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?

Как правило, задачи, которые решает код, очень похожи и часто имеют что-то общее.

Например, при работе с конфигурационными файлами, каждый раз надо выполнять такие действия:

- открытие файла
- удаление (или пропуск) строк, которые начинаются на знак восклицания (для Cisco)
- удаление (или пропуск) пустых строк
- удаление символов перевода строки в конце строк
- преобразование полученного результата в список

Дальше действия могут отличаться, в зависимости от того, что нужно делать.

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.

В этом разделе рассматриваеся ситуация когда функция находится в том же файле.

А в разделе Модули будет рассматриваться как повторно использовать объекты, которые находятся в других скриптах.

Создание функций

Создание функции:

- функции создаются с помощью зарезервированного слова **def**
- за def следуют имя функции и круглые скобки
- внутри скобок могут указываться параметры, которые функция принимает
- после круглых скобок идет двоеточие и с новой строки, с отступом, идет блок кода, который выполняет функция
- 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

> Код функций, которые используются в этом разделе, можно скопировать из файла create_func.py

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 вызыве функции, действия, которые в ней перечислены, будут выполняться. Это чем-то похоже на ACL в сетевом оборудовании: при создании ACL в конфигурации, он ничего не делает до тех пор, пока не будет куда-то применен.

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.

Эта функция ожидает имя файла, в качестве аргумента, и затем выводит содержимое файла:

```
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
!

In [3]: 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
```

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'
```

> Лучше не лениться писать краткие комментарии, которые описывают работу функции. Например, описать, что функция ожидает на вход, какого типа должны быть аргументы и что будет на выходе. Кроме того, лучше написать пару предложений о том, что делает функция. Это очень поможет, когда через месяц-два вы будете пытаться понять, что делает функция, которую вы же написали.

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.

Функция open_file, в примере выше, просто выводит на стандартный поток вывода содержимое файла. Но, чаще всего, от функции нужно получить результат её работы.

В данном случае, если присвоить вывод функции переменной result, результат будет таким:

```
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
```

Переменная result равна `None`. Так получилось из-за того, что функция ничего не возвращает. Она просто выводит сообщение на стандартный поток вывода.

Для того, чтобы функция возвращала значение, которое потом можно, например, присвоить переменной, используется 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
!
```

Теперь в переменой result находится содержимое файла.

В реальной жизни, практически всегда, функция будет возвращать какое-то значение. Вместе с тем, можно использовать выражение print, чтобы дополнительно выводить какие-то сообщения.

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

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.

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)

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

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\nservi
ce timestamps log datetime msec localtime show-timezone year\nservice password-encrypt
ion\nservice sequence-numbers\n!\nno ip domain lookup\n!\nip ssh version 2\n!\n'

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

Обратите внимание, что переменная result по-прежнему осталась равной 'test string'. Несмотря на то, что внутри функции ей присвоено содержимое файла.

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.

> Код функций, которые используются в этом разделе, можно скопировать из файла func_params_args.py

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)
   ...:
```

В данном случае, у функции delete_exclamation_from_cfg два параметра: in_cfg и out_cfg.

Функция открывает файл in_cfg, читает содержимое в список; затем открывает файл out_cfg и записывает в него только те строки, которые не начинаются на знак восклицания.

В данном случае функция ничего не возвращает.

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
!
```

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
```

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-7-66ae381f1c4f> in <module>()
----> 1 delete_exclamation_from_cfg('r1.txt')

TypeError: delete_exclamation_from_cfg() takes exactly 2 arguments (1 given)
```

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)

> Код функций, которые используются в этом разделе, можно скопировать из файла func_params_types.py

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
```

Функция cfg_to_list ожидает два аргумента: cfg_file и delete_exclamation.

Внутри, она открывает файл cfg_file для чтения, проходится по всем строкам и, если аргумент delete_exclamation истина и строка начинается с восклицательного знака, строка пропускается. Оператор `pass` означает, что ничего не выполняется.

Во всех остальных случаях, в строке справа удаляются символы перевода строки и строка добавляется в словарь result.

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.

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
   ....:
```

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']
```

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

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.

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.

> Код функций, которые используются в этом разделе, можно скопировать из файла func_args_types.py

Посмотрим на разные способы передачи аргументов, на примере 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',
 '!']
```

Если при вызове функции поменять аргументы местами, скорее всего, возникнет ошибка, в зависимости от конкретной функции.

