

PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ФУНКЦИИ

ФУНКЦИИ

Функция - это блок кода, выполняющий определенные действия:

- у функции есть имя, с помощью которого можно запускать этот блок кода сколько угодно раз
 - запуск кода функции, называется **вызовом функции**
- при создании функции, как правило, определяются параметры функции.
 - параметры функции определяют какие аргументы функция может принимать
- функциям можно передавать аргументы
 - соответственно, код функции будет выполняться с учетом указанных аргументов

ЗАЧЕМ НУЖНЫ ФУНКЦИИ?

Часто получается, что есть кусок кода, который повторяется. Конечно, его можно копировать из одного скрипта в другой. Но это очень неудобно, так как, при внесении изменений в код, нужно будет обновить его во всех файлах, в которые он скопирован.

Гораздо проще и правильней вынести этот код в функцию (это может быть и несколько функций).

И тогда, в этом файле, или каком-то другом, эта функция просто будет использоваться.

СОЗДАНИЕ ФУНКЦИЙ

СОЗДАНИЕ ФУНКЦИЙ

Создание функции:

- функции создаются с помощью зарезервированного слова `def`
- при создании функции, могут также указываться параметры, которые функция принимает
- первой строкой, опционально, может быть комментарий, так называемая `docstring`
- в функциях может использоваться оператор `return`
 - он используется для прекращения работы функции и выхода из нее
 - чаще всего, оператор `return` возвращает какое-то значение

СОЗДАНИЕ ФУНКЦИЙ

Пример функции:

```
In [1]: def open_file( filename ):  
...:     """Documentation string"""  
...:     with open(filename) as f:  
...:         print(f.read())  
...:
```

Когда функция создана, она ещё ничего не выполняет. Только при вызове функции, действия, которые в ней перечислены, будут выполняться.

ВЫЗОВ ФУНКЦИИ

При вызове функции, нужно указать её имя и передать аргументы, если нужно.

- Параметры - это переменные, которые используются, при создании функции.
- Аргументы - это фактические значения (данные), которые передаются функции, при вызове.

ВЫЗОВ ФУНКЦИИ

```
In [1]: def open_file( filename ):  
...:     """Documentation string"""  
...:     with open(filename) as f:  
...:         print(f.read())  
...:
```

```
In [2]: open_file('r1.txt')  
!  
service timestamps debug datetime msec localtime show-timezone year  
service timestamps log datetime msec localtime show-timezone year  
service password-encryption  
service sequence-numbers  
!  
no ip domain lookup  
!  
ip ssh version 2  
!
```

DOCSTRING

Первая строка в определении функции - это docstring, строка документации. Это комментарий, который используется как описание функции. Его можно отобразить так:

```
In [4]: open_file.__doc__  
Out[4]: 'Documentation string'
```

ОПЕРАТОР RETURN

Оператор `return` используется для прекращения работы функции, выхода из нее, и, как правило, возврата какого-то значения. Функция может возвращать любой объект Python.

ОПЕРАТОР RETURN

Если присвоить вывод функции переменной result, результат будет таким:

```
In [1]: def open_file( filename ):  
...:     """Documentation string"""  
...:     with open(filename) as f:  
...:         print(f.read())  
...:
```

```
In [5]: result = open_file('ospf.txt')  
router ospf 1  
router-id 10.0.0.3  
auto-cost reference-bandwidth 10000  
network 10.0.1.0 0.0.0.255 area 0  
network 10.0.2.0 0.0.0.255 area 2  
network 10.1.1.0 0.0.0.255 area 0
```

```
In [6]: print(result)  
None
```

ОПЕРАТОР RETURN

```
In [7]: def open_file( filename ):
...:     """Documentation string"""
...:     with open(filename) as f:
...:         return f.read()
...:
```

```
In [8]: result = open_file('r1.txt')
```

```
In [9]: print(result)
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

ОПЕРАТОР RETURN

Ещё один важный аспект работы оператора return: выражения, которые идут после return, не выполняются.

То есть, в функции ниже, строка "Done" не будет выводиться, так как она стоит после return:

```
In [10]: def open_file( filename ):
...:     print("Reading file", filename)
...:     with open(filename) as f:
...:         return f.read()
...:         print("Done")
...:
```

```
In [11]: result = open_file('r1.txt')
Reading file r1.txt
```

ПРОСТРАНСТВА ИМЕН. ОБЛАСТИ ВИДИМОСТИ

ПРОСТРАНСТВА ИМЕН. ОБЛАСТИ ВИДИМОСТИ

У переменных в Python есть область видимости. В зависимости от места в коде, где переменная была определена, определяется и область видимости, то есть, где переменная будет доступна.

При использовании имен переменных в программе, Python каждый раз, ищет, создает или изменяет эти имена в соответствующем пространстве имен. Пространство имен, которое доступно в каждый момент, зависит от области в которой находится код.

ПРОСТРАНСТВА ИМЕН. ОБЛАСТИ ВИДИМОСТИ

У Python есть правило LEGB, которым он пользуется при поиске переменных.

Например, если внутри функции, выполняется обращение к имени переменной, Python ищет переменную в таком порядке по областям видимости (до первого совпадения):

- L (local) - в локальной (внутри функции)
- E (enclosing) - в локальной области объемлющих функций (это те функции, внутри которых находится наша функция)
- G (global) - в глобальной (в скрипте)
- B (built-in) - в встроенной (зарезервированные значения Python)

ПРОСТРАНСТВА ИМЕН. ОБЛАСТИ ВИДИМОСТИ

Соответственно есть локальные и глобальные переменные:

- локальные переменные:
 - переменные, которые определены внутри функции
 - эти переменные становятся недоступными после выхода из функции
- глобальные переменные
 - переменные, которые определены вне функции
 - эти переменные 'глобальны' только в пределах модуля
 - например, чтобы они были доступны в другом модуле, их надо импортировать

ПРОСТРАНСТВА ИМЕН. ОБЛАСТИ ВИДИМОСТИ

Пример локальной и глобальной переменной result:

```
In [1]: result = 'test string'

In [2]: def open_file( filename ):
...:     with open(filename) as f:
...:         result = f.read()
...:         return result
...:

In [3]: open_file('r1.txt')
Out[3]: '!\\nservice timestamps debug datetime msec localtime show-timezone year\\nservice timestamps log date

In [4]: result
Out[4]: 'test string'
```

ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

Цель создания функции, как правило, заключается в том, чтобы вынести кусок кода, который выполняет определенную задачу, в отдельный объект. Это позволяет использовать этот кусок кода многократно, не создавая его заново в программе.

Как правило, функция должна выполнять какие-то действия с входящими значениями и на выходе выдавать результат.

При работе с функциями, важно различать:

- **параметры** - это переменные, которые используются, при создании функции.
- **аргументы** - это фактические значения (данные), которые передаются функции, при вызове.

ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

Для того чтобы функция могла принимать входящие значения, ее нужно создать с параметрами:

```
In [1]: def delete_exclamation_from_cfg( in_cfg, out_cfg ):
...:     with open(in_cfg) as in_file:
...:         result = in_file.readlines()
...:     with open(out_cfg, 'w') as out_file:
...:         for line in result:
...:             if not line.startswith('!'):
...:                 out_file.write(line)
...:
```

ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

Файл r1.txt будет использоваться как первый аргумент (in_cfg):

```
In [2]: cat r1.txt
!  
service timestamps debug datetime msec localtime show-timezone year  
service timestamps log datetime msec localtime show-timezone year  
service password-encryption  
service sequence-numbers  
!  
no ip domain lookup  
!  
ip ssh version 2  
!
```

ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

Пример использования функции `delete_exclamation_from_cfg`:

```
In [3]: delete_exclamation_from_cfg('r1.txt', 'result.txt')
```

Файл `result.txt` выглядит так:

```
In [4]: cat result.txt
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
no ip domain lookup
ip ssh version 2
```


ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИЙ

При таком определении функции, надо обязательно передать оба аргумента. Если передать только один аргумент, возникнет ошибка. Аналогично, возникнет ошибка, если передать три и больше аргументов:

```
In [5]: delete_exclamation_from_cfg('r1.txt')
-----
TypeError                                Traceback (most recent call last)
<ipython-input-12-66ae381f1c4f> in <module>()
----> 1 delete_exclamation_from_cfg('r1.txt')

TypeError: delete_exclamation_from_cfg() missing 1 required positional argument: 'out_cfg'
```

ТИПЫ ПАРАМЕТРОВ ФУНКЦИИ

ТИПЫ ПАРАМЕТРОВ ФУНКЦИИ

При создании функции, можно указать, какие аргументы нужно передавать обязательно, а какие нет.

Соответственно, функция может быть создана с параметрами:

- **обязательными**
- **необязательными** (опциональными, параметрами со значением по умолчанию)

ОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Обязательные параметры - определяют какие аргументы нужно передать функции обязательно. При этом, их нужно передать ровно столько, сколько указано параметров функции (нельзя указать большее или меньшее количество аргументов)

Функция с обязательными параметрами:

```
In [1]: def cfg_to_list(cfg_file, delete_exclamation):
.....:     result = []
.....:     with open( cfg_file ) as f:
.....:         for line in f:
.....:             if delete_exclamation and line.startswith('!'):
.....:                 pass
.....:             else:
.....:                 result.append(line.rstrip())
.....:     return result
```

ОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Пример вызова функции:

```
In [2]: cfg_to_list('r1.txt', True)
Out[2]:
['service timestamps debug datetime msec localtime show-timezone year',
'service timestamps log datetime msec localtime show-timezone year',
'service password-encryption',
'service sequence-numbers',
'no ip domain lookup',
'ip ssh version 2']
```

Так как аргументу `delete_exclamation` передано значение `True`, в итоговом словаре нет строк с восклицательными знаками.

ОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ

Вызов функции, со значением False для аргумента delete_exclamation:

```
In [3]: cfg_to_list('r1.txt', False)
Out[3]:
['!',
 'service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 '!',
 'no ip domain lookup',
 '!',
 'ip ssh version 2',
 '!']
```

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ (ПАРАМЕТРЫ СО ЗНАЧЕНИЕМ ПО УМОЛЧАНИЮ)

При создании функции, можно указывать значение по умолчанию для параметра:

```
In [4]: def cfg_to_list(cfg_file, delete_exclamation=True):
.....:     result = []
.....:     with open( cfg_file ) as f:
.....:         for line in f:
.....:             if delete_exclamation and line.startswith('!'):
.....:                 pass
.....:             else:
.....:                 result.append(line.rstrip())
.....:     return result
.....:
```

НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ (ПАРАМЕТРЫ СО ЗНАЧЕНИЕМ ПО УМОЛЧАНИЮ)

Так как теперь у параметра `delete_exclamation` значение по умолчанию равно `True`, соответствующий аргумент можно не указывать при вызове функции, если значение по умолчанию подходит:

```
In [5]: cfg_to_list('r1.txt')
Out[5]:
['service timestamps debug datetime msec localtime show-timezone year',
'service timestamps log datetime msec localtime show-timezone year',
'service password-encryption',
'service sequence-numbers',
'no ip domain lookup',
'ip ssh version 2']
```


НЕОБЯЗАТЕЛЬНЫЕ ПАРАМЕТРЫ (ПАРАМЕТРЫ СО ЗНАЧЕНИЕМ ПО УМОЛЧАНИЮ)

Но, можно и указать, если нужно поменять значение по умолчанию:

```
In [6]: cfg_to_list('r1.txt', False)
Out[6]:
['!',
 'service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 '!',
 'no ip domain lookup',
 '!',
 'ip ssh version 2',
 '!']
```

ТИПЫ АРГУМЕНТОВ ФУНКЦИИ

ТИПЫ АРГУМЕНТОВ ФУНКЦИИ

При вызове функции аргументы можно передавать двумя способами:

- как **ПОЗИЦИОННЫЕ** - передаются в том же порядке, в котором они определены, при создании функции. То есть, порядок передачи аргументов, определяет какое значение получит каждый
- как **КЛЮЧЕВЫЕ** - передаются с указанием имени аргумента и его значения. В таком случае, аргументы могут быть указаны в любом порядке, так как их имя указывается явно.

ТИПЫ АРГУМЕНТОВ ФУНКЦИИ

Позиционные и ключевые аргументы могут быть смешаны, при вызове функции. То есть, можно использовать оба способа, при передаче аргументов одной и той же функции. При этом, сначала должны идти позиционные аргументы, а только потом - ключевые.

ТИПЫ АРГУМЕНТОВ ФУНКЦИИ

```
In [1]: def cfg_to_list(cfg_file, delete_exclamation):
.....:     result = []
.....:     with open( cfg_file ) as f:
.....:         for line in f:
.....:             if delete_exclamation and line.startswith('!'):
.....:                 pass
.....:             else:
.....:                 result.append(line.rstrip())
.....:     return result
.....:
```

ПОЗИЦИОННЫЕ АРГУМЕНТЫ

Позиционные аргументы, при вызове функции, надо передать в правильном порядке (поэтому они и называются позиционные)

```
In [2]: cfg_to_list('r1.txt', False)
Out[2]:
['!',
 'service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 '!',
 'no ip domain lookup',
 '!',
 '',
 '',
 'ip ssh version 2',
 '!']
```

КЛЮЧЕВЫЕ АРГУМЕНТЫ

Ключевые аргументы:

- передаются с указанием имени аргумента
- засчет этого, они могут передаваться в любом порядке

Если передать оба аргумента, как ключевые, можно передавать их в любом порядке:

```
In [4]: cfg_to_list(delete_exclamation=False, cfg_file='r1.txt')
Out[4]:
['!',
 'service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 '!',
 'no ip domain lookup',
 '!',
 'ip ssh version 2',
 '!']
```

КЛЮЧЕВЫЕ АРГУМЕНТЫ

Сначала должны идти позиционные аргументы, а затем ключевые.

Если сделать наоборот, возникнет ошибка:

```
In [5]: cfg_to_list(delete_exclamation=False, 'r1.txt')
File "<ipython-input-3-8f3a3aa16a22>", line 1
    cfg_to_list(delete_exclamation=False, 'r1.txt')
                                   ^
SyntaxError: positional argument follows keyword argument
```


КЛЮЧЕВЫЕ АРГУМЕНТЫ

Но в такой комбинации можно:

```
In [6]: cfg_to_list('r1.txt', delete_exclamation=True)
Out[6]:
['service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 'no ip domain lookup',
 'ip ssh version 2']
```

АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Иногда, необходимо сделать так, чтобы функция принимала не фиксированное количество аргументов, а любое. Для такого случая, в Python можно создавать функцию со специальным параметром, который принимает аргументы переменной длины. Такой параметр может быть, как ключевым, так и позиционным.

ПОЗИЦИОННЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Параметр, который принимает позиционные аргументы переменной длины, создается добавлением перед именем параметра звездочки. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя `*args`

Пример функции:

```
In [1]: def sum_arg(a,*args):  
.....:     print(a, args)  
.....:     return a + sum(args)  
.....:
```

ПОЗИЦИОННЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Вызов функции с разным количеством аргументов:

```
In [2]: sum_arg(1,10,20,30)
```

```
1 (10, 20, 30)
```

```
Out[2]: 61
```

```
In [3]: sum_arg(1,10)
```

```
1 (10,)
```

```
Out[3]: 11
```

```
In [4]: sum_arg(1)
```

```
1 ()
```

```
Out[4]: 1
```

ПОЗИЦИОННЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

```
In [5]: def sum_arg(*args):  
.....:     print(arg)  
.....:     return sum(arg)  
.....:
```

```
In [6]: sum_arg(1, 10, 20, 30)  
(1, 10, 20, 30)  
Out[6]: 61
```

```
In [7]: sum_arg()  
( )  
Out[7]: 0
```

КЛЮЧЕВЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Параметр, который принимает ключевые аргументы переменной длины, создается добавлением перед именем параметра двух звездочек. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя `**kwargs` (от keyword arguments).

```
In [8]: def sum_arg(a,**kwargs):  
.....:     print(a, kwargs)  
.....:     return a + sum(kwargs.values())  
.....:
```

КЛЮЧЕВЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Вызов функцию с разным количеством ключевых аргументов:

```
In [9]: sum_arg(a=10,b=10,c=20,d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[9]: 70
```

```
In [10]: sum_arg(b=10,c=20,d=30,a=10)
10 {'c': 20, 'b': 10, 'd': 30}
Out[10]: 70
```


КЛЮЧЕВЫЕ АРГУМЕНТЫ ПЕРЕМЕННОЙ ДЛИНЫ

Обратите внимание, что, хотя а можно указывать как позиционный аргумент, нельзя указывать позиционный аргумент после ключевого:

```
In [11]: sum_arg(10,b=10,c=20,d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[11]: 70

