

# Лекция 4

Язык программирования Python.

# Домашнее задание

1. Отсортируйте список случайной длины в зависимости от мода по возрастанию и по убыванию
2. Найдите сумму всех чисел меньше 1000, кратных 3 или 5 с помощью функции генератора
3. Запишите в словарь по ключам от 1 до 10, список чисел (подается на входе), которые делятся на соответствующие ключи

# Ключевое слово yield

```
home > serg > tmp > test.py > gen
1  def gen(start, end, step):
2      while start < end:
3          yield start
4          start += step
5
6
7  g = gen(1, 10, 1)
8  print(sum(g))
9
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS

```
[13:07:23] serg :: serg-pc → ~/tmp»
python test.py
45
[13:07:25] serg :: serg-pc → ~/tmp»
```

# Ключевое слово yield

```
1 def gen(start, end, step):
2     while start < end:
3         yield start
4         start += step
5
6
7 g = gen(1, 10, 1)
8 for i in g:
9     print(i)
10
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[13:10:56] serg :: serg-pc → ~/tmp»
python test.py
```

```
1
2
3
4
5
6
7
8
9
```

```
[13:10:57] serg :: serg-pc → ~/tmp»
```

# Ключевое слово yield

```
1  def gen(start, end, step):
2      while start < end:
3          yield start
4          start += step
5
6
7  g = gen(1, 10, 1)
8  for i in g:
9      print(i)
10
11 for i in g:
12     print(i)
13
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS

```
[13:09:18] serg :: serg-pc → ~/tmp»
python test.py
1
2
3
4
5
6
7
8
9
[13:10:31] serg :: serg-pc → ~/tmp»
```

# План занятия

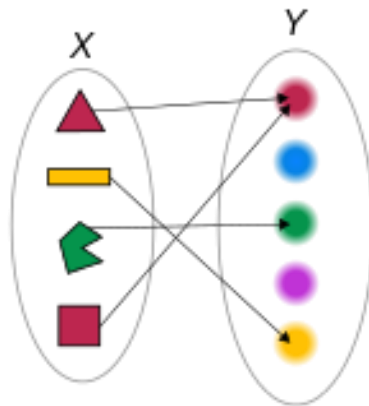
1. Функции
2. Определение
3. Аргументы функции
  - a. Обязательные
  - b. Необязательные
  - c. Передача аргументов
    - i. по значению
    - ii. по ссылке
4. Области видимости переменных
5. Рекурсия
6. Анонимные функции
7. Практика

## Переиспользование кода

**DRY** - don't repeat yourself. Принцип говорит нам о том, что если вы заметили в вашем коде места, которые принципиально делают одну и ту же работу но различаются лишь данными над которыми они работают, то нужно такие места локализовывать и выносить в функции или циклы (в зависимости от ситуации).

# Функция (математика)

Функцией  $f$ , определенной на множестве  $X$  со значениями в множестве  $Y$  называют “правило”  $f(x)$  такое, что  $\forall x \in X \exists! f(x) \in Y$





# Функция (программирование)

**Функция в программировании, или подпрограмма** — фрагмент программного кода, к которому можно обратиться из другого места программы. При этом предполагается, что функция должна возвращать некоторый результат своих вычислений.

# Процедура

**Процедура** — это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из последующих частей программы для выполнения определенных действий. Основное отличие процедуры от функции заключается в том, что она не возвращает никакого результат, в тоже время она может прямо или косвенно изменить состояние программы.

# Встроенные функции Python

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

# Определение функции в Python

Синтаксически функция в Python определяется следующим образом:

```
def function_name(arg1, arg2,...,argN):  
    do_some_work
```

- **def** - сигнализирует интерпретатору, что после него начинается определение функции
- **function\_name** - имя функции определяется программистом
- В скобках через запятую перечисляется список аргументов, которыми функция оперирует
- после “:” в новом программном блоке, определяется логика функция

# Имя функции

Требования к именованию функции перечислены в стандарте [PEP 8](#). В целом требования примерно такие же как и для именования переменных, то есть название должно лаконично и кратко отражать, то что, как вы предполагаете, должна делать функция...

