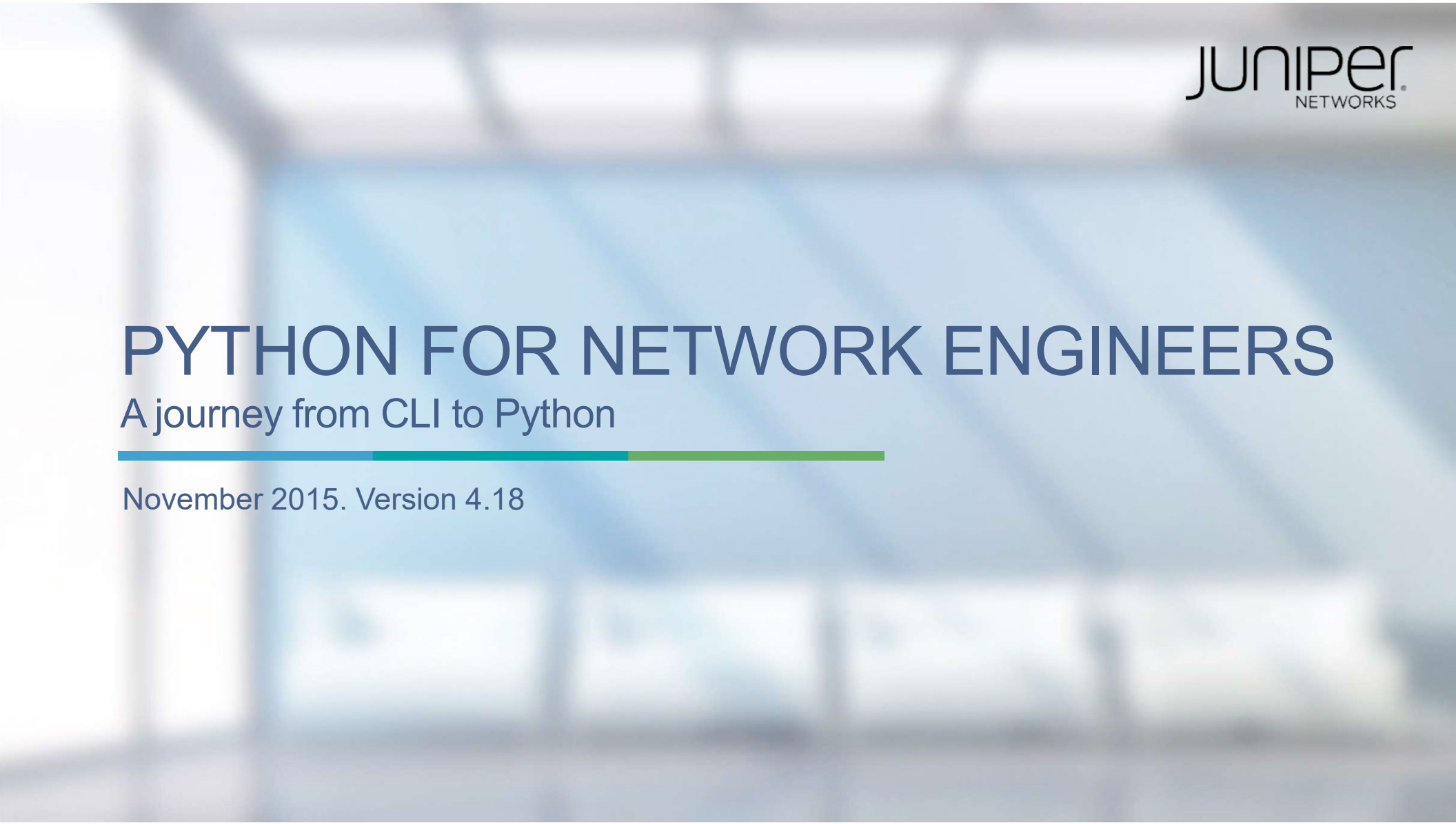


PYTHON FOR NETWORK ENGINEERS

A journey from CLI to Python

November 2015. Version 4.18

DISCLAIMER

This course is useful to network engineers with no python programming knowledge, who wants to use python to manage junos devices.

What makes this course even more interesting, is that it is written for beginners in programming, by a beginner in programming ☺

The examples and code in this document are for learning and educational purposes.

The samples were created with the goals of clarity and ease of understanding.