In [12]: sum_arg(b=10,c=20,d=30,10)
File "<ipython-input-14-71c121dc2cf7>", line 1
    sum_arg(b=10,c=20,d=30,10)
                        ^
SyntaxError: positional argument follows keyword argument
```

РАСПАКОВКА АРГУМЕНТОВ

РАСПАКОВКА АРГУМЕНТОВ

В Python, выражения `*args` и `**kwargs` позволяют выполнять ещё одну задачу - **распаковку аргументов**.

До сих пор, мы вызывали все функции вручную. И, соответственно, передавали все нужные аргументы.

Но, в реальной жизни, как правило, данные необходимо передавать в функцию программно. И часто данные идут в виде какого-то объекта Python.

РАСПАКОВКА ПОЗИЦИОННЫХ АРГУМЕНТОВ

Функция config_interface (файл func_args_var_unpacking.py):

```
def config_interface(intf_name, ip_address, cidr_mask):  
    interface = 'interface {}'  
    no_shut = 'no shutdown'  
    ip_addr = 'ip address {} {}'  
    result = []  
    result.append(interface.format( intf_name ))  
    result.append(no_shut)  
  
    mask_bits = int(cidr_mask.split('/')[ -1])  
    bin_mask = '1'*mask_bits + '0'*(32-mask_bits)  
    dec_mask = [str(int(bin_mask[i:i+8], 2)) for i in range(0,25,8)]  
    dec_mask_str = '.'.join(dec_mask)  
  
    result.append(ip_addr.format( ip_address, dec_mask_str ))  
    return result
```

РАСПАКОВКА ПОЗИЦИОННЫХ АРГУМЕНТОВ

```
In [1]: config_interface('Fa0/1', '10.0.1.1', '/25')
Out[1]: ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.128']

In [2]: config_interface('Fa0/3', '10.0.0.1', '/18')
Out[2]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.192.0']

In [3]: config_interface('Fa0/3', '10.0.0.1', '/32')
Out[3]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']

In [4]: config_interface('Fa0/3', '10.0.0.1', '/30')
Out[4]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.252']

In [5]: config_interface('Fa0/3', '10.0.0.1', '30')
Out[5]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.252']
```

РАСПАКОВКА ПОЗИЦИОННЫХ АРГУМЕНТОВ

Например, список `interfaces_info`, в котором находятся параметры для настройки интерфейсов:

```
In [6]: interfaces_info = [['Fa0/1', '10.0.1.1', '/24'],  
.....:                  ['Fa0/2', '10.0.2.1', '/24'],  
.....:                  ['Fa0/3', '10.0.3.1', '/24'],  
.....:                  ['Fa0/4', '10.0.4.1', '/24'],  
.....:                  ['Lo0', '10.0.0.1', '/32']]
```

РАСПАКОВКА ПОЗИЦИОННЫХ АРГУМЕНТОВ

Если пройтись по списку в цикле и передавать вложенный список, как аргумент функции, возникнет ошибка:

```
In [7]: for info in interfaces_info:
.....:     print(config_interface(info))
.....:
-----
TypeError                                Traceback (most recent call last)
<ipython-input-5-f7d6a9d80d48> in <module>()
      1 for info in interfaces_info:
----> 2     print(config_interface(info))
      3

TypeError: config_interface() missing 2 required positional arguments: 'ip_address' and 'cidr_mask'
```

РАСПАКОВКА ПОЗИЦИОННЫХ АРГУМЕНТОВ

В такой ситуации, пригодится распаковка аргументов.
Достаточно добавить * перед передачей списка, как аргумента, и ошибки уже не будет:

```
In [8]: for info in interfaces_info:
.....:     print(config_interface(*info))
.....:
['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0']
['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0']
['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0']
['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0']
['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']
```

Python сам 'распакует' список info и передаст в функцию элементы списка, как аргументы.

РАСПАКОВКА КЛЮЧЕВЫХ АРГУМЕНТОВ

Аналогичным образом, можно распаковывать словарь, чтобы передать его как ключевые аргументы.

Функция `config_to_list`:

```
def config_to_list(cfg_file, delete_excl=True,
                  delete_empty=True, strip_end=True):
    result = []
    with open( cfg_file ) as f:
        for line in f:
            if strip_end:
                line = line.rstrip()
            if delete_empty and not line:
                pass
            elif delete_excl and line.startswith('!'):
                pass
            else:
                result.append(line)
    return result
```

РАСПАКОВКА КЛЮЧЕВЫХ АРГУМЕНТОВ

Пример использования:

```
In [9]: config_to_list('r1.txt')
Out[9]:
['service timestamps debug datetime msec localtime show-timezone year',
'service timestamps log datetime msec localtime show-timezone year',
'service password-encryption',
'service sequence-numbers',
'no ip domain lookup',
'ip ssh version 2']
```

РАСПАКОВКА КЛЮЧЕВЫХ АРГУМЕНТОВ

Список словарей `cfg`, в которых указано имя файла и все аргументы:

```
In [10]: cfg = [dict(cfg_file='r1.txt', delete_excl=True, delete_empty=True, strip_end=True),
.....:         dict(cfg_file='r2.txt', delete_excl=False, delete_empty=True, strip_end=True),
.....:         dict(cfg_file='r3.txt', delete_excl=True, delete_empty=False, strip_end=True),
.....:         dict(cfg_file='r4.txt', delete_excl=True, delete_empty=True, strip_end=False)]
```

РАСПАКОВКА КЛЮЧЕВЫХ АРГУМЕНТОВ

Если передать словарь функции config_to_list, возникнет ошибка:

```
In [11]: for d in cfg:
.....:     print(config_to_list(d))
.....:

-----
TypeError                                Traceback (most recent call last)
<ipython-input-4-8d1e8defad71> in <module>()
      1 for d in cfg:
----> 2     print(config_to_list(d))
      3

<ipython-input-1-6337ba2bfe7a> in config_to_list(cfg_file, delete_excl, delete_empty, strip_end)
      2             delete_empty=True, strip_end=True):
      3     result = []
----> 4     with open( cfg_file ) as f:
      5         for line in f:
      6             if strip_end:

TypeError: expected str, bytes or os.PathLike object, not dict
```

Ошибка такая, так как все параметры, кроме имени файла, опциональны. И на стадии открытия файла, возникает ошибка, так как вместо файла, передан словарь.

РАСПАКОВКА КЛЮЧЕВЫХ АРГУМЕНТОВ

Если добавить ** перед передачей словаря функции, функция нормально отработает:

```
In [12]: for d in cfg:
...:     print(config_to_list(**d))
...:
['service timestamps debug datetime msec localtime show-timezone year', 'service timestamps log datetime msec
['!', 'service timestamps debug datetime msec localtime show-timezone year', 'service timestamps log datetime
['service timestamps debug datetime msec localtime show-timezone year', 'service timestamps log datetime msec
['service timestamps debug datetime msec localtime show-timezone year\n', 'service timestamps log datetime r
```

Python распаковывает словарь и передает его в функцию как ключевые аргументы.

ПРИМЕР ИСПОЛЬЗОВАНИЯ КЛЮЧЕВЫХ АРГУМЕНТОВ ПЕРЕМЕННОЙ ДЛИНЫ И РАСПАКОВКИ АРГУМЕНТОВ

С помощью аргументов переменной длины и распаковки аргументов, можно передавать аргументы между функциями.

Функция `config_to_list` (файл `kwargs_example.py`):

```
def config_to_list(cfg_file, delete_excl=True,
                  delete_empty=True, strip_end=True):
    result = []
    with open( cfg_file ) as f:
        for line in f:
            if strip_end:
                line = line.rstrip()
            if delete_empty and not line:
                pass
            elif delete_excl and line.startswith('!'):
                pass
            else:
                result.append(line)
    return result
```

Функция берет файл с конфигурацией, убирает часть строк и возвращает остальные строки как список.

Вызов функции в ipython:

```
In [1]: config_to_list('r1.txt')
Out[1]:
['service timestamps debug datetime msec localtime show-timezone year',
'service timestamps log datetime msec localtime show-timezone year',
'service password-encryption',
'service sequence-numbers',
'no ip domain lookup',
'ip ssh version 2']
```

По умолчанию, из конфигурации убираются пустые строки, перевод строки в конце строк и строки, которые начинаются на знак восклицания.

Вызов функции со значением delete_empty=False:

```
In [2]: config_to_list('r1.txt', delete_empty=False)
Out[2]:
['service timestamps debug datetime msec localtime show-timezone year',
 'service timestamps log datetime msec localtime show-timezone year',
 'service password-encryption',
 'service sequence-numbers',
 'no ip domain lookup',
 '',
 '',
 'ip ssh version 2']
```

Задача:

- Создать функцию `clear_cfg_and_write_to_file`, которая с помощью функции `config_to_list`, удаляет лишние строки из конфигурации, а затем записывает строки в указанный файл.
 - при этом, надо не потерять возможность управлять тем, какие строки будут отброшены.
 - то есть, необходимо чтобы функция `clear_cfg_and_write_to_file` поддерживала те же параметры, что и функция `config_to_list`.

Можно просто продублировать все параметры функции и передавать их в функцию `config_to_list`:

```
def clear_cfg_and_write_to_file(cfg, to_file, delete_excl=True,
                                delete_empty=True, strip_end=True):

    cfg_as_list = config_to_list(cfg, delete_excl=delete_excl,
                                  delete_empty=delete_empty, strip_end=strip_end)
    with open(to_file, 'w') as f:
        f.write('\n'.join(cfg_as_list))
```

Но, если воспользоваться возможностью Python принимать аргументы переменной длины, можно сделать функцию `clear_cfg_and_write_to_file` такой:

```
def clear_cfg_and_write_to_file(cfg, to_file, **kwargs):  
    cfg_as_list = config_to_list(cfg, **kwargs)  
    with open(to_file, 'w') as f:  
        f.write('\n'.join(cfg_as_list))
```

ПОЛЕЗНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ

ФУНКЦИЯ PRINT

ФУНКЦИЯ PRINT

Функция print:

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

Функция print выводит все элементы, разделяя их значением `sep`, и завершает вывод значением `end`.

ФУНКЦИЯ PRINT

Все элементы, которые передаются как аргументы, конвертируются в строки:

```
In [4]: def f(a):  
...:     return a  
...:  
  
In [5]: print(1, 2, f, range(10))  
1 2 <function f at 0xb4de926c> range(0, 10)
```


ФУНКЦИЯ PRINT

Для функций `f` и `range` результат равнозначен применению `str()`:

```
In [6]: str(f)
Out[6]: '<function f at 0xb4de926c>'

In [7]: str(range(10))
Out[7]: 'range(0, 10)'
```

SEP

Параметр `sep` контролирует то, какой разделитель будет использоваться между элементами.

По умолчанию используется пробел:

```
In [8]: print(1, 2, 3)  
1 2 3
```

SEP

Но можно изменить значение sep на любую другую строку:

```
In [9]: print(1, 2, 3, sep='|')  
1|2|3
```

```
In [10]: print(1, 2, 3, sep='\n')  
1  
2  
3
```

```
In [11]: print(1, 2, 3, sep='\n'+'-'*10+'\n')  
1  
-----  
2  
-----  
3
```

SEP

В некоторых ситуациях функция print может заменить метод join:

```
In [12]: items = [1,2,3,4,5]
```

```
In [13]: print(*items, sep=', ')
```

```
1, 2, 3, 4, 5
```

END

Параметр `end` контролирует то, какое значение выведется после вывода всех элементов.

По умолчанию используется перевод строки:

```
In [19]: print(1,2,3)  
1 2 3
```

Можно изменить значение `end` на любую другую строку:

```
In [20]: print(1,2,3, end='\n'+ '-'*10)  
1 2 3  
-----
```

FLUSH

По умолчанию при записи в файл или выводе на стандартный поток вывода вывод буферизируется. Функция `print` позволяет отключать буферизацию.

Пример скрипта, который выводит число от 0 до 10 каждую секунду (файл `print_nums.py`):

```
import time

for num in range(10):
    print(num)
    time.sleep(1)
```

Попробуйте запустить скрипт и убедиться, что числа выводятся раз в секунду.

FLUSH

Теперь, аналогичный скрипт, но числа будут выводиться в одной строке (файл `print_nums_online.py`):

```
import time

for num in range(10):
    print(num, end=' ')
    time.sleep(1)
```

FLUSH

Попробуйте запустить функцию. Числа не выводятся по одному в секунду, а выводятся все через 10 секунд.

Это связано с тем, что при выводе на стандартный поток вывода flush выполняется после перевода строки.

Чтобы скрипт отработывал как нужно, необходимо установить flush равным True (файл print_nums_online_fixed.py):

```
import time

for num in range(10):
    print(num, end=' ', flush=True)
    time.sleep(1)
```


ФУНКЦИЯ RANGE

ФУНКЦИЯ RANGE

Функция range возвращает неизменяемую последовательность чисел в виде объекта range.

Синтаксис функции:

```
range(stop)  
range(start, stop[, step])
```

Параметры функции:

- **start** - с какого числа начинается последовательность. По умолчанию - 0
- **stop** - до какого числа продолжается последовательность чисел. Указанное число не включается в диапазон
- **step** - с каким шагом растут числа. По умолчанию 1

ФУНКЦИЯ RANGE

Функция range хранит только информацию о значениях start, stop и step и вычисляет значения по мере необходимости. Это значит, что, независимо от размера диапазона, который описывает функция range, она всегда будет занимать фиксированный объем памяти.

Самый простой вариант range - передать только значение stop:

```
In [1]: range(5)
Out[1]: range(0, 5)

In [2]: list(range(5))
Out[2]: [0, 1, 2, 3, 4]
```

ФУНКЦИЯ RANGE

Если передаются два аргумента, то первый используется как start, а второй - как stop:

```
In [3]: list(range(1, 5))  
Out[3]: [1, 2, 3, 4]
```

И, чтобы указать шаг последовательности, надо передать три аргумента:

```
In [4]: list(range(0, 10, 2))  
Out[4]: [0, 2, 4, 6, 8]  
  
In [5]: list(range(0, 10, 3))  
Out[5]: [0, 3, 6, 9]
```

ФУНКЦИЯ RANGE

С помощью range можно генерировать и убывающие последовательности чисел:

```
In [6]: list(range(10, 0, -1))  
Out[6]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  
  
In [7]: list(range(5, -1, -1))  
Out[7]: [5, 4, 3, 2, 1, 0]
```

ФУНКЦИЯ RANGE

В убывающей последовательности шаг тоже может быть разным:

```
In [8]: list(range(10, 0, -2))  
Out[8]: [10, 8, 6, 4, 2]
```

ФУНКЦИЯ RANGE

Функция поддерживает отрицательные значения start и stop:

```
In [9]: list(range(-10, 0, 1))  
Out[9]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]  
  
In [10]: list(range(0, -10, -1))  
Out[10]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

ФУНКЦИЯ RANGE

Объект range поддерживает все **операции**, которые поддерживают последовательности в Python, кроме сложения и умножения.

Проверка, входит ли число в диапазон, который описывает range:

```
In [11]: nums = range(5)
```

```
In [12]: nums  
Out[12]: range(0, 5)
```

```
In [13]: 3 in nums  
Out[13]: True
```

```
In [14]: 7 in nums  
Out[14]: False
```


ФУНКЦИЯ RANGE

Можно получить конкретный элемент диапазона:

```
In [15]: nums = range(5)
```

```
In [16]: nums[0]
```

```
Out[16]: 0
```

```
In [17]: nums[-1]
```

```
Out[17]: 4
```

ФУНКЦИЯ RANGE

Range поддерживает срезы:

```
In [18]: nums = range(5)
```

```
In [19]: nums[1:]  
Out[19]: range(1, 5)
```

```
In [20]: nums[:3]  
Out[20]: range(0, 3)
```

ФУНКЦИЯ RANGE

Можно получить длину диапазона:

```
In [21]: nums = range(5)
```

```
In [22]: len(nums)
```

```
Out[22]: 5
```

ФУНКЦИЯ RANGE

А также минимальный и максимальный элемент:

```
In [23]: nums = range(5)
```

```
In [24]: min(nums)
```

```
Out[24]: 0
```

```
In [25]: max(nums)
```

```
Out[25]: 4
```

ФУНКЦИЯ RANGE

Кроме того, объект range поддерживает метод index:

```
In [26]: nums = range(1, 7)
```

```
In [27]: nums.index(3)
```

```
Out[27]: 2
```

