

In [95]:
```
# функция-генератор, возвращающая из словаря пары ключ-значение в порядке убывания ключа
def items_in_key_order(d):
    for key in sorted(d, reverse=True):
        yield key, d[key]
```

In [96]:
```
d1 = dict(zip(['ac','vg','ddxf','c','ff', 'bfafakl'], range(6)))
d1
```

Out[96]:
```
{'ac': 0, 'bfafakl': 5, 'c': 3, 'ddxf': 2, 'ff': 4, 'vg': 1}
```

In [97]:
```
list(items_in_key_order(d1))
```

Out[97]:
```
[('vg', 1), ('ff', 4), ('ddxf', 2), ('c', 3), ('bfafakl', 5), ('ac', 0)]
```

In [99]:
```
[(key, d1[key]) for key in sorted(d1, reverse=True)]
```

Out[99]:
```
[('vg', 1), ('ff', 4), ('ddxf', 2), ('c', 3), ('bfafakl', 5), ('ac', 0)]
```

In [100]:
```
list(((key, d1[key]) for key in sorted(d1, reverse=True)))
```

Out[100]:
```
[('vg', 1), ('ff', 4), ('ddxf', 2), ('c', 3), ('bfafakl', 5), ('ac', 0)]
```

In [101]:
```
items_in_key_order(d1)
```

Out[101]:
```
<generator object items_in_key_order at 0x000001E3822C5E08>
```

In [102]:
```
g2 = ((key, d1[key]) for key in sorted(d1, reverse=True))
g2
```

Out[102]:
```
<generator object <genexpr> at 0x000001E3822C53B8>
```

In [103]:
```
next(g2)
```

Out[103]:
```
('vg', 1)
```

In [104]:
```
next(g2)
```

Out[104]:
```
('ff', 4)
```

Благодаря отложенным вычислениям в генераторах (функциях и выражения) можно экономить ресурсы и создавать генераторы бесконечных последовательностей.

In [105]:
```
# бесконечный генератор четвертей:
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

In [106]:
```
result = []
for x in quarters():
    result.append(x)
    if x >= 5.0:
        break
result
```

Out[106]:
```
[0.0,
 0.25,
 0.5,
 0.75,
 1.0,
 1.25,
 1.5,
 1.75,
 2.0,
 2.25,
 2.5,
 2.75,
 3.0,
 3.25,
 3.5,
 3.75,
 4.0,
 4.25,
 4.5,
 4.75,
 5.0]
```

In [108]:
```
list(itl.islice(quarters(10.0), 30))
```

Out[108]:
```
[10.0,
 10.25,
 10.5,
 10.75,
 11.0,
 11.25,
 11.5,
 11.75,
 12.0,
 12.25,
 12.5,
 12.75,
 13.0,
 13.25,
 13.5,
 13.75,
 14.0,
 14.25,
 14.5,
 14.75,
 15.0,
 15.25,
 15.5,
 15.75,
 16.0,
 16.25,
 16.5,
 16.75,
 17.0,
 17.25]
```

In [ ]: