

Contents

1	Python Intro Lab	2
2	Data Structures Lab	2
2.1	First, we'll clone a Github repo with code samples	2
2.2	Now we'll run Python interactively and play with lists and dictionaries within Python	2
2.3	Now we'll run a Python script to open a YAML file and export it as JSON	3
3	eAPI Command Explorer	4
3.1	Try out a Show Command	4
3.2	Try out a Configuration Command	4
4	pyeapi	4
4.1	Use provided script to add a local user to a virtual Arista device	4
5	Git Operations	6
5.1	Git preferences setup	6
5.2	Create your own repository	6
5.3	Branch and merge (Optional)	6
5.4	View code diffs in Github	6
6	Ansible Ad-Hoc Lab	7
6.1	Run a simple Ansible ad-hoc command against localhost	7
6.2	Run an Ansible ad hoc command to install a package	7
6.3	Run an Ansible ad hoc command against a device in inventory	7
7	Ansible Playbooks Lab	8
7.1	Ansible jq Installation Lab	8
7.2	Ansible REST API Interaction	8
8	Jinja	9
9	Markdown Tables	10

1 Python Intro Lab

In this lab, we'll see which Python version are installed where

- Open a terminal on your jumpbox and issue the following commands

```
$ which python
$ python --version
$ which python3
$ python3 --version
```

2 Data Structures Lab

2.1 First, we'll clone a Github repo with code samples

- On your jumpbox, hit **<http://bit.ly/cbautomation>** to head to the Github repo
- Select the green Clone or Download button
- Click the clipboard icon to copy the repo location text
- Open a terminal on your jumpbox

```
$ git clone https://github.com/craigbruenderman/automation-class.git
```

2.2 Now we'll run Python interactively and play with lists and dictionaries within Python

- Open up lab2.2.py in Atom to observe the code
- Issue the following commands in a terminal

```
$ cd ~/automation-class/code/
$ python -i lab2.2.py
```

```
>>> dns_servers
```

- This displays the contents of a Python list called `dns_servers`, which contains two string elements.

```
>>> core1
```

- This displays the contents of a Python dictionary called `core1`, which contains key/value pairs.

```
>>> as_python()
```

- This calls the function `as_python()`, which uses the `type()` and `print()` functions to indicate the data types of the two previous variables, and make the dictionary a bit more readable.

```
>>> as_yaml(core1)
```

- This calls the function `as_yaml()` and passes it our `core1` dictionary, which prints out the dictionary in YAML format.

```
>>> as_json(core1)
```

- This calls the function `as_json()` and passes it our `core1` dictionary, which prints out the dictionary in JSON format.

2.3 Now we'll run a Python script to open a YAML file and export it as JSON

It is common and helpful to define data in YAML that will be consumed programmatically by things like Python and Ansible. You've been provided with a short YAML file, which holds a list of key/value (dictionary) pairs of common attributes of L2 switch ports. Let's use Python to import that YAML file, prove that it does contain a list of dictionaries, convert it to JSON, and write that out to a new file.

- Open ports.yml in Atom to examine it
- List the files in the code directory and notice ports.yml exists, but ports.json doesn't

```
$ ls
```

- Run the Python script to open the YAML file, convert it to JSON, and write out a new file

```
$ python lab2.3.py
```

- Notice ports.json now exists after running the Python script; we'll examine it in another lab

3 eAPI Command Explorer

TODO

This lab will demonstrate the on-switch Command API explorer feature. The Command API provides an interface to experiment with commands and view their request and response structure without having to write a script or program to test it with.

3.1 Try out a Show Command

- Connect to labvm by clicking on labvm in the Lab Frontend
- Launch Google Chrome from DevBox (not your laptop) via the Guacamole interface and hit **https://192.168.0.14**
- Accept the self-signed SSL certificate and use **arista** / **arista** as the credentials
- Enter `show interfaces` and click Submit POST request¹

3.2 Try out a Configuration Command

- Enter the following and Submit

```
enable
configure
vlan 1000
name test
```

- Log into your switch and observe that VLAN 1000 is present

Note: To switch between your desktop and the switch, press *Control+Alt+Shift*, click *arista* at the top right of the menu, click *Home* and then expand *veos* and double click on *leaf1*. To switch back, reverse the process.

4 pyeapi

TODO

In this lab we will still use Python, but this time with Arista's pyeapi module. Think of a module as a library as a way of adding on or enhancing the native capabilities of Python. You might also hear these referred to as libraries.

Pyeapi is a wrapper around eAPI that abstracts common EOS commands into a more programmatic style. This allows someone without a heavy network background to easily interact with an Arista device. In other words, instead of issuing `enable; configure; vlan 100; name foo`, we can use `vlangs.set_name(100, 'foo')`.

While that may look similar, the abstracted way is easier to implement in Python because it shortcuts some of the steps, and someone that only knows they need to create a VLAN can grok it without having to know the EOS command line.

[Click here for pyeapi documentation.](#)

4.1 Use provided script to add a local user to a virtual Arista device

```
#!/usr/bin/python
import pyeapi
node = pyeapi.connect(host='192.168.0.14', username='arista', password='arista', return_node=True)
```

¹Note that it requires the full command to work properly; shorthand commands, such as `sh int` do not work. Any API action is the same way.

```
users = node.api('users')
users.create('testuser', secret='foo')
users.set_privilege('testuser', value='15')
users.set_role('testuser', value='network-admin')
```

What does this script do?