If you are writing code for a real application, you might write some code differently ☺

AGENDA

- INTRODUCTION TO NETWORK AUTOMATION
- THE BASICS ABOUT PYTHON PROGRAMMING
- IP ADDRESSES MANIPULATION
- FILES MANIPULATION
- TEMPLATES WITH JINJA2
- VARIABLES DEFINITION WITH YAML
- JUNOS AUTOMATION WITH PYEZ LIBRARY
- REST CALLS HANDLING
- REGULAR EXPRESSIONS

SOME USE CASES

- Handle configuration changes faster
  - Add a vlan on all your switches
  - Configure a list of interfaces with a vlan
  - Modify SNMP community over 100 devices
- Reduces human errors
  - Configurations are almost always deployed manually. Automated configuration deployment reduces human errors

SOME USE CASES

- Simplify network audit
  - One way to handle this task is to manually open an SSH connection to every device and manually populate a spreadsheet with relevant information. But we can perform this task in a more automated/programmatic fashion.
  - Some examples:
    - Check if your switches uplinks ports are up
    - Check the status of your BGP neighbors (state needs to be “established”)
  - Data Collection before and after a change (a configuration change, an upgrade …)
    - Check if there is any issue after a change (do you have a BGP neighbor not in an “established” state ….)
    - Upgrade all your devices, and then check if each of them is running the new version

EXAMPLE OF OFF-BOX SCRIPTING

- I have 100 devices on the network.
- I have to update all of my devices with a new snmp community
- Think like a programmer:
  - Use Python on a server/laptop as the point-of-control to remotely manage Junos devices.
  - Steps:
    - Find/Create a list of all the devices IPs and Credentials
    - For every device in the list
      - Connect to device
      - load the configuration change
      - commit the configuration change
      - Close the session
    - Repeat with the next device until we finish the list

WHEN USING AUTOMATION

- The Build phase
  - Around the initial design and installation of a network component.
  - ZTP, Python, Openclos, Ansible can help in building network
- The Configure phase
  - Deploy on demand configuration and software changes to the platform.
  - PyEZ, Puppet, Chef can help in configuring or reconfiguring the network for new services
- The Audit phase
  - Deals with automating the process of monitoring operational state of the platform
  - PyEZ, JUNOS REST API, SNMP can be used to help monitoring or auditing the network

PYTHON

WHAT IS PYTHON?

- A programming language
- Popular. Widely used.
- Contribution from a very large community
  - Lots of modules available (repository) extending the capabilities of the language
- Easy to learn
- Indentation matters
- Versions:
  - Python 2.7 is still mostly in use.
    - Recommended version for PyEZ is Python 2.7
  - Python 3.5 adoption is coming.
    - Python 3.X adoption inhibited because of the many community modules that require 2.7

PIP (package manager/installer program)

- pip stand for "Pip Installs Packages" or "Pip Installs Python".
- pip is a package management system used to find, install and manage Python packages.
- Many packages can be found in the Python Package Index (PyPI).
  - This is a repository for Python.
  - There are currently 90000 packages.
  - https://pypi.python.org/pypi.
- You can use pip to find packages in Python Package Index (PyPI) and to install them.

LAB 1 – USE PIP

- Use pip –help to understand the various options

```
pytraining@py-automation-backup:~$ pip --help
```

- Use pip list to list installed packages.
  - ncclient (netconf client used by PyEZ), Paramiko (ssh client used by PyEZ), junos-eznc (PyEZ), pip, jinja2 to manage templates, netaddr to manage ip addresses, requests to handle REST calls, …

```
pytraining@py-automation-master:~$ pip list
```

- Use pip search to searches packages related to contrail or Juniper or vpn or regex .
- Other pip options frequently used are install and uninstall ...

LAB 1 - VARIABLES

- Variables store values
- Declare the name of the variable and use the assignment operator = to assign the value
- We do not declare the variable type. The value assigned to the variable determines the type
  - Integer, Floats, String, Lists, Dictionaries, and more types.
  - Dynamically typed

LAB 1 - INTEGERS

- Assign an integer to a variable
  - You can add spaces around '=' for readability:
    - a=b or a = b (both work in python. Last option is more readable)
- Integers are whole numbers

```
>>> a = 192
>>> a
192
>>> type(a)
<type 'int'>
>>>
```

- You can use Type(a) or Type (a). Both work in python. We generally do not use a space between function name and (). This is more readable.

LAB 1 - INTEGERS

- Manipulate integers with arithmetic operators

```
>>> a
192
>>> b = 10
>>> type(b)
<type 'int'>
>>> a+b
202
>>> b%3
1
>>> b/3
3
>>> b/3.0
3.3333333333333335
>>>
```

LAB 1 - INTEGERS

- Convert an integer into a string with the built-in function str

```
>>> a
192
>>> str(a)
'192'
>>>
```

- Convert an integer into an hexadecimal with the built-in function hex

```
>>> hex(a)
'0xc0'
>>> 0xff
255
```

- Convert an integer into an binary with the built-in function bin

```
>>> bin(a)
'0b11000000'
>>> 0b1111
15
```

LAB 1 - COMPARAISON OPERATORS

- Comparison operators compare two values and return a Boolean

```
>>> a
192
>>> b
10
>>> a==b
False
>>> a!=b
True
>>> a>b
True
>>> a<=b
False
```

LAB 1 - DIR

- dir() returns a list of the names in the current scope

```
>>> a
192
>>> b
10
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b']
>>>
```

LAB 2 - STRINGS

- Assign a string to a variable
  - Use single quotes ' or double quotes "
  - Use \n for newline and \t for tab

```
>>> hostname="ex4200-1"
>>> type(hostname)
<type 'str'>
>>> hostname
'ex4200-1'
>>> print hostname
ex4200-1
>>>
>>> byte="192"
>>> type(byte)
<type 'str'>
>>>
```

LAB 2 - STRINGS

- To create a multi lines string, use three single quotes or three double quotes

```
>>> banner='''you are accessing a restricted system.
be advised your actions are logged and audited are performed daily'''
>>> type(banner)
<type 'str'>
>>> print banner
you are accessing a restricted system.
be advised your actions are logged and audited are performed daily
>>>
```

LAB 2 - STRINGS

- Use a membership operator (in, not in) to check if a string is a substring of another one

```
>>> ip='192.168.10.254'
>>> ip
'192.168.10.254'
>>> type(ip)
<type 'str'>
>>> '192' in ip
True
>>> '193' in ip
False
>>> '193' not in ip
True
>>>
```

LAB 2 - STRINGS

- Convert a string into an integer

```
>>> byte="192"
>>> type(byte)
<type 'str'>
>>> int(byte)
192
>>> type (int(byte))
<type 'int'>
>>>
```

STRINGS

- To get the list of available functions for strings, use the built-in function dir with the argument str (you can also use an instance of str as the argument)
  - Some functions for strings are: upper, lower, join, split, splitlines, strip, …

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b',
'banner', 'byte', 'hostname', 'ip']
>>> dir(str)
[ ... 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

STRINGS

- Get help with strings:

```
>>> help(str)
```

- Get help with a function. Example with the function upper

```
>>> help(banner.upper)
Help on built-in function upper:

upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.
>>>
```

STRINGS

- Convert a string to uppercase. Use the built-in function upper

```
>>> print banner
you are accessing a restricted system.
be advised your actions are logged and audited are performed daily
>>> help (banner.upper)
Help on built-in function upper:

upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.

>>> print banner.upper()
YOU ARE ACCESSING A RESTRICTED SYSTEM.
BE ADVISED YOUR ACTIONS ARE LOGGED AND AUDITED ARE PERFORMED DAILY
>>>
```

COMMENTS

- The symbol # indicates a comment

```
>>> # this is a comment
>>> domain="juniper.net" #this is another comment
>>> domain
'juniper.net'
>>> print domain
juniper.net
>>>
```

PRINT

```
>>> hostname="ex4200-1"
>>> ip="172.30.179.101"
>>>
>>> # use + to concatenate strings
>>> print "the device hostname is " + hostname
the device hostname is ex4200-1
>>> print "the device " + hostname + " has the ip " + ip
the device ex4200-1 has the ip 172.30.179.101
>>>
>>> print "the device %s has the ip %s" % (hostname, ip)
the device ex4200-1 has the ip 172.30.179.101
>>> print "%s %s" % (hostname,ip)
ex4200-1 172.30.179.101
>>>
>>> print "the device {0} has the ip {1}".format(hostname, ip)
the device ex4200-1 has the ip 172.30.179.101
>>> help (str.format)
```

Deprecated syntax, does not work in 3.x
Prefer format() for forward compatibility

LISTS

- A collection of items
- Items are ordered
- Items separated by commas and enclosed within square brackets ([])
- A list is iterable: a “for loop” iterates over its items

LAB 3 - LISTS

- Create a list

```
>>> my_devices_list=["172.30.108.11", "172.30.108.14", "172.30.108.141",
"172.30.108.133", "172.30.108.254"]
>>> my_devices_list
['172.30.108.11', '172.30.108.14', '172.30.108.141', '172.30.108.133',
'172.30.108.254']
>>> type (my_devices_list)
<type 'list'>
>>>
```

LAB 3 - LISTS

- Get part of a list

```
>>> my_devices_list
['172.30.108.11', '172.30.108.14', '172.30.108.141', '172.30.108.133',
'172.30.108.254']
>>> my_devices_list [0]
'172.30.108.11'
>>> my_devices_list[-1]
'172.30.108.254'
>>> my_devices_list [1:]
['172.30.108.14', '172.30.108.141', '172.30.108.133', '172.30.108.254']
>>> my_devices_list [1:-1]
['172.30.108.14', '172.30.108.141', '172.30.108.133']
>>> my_devices_list [1:3]
['172.30.108.14', '172.30.108.141']
>>>
```

LAB 3 -LISTS

- Check if an item is a list with a membership operator (in, not in)

```
>>> my_devices_list
['172.30.108.11', '172.30.108.14', '172.30.108.133', '172.30.108.254']
>>> "172.30.108.14" in my_devices_list
True
>>> "172.30.108.14" not in my_devices_list
False
>>> "172.30.108.187" in my_devices_list
False
>>>
```

LAB 3 - LISTS

- Get the list of available functions
  - Some functions are: len, sort, index, insert, append, count, ...

```
>>> dir(my_devices_list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

LAB 3 - LISTS

- Get help

```
>>> help(my_devices_list)
```

- Get help for one function. Example with the function insert.

```
>>> help(my_devices_list.insert)
```

LAB 3 - LISTS

- Insert an element with the built-in function insert

```
>>> my_devices_list
['172.30.108.11', '172.30.108.14', '172.30.108.133', '172.30.108.254',
'172.30.108.133']
>>>
>>> my_devices_list.insert(2,'172.30.108.176')
>>> my_devices_list
['172.30.108.11', '172.30.108.14', '172.30.108.176', '172.30.108.133',
'172.30.108.254', '172.30.108.133']
>>>
>>> my_devices_list.insert(0,'172.30.108.10')
>>> my_devices_list
['172.30.108.10', '172.30.108.11', '172.30.108.14', '172.30.108.176',
'172.30.108.133', '172.30.108.254', '172.30.108.133']
>>>
>>> my_devices_list.insert(-1,'172.30.108.132')
>>> my_devices_list
['172.30.108.10', '172.30.108.11', '172.30.108.14', '172.30.108.176',
'172.30.108.133', '172.30.108.254', '172.30.108.132', '172.30.108.133']
>>>
>>> help(my_devices_list.insert)
```

DICTIONARIES

- Collection of key-value pairs
- We use dictionaries to associate values to keys
- Keys are unique
- Items are unordered
- Use curly {} brackets to declare the dictionary.
  - Separate the key and value with colons
  - Use commas between each pair
- Use square brackets [] to retrieve the value for a key
- A dictionary is iterable: a “for loop” iterates over its keys.

LAB 4 - DICTIONARIES

```
>>> this_is_a_dictionary={'domain': 'jnpr.net', 'hostname': 'LAB-EX-VC-
Backbone', "time_zone": 'Europe/Paris', "name_server": "195.68.0.1"}
>>> this_is_a_dictionary
{'domain': 'jnpr.net', 'hostname': 'LAB-EX-VC-Backbone', 'time_zone':
'Europe/Paris', 'name_server': '195.68.0.1'}
>>> type (this_is_a_dictionary)
<type 'dict'>
>>>
```

LAB 4 - DICTIONARIES

- Print a dictionary with pprint (pretty print)
- pprint is not a built-in function, pprint is provided by the module ‘pprint’

```
>>> print this_is_a_dictionary
{'domain': 'jnpr.net', 'hostname': 'LAB-EX-VC-Backbone', 'time_zone':
'Europe/Paris', 'name_server': '195.68.0.1'}
>>>
>>> from pprint import pprint
>>> dir()
>>> pprint (this_is_a_dictionary)
{'domain': 'jnpr.net',
 'hostname': 'LAB-EX-VC-Backbone',
 'name_server': '195.68.0.1',
 'time_zone': 'Europe/Paris'}
>>>
>>> from pprint import pprint as pp
>>> dir()
>>> pp (this_is_a_dictionary)
{'domain': 'jnpr.net',
 'hostname': 'LAB-EX-VC-Backbone',
 'name_server': '195.68.0.1',
 'time_zone': 'Europe/Paris'}
```

LAB 4 - DICTIONARIES

- Get the keys and values

```
>>> this_is_a_dictionary.keys()
['domain', 'hostname', 'time_zone', 'name_server']
>>> this_is_a_dictionary.values()
['jnpr.net', 'LAB-EX-VC-Backbone', 'Europe/Paris', '195.68.0.1']
>>>
```

LAB 4 - DICTIONARIES

- Query the dictionary using square brackets [ ]

```
>>> this_is_a_dictionary["hostname"]
'LAB-EX-VC-Backbone'
>>> this_is_a_dictionary["domain"]
'jnpr.net'
>>>
```

FOR LOOPS

- Syntax
  - Indentation matters: use spaces for the indented block of statements

```
for (expression):
    statement(s)
```

- Use “for loops” if you have something you need to iterate
  - with a list, it iterates over its items
  - with a dictionary, it iterates over its keys
  - with a file, it iterates over its lines
  - with a string, it iterates over its characters.
  - …
- For each iteration of the iterable, the “for loop” execute the statements in the loop body
- Alternatively, use a “for loop” to do something a fixed number of times

LAB 5 - FOR LOOPS WITH A LIST

- If we use a for loop with a list, it iterates over its items.

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14',
'172.30.108.176', '172.30.108.254']
>>> for device in my_devices_list:
      print ("the current device is: " + device)



the current device is: 172.30.108.11
the current device is: 172.30.108.133
the current device is: 172.30.108.133
the current device is: 172.30.108.14
the current device is: 172.30.108.176
the current device is: 172.30.108.254
>>>
```

LAB 5 - FOR LOOPS WITH A DICTIONARY

- If we use a for loop with a dictionary, it iterates over its keys

```
>>> this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'hostname': 'LAB-EX-VC-Backbone',
'domain': 'jnpr.net', 'name_server': '195.68.0.1', 'time_zone':
'Europe/Paris'}
>>> for key in this_is_a_dictionary:
      print key



ntp_server
hostname
domain
name_server
time_zone
>>>
```

LAB 5 - FOR LOOPS WITH A DICTIONARY

- If we use a for loop with a dictionary, it iterates over its keys

```
>>> this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'hostname': 'LAB-EX-VC-Backbone',
'domain': 'jnpr.net', 'name_server': '195.68.0.1', 'time_zone':
'Europe/Paris'}
>>> for key in this_is_a_dictionary:
      print this_is_a_dictionary[key]



172.17.28.5
LAB-EX-VC-Backbone
jnpr.net
195.68.0.1
Europe/Paris
>>>
```

CONDITIONALS: IF…ELIF…ELSE

- Syntax:
  - Indentation matters: use spaces for the indented blocks of statements

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
else:
    statement(s)
```

- Executes some statements if an expression evaluates to true
- Elif and Else are optional
- Elif means “else if”
- Else specify actions to take if no condition was met previously.

LAB 6 - IF…ELIF…ELSE

- Use a comparison operator in the if or elif expression

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14',    '172.30.108.176',
'172.30.108.254']
>>> for device in my_devices_list:
     if device=='172.30.108.14':
      print "172.30.108.14 was found in my_devices_list"


172.30.108.14 was found in my_devices_list
>>>
```

LAB 6 - IF…ELIF…ELSE

- Use a comparison operator in the if or elif expression

```
>>> this_is_a_dictionary
{'ntp_server': '172.17.28.5', 'hostname': 'LAB-EX-VC-Backbone', 'domain': 'jnpr.net',
'time_zone': 'Europe/Paris'}
>>> for key in this_is_a_dictionary :
     if key=='domain':
      print "the domain is " + this_is_a_dictionary[key]


the domain is jnpr.net
>>>
```

LAB 6 - IF…ELIF…ELSE

- Use a membership operator in the if or elif expression

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14',    '172.30.108.176',
'172.30.108.254']
>>> '172.30.108.14' in my_devices_list
True
>>> if '172.30.108.14' in my_devices_list:
     print "172.30.108.14 was found in my_devices_list"
    else:
     print "172.30.108.14 was not found in my_devices_list "


