Аргументы, Лямбда-функции, Списочные сокращения, Декораторы

Множественные аргументы

In [4]:

```
def sum(a, b):
    return a + b

sum(5, 6)
```

Out[4]:

```
11
```

In [5]:

```
sum(5,2,1,4)
```

```
---------------------------------------------------------------------
--------
TypeError                                 Traceback (most recent ca
ll last)
<ipython-input-5-6b8d6e787439> in <module>
----> 1 sum(5,2,1,4)

TypeError: sum() takes 2 positional arguments but 4 were given
```

In [80]:

```
def sum_infinite(*numbers):
    a = 0
    for i in numbers:
        a += i
    return a

sum_infinite(1,2,34,4,5,65,7)
```

Out[80]:

```
118
```

In [8]:

```
def info(**data):
    for variable, value in data.items():
        print("{}: {}".format(variable, value))

info(Name = "Anna", City = "Rome")
```

```
Name: Anna
City: Rome
```

Лямбда-функции

In [11]:

```
def sum(a, b):
    return a + b

sum_lambda = lambda x, y: x + y

print(sum(2, 3))
print(sum_lambda(2, 3))
```

```
5
5
```

Лямбда-функции - это удобный и короткий способ записи обычной функции. Оне не дает преимуществ в скорости или экономии памяти

Важно понимать, что все функции могут принимать и нулевое колличество элементов. Лямбда функции не исключение

In [12]:

```
helloWorld = lambda: "Hello World!"

helloWorld()
```

Out[12]:

```
'Hello World!'
```

In [17]:

```
v = lambda **x : print(x)
v(LL = 5, BB = 6, CC = 7)
```

```
{'LL': 5, 'BB': 6, 'CC': 7}
```

In [18]:

```
v = lambda *x : print(x)
v(5, 6, 7)
```

```
(5, 6, 7)
```

Списочные сокращения

Существует удобная форма создания и записи списков. Списочные сокращения призваны упросить и ускорить генерацию либо сохранение данных. Предположим, мы хотим иметь список квадратов первой сотни натуральных чисел.

In [21]:

```
sq = []
for i in range(1, 101):
    sq.append(i ** 2)
```

In [22]:

```
print(sq)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 25
6, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841,
900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 168
1, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 270
4, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 396
9, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 547
6, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 722
5, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 921
6, 9409, 9604, 9801, 10000]
```

Мы можем сократить это до кода из одной строки

In [23]:

```
squares = [x ** 2 for x in range(1, 101)]
```

In [81]:

```
b  = ["AA AA", "BB BB"]
```

In [82]:

```
v = [x.split() for x in b]
```

In [83]:

```
v
```

Out[83]:

```
[['AA', 'AA'], ['BB', 'BB']]
```

In [24]:

```
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 25
6, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841,
900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 168
1, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 270
4, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 396
9, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 547
6, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 722
5, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 921
6, 9409, 9604, 9801, 10000]
```

In [86]:

```
a = [x * y for x in [4,5,6] for y in [1, 2 ,3]]
```

```
  File "<ipython-input-86-a182fcf9ce32>", line 1
    a = [x * y for x in range(1,10) and for y in [1, 2 ,3]]
                                        ^
SyntaxError: invalid syntax
```

In [85]:

```
a
```

Out[85]:

```
[1,
 2,
 3,
 2,
 4,
 6,
 3,
 6,
 9,
 4,
 8,
 12,
 5,
 10,
 15,
 6,
 12,
 18,
 7,
 14,
 21,
 8,
 16,
 24,
 9,
 18,
 27]
```

Аналогично можно проделать для словарей

In [38]:

```
a = {x : x ** 2 for x in range(100) }
```

In [40]:

```
print(a)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81,
10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225, 16: 256, 17:
289, 18: 324, 19: 361, 20: 400, 21: 441, 22: 484, 23: 529, 24: 576,
25: 625, 26: 676, 27: 729, 28: 784, 29: 841, 30: 900, 31: 961, 32:
1024, 33: 1089, 34: 1156, 35: 1225, 36: 1296, 37: 1369, 38: 1444, 3
9: 1521, 40: 1600, 41: 1681, 42: 1764, 43: 1849, 44: 1936, 45: 202
5, 46: 2116, 47: 2209, 48: 2304, 49: 2401, 50: 2500, 51: 2601, 52:
2704, 53: 2809, 54: 2916, 55: 3025, 56: 3136, 57: 3249, 58: 3364, 5
9: 3481, 60: 3600, 61: 3721, 62: 3844, 63: 3969, 64: 4096, 65: 422
5, 66: 4356, 67: 4489, 68: 4624, 69: 4761, 70: 4900, 71: 5041, 72:
5184, 73: 5329, 74: 5476, 75: 5625, 76: 5776, 77: 5929, 78: 6084, 7
9: 6241, 80: 6400, 81: 6561, 82: 6724, 83: 6889, 84: 7056, 85: 722
5, 86: 7396, 87: 7569, 88: 7744, 89: 7921, 90: 8100, 91: 8281, 92:
8464, 93: 8649, 94: 8836, 95: 9025, 96: 9216, 97: 9409, 98: 9604, 9
9: 9801}
```