ФУНКЦИЯ SORTED

ФУНКЦИЯ SORTED

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']
```

```
In [2]: sorted(list_of_words)
```

```
Out[2]: ['', 'dict', 'list', 'one', 'two']
```

```
In [3]: sorted(list_of_words, reverse=True)
```

```
Out[3]: ['two', 'one', 'list', 'dict', '']
```

ФУНКЦИЯ SORTED

```
In [4]: tuple_of_tuples = (('hostname', 'london_r1'),
.....: ('location', '21 New Globe Walk'),
.....: ('vendor', 'Cisco'),
.....: ('model', '4451'),
.....: ('IOS', '15.4'),
.....: ('IP', '10.255.0.1'))
```

```
In [5]: sorted(tuple_of_tuples)
```

```
Out[5]:
```

```
[('IOS', '15.4'),
 ('IP', '10.255.0.1'),
 ('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('model', '4451'),
 ('vendor', 'Cisco')]
```


ФУНКЦИЯ SORTED

```
In [6]: list_of_words
Out[6]: ['one', 'two', 'list', '', 'dict']

In [7]: sorted(list_of_words, key=len)
Out[7]: ['', 'one', 'two', 'list', 'dict']

In [8]: sorted(list_of_words, key=len, reverse=True)
Out[8]: ['list', 'dict', 'one', 'two', '']
```

ФУНКЦИЯ SORTED

```
In [18]: from operator import itemgetter
```

```
In [19]: list_of_tuples = [  
...:     ('IT_VLAN', 320),  
...:     ('Mngmt_VLAN', 99),  
...:     ('User_VLAN', 1010),  
...:     ('DB_VLAN', 11)]
```

```
In [20]: sorted(list_of_tuples, key=itemgetter(1))
```

```
Out[20]: [('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320), ('User_VLAN', 1010)]
```

ФУНКЦИЯ ZIP()

ФУНКЦИЯ ZIP()

Функция zip():

- на вход функции передаются последовательности
- zip() возвращает итератор с кортежами, в котором n-ый кортеж состоит из n-ых элементов последовательностей, которые были переданы как аргументы
 - если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности
- последовательность элементов соблюдается

ФУНКЦИЯ ZIP()

```
In [1]: a = [1,2,3]
```

```
In [2]: b = [100,200,300]
```

```
In [3]: list(zip(a,b))
```

```
Out[3]: [(1, 100), (2, 200), (3, 300)]
```

Использование zip() со списками разной длины:

```
In [4]: a = [1,2,3,4,5]
```

```
In [5]: b = [10,20,30,40,50]
```

```
In [6]: c = [100,200,300]
```

```
In [7]: list(zip(a,b,c))
```

```
Out[7]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

ФУНКЦИЯ ZIP()

```
In [8]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [8]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.1']

In [9]: list(zip(d_keys,d_values))
Out[9]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [10]: dict(zip(d_keys,d_values))
Out[10]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}

In [11]: r1 = dict(zip(d_keys,d_values))
```

ФУНКЦИЯ ZIP()

```
In [10]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
```

```
In [11]: data = {
.....: 'r1': ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.1'],
.....: 'r2': ['london_r2', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.2'],
.....: 'sw1': ['london_sw1', '21 New Globe Walk', 'Cisco', '3850', '3.6.XE', '10.255.0.101']
.....: }
```

```
In [12]: london_co = {}
```

```
In [13]: for key in data.keys():
.....:     london_co[key] = dict(zip(d_keys, data[key]))
.....:
```

```
In [14]: london_co
```

```
Out[14]:
```

```
{'r1': {'IOS': '15.4',
        'IP': '10.255.0.1',
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
        'IP': '10.255.0.2',
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
```

ФУНКЦИЯ ALL()

ФУНКЦИЯ ALL()

Функция `all()` возвращает `True`, если все элементы истина (или объект пустой).

```
In [54]: all([False, True, True])  
Out[54]: False
```

```
In [55]: all([True, True, True])  
Out[55]: True
```

```
In [56]: all([])  
Out[56]: True
```

ФУНКЦИЯ ALL()

```
In [57]: IP = '10.0.1.1'
```

```
In [58]: digits = [ i for i in IP.split('.') if i.isdigit() ]
```

```
In [59]: digits
```

```
Out[59]: ['10', '0', '1', '1']
```

```
In [60]: all( i.isdigit() for i in IP.split('.'))
```

```
Out[60]: True
```

```
In [61]: all( i.isdigit() for i in '10.1.1.a'.split('.'))
```

```
Out[61]: False
```

ФУНКЦИЯ ANY()

ФУНКЦИЯ ANY()

Функция `any()` возвращает `True`, если хотя бы один элемент истина (или объект пустой).

```
In [62]: any([False, True, True])
```

```
Out[62]: True
```

```
In [63]: any([False, False, False])
```

```
Out[63]: False
```

```
In [64]: any([])
```

```
Out[64]: False
```

```
In [65]: any( i.isdigit() for i in '10.1.1.a'.split('.'))
```

```
Out[65]: True
```

АНОНИМНАЯ ФУНКЦИЯ LAMBDA

АНОНИМНАЯ ФУНКЦИЯ LAMBDA

В анонимной функции lambda:

- может содержаться только одно выражение
- аргументов может передаваться сколько угодно

```
In [1]: sum_arg = lambda a, b: a + b
```

```
In [2]: sum_arg(1,2)
```

```
Out[2]: 3
```

```
In [3]: sum_arg(10,22)
```

```
Out[3]: 32
```

АНОНИМНАЯ ФУНКЦИЯ LAMBDA

```
In [1]: sum_arg = lambda a, b: a + b
```

```
In [2]: sum_arg(1,2)
```

```
Out[2]: 3
```

```
In [3]: sum_arg(10,22)
```

```
Out[3]: 32
```

Аналогичная обычная функция:

```
In [4]: def sum_arg(a, b):
```

```
.....:     return a + b
```

```
.....:
```

```
In [5]: sum_arg(1,5)
```

```
Out[5]: 6
```

АНОНИМНАЯ ФУНКЦИЯ LAMBDA

Сортировка элементов с помощью lambda:

```
list_of_lists = [  
    ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0'],  
    ['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0'],  
    ['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0'],  
    ['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0'],  
    ['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']]  
  
In [6]: sorted(list_of_lists)  
Out[6]:  
[['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0'],  
 ['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0'],  
 ['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0'],  
 ['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0'],  
 ['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']]  
  
In [7]: sorted(list_of_lists, key=lambda x: x[2])  
Out[7]:  
[['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255'],  
 ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0'],  
 ['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0'],  
 ['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0'],  
 ['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0']]
```


ФУНКЦИЯ MAP()

ФУНКЦИЯ MAP()

Функция `map()` применяет указанную функцию к каждому элементу последовательности и возвращает список результатов.

