Конспект курса "Python для сетевых инженеров"

Подготовка к работе

Полезные ссылки

Сайт обучения Книга Python для сетевых инженеров Книга про Git Удобный сайт по модулям питон Официальный сайт Python Книга по Python - "Python trick the book"

VmWare

Для VmWare player нет штатного средства задать статический адрес для виртуальной машины. (В Pro версии есть Vm Network editor)

Правим файлы конфигурации:

C:\ProgramData\VMware\vmnetdhcp.conf Добавляем в конец файла

```
#Python
host VMnet8 {
    hardware ethernet 00:0C:29:D6:D1:F5;
    fixed-address 192.168.8.132;
}
```

Mac-адрес смотреть в виртуалке или в меню **Vmware Workstation / Player, Settings > Network Adapter > Advanced**

Перезапускаем службу и перестартовывает виртуалку

```
net stop vmnetdhcp
net start vmnetdhcp
```

Pip

Pip - это система управления пакетами. В зависимости от того, как установлен и настроен Python в системе, может потребоваться использовать pip3 вместо pip. `pip --version`

Установка модулей

```
pip install tabulate
pip uninstall tabulate
pip3 install --upgrade tabulate

python3.7 -m pip install tabulate
```

Виртуальные окружения

В Python есть несколько вариантов для создания виртуальных окружений. Использовать можно любой из них. Для начала можно использовать virtualenvwrapper

`sudo pip3.7 install virtualenvwrapper`

После установки, в файле `.bashrc`, находящимся в домашней папке текущего пользователя, нужно добавить несколько строк:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.7
export WORKON_HOME=~/venv
. /usr/local/bin/virtualenvwrapper.sh
```

Перезапуск командного интерпретатора: `exec bash`

Работа с виртуальными окружениями

`mkvirtualenv --python=/usr/local/bin/python3.7 pyneng` - создание нового `workon pyneng` - переход в созданное виртуальное окружение `deactivate` - выход из виртуального окружения `rmvirtualenv test` - удаление виртуального окружения `lssitepackages` - какие пакеты установлены в виртуальном окружении

Git

Дополнительные ресурсы, которые позволят глубже познакомиться с Git:

GitHowTo - интерактивный howto на русском Pro Git book - Минимально необходимые знания для работы с Git и GitHub. Книга на русском

Установка Git

Можно установить Git локально и запустить web-интерфейс. Например, GitLab. Вариантов много.

apt install git

Настройка Git

Для начала работы с Git, необходимо указать имя и e-mail пользователя, которые будут использоваться для синхронизации локального репозитория с репозиторием на GitHub:

```
git config --global user.name "username"
git config --global user.email "username.user@example.com"
```

global - используется для всех проетов Посмотреть настройки Git можно таким образом:

```
git config --list
```

Файл исключений (эти файлы не отслеживаются Git)

```
.gitignore
    *.un~ (имя файла)
    .ssh (имя каталога)
```

Настройка отображения кириллических символов

```
git config --global core.quotepath false
```

Подключение к своему репозиторию по SSH

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/github_rsa
```

Проверка:

```
ssh -T git@github.com
Hi username! You've successfully authenticated, but GitHub does not provide shell
access.
```

Команды Git

`git help команда` - help по команде `git init` - инициализация репозитария (создается каталог .git, не забыть создать и перейти в каталог) `git remote -v` - отображение свойств репозитария на удаленном сервере `Ctrl-R` - вход в режим Git, когда он ищет вводимые символы в истории команд `git checkout -- file` - откатить изменения в файле

- клонирование репозитария

`git clone https://github.com/natenka/pyneng-examples-exercises`

- работа `git status` `git pull` - обновление локального репозитория `git add .` - добавляем в stage все файлы (рекурсивно вниз от текущего репозитория) `git add r[1-3].txt` `git commit -a` - add + commit `git reset HEAD имя_файла` - убрать файл из stage area `git rm -cached имя_файла_или_каталога` - убрать из репозитория (файл на диске остается) `git commit -m "комментарий"` - коммитим измененния `git push origin master` - загружаем в репозитарий изменения

- просмотр `git diff` - показывает, какие изменения были внесены с момента последнего коммита `git diff --cached` - diff в staged area `git diff --staged` - отличия между staging и последним коммитом `git log -p origin/master..` – она покажет, какие изменения вы собираетесь добавлять в свой репозиторий на GitHub `git log -p -1` - какие именно изменения были внесены за 1 коммит `git log -p ..origin/master` - какие изменения были выполнены с момента последней синхронизации

Отображение статуса репозитория в приглашении

```
cd ~
git clone https://github.com/magicmonty/bash-git-prompt.git .bash-git-prompt --
depth=1
```

А затем добавить в конец файла .bashrc такие строки:

```
GIT_PROMPT_ONLY_IN_REPO=1
source ~/.bash-git-prompt/gitprompt.sh
```

Github

Чтобы сдать сделанные задания, нужно в проделать ряд манипуляций в интерфейсе GitHub или использовать скрипт см. раздел инструменты на сайте https://pyneng.github.io/

Работа с проверенными заданиями https://pyneng.github.io/docs/checked-tasks-git/

Visual Studio

Установим дополнения (Extensions)

- **Markdown extension pack** - появляется возможность просмотра "на лету"
- **Git history** - просмотр изменений в Git

Подключение к удаленному хосту

Инструкция - Remote Development using SSH Extension

Windows

Начало работы с Python в Windows для начинающих

Python 3.7.6

Виртуальное окружение в Python на Windows

Установка и настройка Git

Инструкции: Установка Git на Windows Downloading Git GitHub Desktop Подключение к Git по SSH в Windows

Настройка Python

```
pip install virtualenvwrapper-win
mkvirtualenv pyneng
workon pyneng
pip install mu-editor
```

Python - интерпретируемый язык (можно выполнять построчно) REPL = интерпритатор **ipyton** - удобный интерпритатор **jupiter** - интерпритатор, работающий в браузере

Подготовка

Модули ставятся через pip

```
pip --version
```

`Python2.7 -m pip install xxx` - установка модуля под версию 2.7

Лучше работать через виртуальные окружения

`pip list` - список пакетов

модули лучше обновлять после тестиорвания в новом вирт.окружунии

Виртуальные окружения

нужно использовать только 1 способ работы с вирт.окр рекомендуется использовать `virtualenvwrapper`

- команды `which python3.7` - путь в системе `mkvirtualenv --python="путь к python"` создание вирт.окружения (рекомендуется в имени указать версию питон) `rmvirtualenv имя` - удалить окружение `workon [tab]` - просмотр всех окружений `workon имя` - переход в окружение `deactivate` - выход
- как удобно проверять переход на новую версию пакета `cpvirtualenv имя` - скопировать окружение `обновить нужные пакеты`
- как удобно проверять переход на новую версию питон `pip freeze` - список модулей в вирт окруж `создать новое окружение создать файл requirements pip install -r ???` - устанавливает все модули,указанные в файле requirements

Python

Синтаксис Python

.py - расширение для скриптов питона **pep8** - документ, описывающий правила написания кода

Отступы - это имеют значения, они отделяют блоки кода Рекомендуется в качестве отступа использовать 4 пробела (нужно настроить tab=4 пробела)

- Комментарии `# ...` - однострочный

```
"""
многострочный
комментарий
"""
```

- ipython - удобный интерпретатор `%` - специфично для ipython `%history` - история `pg-up, pg-down` `ctrl-r` - поиск по истории `ctrl-a, b, u` работают `функция?` - помощь по функции `dir()` - список переменных

Типы данных

Строки

неизменяемый упорядоченный тип данных => можно обращаться по индексу `строка[0]` заключается в одинарные или двойные кавычки рекомендуется выбрать один вариант (или ', или ")

`\n` - перевод строки (зависит от ОС, может быть `\r\n`) `\r` - возврат каретки в начало строки

- Срезы

`строка[1:5:2]` -

- Методы

```
test = "тестовая строка"
test.метод()

ipyton
test.метод? - описание

.count("xx") - посчичать кол-во xx в строке
.find("xxx") - найти позицию строки xxx

.startswith.("Fast") - начинается ли строка с Fast
.endswith.("0/1") - заканчивается ли строка на 0/1

.replace("старое значение", "новое значение")
.strip() - удаление пробельных символов из начала и конца строки
.strip("[]") - удаление символов [ и ] из начала и конца строки

.split() - разделение строки по пробельным символам
.split("xx") - разделение строки по символам xx
```

Списки (List)

список - **изменяемый** упорядоченный тип данных задается в [] через запятую можно обращаться по индексу []

`vlans2 = vlans` - создать ссылку на список (т.к. тип - изменяемый); `id(vlans)` будет равен `id(vlans2)`
`vlans + vlans2` - объединить список

- Срезы

```
vlans = [1, 10, 30, 100, 5, 15]
vlans[0] - элемент 0 = 1
vlans[2:5] - с
vlans[2:5:2] - с шагом 2
```

- Список списков

```
interfaces = [
    ["Fa0/1", "192.168.1.1"],
    ["Fa0/2", "192.168.1.2"],
    ["Fa0/3", "192.168.1.3"],
]

interfaces[0][1]
Fa0/1
```

- Функции `len(список)` - количество элементов `sorted(список)` - упорядочить (всегда возвращает список) `del interfaces[2]` - удалить элемент 2 `id(объект)` - идентификатор
- Методы `.sort()` - отсортировать `";".join(["a", "b", "c"])` - объединить = a;b;c `.copy()` - сделать копию списка `.append(xx)` - добавить один элемент в конец списка `.extend(список)` - добавить несколько элементов в конец списка `.remove("xx")` - удалить первый элемент "xx" из списка `.insert(1, "xx")` - вставить новый элемент "xx" в позицию 1

Словари (Dict)

Словарь (dict) - изменяемый тип данных Упорядоченный тип данных, как записали, так и хранится (начиная с версии 3.7) Все данные хранятся в виде пар ключ:значение

{ключ1: значение, ключ2: значение} `name in london` - есть ли ключ в словаре промежуточные переменные делают ссылку `london2 = london` - создание ссылки на словарь `london['key']` - получить значение по ключу key `london['new_key'] = "dddd"` - добавление новой пары ключ/значение

- Методы словарей (dict) `.keys()` - объект view, список ключей `.values()` - объект view, список значений `.items()` - объект view, список пар ключ/значение объекты view можно преобразовывать в список list() `.copy()` - создать копию `.clear()` - очистить, можно `london = {} .get("key", "что возвращать, если ключа нет")` - получить значение по ключу или None,

если ключа нет `.setdefault(ios, "значение")` - если ключ есть, то возвращает значение, если ключа нет, то создается новый ключ с пустым значением или со значением второго параметра `update({key1:a, key2:b})` - добавить и обновить пары `london_co[sw2] = dict.fromkeys(list(london_co['sw1'].keys()))` - создать новый словарь sw2 с пустыми значениями `dict.fromkeys(sw2, "xxx")` - создать новый словарь sw2 со значениями xxx

Кортежи (Tuple)

Кортежи (Tuple) - неизменяемый тип данных Это список с правами только на чтение Удобно получать из внешних источников, чтобы случайно не изменить

`list(кортеж)` - конвертация в список

- Методы `.count .index`

Множества (Set)

Множества (Set) - изменяемый неупорядоченный тип данных содержит только уникальные значения отображается как {x, y, z} `set(список)` - конвертация списка в множество

- Методы `.add()` - добавить элемент `.discard()` - удалить элемент `.clear()` - очистить операции с множествами: `.intersection() или &` - найти пересечение множеств `.union() или |` - найти объединение множеств `.difference()` - найти отличие множеств `.symmetric_difference` - найти уникальные значения из множеств `issubset() issuperset()`

Булевы значения

- True число, строка или непустой объект
- False None, 0, пустой объект - это False

`bool(объект)` - возвращает True/False

Преобразование типов

'type(объект)' - выводит тип объекта 'type(vlans) = set' - проверка типа (True/False)

СкриптЫ

Запуск

Первая строка скрипта должна быть

```
#!/usr/bin/env python3
```

chmod +x имя_скрипта.py

Аргументы

`from sys import argv` - импортируем модуль `argv` - список (List), содержащий аргумены, переданные скрипту `argv[0]` - имя скрипта `argv[1..n]` - параметры