Так же можно проделать и с множествами и tuple

In [41]:

```
a = {x**2 for x in range(100)}
b = tuple(x**2 for x in range(100))
print(a)
print(b)
```

```
{0, 1, 1024, 4096, 4, 9216, 9, 16, 529, 3600, 4624, 25, 36, 2601, 4
9, 7225, 3136, 64, 576, 1089, 6724, 1600, 2116, 5184, 7744, 9801, 8
1, 8281, 6241, 100, 625, 121, 4225, 1156, 8836, 3721, 144, 1681, 27
04, 5776, 4761, 2209, 676, 169, 3249, 9409, 196, 1225, 5329, 729, 2
25, 1764, 7396, 6889, 7921, 2809, 256, 2304, 6400, 3844, 4356, 784,
1296, 8464, 289, 3364, 4900, 5929, 1849, 9025, 324, 841, 1369, 240
1, 2916, 5476, 361, 3969, 900, 9604, 4489, 400, 1936, 7056, 7569, 3
481, 6561, 1444, 8100, 5041, 441, 961, 2500, 6084, 8649, 3025, 484,
2025, 1521, 5625}
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 84
1, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2
704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 39
69, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 547
6, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 722
5, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 921
6, 9409, 9604, 9801)
```

Зачем здесь tuple перед скобками станет понятней чуть позднее

Так же существует возможность генерировать словари с помощью сочетания lambda функций и функций map или filter

Итераторы

Итератор — это объект-абстракция, который позволяет брать из источника, будь это stdin или, скажем, какой-то большой контейнер, элемент за элементом, при этом итератор знает только о том объекте, на котором он в текущий момент остановился.

В Python (и не только в нем) есть два понятия, которые звучат практически одинаково, но обозначают разные вещи, — iterator и iterable. Первое — это объект, который реализует описанный выше интерфейс, а второе — контейнер, который может служить источником данных для итератора. Чтобы получить итератор для iterable, в Python есть специальная функция iter

Например, всем известный список является итерируемым объектом.

In [45]:

```
nucleotides = ["A", "T", "G", "C"]

nucleotides_iterator = iter(nucleotides)
print (type(nucleotides_iterator), nucleotides_iterator)
```

```
<class 'list_iterator'> <list_iterator object at 0x7f3765726748>
```

In [46]:

```
print (next(nucleotides_iterator))
print (next(nucleotides_iterator))
print (next(nucleotides_iterator))
print (next(nucleotides_iterator))
```

```
A
T
G
C
```

In [47]:

```
def print_nucleotides():
    for n in nucleotides:
        print (n)

print_nucleotides()
```

```
A
T
G
C
```

In [ ]:

```

```

In [ ]:

```

```

Декораторы

Мы помним, что функции в питоне могут являться переменными, и, например, можем написать функцию, которая создает други функции

In [56]:

```
def create_add(y):
    def new_func(x):
        return y + x
    return new_func

add5 = create_add(5)
add5(9)
```

Out[56]:

```
14
```

Допустим, у нас возникает задача отладки, для начала просто ввыводе сообщений при запуске функции и при завершении ее работы.

In [31]:

```
def decor(func):
    def decorated_func(*args, **kwargs):
        print ("Function started")
        result = func(*args, **kwargs)
        print ("Function ended")
        return result
    return decorated_func
```

In [57]:

```
decor_add = decor(add5)

decor_add(5)
```

```
Function started
Function ended
```

Out[57]:

```
10
```

Для работы с декораторами, в питоне есть специальная конструкция

In [33]:

```
def simple():
    return "Hello"

@decor
def not_simple():
    return "Hello"
```

In [34]:

```
simple()
```

Out[34]:

```
'Hello'
```

In [35]:

```
not_simple()
```

```
Function started
Function ended
```

Out[35]:

```
'Hello'
```

`Декоратору непросто передать дополнительные аргументы, т.к он принимает первым аргументом функцию. На самом деле, для этого надо сделать не декоратор, а функцию, возвращающую декоратор

Напишем, например, декоратор, который будет писать в лог текущее время и дату и свое имя

In [36]:

```
def log_wrapper(name):
    import datetime
    def real_decorator(func):
        def decorated_func(*args, **kwargs):
            now = datetime.datetime.now()
            print ("{}, Date: {}, Function started".format(name, now))
            result = func(*args, **kwargs)
            now = datetime.datetime.now()
            print ("{}, Date: {}, Function ended".format(name, now))
            return result
        return decorated_func
    return real_decorator
```

In [37]:

```
@log_wrapper("Simple logger")
def fibonacci(n):
    f0, f1 = 0, 1
    for i in range(n):
        f0, f1 = f1, f0 + f1
    return f0

res = fibonacci(1000000)
```

```
Simple logger, Date: 2020-03-10 08:43:01.323938, Function started
Simple logger, Date: 2020-03-10 08:43:15.604734, Function ended
```

Map Filter

In [48]:

```
squares = [x ** 2 for x in range(101)]
print (squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 84
1, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2
704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 39
69, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 547
6, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 722
5, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 921
6, 9409, 9604, 9801, 10000]
```