Примеры:

**sort**(....) # sorting some data

**search**(...) # search something in something

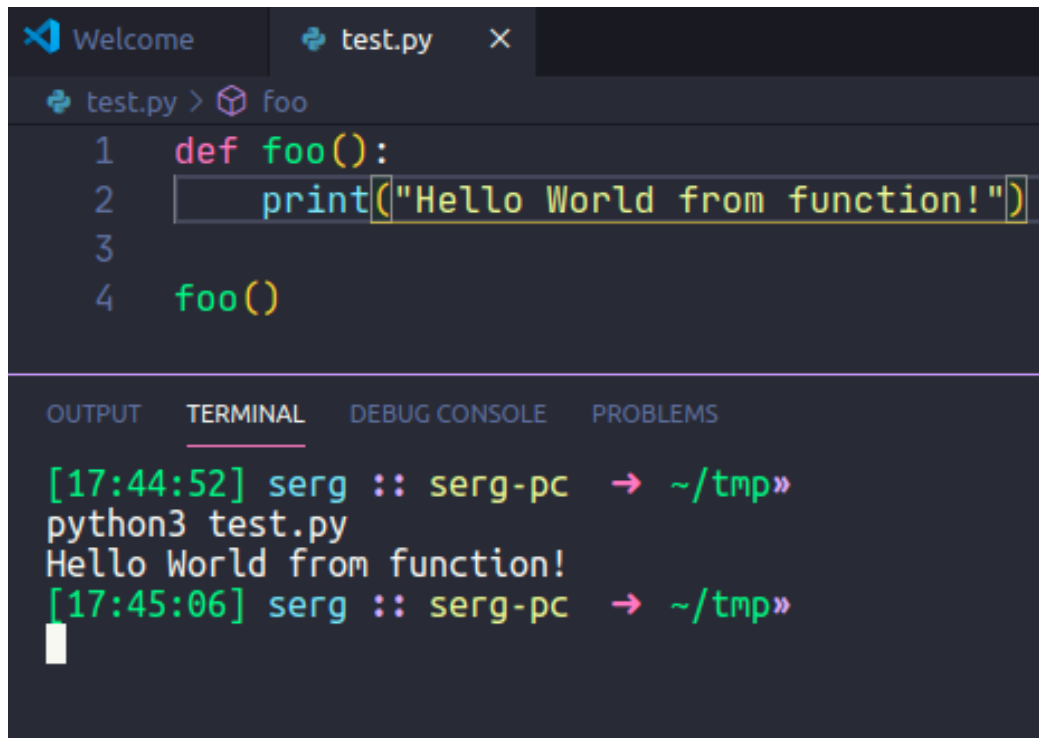
**a**(...) # bad name no clue what this function suppose to do

# Вызов функции

1. Вызов функции происходит через указание имени функции
2. В круглых скобках перечисляются аргументы функции, если таковые есть.
3. Результат работы функции может быть проигнорирован а может быть присвоен для последующего использования

```
def my_function(r, pi):  
    return r**2 * pi  
  
area = my_function(2, 3.1415)  
print(f"Area of circle is {area}")
```

# Определение функции в Python



The image shows a Visual Studio Code editor window with a file named `test.py` open. The editor displays the following Python code:

```
1 def foo():  
2     print("Hello World from function!")  
3  
4 foo()
```

Below the code editor, the **TERMINAL** tab is active, showing the execution of the script:

```
[17:44:52] serg :: serg-pc → ~/tmp»  
python3 test.py  
Hello World from function!  
[17:45:06] serg :: serg-pc → ~/tmp»
```

# Возврат (return)

Остановка  
выполнение  
функции  
ключевое слово  
**return**

```
1  def triple_sumator(a, b, c):  
2      return a + b + c  
3  
4  print(triple_sumator(1, 2, 3))
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS

```
[18:24:49] serg :: serg-pc → ~/tmp»  
python3 test.py  
6  
[18:24:50] serg :: serg-pc → ~/tmp»  
□
```



# Завершение работы функции (return)

**return** - может возвращать несколько значений при этом результат работы функции нужно перечислить через запятую после ключевого слова **return** и результат будет являться значением типа **tuple**

```
def many_result_example():  
    return 1,2,3,4  
  
result = many_result_example()  
print(type(result))  
print(result)
```

```
<class 'tuple'>  
(1, 2, 3, 4)
```

# Распаковка результата

Если функция возвращает несколько значений зачастую удобно распаковывать результаты сразу в переменные.

```
def many_result_example():  
    return 1,2,3,4  
  
val1, val2, val3, val4 = many_result_example()  
print(f"{val1=} {val2=} {val3=} {val4=}")
```

```
val1=1 val2=2 val3=3 val4=4  
[2021-10-04 17:44:55] [WARNING] [app.dtc  
88 142:6379 111 ') 'current attempt'
```

# Аргументы функции

Также можно рассматривать как набор данных над которыми функция выполняет свою работу.

# Аргументы функции(Обязательные)

**Обязательные** аргументы должны всегда передаваться при вызове функции нарушение этого влечет исключение времени выполнения

# Аргументы функции(Обязательные)

```
6 def require_args(arg1, arg2):  
7     print(arg1, arg2)  
8  
9 require_args(1, 2)  
10 require_args(1)  
11
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

[18:31:47] serg :: serg-pc → ~/tmp»

python3 test.py

1 2

Traceback (most recent call last):

File "test.py", line 10, in <module>

require\_args(1)

TypeError: require\_args() missing 1 required positional argument: 'arg2'

# Аргументы функции(Необязательные)

**Необязательные** аргументы или **именованные** аргументы - аргументы которые могут отсутствовать при вызове функции, при этом такие аргументы должны иметь значение по умолчанию, которые будет подставлены во время работы функции.

```
def foo(arg1, arg2, arg3=None, arg4=None):  
    print(f"{arg1=}, {arg2=}, {arg3=}, {arg4=}")
```

```
foo(1,2)  
foo(3,4, "Hello", "World")  
foo(1,2,arg3="Hello", arg4="World")
```

```
→ python3.8 my_first_script.py  
arg1=1, arg2=2, arg3=None, arg4=None  
arg1=3, arg2=4, arg3='Hello', arg4='World'  
arg1=1, arg2=2, arg3='Hello', arg4='World'
```

# Аргументы функции(Необязательные)

```
5
6 def not_require_args(arg1, arg2=2):
7     print(arg1, arg2)
8
9 not_require_args(1, 3)
10 not_require_args(1)
11
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[18:32:55] serg :: serg-pc → ~/tmp»
python3 test.py
1 3
1 2
```

# Вычисление аргументов функции

**ATTENTION!** Язык гарантирует, что аргументы функции будут вычислены до вызова функции то есть, если к примеру один или несколько аргументов являются результатом работы других функций, то сначала будут вызваны эти функции.

```
import random

def foo(arg1, arg2):
    print(f"{arg1=} {arg2=}")
    return arg1 + arg2

def bar():
    return random.randint(1, 100)

print(foo(bar(), bar()))
```

```
arg1=70 arg2=32
102
```



# Неопределенное количество аргументов

Python не предоставляет возможности переопределять функции в зависимости от типа аргументов, как например это позволяет делать C++/C. То есть в пределах одного модуля желательно чтобы имя функции было уникально.

```
def my_f(a,b):  
    print("Hello")  
  
def my_f(b,v):  
    print("World")  
  
my_f(1,2)
```

# Неопределенное количество аргументов

Для того чтобы у вас была возможность определять поведение функции в зависимости от набора аргументов можно воспользоваться конструкцией:

```
def function_name(*args, **kwargs):  
    do_some_work
```

При этом **\*args** - можно интерпретировать как список не именованных аргументов, а **\*\*kwargs** - как словарь именованных аргументов, где ключ это имя аргумента а значение - значение аргумента.