`import pyeapi` - this imports the pyeapi module.

`node = pyeapi.connect(host='192.168.0.14', username='arista', password='arista', return_node=True)` - instantiates the variable `node`, and uses `pyeapi` to create a connection to the switch using the username of `arista` and the password `arista`

`return_node` - allows you to use the `node` object to consume the API with - don't focus on this one for now, let's just roll with the rest of the script

`users = node.api('users')` - creates a new variable `users` and specifies the API submodule `users`; this tells Python we want to create a user using `pyeapi`

`users.create('testuser', secret='foo')` - Using the Users API, we use the `create` method; `testuser` is the user- name, and the password is `foo`

`users.set_privilege('testuser', value='15')` - Using the Users API, we use the `set_privilege` method; `testuser` is the username which was created in the last step, and the privilege level is 15

`users.set_role('testuser', value='network-admin')` - Using the Users API, we use the `set_role` method; `testuser` is the username which was created in the last step, and the Arista role is the predefined default role of `network-admin`

5 Git Operations

5.1 Git preferences setup

- Issue the following commands

```
$ git --version
$ git config --global user.name "John Doe"
$ git config --global user.email "john.doe@dogpile.com"
$ git config --list
```

5.2 Create your own repository

- Issue the following commands

```
$ cd
$ mkdir muh-bad-codez
$ cd muh-bad-codez
$ git init
$ echo "Info here" >> README.md
$ git add README.md
$ git status
$ git commit -m "My first commit"
$ git status
$ git reflog
```

5.3 Branch and merge (Optional)

Continuing with the Git repository just created, we'll be creating a branch.

```
$ git branch trysomething
$ git checkout trysomething
$ touch testfile
$ git add testfile
$ git commit -m "Added a test file"
$ git checkout master
$ git merge trysomething
$ git branch -d trysomething
```

5.4 View code diffs in Github

- Visit <http://bit.ly/cbautomation>
- Click through **commits**
- Select any commit
- Choose Split view to compare it to the previous commit

6 Ansible Ad-Hoc Lab

6.1 Run a simple Ansible ad-hoc command against localhost

Let's begin learning Ansible with a simple example of issuing ad-hoc commands. ad-hoc commands are essentially one-off commands - something you might issue once, but not ever need to repeat again.

```
$ ansible localhost -m raw -a "ping -c 3 192.168.0.14"
```

With this ad-hoc command, we're invoking Ansible against localhost, and using the raw command module to send 3 pings to 192.168.0.14. Nothing Earth shattering here.

6.2 Run an Ansible ad hoc command to install a package

- Issue the following command

```
$ ansible localhost -m apt -a "name=yamllint" --become -k -c local
```

- Provide your jumpbox password when prompted

With this ad hoc command, we're invoking Ansible against localhost, using the apt package management module to install the yamllint² package on your jumpbox, as super user with a prompted password. The connection method is local.

6.3 Run an Ansible ad hoc command against a device in inventory

Both ad hoc commands we've run so far were harder than simply invoking the actions they performed directly on your jumpbox.

Let's up the ante by running an ad-hoc command against a remote device, but first we'll need an inventory file. Inventory files define groups of devices along with their IPs and, optionally, variables. You'll typically reference these groups to scope the execution of Ansible playbook tasks.

- Open the ~/automation-class/ansible/hosts inventory file in Atom
- Notice this inventory consists of groups of devices, their common names, and the IPs they should be connected to via
- Uncomment the [veos] group and its member device and save it
- Your inventory file 'hosts' should now look like this

```
[webserver]
web1 ansible_host=10.0.0.1
web2 ansible_host=10.0.0.2
```

```
[database_servers]
db1 ansible_host=172.16.99.1
```

```
[veos]
switch1 ansible_host=192.168.0.14
```

- Issue the following command

```
$ ansible veos -i hosts -m raw -a "show version" -u arista -k
```

That time, we again ran an ad hoc command, but with two differences. First, we referred to a group of (1) devices specified in an inventory file. Second, the commands were not executed locally, but rather on the remote device after authenticating to it. This is how Ansible typically operates.

²yamllint is an excellent utility to check for proper YAML syntax

7 Ansible Playbooks Lab

7.1 Ansible jq Installation Lab

While ad hoc commands can be useful, the real power of Ansible comes from using orchestrated playbooks. This lab will install the jq package for beautifying JSON on your jumpbox, similar to what was done previously with an ad hoc command. However, this time, we'll do it using an Ansible playbook.

- Open ports.json in Atom - Not very easy to look at

```
$ cd ~/automation-class/code/ansible
$ ansible-playbook --version
$ ansible-playbook lab8.1.yml
```

- Now we can use the jq utility to pretty print JSON

```
$ jq . ../ports.json
```

- Ahh, much better

7.2 Ansible REST API Interaction

Here, we'll be using the Ansible Twilio API³ module to send messages for fun and profit. There's money on the line in this lab - things just got real. Edit lab8.2.yml and use Ansible to send me an MMS with exactly the following. The first correctly formed MMS I get wins the prize.

- Your email
- Your VEOS switch serial number
- Heart-shaped Ansible image

³This is my personal Twilio account, be gentle.

8 Jinja

TODO

9 Markdown Tables

First Name	Last Name	Location	Allegiance
Mance	Rayder	North of the Wall	Wildlings
Margaery	Tyrell	The Reach	House Tyrell
Danerys	Targaryen	Meereen	House Targaryen
Tyrion	Lannister	King's Landing	House Lannister