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']
```

```
In [2]: map(str.upper, list_of_words)
```

```
Out[2]: <map at 0xb45eb7ec>
```

```
In [3]: list(map(str.upper, list_of_words))
```

```
Out[3]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

ФУНКЦИЯ MAP()

Конвертация в числа:

```
In [3]: list_of_str = ['1', '2', '5', '10']
```

```
In [4]: list(map(int, list_of_str))
```

```
Out[4]: [1, 2, 5, 10]
```

ФУНКЦИЯ MAP()

Вместе с map удобно использовать lambda:

```
In [5]: vlans = [100, 110, 150, 200, 201, 202]
```

```
In [6]: list(map(lambda x: 'vlan {}'.format(x), vlans))
```

```
Out[6]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

ФУНКЦИЯ MAP()

Если функция, которую использует `map()`, ожидает два аргумента, то передаются два списка:

```
In [7]: nums = [1, 2, 3, 4, 5]
```

```
In [8]: nums2 = [100, 200, 300, 400, 500]
```

```
In [9]: list(map(lambda x, y: x*y, nums, nums2))
```

```
Out[9]: [100, 400, 900, 1600, 2500]
```

MAP VS LIST COMPREHENSIONS

Как правило, вместо map можно использовать list comprehension. Чаще всего, вариант с list comprehension более понятный. А в некоторых случаях, даже быстрее.

MAP VS LIST COMPREHENSIONS

Перевести все строки в верхний регистр:

```
In [48]: list_of_words = ['one', 'two', 'list', '', 'dict']
```

```
In [49]: [ str.upper(word) for word in list_of_words ]
```

```
Out[49]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

MAP VS LIST COMPREHENSIONS

Конвертация в числа:

```
In [50]: list_of_str = ['1', '2', '5', '10']
```

```
In [51]: [ int(i) for i in list_of_str ]
```

```
Out[51]: [1, 2, 5, 10]
```


MAP VS LIST COMPREHENSIONS

Форматирование строк:

```
In [52]: vlans = [100, 110, 150, 200, 201, 202]
```

```
In [53]: [ 'vlan {}'.format(x) for x in vlans ]
```

```
Out[53]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

MAP VS LIST COMPREHENSIONS

Для получения пар элементов, используется zip:

```
In [54]: nums = [1, 2, 3, 4, 5]

In [55]: nums2 = [100, 200, 300, 400, 500]

In [56]: [ x*y for x, y in zip(nums,nums2) ]
Out[56]: [100, 400, 900, 1600, 2500]
```

ФУНКЦИЯ FILTER()

ФУНКЦИЯ `FILTER()`

Функция `filter()` применяет функцию ко всем объектам списка, и возвращает те объекты, для которых функция вернула `True`.

Например, вернуть только те строки, в которых находятся числа:

```
In [1]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']
```

```
In [2]: filter(str.isdigit, list_of_strings)
```

```
Out[2]: <filter at 0xb45eb1cc>
```

```
In [3]: list(filter(str.isdigit, list_of_strings))
```

```
Out[3]: ['100', '1', '50']
```

ФУНКЦИЯ FILTER()

Например, из списка чисел оставить только нечетные:

```
In [48]: list(filter(lambda x: x%2, [10, 111, 102, 213, 314, 515]))  
Out[48]: [111, 213, 515]
```

Аналогично, только четные:

```
In [49]: list(filter(lambda x: not x%2, [10, 111, 102, 213, 314, 515]))  
Out[49]: [10, 102, 314]
```

ФУНКЦИЯ FILTER()

```
In [50]: list_of_words = ['one', 'two', 'list', '', 'dict']
```

```
In [51]: list(filter(lambda x: len(x) > 2, list_of_words))
```

```
Out[51]: ['one', 'two', 'list', 'dict']
```

```
In [52]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']
```

```
In [53]: list(filter(lambda x: x.isdigit(), list_of_strings))
```

```
Out[53]: ['100', '1', '50']
```

FILTER VS LIST COMPREHENSIONS

Вернуть только те строки, в которых находятся числа:

```
In [7]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']
```

```
In [8]: [ s for s in list_of_strings if s.isdigit() ]
```

```
Out[8]: ['100', '1', '50']
```

FILTER VS LIST COMPREHENSIONS

Нечетные/четные числа:

```
In [9]: nums = [10, 111, 102, 213, 314, 515]
```

```
In [10]: [ n for n in nums if n % 2 ]
```

```
Out[10]: [111, 213, 515]
```

```
In [11]: [ n for n in nums if not n % 2 ]
```

```
Out[11]: [10, 102, 314]
```


FILTER VS LIST COMPREHENSIONS

Из списка слов оставить только те, у которых количество букв больше двух:

```
In [12]: list_of_words = ['one', 'two', 'list', '', 'dict']  
  
In [13]: [ word for word in list_of_words if len(word) > 2 ]  
Out[13]: ['one', 'two', 'list', 'dict']
```