# Неопределенное количество аргументов

```
def many_args_func(*args, **kwargs):  
    for arg in args:  
        print(arg)  
  
    for k, v in kwargs.items():  
        print(k, v)  
  
many_args_func(1,2,3, arg1=3, arg2=4)
```

```
1  
2  
3  
arg1 3  
arg2 4
```

Одним из примеров встроенных функции с динамическим количеством аргументов является функция `print()`

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

# Чистые функции

Функция которая

1. Детерминированная - для одного и того же входа результат работы будет одинаковый всегда
2. Функция не имеет побочных эффектов - то есть не меняет состояние программы в какой бы момент она не была запущена

# Функции первого класса

В Python функции являются функциями (объектами) первого класса.

Язык поддерживает:

1. возможность инициализацию переменных функциями
2. позволяет возвращать функциональный объект как результат работы другой функции,
3. а также позволяет передавать функции как аргументы для другой функции

1

```
def foo():  
    print("Hello World!")  
  
var1 = foo  
  
var1()
```

```
def foo():  
    print("Hello World!")  
  
var1 = foo  
  
var1()
```

2

```
def bar():  
    return foo
```

```
var2 = bar()  
var2()  
bar()()
```

```
Hello World!  
Hello World!
```

3

```
def foobar(in_f):  
    in_f()  
  
foobar(foo)
```

```
Hello World!
```



# Классификация объектов

1. **Мутальные** - или изменяемые, значение этих объектов могут меняться в течении жизни меняться будут именно память. Если вы инициализировали переменную значением такого типа, то это значение можно менять и все переменные ссылающиеся на этот объект будут видеть изменения
2. **Иммутабельные** - неизменяемые эти объекты не могут менять своего значения во время жизни. Переменные инициализированные таким типом данных не смогут менять то значение, которое лежит в памяти.

# Аргументы функции(Передача аргументов)

Class	Description	Immutable?
<b>bool</b>	Boolean value	✓
<b>int</b>	integer (arbitrary magnitude)	✓
<b>float</b>	floating-point number	✓
<b>list</b>	mutable sequence of objects	
<b>tuple</b>	immutable sequence of objects	✓
<b>str</b>	character string	✓
<b>set</b>	unordered set of distinct objects	
<b>frozenset</b>	immutable form of set class	✓
<b>dict</b>	associative mapping (aka dictionary)	

# Примеры

```
In [5]: l = [1,2,3]
```

```
In [6]: l1 = l
```

```
In [7]: l[0] = 10
```

```
In [8]: l1  
Out[8]: [10, 2, 3]
```

```
In [9]: l  
Out[9]: [10, 2, 3]
```

```
In [10]: s = "some string object"
```

```
In [11]: s[0] = "a"
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-11-ff8d72ff8633> in <module>  
----> 1 s[0] = "a"
```

```
TypeError: 'str' object does not support item assignment
```

```
In [12]: t = (1,2,3)
```

```
In [13]: t[0] = 10
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-13-88963aa635fa> in <module>  
----> 1 t[0] = 10
```

```
TypeError: 'tuple' object does not support item assignment
```

```
In [14]: x = 1
```

```
In [15]: id(x)  
Out[15]: 9666944
```

```
In [16]: x += 1
```

```
In [17]: x  
Out[17]: 2
```

```
In [18]: id(x)  
Out[18]: 9666976
```

# Передача аргументов - по значению

Иммутабельные объект/переменные **всегда** передаются по значению это значит ровно то, что в функцию будет передана копия объекта, и при изменение объекта внутри функции оригинал не будет поменян:

```
In [1]: def foo(x):  
...:     x += 100  
...:     print(x)  
...:  
  
In [2]: x = 10  
  
In [3]: foo(x)  
110  
  
In [4]: print(x)  
10
```