В случае с функцией cfg_to_list, получится такой результат:

```
In [3]: cfg_to_list(False, 'r1.txt')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-e6da7e2657eb> in <module>()
----> 1 cfg_to_list(False, 'r1.txt')

<ipython-input-15-21a013e5e92c> in cfg_to_list(cfg_file, delete_exclamation)
      1 def cfg_to_list(cfg_file, delete_exclamation):
      2     result = []
----> 3     with open( cfg_file ) as f:
      4         for line in f:
      5             if delete_exclamation and line.startswith('!'):

TypeError: coercing to Unicode: need string or buffer, bool found
```

Ключевые аргументы

Ключевые аргументы:

- передаются с указанием имени аргумента
- за счет 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',
 '!']
```

Но, обратите внимание, что всегда 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-19-5efdee7ce6dd>", line 1
    cfg_to_list(delete_exclamation=False, 'r1.txt')
SyntaxError: non-keyword arg after keyword arg
```

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']
```

> В реальной жизни, зачастую намного понятней и удобней указывать флаги, такие как delete_exclamation, как ключевой аргумент. Если задать хорошее название параметра, за счет указания его имени, сразу будет понятно, что именно делает этот аргумент.
>
> Например, в функции cfg_to_list, понятно, что аргумент delete_exclamation приводит к удалению восклицательных знаков.

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.

> Даже если вы не будете использовать этот прием в своих скриптах, есть большая вероятность, что, вы встретите его в чужом коде.

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)
  ....:
```

Функция sum_arg создана с двумя параметрами:

- параметр `a`
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр `*args` - ожидает аргументы переменной длины
  - сюда попадут все остальные аргументы в виде кортежа
  - эти аргументы могут отсутствовать

Вызов функцию 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
```

Можно создать и такую функцию:

```
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).

Пример функции:

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

Функция sum_arg создана с двумя параметрами:

- параметр `a`
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен

- параметр `**kwargs` - ожидает ключевые аргументы переменной длины
  - сюда попадут все остальные ключевые аргументы в виде словаря
  - эти аргументы могут отсутствовать

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
```

Обратите внимание, что, хотя `a` 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-6-71c121dc2cf7>", line 1
    sum_arg(b=10,c=20,d=30,10)
SyntaxError: non-keyword arg after keyword arg
```

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

Для примера, используем функцию config_interface (файл func_args_var_unpacking.py):

```
def config_interface(intf_name, ip_address, cidr_mask):
    interface = 'interface %s'
    no_shut = 'no shutdown'
    ip_addr = 'ip address %s %s'
    result = []
    result.append(interface % intf_name)
    result.append(no_shut)

    mask_bits = int(cidr_mask.split('/')[-1])
    bin_mask = '1'*mask_bits + '0'*(32-mask_bits)
    dec_mask = '.'.join([ str(int(bin_mask[i:i+8], 2)) for i in [0,8,16,24] ])

    result.append(ip_addr % (ip_address, dec_mask))
    return result
```

Функция ожидает как аргумент:

- intf_name - имя интерфейса
- ip_address - IP-адрес
- cidr_mask - маску в формате CIDR (допускается и формат /24 и просто 24)

На выходе, она выдает список строк, для настройки интерфейса.

Например:

```
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']
```

Допустим, теперь нужно вызвать функцию и передать ей информацию, которая была получена из другого источника, например из БД.

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']]
```

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-32-fb83ecc1fbcf> in <module>()
      1 for info in interfaces_info:
----> 2     print config_interface(info)
      3

TypeError: config_interface() takes exactly 3 arguments (1 given)
```

Ошибка вполне логичная: функция ожидает три аргумента, а ей передан 1 аргумент - список.

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.

> Таким же образом можно распаковывать и кортеж.

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
```

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

> Весь код функции можно вставить в ipython с помощью команды %cpaste.

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']
```

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_en
d=True),
   ....:        dict(cfg_file='r3.txt', delete_excl=True, delete_empty=False, strip_en
d=True),
   ....:        dict(cfg_file='r4.txt', delete_excl=True, delete_empty=True, strip_end=
False)]
```

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-40-1affbd99c2f5> in <module>()
      1 for d in cfg:
----> 2     print config_to_list(d)
      3

<ipython-input-35-6337ba2bfe7a> 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: coercing to Unicode: need string or buffer, dict found
```

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

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 times
tamps log datetime msec localtime show-timezone year', 'service password-encryption',
'service sequence-numbers', 'no ip domain lookup', 'ip ssh version 2']
['!', 'service timestamps debug datetime msec localtime show-timezone year', 'service
timestamps log datetime msec localtime show-timezone year', 'service password-encrypti
on', 'service sequence-numbers', '!', 'no ip domain lookup', '!', 'ip ssh version 2',
'!']
['service timestamps debug datetime msec localtime show-timezone year', 'service times
tamps log datetime msec localtime show-timezone year', 'service password-encryption',
'service sequence-numbers', '', '', '', 'ip ssh version 2', '']
['service timestamps debug datetime msec localtime show-timezone year\n', 'service tim
estamps log datetime msec localtime show-timezone year\n', 'service password-encryptio
n\n', 'service sequence-numbers\n', 'no ip domain lookup\n', 'ip ssh version 2\n']
```

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. Посмотрим на примере.

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
```

> Весь код функции можно вставить в ipython с помощью команды `%cpaste` .

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']
```

Теперь пустые строки появились в списке.

Сделаем 'оберточную' функцию clear_cfg_and_write_to_file, которая берет файл конфигурации, с помощью функции config_to_list, удаляет лишние строки и затем записывает строки в указанный файл.

Но, при этом, мы не хотим терять 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.

Конечно, можно просто продублировать все параметры функции и передать 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))
```

В функции clear_cfg_and_write_to_file явно прописаны её аргументы, а всё остальное попадет в переменную `kwargs`. Затем переменная `kwargs` передается, как аргумент, в функцию config_to_list. Но, так как переменная `kwargs` это словарь, её надо распаковать, при передаче функции config_to_list.

Так функция clear_cfg_and_write_to_file выглядит проще и понятней. И, главное, в таком варианте, в функцию config_to_list можно добавлять аргументы, без необходимости дублировать их в функции clear_cfg_and_write_to_file.

> В этом примере, `**kwargs` используется и для того, чтобы указать, что функция clear_cfg_and_write_to_file может принимать аргументы переменной длины, и для того, чтобы 'распаковать' словарь kwargs, когда мы передаем его в функцию config_to_list.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом zip или tar.gz.

Также для курса подготовлены две виртуальные машины на выбор: Vagrant или VMware. В них установлены все пакеты, которые используются в курсе.

> Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.
>
> Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.
>
> Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 7.1

Создать функцию, которая генерирует конфигурацию для access-портов.

Параметр access ожидает, как аргумент, словарь access-портов, вида:

```
{'FastEthernet0/12':10,
 'FastEthernet0/14':11,
 'FastEthernet0/16':17,
 'FastEthernet0/17':150}
```

Функция должна возвращать список всех портов в режиме access с конфигурацией на основе шаблона access_template.

В конце строк в списке не должно быть символа перевода строки.

Пример итогового списка:

```
[
'interface FastEthernet0/12',
'switchport mode access',
'switchport access vlan 10',
'switchport nonegotiate',
'spanning-tree portfast',
'spanning-tree bpduguard enable',
'interface FastEthernet0/17',
'switchport mode access',
'switchport access vlan 150',
'switchport nonegotiate',
'spanning-tree portfast',
'spanning-tree bpduguard enable',
...]
```

Проверить работу функции на примере словаря access_dict.

```
def generate_access_config(access):
    """
    access - словарь access-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/12':10,
          'FastEthernet0/14':11,
          'FastEthernet0/16':17}

    Возвращает список всех портов в режиме access с конфигурацией на основе шаблона
    """
    access_template = ['switchport mode access',
                       'switchport access vlan',
                       'switchport nonegotiate',
                       'spanning-tree portfast',
                       'spanning-tree bpduguard enable']


access_dict = { 'FastEthernet0/12':10,
                'FastEthernet0/14':11,
                'FastEthernet0/16':17,
                'FastEthernet0/17':150 }
```

Задание 7.1a

Сделать копию скрипта задания 7.1.