In [49]:

```
squares = list(map(lambda x : x ** 2, range(101)))
print (squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 84
1, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2
704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 39
69, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 547
6, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 722
5, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 921
6, 9409, 9604, 9801, 10000]
```

In [50]:

```
div7_5 = [x for x in range(1000) if x % 7 == 5]
div7_5[0:10]
```

Out[50]:

```
[5, 12, 19, 26, 33, 40, 47, 54, 61, 68]
```

In [89]:

```
div7_5 = list(filter(lambda x : x % 7 == 5, range(1000)))
print (div7_5)
```

```
[5, 12, 19, 26, 33, 40, 47, 54, 61, 68, 75, 82, 89, 96, 103, 110, 1
17, 124, 131, 138, 145, 152, 159, 166, 173, 180, 187, 194, 201, 20
8, 215, 222, 229, 236, 243, 250, 257, 264, 271, 278, 285, 292, 299,
306, 313, 320, 327, 334, 341, 348, 355, 362, 369, 376, 383, 390, 39
7, 404, 411, 418, 425, 432, 439, 446, 453, 460, 467, 474, 481, 488,
495, 502, 509, 516, 523, 530, 537, 544, 551, 558, 565, 572, 579, 58
6, 593, 600, 607, 614, 621, 628, 635, 642, 649, 656, 663, 670, 677,
684, 691, 698, 705, 712, 719, 726, 733, 740, 747, 754, 761, 768, 77
5, 782, 789, 796, 803, 810, 817, 824, 831, 838, 845, 852, 859, 866,
873, 880, 887, 894, 901, 908, 915, 922, 929, 936, 943, 950, 957, 96
4, 971, 978, 985, 992, 999]
```

In [ ]:

```

```

Однострочный IF ELSE

Важно, что в Python есть однострочный if else. Называется тернарным оператором, так как использует три аргумента - условие, что вовзращать, если условие истинно и что возвращать, если условие ложно

In [53]:

```
a = 5 if 14 % 7 == 0 else 7
print (a)
codon = "UAA"
a = "StopCodon" if codon in ("UAA",  "UAG",  'UGA') else "Tryptophan" if codon =
= 'UGG' else "Other"
print (a)
```

```
5
StopCodon
```

In [54]:

```
odd_or_even = ["Odd" if x % 2 == 1 else "Even" for x in range(100)]
odd_or_even[0:10]
```

Out[54]:

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Eve
n', 'Odd']
```

Почему map, filter и reduce хороши в меру? (Как и вообще функциональное программирование в Python)*

Как проверить, является ли число 101 простым?

In [115]:

```
if reduce(lambda x, y : 0 if x % y == 0 else x, range(2,101), 101):
    print ("Prime")
else:
    print ("Not prime")
```

```
Prime
```

Как обобщить на любое натуральное число

In [116]:

```
is_prime = lambda z : False if z == 1 else True if reduce(lambda x,
                                                          y : 0 if x % y == 0 el
se x, range(2,z), z) else False
print (is_prime(10), is_prime(1), is_prime(2), is_prime(17))
```

```
False False True True
```

Возьмем только простые числа из первых 1000

In [117]:

```
primes = list(filter(lambda z : False if z == 1 else True if reduce(lambda x,
                                                                    y : 0 if x % y == 0 el
se x, range(2,z), z) else False,
            range(1, 1001)))
print (primes[0:10])
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Теперь оставим получим от них квадраты

In [118]:

```
prime_squares = map(lambda x : x ** 2, (filter(lambda z : False if z == 1 else T
rue if reduce(lambda x,
                                                                    y : 0 if x % y == 0 el
se x, range(2,z), z) else False,
            range(1, 1001))))
print (list(prime_squares)[0:10])
```

```
[4, 9, 25, 49, 121, 169, 289, 361, 529, 841]
```

И оставим их них те, что на втором месте имеют 2

In [119]:

```
prime_squares_filtered = filter(lambda x : (x % 100) // 10  == 2, map(lambda x :
x ** 2, (filter(lambda z : False if z == 1 else True if reduce(lambda x,
                           y : 0 if x % y == 0 else x, range(2,z), z) else Fals
e, range(1, 1001)))) )

print (list(prime_squares_filtered)[0:10])
```

```
[25, 121, 529, 3721, 5329, 7921, 16129, 19321, 29929, 44521]
```

In [75]:

```
a = ["EEEE\n" for _ in range(100)]

f = open("innnn", "w")
for i in a:
    f.write(i)
```

In [76]:

```
ls
```

```
in  innnn  LectN.ipynb
```

In [78]:

```
a = [x.strip("\n") for x in open("innnn", "r")]

a
```

Out[78]:

```
['EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
```

```
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE',
 'EEEE']
```

In [71]:

```
cat in
```

In [94]:

```
a = [(lambda : x) for x in range(10)]
```

In [95]:

```
a[1]()
```

Out[95]:

9

In [96]:

```
a[2]()
```

Out[96]:

9

In [ ]:

```

```