172.30.108.14 was found in my_devices_list
>>>
```

PYTHON BUILDING BLOCKS

PYTHON BUILDING BLOCKS

- Module:
  - A file with Python code. A python file.
  - The file name is the module name with the suffix .py appended (module.py).
  - A module can define functions, classes, variables ...
- Package: several python modules all together in a directory, accompanied with a file named __init__.py. The file __init__.py can be empty.
- Function:
  - A function returns a value. Call a function passing arguments.
  - There are many built-in functions. You can also create your own functions.
  - A function is defined once and can be called multiple times.

PYTHON BUILDING BLOCKS

- Class:
  - Classes define objects.
  - Call a class passing arguments. The returned value is an object. So each instance of a class is an object.
  - A class defines functions available for this object (in a class, these functions are called methods)
- Method:
  - A method is a function defined in a class.
  - To call a method, we first need to create an instance of the class. Once you have an instance of a class, you can call a method for this object.

MODULE __builtin__

- Python has a number of functions built into it that are always available.
- The module __builtins__ contains built-in functions which are automatically available. You don’t have to import this module. You don’t have to import these built-in functions.

MODULES FOR NETWORK ENGINEERS

- Python allows you to import modules to reuse code.
  - Good programmers write good code; great programmers reuse/steal code ☺
  - Importing a module is done without using the .py extension
- Anyone can create modules for private uses or to share with community
- Some very nice Python modules for network engineers:
  - netaddr: a Python library for representing and manipulating network addresses
  - re: regular expressions
  - requests: rest api manipulation
  - jinja2: generate documents based on templates
  - Yaml: “users to programs” communication (to define variables)
  - PyEZ: Python library to interact with Junos devices

MANIPULATE IP ADDRESSES WITH PYTHON

PYTHON NETADDR PACKAGE

- There are many Python modules to manipulate IP addresses: ipaddr (google contribution), ipaddress (easy but requires python 3), IPy, netaddr, …
- netaddr is a Python package to manipulate IP addresses and subnets.
  - IPAddress is a class in module netaddr.ip. An IPAddress instance is an individual IPv4 or IPv6 address object (without net mask)
  - IPNetwork is a class in module netaddr.ip. An IPNetwork instance is an IPv4 or IPv6 network or subnet object

THE CLASS IPADDRESS

- Import the class IPAddress

```
>>> from netaddr import IPAddress
```

- Instantiate the class IPAddress

To instantiate a class, declare a variable and call the class passing arguments. This assigns the returned value (the newly created object) to the variable.

```
>>> ip=IPAddress('192.0.2.1')
>>> type(ip)
<class 'netaddr.ip.IPAddress'>
>>> ip
IPAddress('192.0.2.1')
>>> print ip
192.0.2.1
```

- Then you can easily play with the created objects using methods and properties.

THE CLASS IPADDRESS

- Use dir() to check if the class IPAddress and its instance the variable ip are the current scope
- Use dir (IPAddress) to list the available methods and properties with this class.
- Use help (IPAddress) to get help with this class.
- To get help about a method or property of this class, use help (IPAddress.method) or help (IPAddress.property).

```
>>> dir()

>>> dir(IPAddress)

>>> help(IPAddress)

>>> help (IPAddress.version)
Help on property:

    the IP protocol version represented by this IP object.
```

THE CLASS IPADDRESS

- Some methods and properties:

```
>>> ip
IPAddress('192.0.2.1')
>>> ip.version
4
>>> ip.is_private()
False
>>> ip.is_unicast()
True
>>> ip.is_multicast()
False
>>> ip.bits()
'11000000.00000000.00000010.00000001‘
>>>
>>> ip
IPAddress('192.0.2.1')
>>> ip+1
IPAddress('192.0.2.2')
>>> ip+255
IPAddress('192.0.3.0')
```

LAB 7 - THE CLASS IPNEWORK

- The class IPNetwork is define in the package netaddr.ip
- Each instance of the class IPNetwork is an object (a subnet)
- Once you have created an instance of the class IPNetwork, you can use the methods defined in the class IPNetwork with this subnet.
  - The method “next” returns the adjacent subnet succeeding the `IPNetwork` object.
  - the method “previous” returns the adjacent subnet preceding the `IPNetwork` object.

LAB 7 - THE CLASS IPNEWORK

- Import the class IPNetwork

```
>>> from netaddr import IPNetwork
```

- Instantiate the class IPNetwork

To instantiate a class, declare a variable and call the class passing arguments. This assigns the returned value (the newly created object) to the variable.

```
>>> net=IPNetwork('192.0.2.0/24')
>>> type(net)
<class 'netaddr.ip.IPNetwork'>
>>> net
IPNetwork('192.0.2.0/24')
>>> print net
192.0.2.0/24
>>> net[0]
IPAddress('192.0.2.0')
>>> net[-1]
IPAddress('192.0.2.255')
>>> print net[-1]
192.0.2.255
```

LAB 7 - THE CLASS IPNEWORK

- Use dir() to check if the class IPNetwork and its instance the variable net are the current scope
- Use dir(IPNetwork), help(IPNetwork), or help(IPNetwork.method) to learn more about the class.