Условия

```
if/elif/else
```

- Синтаксис

```
if a == b:
    pass
elif a > b:
    pass
else:
    pass
```

Pytest

Pytest - средство для автоматизированного тестирования кода

`pytest test\exercise_n_1.py` - вызов теста exercise_n_1.py запускать из каталога n_exercise

Настройка pytest под Windows 10

```
pip install pytest
pip install pyyaml
```

Регулярные выражения

шпаргалка по регулярным выражениям - pythex.org шпаргалка по регулярным выражениям - pyregex.org

`import re` - импортируем станд модуль Python `m = re.search(regex, line)` - поиск первого совпадения `m == None`, если нет совпадения

`m.group()` - возвращает строку, совпавшую с regex `m.group(1)` - возвращает первую группу `m.group(3, 1, 2)` - возвращает строку с определенной последовательностью групп `m.groups()` - возвращает кортеж из всех групп

Нужно использовать raw строки `r"регулярное выражение \w \w"` - не требует экранирования каждого спецсимвола

Спецсимволы

`\d` - все цифры `\D` - все символы, кроме цифр `\w` - все цифры и буквы `\W` - все символы, кроме цифр и букв `\s` - все пробельные символы `\S` - все символы, кроме пробельных

`regex+` - 1 и более совпадений `regex*` - 0 или более совпадений `regex?` - 0 или 1 повторение предыдущего символа `regex{n}` - повторение совпадения n раз

`\.` - любой символ (кроме перевода строки) `^` - начало строки `$` - конец строки `[abc]` - любоЙ симол из указанных в скобках `[^abc]` - любоЙ симол, кроме указанных в скобках `[^abc]` - любоЙ симол, кроме указанных в скобках `a|b` - выражение a или выражение b `0/0|0/1` - 0/0 или 0/1 `()` - группировка символов

Жадность спецсимволов повторения

По умолчанию спецсимволы повторения `+`, `*` захватывают совпадение максимальной длины `+?`, `*?` захватывают совпадение минимальной длины

Группы

Все, что в скобках попадает в группы: `m = re.search("...regex...(xxx)...regex...", line)` Запоминается последнее совпадение с группой `m.group(0)` - все совпадение `m.group(1)` - первая группа `m.group(n)` - n-ная группа

Именованные группы `(?P<name>regex)` - Именованная группа name `m.group(name)` - группа name `m.groupdict()` - словарь именованных групп

`(?:regex)` - Группа без захвата.

Функции модуля re

`.start()` - индекс начала совпадения `.end()` - индекс конца совпадения `.span()` - возвращает кортеж (начало, конец)

- .search(regex, line)

`m = re.search('regex', line)` - поиск рег.выражения regex в строке line. Обрабатывает построчно, возвращает первое совпадение `m.lastgroup` - переменная, содержит имя последней найденной группы `m.lastindex` - переменная, содержит номер последней найденной группы `m = re.search(r'(/d+) /1)', line)` - поиск группы цифр, а потом еще раз такое же число

- .findall(regex, line)

Ищутся все повторения если нет групп - возвращает список строк ? если 1 группа - возвращает список кортежей если несколько групп - возвращает список кортежей

- .finditer(regex, line)

это итератор, по отдает по одному совпадению в цикле. Поддерживает все функции .search (.group() и т.д.)

```
result = re.finditer(regex, line)
for match in result:
    pass
```

- .compile()

функция compile заранее компилирует регулярное выражение для дальнейшего использования в коде. Плюс есть еще несколько функций:

- .split() - разбиение на подстроки
- .sub() - замена

`m = re.sub('regex', line)` - поиск замена по регулярному выражению

Флаги

описание флагов

флаги изменяют поведение регулярных выражений несколько флагов передают через | "re.DOTALL | re.IGNORECASE"

`re.DOTALL` - режим мультилайн (`.+` и `.*` работают до конца файла, а не до конца строки) `re.ASCII (re.A)` - режим `re.IGNORECASE (re.I)` - режим `re.MULTILINE (re.M)` - многострочный режим `re.DOTALL (re.S)` - режим `re.VERBOSE (re.X)` - режим `re.LOCALE (re.L)` - режим `re.DEBUG` - режим

Кодировка в Python

Unicode

Unicode - каждому символу соответстует свой номер ASCII - 1 байт, каждому символу соответствует свой код UTF-8 - может занимать от 1 до 3 байт, описывает как символ Unicode кодируется в байты. В основном будет использоваться при работе с оборудованием

