PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

МОДУЛИ

МОДУЛИ

Модуль в Python это обычный текстовый файл с кодом Python и расширением **.py**. Они позволяют логически упорядочить и сгруппировать код.

Разделение на модули может быть, например, по такой логике:

- разделение данных, форматирования и логики кода
- группировка функций и других объектов по функционалу

Модули хороши тем, что позволяют повторно использовать уже написанный код и не копировать его (например, не копировать когда-то написанную функцию).

ИМПОРТ МОДУЛЯ

ИМПОРТ МОДУЛЯ

В Python есть несколько способов импорта модуля:

- import module
- import module as
- from module import object
- from module import *

IMPORT MODULE

```
In [1]: dir()
Out[1]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'quit']

In [2]: import os

In [3]: dir()
Out[3]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'os',
 'quit']

In [4]: os.getlogin()
Out[4]: 'natasha'
```

IMPORT MODULE AS

Конструкция **import module as** позволяет импортировать модуль под другим именем (как правило, более коротким):

```
In [1]: import subprocess as sp

In [2]: sp.check_output('ping -c 2 -n  8.8.8.8', shell=True)
Out[2]: 'PING 8.8.8.8 (8.8.8.8): 56 data bytes\n64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=49.880 ms\n64
```

FROM MODULE IMPORT OBJECT

Вариант **from module import object** удобно использовать, когда из всего модуля нужны только одна-две функции:

```
In [1]: from os import getlogin, getcwd
```

Теперь эти функции доступны в текущем именном пространстве:

```
In [2]: dir()
Out[2]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'getcwd',
 'getlogin',
 'quit']
```

FROM MODULE IMPORT OBJECT

Их можно вызывать без имени модуля:

```
In [3]: getlogin()
Out[3]: 'natasha'

In [4]: getcwd()
Out[4]: '/Users/natasha/Desktop/Py_net_eng/code_test'
```

FROM MODULE IMPORT *

Вариант **from module import *** импортирует все имена модуля в текущее именное пространство:

```
In [1]: from os import *

In [2]: dir()
Out[2]:
['EX_CANTCREAT',
 'EX_CONFIG',
 ...
 'wait',
 'wait3',
 'wait4',
 'waitpid',
 'walk',
 'write']
```

Такой вариант импорта лучше не использовать. При таком импорте, по коду не понятно, что какая-то функция взята из модуля os, например. Это заметно усложняет понимание кода.

СОЗДАНИЕ СВОИХ МОДУЛЕЙ

СОЗДАНИЕ СВОИХ МОДУЛЕЙ

Файл sw_int_templates.py:

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

trunk_template = ['switchport trunk encapsulation dot1q',
                  'switchport mode trunk',
                  'switchport trunk allowed vlan']

l3int_template = ['no switchport', 'ip address']
```

Файл sw_data.py:

```
sw1_fast_int = {
                'access':{
                        '0/12':'10',
                        '0/14':'11',
                        '0/16':'17'}}
```

СОЗДАНИЕ СВОИХ МОДУЛЕЙ

Совмещаем всё вместе в файле generate_sw_int_cfg.py:

```
import sw_int_templates
from sw_data import sw1_fast_int


def generate_access_cfg(sw_dict):
    result = []
    for intf in sw_dict['access']:
        result.append('interface FastEthernet' + intf)
        for command in sw_int_templates.access_template:
            if command.endswith('access vlan'):
                result.append(' {} {}'.format(command,
                                              sw_dict['access'][intf]))
            else:
                result.append(' {}'.format(command))
    return result


print('\n'.join(generate_access_cfg(sw1_fast_int)))
```

СОЗДАНИЕ СВОИХ МОДУЛЕЙ

Результат выполнения скрипта:

```
$ python generate_sw_int_cfg.py
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable
```

IF NAME == "MAIN"

Иногда, скрипт, который вы создали, может выполняться и самостоятельно, и может быть импортирован как модуль, другим скриптом.

Файл sw_cfg_templates.py с шаблонами конфигурации:

```
basic_cfg = """
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
!
"""

lines_cfg = """
!
line con 0
 logging synchronous
 history size 100
line vty 0 4
 logging synchronous
 history size 100
 transport input ssh
!
"""
```

IF NAME == "MAIN"

В файле generate_sw_cfg.py импортируются шаблоны из sw_cfg_templates.py и функции из предыдущих файлов:

```
from sw_data import sw1_fast_int
from generate_sw_int_cfg import generate_access_cfg
from sw_cfg_templates import basic_cfg, lines_cfg


print(basic_cfg)
print('\n'.join(generate_access_cfg(sw1_fast_int)))
print(lines_cfg)
```

В результате, должны отобразиться такие части конфигурации, по порядку: шаблон basic_cfg, настройка интерфейсов, шаблон lines_cfg.

IF NAME == "MAIN"

Результат выполнения:

```
$ python generate_sw_cfg.py
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable

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
!

interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
```

IF NAME == "MAIN"

Полученный вывод не совсем правильный: перед строками шаблона basic_cfg, идет лишняя конфигурация интерфейсов.

Так получилось из-за строки print в файле generate_sw_int_cfg.py:

```
print '\n'.join(generate_access_cfg(sw1_fast_int))
```

Когда скрипт импортирует какой-то модуль, всё, что находится в модуле, выполняется.

IF NAME == "MAIN"

Файл generate_sw_int_cfg2.py:

```
import sw_int_templates
from sw_data import sw1_fast_int


def generate_access_cfg(sw_dict):
    result = []
    for intf in sw_dict['access']:
        result.append('interface FastEthernet' + intf)
        for command in sw_int_templates.access_template:
            if command.endswith('access vlan'):
                result.append(' {} {}'.format( command, sw_dict['access'][intf] ))
            else:
                result.append(' {}'.format( command ))
    return result

if __name__ == "__main__":
    print('\n'.join(generate_access_cfg(sw1_fast_int)))
```

IF NAME == "MAIN"

```
if __name__ == "__main__":
    print('\n'.join(generate_access_cfg(sw1_fast_int)))
```

Переменная __name__ это специальная переменная, которая выставляется равной "__main__", если если файл запускается как основная программа. И выставляется равной имени модуля, если модуль импортируется.

Таким образом, условие if __name__ == "__main__" проверяет был ли файл запущен напрямую.

IF NAME == "MAIN"

```
$ python generate_sw_cfg.py

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
!

interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable

!
line con 0
 logging synchronous
```

ПОЛЕЗНЫЕ МОДУЛИ

МОДУЛЬ SUBPROCESS

МОДУЛЬ SUBPROCESS

Модуль subprocess позволяет создавать новые процессы. При этом, он может подключаться к стандартным потокам ввода/вывода/ошибок и получать код возврата.

С помощью subprocess, можно, например, выполнять любые команды Linux из скрипта. И, в зависимости от ситуации, получать вывод или только проверять, что команда выполнилась без ошибок.

ФУНКЦИЯ SUBPROCESS.RUN()

Функция subprocess.run() - основной способ работы с модулем subprocess. Самый простой вариант использования функции, запуск её таким образом:

```
In [1]: import subprocess

In [2]: result = subprocess.run('ls')
ipython_as_mngmt_console.md  README.md         version_control.md
module_search.md             useful_functions
naming_conventions           useful_modules
```

ФУНКЦИЯ SUBPROCESS.RUN()

В переменной result теперь содержится специальный объект CompletedProcess. Из этого объекта можно получить информацию о выполнении процесса, например, о коде возврата:

```
In [3]: result
Out[3]: CompletedProcess(args='ls', returncode=0)

In [4]: result.returncode
Out[4]: 0
```

ФУНКЦИЯ SUBPROCESS.RUN()

Если необходимо вызвать команду с аргументами, её нужно передавать таким образом (как список):

```
In [5]: result = subprocess.run(['ls', '-ls'])
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:28 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

ФУНКЦИЯ SUBPROCESS.RUN()

При попытке выполнить команду с использованием wildcard выражений, например, использовать *, возникнет ошибка:

```
In [6]: result = subprocess.run(['ls', '-ls', '*md'])
ls: cannot access *md: No such file or directory
```

МОДУЛЬ SUBPROCESS

Чтобы вызывать команды, в которых используются wildcard выражения, нужно добавлять аргумент shell и вызывать команду таким образом:

```
In [7]: result = subprocess.run('ls -ls *md', shell=True)
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

МОДУЛЬ SUBPROCESS

Ещё одна особенность функции run() - она ожидает завершения выполнения команды. Если попробовать, например, запустить команду ping, то этот аспект будет заметен:

```
In [8]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=54.4 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 54.498/54.798/55.116/0.252 ms
```

ПОЛУЧЕНИЕ РЕЗУЛЬТАТА ВЫПОЛНЕНИЯ КОМАНДЫ

По умолчанию, функция run возвращает результат выполнения команды на стандартный поток вывода.

Если нужно получить результат выполнения команды, надо добавить аргумент stdout и указать ему значение subprocess.PIPE:

```
In [9]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE)
```

ПОЛУЧЕНИЕ РЕЗУЛЬТАТА ВЫПОЛНЕНИЯ КОМАНДЫ

Теперь можно получить результат выполнения команды таким образом:+

```
In [10]: print(result.stdout)
b'total 28\n4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md\n4 -rw-r--r-- 1 vag
```

ПЕРЕВОД РЕЗУЛЬТАТА В UNICODE

Вариант с decode:

```
In [11]: print(result.stdout.decode('utf-8'))
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:30 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

ПЕРЕВОД РЕЗУЛЬТАТА В UNICODE

Вариант с encoding:

```
In [12]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE, encoding='utf-8')

In [13]: print(result.stdout)
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:31 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

ОТКЛЮЧЕНИЕ ВЫВОДА

Иногда достаточно получения кода возврата и нужно отключить вывод результата выполнения на стандартный поток вывода и при этом сам результат не нужен. Это можно сделать передав функции run аргумент stdout со значением subprocess.DEVNULL

```
In [14]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.DEVNULL)

In [15]: print(result.stdout)
None

In [16]: print(result.returncode)
0
```

РАБОТА СО СТАНДАРТНЫМ ПОТОКОМ ОШИБОК

Если команда была выполнена с ошибкой или не отработала корректно, вывод команды попадет на стандартный поток ошибок. Получить этот вывод можно так же, как и стандартный поток вывода:

```
In [17]: result = subprocess.run(['ping', '-c', '3', '-n', 'a'], stderr=subprocess.PIPE, encoding='utf-8')
```

РАБОТА СО СТАНДАРТНЫМ ПОТОКОМ ОШИБОК

Теперь в result.stdout пустая строка, а в result.stderr находится стандартный поток вывода:

```
In [18]: print(result.stdout)
None

In [19]: print(result.stderr)
ping: unknown host a

In [20]: print(result.returncode)
2
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ МОДУЛЯ

Пример использования модуля subprocess (файл subprocess_basic_run.py):+

```
import subprocess

reply = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])

if reply.returncode == 0:
    print("Alive")
else:
    print("Unreachable")
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ МОДУЛЯ

Результат выполнения будет таким:

```
$ python subprocess_basic_run.py
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=54.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 53.962/54.145/54.461/0.293 ms
Alive
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ МОДУЛЯ

Попробуем собрать всё в финальную функцию, которая будет проверять доступность IP-адреса (файл subprocess_ping_function.py):

```
import subprocess


def ping_ip(ip_address):
    reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           encoding='utf-8')
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stderr

print(ping_ip('8.8.8.8'))
print(ping_ip('a'))
```

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ МОДУЛЯ

Результат выполнения будет таким:

```
$ python subprocess_ping_function.py
(True, 'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=63.8 ms\
(False, 'ping: unknown host a\n')
```

МОДУЛЬ OS

МОДУЛЬ OS

Модуль os позволяет работать с файловой системой, с окружением, управлять процессами.

МОДУЛЬ OS

Модуль os позволяет создавать каталоги:

```
In [1]: import os

In [2]: os.mkdir('test')

In [3]: ls -ls
total 0
0 drwxr-xr-x  2 nata  nata  68 Jan 23 18:58 test/
```

МОДУЛЬ OS

Кроме того, в модуле есть соответствующие проверки на существование. Например, если попробовать повторно создать каталог, возникнет ошибка:

```
In [4]: os.mkdir('test')
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-4-cbf3b897c095> in <module>()
----> 1 os.mkdir('test')

FileExistsError: [Errno 17] File exists: 'test'
```

В таком случае, пригодится проверка os.path.exists:

```
In [5]: os.path.exists('test')
Out[5]: True

In [6]: if not os.path.exists('test'):
   ...:     os.mkdir('test')
   ...:
```

МОДУЛЬ OS

Метод listdir, позволяет посмотреть содержимое каталога:

```
In [7]: os.listdir('.')
Out[7]: ['cover3.png', 'dir2', 'dir3', 'README.txt', 'test']
```

С помощью проверок os.path.isdir и os.path.isfile, можно получить отдельно список файлов и список каталогов:

```
In [8]: dirs = [ d for d in os.listdir('.') if os.path.isdir(d)]

In [9]: dirs
Out[9]: ['dir2', 'dir3', 'test']

In [10]: files = [ f for f in os.listdir('.') if os.path.isfile(f)]

In [11]: files
Out[11]: ['cover3.png', 'README.txt']
```

МОДУЛЬ OS

Также в модуле есть отдельные методы для работы с путями:

```
In [12]: os.path.basename(file)
Out[12]: 'README.md'

In [13]: os.path.dirname(file)
Out[13]: 'Programming/PyNEng/book/16_additional_info'

In [14]: os.path.split(file)
Out[14]: ('Programming/PyNEng/book/16_additional_info', 'README.md')
```

МОДУЛЬ ARGPARSE

МОДУЛЬ ARGPARSE

argparse - это модуль для обработки аргументов командной строки.

Примеры того, что позволяет делать модуль:

- создавать аргументы и опции, с которыми может вызываться скрипт
- указывать типы аргументов, значения по умолчанию
- указывать какие действия соответствуют аргументам
- выполнять вызов функции, при указании аргумента
- отображать сообщения с подсказами по использованию скрипта

МОДУЛЬ ARGPARSE

Пример скрипта ping_function.py:

```
import subprocess
from tempfile import TemporaryFile
import argparse

def ping_ip(ip_address, count):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    with TemporaryFile() as temp:
        try:
            output = subprocess.check_output(['ping', '-c', str(count), '-n', ip_address],
                                             stderr=temp)
            return 0, output.decode('utf-8')
        except subprocess.CalledProcessError as e:
            temp.seek(0)
            return e.returncode, temp.read().decode('utf-8')
```

МОДУЛЬ ARGPARSE

Пример скрипта ping_function.py (продолжение):

```
parser = argparse.ArgumentParser(description='Ping script')

parser.add_argument('-a', action="store", dest="ip")
parser.add_argument('-c', action="store", dest="count", default=2, type=int)

args = parser.parse_args()
print(args)

rc, message = ping_ip( args.ip, args.count )
print(message)
```

МОДУЛЬ ARGPARSE

Создание парсера:

```
parser = argparse.ArgumentParser(description='Ping script')
```

МОДУЛЬ ARGPARSE

Добавление аргументов:

- аргумент, который передается после опции -a, сохранится в переменную ip

```
parser.add_argument('-a', action="store", dest="ip")
```

- аргумент, который передается после опции -c, будет сохранен в переменную count, но, прежде, будет конвертирован в число. Если аргумент не было указан, по умолчанию, будет значение 2

```
parser.add_argument('-c', action="store", dest="count", default=2, type=int)
```

МОДУЛЬ ARGPARSE

Строка args = parser.parse_args() указывается, после того как определены все аргументы.

После её выполнения, в переменной args содержатся все аргументы, которые были переданы скрипту. К ним можно обращаться, используя синтаксис args.ip.

МОДУЛЬ ARGPARSE

Если переданы оба аргумента:

```
$ python ping_function.py -a 8.8.8.8 -c 5
Namespace(count=5, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.673 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.902 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=48.696 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=48 time=50.040 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=48 time=48.831 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.673/49.228/50.040/0.610 ms
```

Namespace - это объект, который возвращает метод parse_args()

МОДУЛЬ ARGPARSE

Передаем только IP-адрес:

```
$ python ping_function.py -a 8.8.8.8
Namespace(count=2, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.563 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.616 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.563/49.090/49.616/0.526 ms
```

МОДУЛЬ ARGPARSE

Вызов скрипта без аргументов:

```
$ python ping_function.py
Namespace(count=2, ip=None)
Traceback (most recent call last):
  File "ping_function.py", line 31, in <module>
    rc, message = ping_ip( args.ip, args.count )
  File "ping_function.py", line 16, in ping_ip
    stderr=temp)
  File "/usr/local/lib/python3.6/subprocess.py", line 336, in check_output
    **kwargs).stdout
  File "/usr/local/lib/python3.6/subprocess.py", line 403, in run
    with Popen(*popenargs, **kwargs) as process:
  File "/usr/local/lib/python3.6/subprocess.py", line 707, in __init__
    restore_signals, start_new_session)
  File "/usr/local/lib/python3.6/subprocess.py", line 1260, in _execute_child
    restore_signals, start_new_session, preexec_fn)
TypeError: expected str, bytes or os.PathLike object, not NoneType
```

МОДУЛЬ ARGPARSE

Если бы функция была вызвана без аргументов, когда не используется argparse, возникла ошибка, что не все аргументы указаны.

Но, из-за argparse, фактически аргумент передается, только он равен None. Это видно в строке Namespace(count=2, ip=None).

МОДУЛЬ ARGPARSE

В таком скрипте, очевидно, IP-адрес необходимо указывать всегда. И в argparse можно указать, что аргумент является обязательным.

Надо изменить опцию -a: добавить в конце required=True:

```
parser.add_argument('-a', action="store", dest="ip", required=True)
```

МОДУЛЬ ARGPARSE

Теперь, если вызвать скрипт без аргументов, вывод будет таким:

```
$ python ping_function.py
usage: ping_function.py [-h] -a IP [-c COUNT]
ping_function.py: error: the following arguments are required: -a
```

Теперь отображается понятное сообщение, что надо указать обязательный аргумент. И подсказка usage.

МОДУЛЬ ARGPARSE

Также, благодаря argparse, доступен help:

```
$ python ping_function.py -h
usage: ping_function.py [-h] -a IP [-c COUNT]

Ping script

optional arguments:
  -h, --help  show this help message and exit
  -a IP
  -c COUNT
```

Обратите внимание, что в сообщении, все опции находятся в секции optional arguments. argparse сам определяет, что указаны опцию, так как они начинаются с - и в имени только одна буква.

МОДУЛЬ ARGPARSE

Зададим IP-адрес, как позиционный аргумент.

Файл ping_function_ver2.py:

```
import subprocess
from tempfile import TemporaryFile

import argparse

def ping_ip(ip_address, count=3):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    with TemporaryFile() as temp:
        try:
            output = subprocess.check_output(['ping', '-c', str(count), '-n', ip_address],
                                             stderr=temp)
            return 0, output.decode('utf-8')
        except subprocess.CalledProcessError as e:
            temp.seek(0)
            return e.returncode, temp.read().decode('utf-8')
```

МОДУЛЬ ARGPARSE

Файл ping_function_ver2.py (продолжение):

```
parser = argparse.ArgumentParser(description='Ping script')

parser.add_argument('host', action="store", help="IP or name to ping")
parser.add_argument('-c', action="store", dest="count", default=2, type=int,
                    help="Number of packets")

args = parser.parse_args()
print(args)

rc, message = ping_ip( args.host, args.count )
print(message)
```

МОДУЛЬ ARGPARSE

Теперь, вместо указания опции -a, можно просто передать IP-адрес. Он будет автоматически сохранен в переменной host. И автоматически считается обязательным.

То есть, теперь не нужно указывать required=True и dest="ip".

МОДУЛЬ ARGPARSE

Теперь вызов скрипта выглядит так:

```
$ python ping_function_ver2.py 8.8.8.8 -c 2
Namespace(host='8.8.8.8', count=2)
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=49.203 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=51.764 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 49.203/50.484/51.764/1.280 ms
```

МОДУЛЬ ARGPARSE

А сообщение help так:

```
$ python ping_function_ver2.py -h
usage: ping_function_ver2.py [-h] [-c COUNT] host

Ping script

positional arguments:
  host        IP or name to ping

optional arguments:
  -h, --help  show this help message and exit
  -c COUNT    Number of packets
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Файл parse_dhcp_snooping.py:

```
# -*- coding: utf-8 -*-
import argparse

DFLT_DB_NAME = 'dhcp_snooping.db'
DFLT_DB_SCHEMA = 'dhcp_snooping_schema.sql'

def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))


def add(args):
    if args.sw_true:
        print("Adding switch data to database")
    else:
        print("Reading info from file(s) \n{}".format(', '.join(args.filename)))
        print("\nAdding data to db {}".format(args.db_file))


def get(args):
    if args.key and args.value:
        print("Geting data from DB: {}".format(args.db_file))
        print("Request data for host(s) with {} {}".format((args.key, args.value)))
    elif args.key or args.value:
        print("Please give two or zero args\n")
        print(show_subparser_help('get'))
    else:
        print("Showing {} content...".format(args.db_file))
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Файл parse_dhcp_snooping.py:

```
parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(title='subcommands',
                                   description='valid subcommands',
                                   help='description')


create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults( func=create )
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Файл parse_dhcp_snooping.py:

```
add_parser = subparsers.add_parser('add', help='add data to db')
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
add_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db name')
add_parser.add_argument('-s', dest='sw_true', action='store_true',
                        help='add switch data if set, else add normal data')
add_parser.set_defaults( func=add )


get_parser = subparsers.add_parser('get', help='get data from db')
get_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db name')
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
get_parser.add_argument('-v', dest="value", help='value of key')
get_parser.add_argument('-a', action='store_true', help='show db content')
get_parser.set_defaults( func=get )

if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Вложенные парсеры будут отображаться как команды. Но, фактически, они будут использоваться как обязательные аргументы.

С помощью вложенных парсеров, создается иерархия аргументов и опций. Аргументы, которые добавлены во вложенный парсер, будут доступны как аргументы этого парсера.

Например, в этой части, создан вложенный парсер create_db и к нему добавлена опция -n:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', dest='name', default=DFLT_DB_NAME,
                           help='db filename')
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Синтаксис создания вложенных парсеров и добавления к ним аргументов, одинаков:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults( func=create )
```

Метод add_argument добавляет аргумент. Тут синтаксис точно такой же, как и без использования вложенных парсеров.

ВЛОЖЕННЫЕ ПАРСЕРЫ

В строке указывается, что, при вызове парсера create_parser, будет вызвана функция create.

```
create_parser.set_defaults( func=create )
```

Функция create получает как аргумент, все аргументы, которые были переданы. И, внутри функции, можно обращаться к нужным:

```
def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Если вызвать help для этого скрипта, вывод будет таким:

```
$ python parse_dhcp_snooping.py -h
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help           show this help message and exit

subcommands:
  valid subcommands

  {create_db,add,get}  description
    create_db          create new db
    add                add data to db
    get                get data from db
```

ВЛОЖЕННЫЕ ПАРСЕРЫ

Обратите внимание, что каждый вложенный парсер, который создан в скрипте, отображается как команда в подсказке usage:

```
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...
```

У каждого вложенного парсера теперь есть свой help:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]

optional arguments:
  -h, --help      show this help message and exit
  -n db-filename  db filename
  -s SCHEMA       db schema filename
```

Кроме вложенных парсеров, в этом примере также есть несколько новых возможностей argparse.

METAVAR

В парсере create_parser используется новый аргумент - metavar:

```
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
```

METAVAR

Аргумент metavar позволяет указывать имя аргумента для вывода в сообщении usage и help:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]

optional arguments:
  -h, --help      show this help message and exit
  -n db-filename  db filename
  -s SCHEMA       db schema filename
```

Посмотрите на разницу между опциями -n и -s:

- после опции -n, и в usage, и в help, указывается имя, которое указано в параметре metavar
- после опции -s указывается имя переменной, в которую сохраняется значение

NARGS

В парсере add_parser используется nargs:

```
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
```

nargs позволяет указать, что в этот аргумент должно попасть определенное количество элементов. В этом случае, все аргументы, которые были переданы скрипту, после имени аргумента filename, попадут в список nargs. Но должен быть передан хотя бы один аргумент.

NARGS

Сообщение help, в таком случае, выглядит так:

```
$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      file(s) to add to db

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  db name
  -s            add switch data if set, else add normal data
```

NARGS

Если передать несколько файлов, они попадут в список. А, так как функция add, просто выводит имена файлов, вывод получится таким:

```
$ python parse_dhcp_snooping.py add filename test1.txt test2.txt
Reading info from file(s)
filename, test1.txt, test2.txt

Adding data to db dhcp_snooping.db
```

NARGS

nargs поддерживает такие значения:

- N - должно быть указанное количество аргументов. Аргументы будут в списке (даже, если указан 1)
- ? - 0 или 1 аргумент
- * - все аргументы попадут в список
- + - все аргумнеты попадут в список, но должен быть передан, хотя бы, один аргумент

CHOICES

В парсере get_parser используется choices:

```
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
```

Для некоторых аргументов, важно, чтобы значение было выбрано только из определенных вариантов. Для таких случаев, можно указывать choices.

CHOICES

Для этого парсера, help выглядит так:

```
$ python parse_dhcp_snooping.py get -h
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]

optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          db name
  -k {mac,ip,vlan,interface,switch}
                        host key (parameter) to search
  -v VALUE              value of key
  -a                    show db content
```

CHOICES

А, если выбрать неправильный вариант:

```
$ python parse_dhcp_snooping.py get -k test
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'test' (choose from 'mac', 'ip', 'vlan', 'i
```

ИМПОРТ ПАРСЕРА

В файле parse_dhcp_snooping.py, последние две строки будут выполняться только в том случае, если скрипт был вызван как основной.

```
if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

А значит, если импортировать файл, эти строки не будут вызваны.

ИМПОРТ ПАРСЕРА

Попробуем импортировать парсер в другой файл (файл call_pds.py):

```
from parse_dhcp_snooping import parser

args = parser.parse_args()
args.func(args)
```

ИМПОРТ ПАРСЕРА

Вызов сообщения help:

```
$ python call_pds.py -h
usage: call_pds.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help           show this help message and exit

subcommands:
  valid subcommands

  {create_db,add,get}  description
    create_db          create new db
    add                add data to db
    get                get data from db
```

ИМПОРТ ПАРСЕРА

Вызов аргумента:

```
$ python call_pds.py add test.txt test2.txt
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

Всё работает без проблем.

ПЕРЕДАЧА АРГУМЕНТОВ ВРУЧНУЮ

И, последняя особенность argparse - возможность передавать аргументы вручную.

Аргументы можно передать как список, при вызове метода parse_args() (файл call_pds2.py):

```
from parse_dhcp_snooping import parser, get

args = parser.parse_args('add test.txt test2.txt'.split())
args.func(args)
```

Необходимо использовать метод split(), так как метод parse_args, ожидает список аргументов.

ПЕРЕДАЧА АРГУМЕНТОВ ВРУЧНУЮ

Результат будет таким, как если бы скрипт был вызван с аргументами:

```
$ python call_pds2.py
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

PYTHON PACKAGE

PYTHON PACKAGE

Пакет Python - это набор модулей, которые организованы по каталогам. Каталоги задают структуру пакета

Пакет Python и обычны набор скриптов Python отличаются тем, что в пакете, в каждом каталоге должен находиться специальный файл - __init__.py

PYTHON PACKAGE

Пример структуры пакета:

```
$ tree my_scripts/
my_scripts/
├── __init__.py
├── configs
│   ├── __init__.py
│   └── cisco.py
├── connect.py
└── parse
    ├── __init__.py
    ├── cisco.py
    └── juniper.py
```

Файлы __init__.py пустые.

PYTHON PACKAGE

Файл connect.py:

```
print('Import connect.py')

def connect_ssh(ip):
    print('Connect SSH to {}'.format(ip))


def connect_telnet(ip):
    print('Connect Telnet to {}'.format(ip))
```

PYTHON PACKAGE

Файл configs/cisco.py:

```
print('Import configs/cisco.py')

basic_cfg = """
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
!
"""

lines_cfg = """
!
line con 0
 logging synchronous
 history size 100
line vty 0 4
 logging synchronous
 history size 100
 transport input ssh
!
"""
```

PYTHON PACKAGE

Файл parse/cisco.py:

```
print('Import parse/cisco.py')

def parse_with_re(command):
    print('Parse command {} with regex'.format(command))


def parse_with_textfsm(command):
    print('Parse command {} with texfsm'.format(command))
```

PYTHON PACKAGE

Файл parse/juniper.py:

```
print('Import parse/juniper.py')

def parse_with_re(command, regex):
    print('Parse command {} with regex {}'.format(command,
                                                  regex))


def parse_with_textfsm(command, template):
    print('Parse command {} with texfsm {}'.format(command,
                                                   template))
```

PYTHON PACKAGE

Импорт модулей/функций из пакета:

```
In [1]: import my_scripts.connect
Import connect.py

In [2]: dir(my_scripts.connect)
Out[2]:
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'connect_ssh',
 'connect_telnet']
```

PYTHON PACKAGE

```
In [3]: my_scripts.connect.connect_ssh('10.1.1.1')
Connect SSH to 10.1.1.1

In [4]: my_scripts.connect.connect_telnet('10.1.1.1')
Connect Telnet to 10.1.1.1
```

PYTHON PACKAGE

```
In [5]: import my_scripts.parse.cisco as parse_cisco
Import parse/cisco.py

In [6]: parse_cisco.parse_with_re
Out[6]: <function my_scripts.parse.cisco.parse_with_re>
```

PYTHON PACKAGE

```
In [7]: import my_scripts.configs.cisco as cfg_cisco
Import configs/cisco.py

In [8]: cfg_cisco.basic_cfg
Out[8]: '\nservice timestamps debug datetime msec localtime show-timezone year\nservice timestamps log datet
```

PYTHON PACKAGE

Можно упростить импорт, настроив __init__.py:

```
from .connect import *
from .parse import cisco
from .parse import juniper as parse_juniper
from .configs.cisco import *
```

PYTHON PACKAGE

Теперь, если выполнить import my_scripts:

```
In [1]: import my_scripts
Import connect.py
Import parse/cisco.py
Import parse/juniper.py
Import configs/cisco.py
```

PYTHON PACKAGE

```
In [4]: dir(my_scripts)
Out[4]:
[
 'basic_cfg',
 'cisco',
 'configs',
 'connect',
 'connect_ssh',
 'connect_telnet',
 'lines_cfg',
 'parse',
 'parse_juniper']
```

PYTHON PACKAGE (ГЛОБАЛЬНО)

Для того чтобы можно было импортировать пакет, его необходимо разместить в одном из каталогов, в котором Python ищет модули или добавить новый путь:

```
In [1]: import sys

In [2]: sys.path
Out[2]:
['',
 '/home/vagrant/venv/py3_convert/bin',
 '/home/vagrant/venv/py3_convert/lib/python36.zip',
 '/home/vagrant/venv/py3_convert/lib/python3.6',
 '/home/vagrant/venv/py3_convert/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6',
 '/home/vagrant/venv/py3_convert/lib/python3.6/site-packages',
 '/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/IPython/extensions',
 '/home/vagrant/.ipython']
```

PYTHON PACKAGE (ГЛОБАЛЬНО)

Для своих пакетов можно использовать каталог:

```
$ python3.6 -m site --user-site
/home/vagrant/.local/lib/python3.6/site-packages
```

После того как каталог будет создан, он автоматически будет добавлен в пути поиска модулей:

```
$ mkdir -p /home/vagrant/.local/lib/python3.6/site-packages
```

PYTHON PACKAGE (ГЛОБАЛЬНО)

```
In [1]: import sys

In [2]: sys.path
Out[2]:
['',
 '/usr/local/bin',
 '/usr/local/lib/python36.zip',
 '/usr/local/lib/python3.6',
 '/usr/local/lib/python3.6/lib-dynload',
 '/home/vagrant/.local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/site-packages/IPython/extensions',
 '/home/vagrant/.ipython']
```

PYTHON PACKAGE (В ВИРТУАЛЬНОМ ОКРУЖЕНИИ)

Для того чтобы можно было импортировать пакет, его необходимо разместить в одном из каталогов, в котором Python ищет модули или добавить новый путь:

```
['',
 '/home/vagrant/venv/py3_convert/bin',
 '/home/vagrant/venv/py3_convert/lib/python36.zip',
 '/home/vagrant/venv/py3_convert/lib/python3.6',
 '/home/vagrant/venv/py3_convert/lib/python3.6/lib-dynload',
 '/usr/local/lib/python3.6',
 '/home/vagrant/venv/py3_convert/lib/python3.6/site-packages',
 '/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/IPython/extensions',
 '/home/vagrant/.ipython']
```

PYTHON PACKAGE (В ВИРТУАЛЬНОМ ОКРУЖЕНИИ)

В виртуальном окружении можно размещать пакеты в пути:

```
/home/vagrant/venv/py3_convert/lib/python3.6/site-packages
```