# Передача аргументов - по ссылке

Мутабельные объект/переменные **всегда** передаются по ссылке - это значит ровно то, что в функцию будет передана ссылка на оригинал, и при изменение объекта внутри функции оригинал также изменится:

```
In [20]: def foo(l):  
...:     l[0] = 100  
...:  
  
In [21]: l = [1,2,3]  
  
In [22]: l  
Out[22]: [1, 2, 3]  
  
In [23]: foo(l)  
  
In [24]: l  
Out[24]: [100, 2, 3]
```

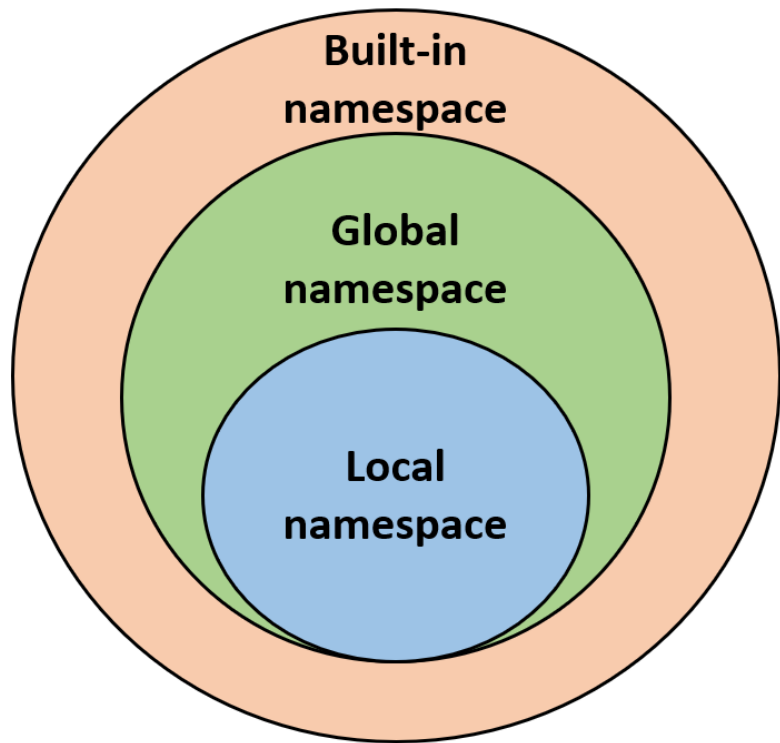
# Что делать если я не хочу менять мутабельный объект в функции?

```
In [26]: from copy import deepcopy  
  
In [27]: l = [1,2,3]  
  
In [28]: l  
Out[28]: [1, 2, 3]  
  
In [29]: foo(deepcopy(l))  
  
In [30]: l  
Out[30]: [1, 2, 3]
```

# Области видимости переменных

В Python, переменные, на которые ссылаются только внутри функции, считаются глобальными. Если переменной присваивается новое значение где-либо в теле функции, считается, что она локальная, и, если вам нужно, то нужно явно указывать её глобальной.

# Области видимости переменных



Type of Namespaces

```
# Global namespace
var1 = 1

def foo():
    # local function namespace
    # you have access to global
    # namespace only for read
    var2 = 2
    def bar():
        # inner local namespace
        # also have access to global
        # and for local namespace of outer
        # function but outer function
        # has not access for variables
        # defined here
        var3 = 3
```



# Ключевое слово global

Для того чтобы поменять глобальные переменные определенный в глобальном неймспейсе нужно использовать ключевое слово **global**

```
def change_global_var():  
    global var1  
    var1 = 200  
  
print(f"Before changing {var1=}")  
change_global_var()  
print(f"After changing {var1=}")
```

```
sergey@CM003 ~/tmp via 🐍 v2.7.18 took 7s  
→ python3 scope.py  
Before changing var1=1  
After changing var1=200
```

# Ключевое слово nonlocal

Если вы предполагаете, что функция определенная внутри другой функции должна менять состояние родительской функции, то нужно пользоваться ключевым словом **nonlocal**

```
def change_local_var():  
    var2 = 1  
    def foo():  
        nonlocal var2  
        var2 = 200  
    print(f"Before changing local variable {var2=}")  
    foo()  
    print(f"After changing local variable {var2=}")  
  
change_local_var()
```

```
Before changing local variable var2=1  
After changing local variable var2=200
```