`ord(символ)` - возвращает Unicode-номер символа `chr(десятичный_код)` - возвращает Unicode символ по его номеру

Байтовая строка

b"строка" = байтовая строка если в строке только ASCII символы, то вывод будет содержать нормальные символы, если в строке, например, кириллица, то в выводе будут коды \xd2\xb0.

print(b"строка") - переводы строк не интерпретируются, а отображаются \n

Функции конвертации `строка.encode()` - из str получаем byte `строка.encode("UTF-8")` - из str получаем byte `строка.encode("UTF-8", 'replace')` - из str получаем byte. Нераспознанные символы заменяются ? `байтовая_строка.decode()` - из byte получаем str `байтовая_строка.decode("UTF-8")` - из byte получаем str `байтовая_строка.decode("UTF-8", 'replace')` - из byte получаем str. Нераспознанные символы заменяются ?

В Python кодировка по умолчанию UTF-8:

```
import sys
sys.getdefaultencoding()
```

Правило "сэндвича" - на входе и выходе преобразуем в байты (байтовые строки), внутри программа используем Unicode (обычные строки)

Есть два варианта работы:

- в функции указываем, в какой кодировке хотим получать вывод
- получам byte и используем .decode()

Подключение к оборудованию

Хранение паролей

- input
- getpass
- переменные окружения

Модули для подключения к оборудованию

в курсе рассматривается автоматизация при подключении черер SSH, telnet если нужно управление через web - можно использовать selenium

Pexpect - модуль для интерактивной работы (ожидание вывода)

документация pexpect

`pip install pexpect` - установка в pexpect можно использовать регулярные выражения
`r1.expect(r"\S+#")` - ожидаем любое количество непробельных символов и затем #
`r1.match.group()` = `\bR1#`

если искомая строка в `.expect()` не найдена, будет выведено исключение если исключение `EOF` - значит, что сессия закрыта

```
import pexpect
with pexpect.spawn("ssh cisco@192.168.100.1") as r1  # открыть сессию
    r1.expect("Password")  # ждать в ответе Password. Если код возврата 0, то
вхождение найдено
    r1.sendline("cisco")  # 6 - отправлено 6 байт
    r1.expect(">")  # ждем символ >
    r1.sendline("enable")  # 7 - отправлено 7 байт
    r1.expect("Pass")  # ждать в ответе Pass
    r1.sendline("cisco")
    r1.expect("#")
    r1.sendline("terminal length 0")
    r1.expect("\S+[>#]")
    r1.sendline("sh ip int br")
    r1.expect("[>#]")  # регулярка, ждем в выводе > или #
    result = r1.before  # вывод строки в байтах, которое вывело оборудование до
совпадения expect
    result = result.decode("UTF-8")  # если оборудование поддерживает кириллицу
    # r1.close  # без использования with (закрыть сессию)
```

`.expect()` - читает из буфера, куда попадает весь вывод с оборудования. Поэтому после каждой команды нужно давать команду `r1.expect(">#]") returncode = .expect(["line1", "line2", pexpect.TIMEOUT, pexpect.EOF])` - можно передавать несколько строк `returncode == 0` - если найден line1 `returncode == 1` - если найден line2 и т.д.

```
лучше сделать в цикле:
.expect("что ждем")
.sendline("command")
result = .before
result = result.decode("UTF-8")  # сразу превращать в строку
```

обработка постраничного режима

лучше всегда вначале скрипта отключить paging. Если такого режима у оборудования нет, то можно читать постранично `^H = \x08` - особенности cisco. вокруг --More-- есть символы backspace, которые мешают. Их можно заменять.