```
>>> dir()

>>> dir(IPNetwork)

>>> help(IPNetwork)

>>> help (IPNetwork.broadcast)
Help on property:

    The broadcast address of this `IPNetwork` object
```

LAB 7 - THE CLASS IPNETWORK

- Some properties:

```
>>> net.version
4
>>> net.netmask
IPAddress('255.255.255.0')
>>> net.hostmask
IPAddress('0.0.0.255')
>>> net.network
IPAddress('192.0.2.0')
>>> net.broadcast
IPAddress('192.0.2.255')
>>> net.size
256
>>> net.prefixlen
24
```

- Some methods:

```
>>> net.is_unicast()
True
>>> net.is_private()
False
>>> net.is_reserved()
False
>>> net.next()
IPNetwork('192.0.3.0/24')
>>> net.previous()
IPNetwork('192.0.1.0/24')
```

LAB 7 – MANIPULATE IP ADDRESSES

- Test if an IP address belongs to a subnet. Use the membership operator (in, not in)

```
>>> from netaddr import IPNetwork, IPAddress
>>> net=IPNetwork('192.0.2.0/24')
>>> ip=IPAddress('192.0.2.1')
>>> if ip in net:
     print str(ip) + " is in "+ str (net)
    else:
     print str(ip) + " is not in "+ str (net)

192.0.2.1 is in 192.0.2.0/24
```

LAB 7 - MANIPULATE IP ADDRESSES

- Generates the IP addresses for a subnet

```
>>> from netaddr import IPNetwork
>>> net=IPNetwork('192.0.2.0/29')
>>> for ip in net:
     print ip

192.0.2.0
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
192.0.2.5
192.0.2.6
192.0.2.7
```

MANIPULATE FILES WITH PYTHON

LAB 8 - OPEN A FILE

- Use the built-in function open in module __builtin__
- Read mode is the default mode. The other modes are write and append.
- With read mode, the file pointer is placed at the beginning of the file.
- Open the file list_of_ip.txt with read mode

```
>>> help(open)
>>> f=open("python_basics/list_of_ip.txt", "r")
>>> f
<open file 'python_basics/list_of_ip.txt', mode 'r' at 0x000000000317AD20>
>>> type(f)
<type 'file'>
```

LAB 8 - CLOSE A FILE

- To close the file, use the method close

```
>>> f.close()
>>> f
<closed file 'python_basics/list_of_ip.txt', mode 'r' at 0x000000000317AD20>
>>> f.closed
True
```

LAB 8 - GET THE AVAILABLE METHODS

- Use the dir builtin function with the argument file to get the list of available methods for files (f is an instance of the class file. So you can also use f as the argument)
  - Some methods are: close, write, read, readline, seek, ….

```
>>> dir(file)
```

LAB 8 - GET HELP

- Use help(f) to get help on the object f. You can also use help(file) to get help with the class file.

```
>>> help(file)
>>> help(f)
```

- To get help on a method the class file, use help(file.method) or help(f.method).
  - Example with the method write:

```
>>> help(f.write)
Help on built-in function write:
write(...)
    write(str) -> None.  Write string str to file.

    Note that due to buffering, flush() or close() may be needed before
    the file on disk reflects the data written.
>>>
```

LAB 8 - READ A FILE

- The method read reads an open file and returns a string.
- If there is no argument, read until EOF is reached.

```
>>> f=open("python_basics/list_of_ip.txt","r")
>>> f
<open file 'python_basics/list_of_ip.txt', mode 'r' at 0x000000000317AE40>
>>> help(f.read)
>>> s=f.read()
>>> type(s)
<type 'str'>
>>> s
'172.30.179.101\n172.30.179.102\n172.30.179.103\n172.30.179.104\n172.30.179.105\n'
>>> print s
172.30.179.101
172.30.179.102
172.30.179.103
172.30.179.104
172.30.179.105
>>> f.close()
```

LAB 8 - WRITE CONTENT ON A FILE

- To open a file with write mode, use "w".
  - If the file doesn’t exist, python will create it.
  - If the file already exists, python will overwrite its content.
- To open a file with append mode, use "a".
  - The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. You can write content without overwriting the file content.
  - If the file does not exist, it creates a new file for writing.