Дополнить скрипт:

- ввести дополнительный параметр, который контролирует будет ли настроен port-security
    - имя параметра 'psecurity'

- по умолчанию значение False

Проверить работу функции на примере словаря access_dict, с генерацией конфигурации port-security и без.

```
def generate_access_config(access):
    """
    access - словарь access-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/12':10,
          'FastEthernet0/14':11,
          'FastEthernet0/16':17 }

    psecurity - контролирует нужна ли настройка Port Security. По умолчанию значение F
alse
        - если значение True, то настройка выполняется с добавлением шаблона port_secu
rity
        - если значение False, то настройка не выполняется

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """

    access_template = ['switchport mode access',
                       'switchport access vlan',
                       'switchport nonegotiate',
                       'spanning-tree portfast',
                       'spanning-tree bpduguard enable']

    port_security = ['switchport port-security maximum 2',
                     'switchport port-security violation restrict',
                     'switchport port-security']

access_dict = { 'FastEthernet0/12':10,
                'FastEthernet0/14':11,
                'FastEthernet0/16':17,
                'FastEthernet0/17':150 }
```

Задание 7.1b

Сделать копию скрипта задания 7.1a.

Изменить скрипт таким образом, чтобы функция возвращала не список команд, а словарь:

- ключи: имена интерфейсов, вида 'FastEthernet0/12'
- значения: список команд, который надо выполнить на этом интерфейсе:

```
     ['switchport mode access',
      'switchport access vlan 10',
      'switchport nonegotiate',
      'spanning-tree portfast',
      'spanning-tree bpduguard enable']
```

Проверить работу функции на примере словаря access_dict, с генерацией конфигурации port-security и без.

```
def generate_access_config(access):
    """
    access - словарь access-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/12':10,
          'FastEthernet0/14':11,
          'FastEthernet0/16':17 }

    psecurity - контролирует нужна ли настройка Port Security. По умолчанию значение F
alse
        - если значение True, то настройка выполняется с добавлением шаблона port_secu
rity
        - если значение False, то настройка не выполняется

    Функция возвращает словарь:
    - ключи: имена интерфейсов, вида 'FastEthernet0/1'
    - значения: список команд, который надо выполнить на этом интерфейсе
    """

    access_template = ['switchport mode access',
                       'switchport access vlan',
                       'switchport nonegotiate',
                       'spanning-tree portfast',
                       'spanning-tree bpduguard enable']

    port_security = ['switchport port-security maximum 2',
                     'switchport port-security violation restrict',
                     'switchport port-security']

access_dict = { 'FastEthernet0/12':10,
                'FastEthernet0/14':11,
                'FastEthernet0/16':17,
                'FastEthernet0/17':150 }
```

Задание 7.2

Создать функцию, которая генерирует конфигурацию для trunk-портов.

Параметр trunk - это словарь trunk-портов.

Словарь trunk имеет такой формат (тестовый словарь trunk_dict уже создан):

```
{ 'FastEthernet0/1':[10,20],
  'FastEthernet0/2':[11,30],
  'FastEthernet0/4':[17] }
```

Функция должна возвращать список команд с конфигурацией на основе указанных портов и шаблона trunk_template.

В конце строк в списке не должно быть символа перевода строки.

Проверить работу функции на примере словаря trunk_dict.

```
def generate_trunk_config(trunk):
    """
    trunk - словарь trunk-портов для которых необходимо сгенерировать конфигурацию.

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    trunk_template = ['switchport trunk encapsulation dot1q',
                      'switchport mode trunk',
                      'switchport trunk native vlan 999',
                      'switchport trunk allowed vlan']

trunk_dict = { 'FastEthernet0/1':[10,20,30],
               'FastEthernet0/2':[11,30],
               'FastEthernet0/4':[17] }
```

Задание 7.2a

Сделать копию скрипта задания 7.2

Изменить скрипт таким образом, чтобы функция возвращала не список команд, а словарь:

- ключи: имена интерфейсов, вида 'FastEthernet0/1'
- значения: список команд, который надо выполнить на этом интерфейсе

Проверить работу функции на примере словаря trunk_dict.

```
def generate_trunk_config(trunk):
    """
    trunk - словарь trunk-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/1':[10,20],
          'FastEthernet0/2':[11,30],
          'FastEthernet0/4':[17] }

    Возвращает словарь:
    - ключи: имена интерфейсов, вида 'FastEthernet0/1'
    - значения: список команд, который надо выполнить на этом интерфейсе
    """
    trunk_template = ['switchport trunk encapsulation dot1q',
                      'switchport mode trunk',
                      'switchport trunk native vlan 999',
                      'switchport trunk allowed vlan']

trunk_dict = { 'FastEthernet0/1':[10,20,30],
               'FastEthernet0/2':[11,30],
               'FastEthernet0/4':[17] }
```