если искомая строка в `.expect()` не найдена, будет выведено исключение `r1.sendline("terminal length 0")` - отключение paging на cisco для текущей сессии

Telnetlib

встроенный модуль python для telnet нужно передавать байтовую строку нужно передавать перевод строки

```
import telnetlib
r1 = telntlib.Telnet("192.168.100.1")
r1.readuntil(b"Username")
r1.write(b"cisco\n")
r1.write(f"{username}\n".encode("utf-8"))
r1.readuntil(b"Password")
r1.write(b"cisco\n")
r1.readuntil(b"Password")
r1.write(b"cisco\n")
r1.readuntil(b">")
r1.write(b"cisco\n")
r1.readuntil(b"#")
r1.write(b"sh ip int br\n")
r1.readuntil(b"#", timeout=2)
```

методы

`.read_very_eager()` - считывает весь вывод из буфера. обязательно использовать паузу

```
import time
r1.write(...)
```

```
r1.write(...)
time.sleep(3)
```

`.read_until("#")` - считывает из буфера до строки. `index, match, page = r1.expect(b"--More--", b"#")` - может регулярки, возвращает коржеж из 3 элементов. Далее if index == 0: и т.п.

Paramiko

```
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(hostname="192.168.100.1", uesrname="cisco", password="cisco",
look_for_keys=False)
ssh = client.invoke_shell()
ssh.send('enable\n')
```

`ssh.receive(3000)` - сколько байт получить из вывода. Если вывода нет, то по таймауту выйдет ошибка

```
import time  # для паузы
import socket  # для паузы
with client.invoke_shell() as ssh
    ...
    ssh.send()
    time.sleep(1000)  # 1 сек
    ssh.settimeout()  # сколько .resv() ждет вывода информации
    ssh.receive()
```

- Считываем большой вывод частями

```
whyle True:
    try:
        part = ssh.recv(50).decode("UTF-8")
        output += part
    except socket.timeout:  # если все считали, то выходим из цикла по таймауту
        break
```

- Считываем большой вывод до приглашения

```
PROMPT = "#"
whyle True:
    try:
        part = ssh.recv(50).decode("UTF-8")
        output += part
        if PROMPT in part:  # если в куске, который мы считали, есть PROMPT, то
выходим из цикла
```

```
            break
    except socket.timeout:  # если все считали, то выходим из цикла по таймауту
        break
```

Netmiko

pip install netmiko

TextFSM

Для обработки текста со сложной логикой Шаблоны обработки текста всегда во внешнем файле В TextFSM Передается строка (можно многострочную) TextFSM обрабатывает ТОЛЬКО построчно и возвращает список списков.

Порядок работы:

```
import textFSM
fsm = textfsm.TextFSM('шаблон')
result = fsm.ParseText('строка где искать')
print(result)
```

Шаблон TextFSM

```
Value VAR1 (regex1)   # Сначала опишем переменные (именованные группы)
Value VAR2 (regex2)

Start   # Начало
  '^... {VAR1} ... {VAR2} -> Record'   # Опишем строку для поиска, начинается с '
^'; Результаты - в Record
```

`fsm.header` - список заголовков (переменных)

Логика работы

1. проверка осуществляется построчно
2. строка обрабатывается до первого совпадения правила. Другие правила не проверяются (по умолчанию, можно поменять поведение)
3. `Record` записывает переменные в список и обнуляет переменную (по умолчанию, можно поменять поведение)
4. `Record` не записывает в список, если нет ни одной переменной
5. Для записи переменных из нескольких строк, можно либо в `Record` записывать признак конца блока или писать его в последнюю переменную
6. В конце шаблона после EOF всегда есть неявное правило `.* -> Record`,

Value

7. `Value Filldown[Fillup] VAR (regex)` - не обнулять переменную после `Record`
8. `Value Requared VAR (regex)` - записывать Record, только если есть переменная VAR
9. `Value List VAR (regex)` - записывать список переменных VAR
10. `-> Next.Record` - по умолчанию. записать и обрабатывать след.строку
11. `-> Continue.Record` - записать и обрабатывать эту же строку

Обработка состояний

В шаблоне:

```
Start
  regex1 -> State1 # если нашли regex1, то идем к состоянию State1

State1
  ...{VAR1}...
  ...
  regex -> Start # если нашли regex, то возврат к состоянию Start
```

Clitable

для автоматического сопоставления шаблона команде

Версия textfsm >= 1.1.0

```
pip show textfsm # актуальная версия > 1.1.0
pip install textfsm
```

```
from textfsm import clitable
cli = clitable.CliTable("index", "path_to_templates")
cli.ParseCmd(output, {"Command": "sh ipint br"})
list(cli.header)
[list(row) for row in cli]

print(cli)
print(cli.FormattedTable())
```