# Рекурсия

Конструкция функциональных вызовов, которые предполагают функции самой себя. Рекурсия предполагает наличие условия остановки, отсутствие которого приведет к завершению программы с ошибкой. К ошибке скорее всего приведет исчерпание памяти - что является основным недостатком рекурсии...

```
function Fibo(n)  
    if n = 1 or n = 2  
        return 1  
    endif  
    return Fibo(n - 1) + Fibo(n - 2)  
endfunction
```

# Рекурсия (Python)

Глубина рекурсии в Python ограничена системной переменной recursion\_limit - по умолчанию этот параметр ограничен числом 400. НО его можно менять.

```
11
12 def fib(n):
13     if n == 1 or n == 0:
14         return n
15
16     return fib(n - 1) + fib(n - 2)
17
18 N = int(input("Input num: "))
19 print(f"{N} fib number {fib(N)}")
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
[18:52:20] serg :: serg-pc → ~/tmp»
python3 test.py
Input num: 10
10 fib number 55
```

# Рекурсия (Python)

Write code in Python 3.6

(drag lower right corner to resize code editor)

```
1 def fib(n):  
2     if n == 1 or n == 0:  
3         return n  
4  
5     return fib(n - 1) + fib(n - 2)  
6  
7 fib(4)
```

→ line that just executed

→ next line to execute

Frames

Objects

Global frame

fib

function  
fib(n)

fib

n 4

fib

n 2

fib

n 1

Return  
value 1



# Связь рекурсии и цикла

```
def fac_rec(n):  
    if n == 1:  
        return n  
    return n * fac_rec(n - 1)  
  
def fac_loop(n):  
    result = 1  
    while n != 1:  
        result *= n  
        n -= 1  
    return result  
  
print(fac_rec(5))  
print(fac_loop(5))
```

```
→ python3 recursion.py  
120  
120
```

# Анонимные функции

Специальный вид функций предполагающий отсутствие специального идентификатора. В Python реализуются через конструкцию лямбда функций - это однострочные выражения которые могут принимать несколько параметров:

**lambda** arg1, arg2,..., argN: some\_operation

# Анонимные функции

```
In [5]: f = lambda x, y: x + y
```

```
In [6]: f(1,2)
```

```
Out[6]: 3
```



# Анонимные функции

```
In [20]: class A:
...:     def __init__(self, i):
...:         self.i = i
...:

In [21]: l = [A(i) for i in range(10)]

In [22]: for a in l:
...:     print(a.i)
...:

0
1
2
3
4
5
6
7
8
9
```

# Анонимные функции

```
In [23]: random.shuffle(l)
```

```
In [24]: for a in l:  
...:     print(a.i)  
...:
```

8

1

4

5

6

9

0

7

3

2

# Анонимные функции

```
In [26]: sorted_l = sorted(l, key=lambda a: a.i)
```

```
In [27]: for a in sorted_l:  
...:     print(a.i)  
...:
```

0

1

2

3

4

5

6

7

8

9

# Анонимные функции

```
In [39]: l
Out[39]:
['MDZHYEMP11',
 '5VWUWMU9T0',
 '0U2SKU8VX1',
 'F21C72MZ92',
 'FFXKK6MJ0N',
 'RFJPR711CU',
 'A8CWUSASBX',
 'DYA8Q1KGK3',
 'OUIXMRZBLJ',
 'LGRNFS46DY']

In [40]: list(map(lambda s: s.lower(), l))
Out[40]:
['mdzhyemp11',
 '5vwuwmu9t0',
 '0u2sku8vx1',
 'f21c72mz92',
 'ffxkk6mj0n',
 'rfjpr711cu',
 'a8cwusasbx',
 'dya8q1kgk3',
 'ouixmrzblj',
 'lgrnfs46dy']
```