Задание 7.3

Создать функцию get_int_vlan_map, которая обрабатывает конфигурационный файл коммутатора и возвращает два объекта:

- словарь портов в режиме access, где ключи номера портов, а значения access VLAN:

```
{'FastEthernet0/12':10,
 'FastEthernet0/14':11,
 'FastEthernet0/16':17}
```

- словарь портов в режиме trunk, где ключи номера портов, а значения список разрешенных VLAN:

```
 {'FastEthernet0/1':[10,20],
  'FastEthernet0/2':[11,30],
  'FastEthernet0/4':[17]}
```

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла config_sw1.txt

Задание 7.3a

Сделать копию скрипта задания 7.3.

Дополнить скрипт:

- добавить поддержку конфигурации, когда настройка access-порта выглядит так:

```
interface FastEthernet0/20
  switchport mode access
  duplex auto
```

То есть, порт находится в VLAN 1

В таком случае, в словарь портов должна добавляться информация, что порт в VLAN 1

Пример словаря:

```
{'FastEthernet0/12':10,
 'FastEthernet0/14':11,
 'FastEthernet0/20':1 }
```

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла config_sw2.txt

Задание 7.4

Создать функцию, которая обрабатывает конфигурационный файл коммутатора и возвращает словарь:

- Все команды верхнего уровня (глобального режима конфигурации), будут ключами.
- Если у команды верхнего уровня есть подкоманды, они должны быть в значении у соответствующего ключа, в виде списка (пробелы вначале можно оставлять).
- Если у команды верхнего уровня нет подкоманд, то значение будет пустым списком

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла config_sw1.txt

При обработке конфигурационного файла, надо игнорировать строки, которые начинаются с '!', а также строки в которых содержатся слова из списка ignore.

Для проверки надо ли игнорировать строку, использовать функцию ignore_command.

```
ignore = ['duplex', 'alias', 'Current configuration']

def ignore_command(command, ignore):
    """
    Функция проверяет содержится ли в команде слово из списка ignore.

    command - строка. Команда, которую надо проверить
    ignore - список. Список слов

    Возвращает True, если в команде содержится слово из списка ignore, False - если не
т
    """
    ignore_command = False

    for word in ignore:
        if word in command:
            return True
    return ignore_command


def config_to_dict(config):
    """
    config - имя конфигурационного файла коммутатора

    Возвращает словарь:
    - Все команды верхнего уровня (глобального режима конфигурации), будут ключами.
    - Если у команды верхнего уровня есть подкоманды,
      они должны быть в значении у соответствующего ключа, в виде списка (пробелы внач
але можно оставлять).
    - Если у команды верхнего уровня нет подкоманд, то значение будет пустым списком
    """
    pass
```

Задание 7.4a

Задача такая же, как и задании 7.4. Проверить работу функции надо на примере файла config_r1.txt

Обратите внимание на конфигурационный файл. В нем есть разделы с большей вложенностью, например, разделы:

- interface Ethernet0/3.100
- router bgp 100

Надо чтобы функция config_to_dict обрабатывала следующий уровень вложенности. При этом, не привязываясь к конкретным разделам. Она должна быть универсальной, и сработать, если это будут другие разделы.

Теперь:

- если уровня 2, то команды верхнего уровня будут ключами словаря, а команды подуровней - списками;
- если уровня 3, то самый вложенный должен быть списком, а остальные - словарями.

На примере interface Ethernet0/3.100

```
{'interface Ethernet0/3.100':{
                     'encapsulation dot1Q 100':[],
                     'xconnect 10.2.2.2 12100 encapsulation mpls':
                         ['backup peer 10.4.4.4 14100',
                          'backup delay 1 1']}}
```

```
ignore = ['duplex', 'alias', 'Current configuration']

def check_ignore(command, ignore):
    """
    Функция проверяет содержится ли в команде слово из списка ignore.

    command - строка. Команда, которую надо проверить
    ignore - список. Список слов

    Возвращает True, если в команде содержится слово из списка ignore, False - если не
т

    """
    ignore_command = False

    for word in ignore:
        if word in command:
            ignore_command = True
    return ignore_command


def config_to_dict(config):
    """
    config - имя конфигурационного файла
    """
    pass
```