```
>>> f=open("python_basics/list_of_ip.txt","a")
>>> f
<open file 'python_basics/list_of_ip.txt', mode 'a' at 0x000000000317AD20>
>>> help(f.write)
>>> f.write("172.30.179.106\n")
>>> f.write("172.30.179.107\n")
>>> f.close()
```

USE TEMPLATES WITH PYTHON

JINJA2 PACKAGE

- Jinja2 is a Python package used to generate documents based on templates.
- There are other templating engines for Python: jinja2 is simple, rich, stable and widely used.
- Jinja2 files use a .j2 file extension
- Variables are marked in the template
  - use a {{ variable-name }} syntax.
- Supports some control structures (if and for).
  - use a {% ... %} syntax.
- We will use Jinja2 to handle junos templates

LAB 9 - JINJA2

```
>>> # import the class Template from module jinja2.environment
>>> from jinja2 import Template
>>>
>>> # template is an instance of the class Template.
>>> template=Template("set system host-name {{hostname}}")
>>> type (template)
<class 'jinja2.environment.Template'>
>>>
>>> # render is a method from class Template
>>> print template.render(hostname="EX_RACK3_RAW12")
set system hostname EX_RACK3_RAW12
>>>
>>> # template2 is another instance of class Template
>>> template2=Template("set vlan {{vlanname}} vlan-id {{vlanid}}")
>>> print (template2.render(vlanname="v10",vlanid="10"))
set vlan v10 vlan-id 10
```

LAB 9 - JINJA2

- Lets use a for loop in the jinja2 template

```
more jinja2_basics/template_int_vlan_2.j2
{%- for interface in interfaces_list %}
set interfaces {{ interface }} unit 0 family ethernet-switching port-mode access vlan members {{
vlan_name }}
{%- endfor %}
```

```
>>> from jinja2 import Template # import the class Template from module jinja2.environment
>>> s=open("jinja2_basics/template_int_vlan_2.j2").read()
>>> print s
{%- for interface in interfaces_list %}
    set interfaces {{ interface }} unit 0 family ethernet-switching port-mode access vlan members
{{ vlan_name }}
{%- endfor %}
>>> template=Template(s)
>>> print template.render(interfaces_list=["ge-0/0/4", "ge-0/0/5", "ge-0/0/6"], vlan_name="v14")
set interfaces ge-0/0/4 unit 0 family ethernet-switching port-mode access vlan members v14
set interfaces ge-0/0/5 unit 0 family ethernet-switching port-mode access vlan members v14
set interfaces ge-0/0/6 unit 0 family ethernet-switching port-mode access vlan members v14
>>>
```

DEFINE PYTHON LISTS AND DICTIONARIES USING YAML FILES

YAML

- YAML stands for "Yaml Ain't Markup Language"
- Yaml is human-readable language.
  - Less markup than XML.
  - A superset of JSON.
- Used for “users to programs” communication
  - For users to read/change data.
  - Used to communicate with program.
  - Designed to translate to structures which are common to various languages (cross language: Python, Perl, Ruby, etc).
  - Used to define variables value.

YAML SYNTAX

- Yaml files use a .yaml or .yml extension
- Yaml documents begin with three dashes - - -
- Comments begin with #
- Strings are unquoted
- Indentation with one or more spaces
  - never with tabulations
- Lists: one member per line.
  - Hyphen + space for each item.
- Keys are separated from values by a colon + space.

YAML SYNTAX FOR A LIST

- device_list.yml is a yaml file.
  - This is a YAML list
  - There is one item per line
  - Hyphen + space for each new item

```
more yaml_basics/device_list.yml
---
#IPs of the devices in the lab environment
    - 172.30.179.101
    - 172.30.179.102
    - 172.30.179.103
    - 172.30.179.104
    - 172.30.179.105
```

TRANSFORM A YAML FILE INTO A PYTHON STRUCTURE

- Open a yaml file

```
>>> f=open('yaml_basics/device_list.yml')
>>> f
<open file 'yaml_basics/device_list.yml', mode 'r' at 0x00000000044468A0>
>>> type (f)
<type 'file'>
```

- Read the file and return a string

```
>>> s=f.read()
>>> type (s)
<type 'str'>
>>> print s
---
#IPs of devices in the lab environment
    - 172.30.179.101
    - 172.30.179.102
    - 172.30.179.103
    - 172.30.179.104
    - 172.30.179.105
```

TRANSFORM A YAML FILE INTO A PYTHON STRUCTURE

- Import the yaml package

```
>>> import yaml
```

- Use the load function to read a string and produce the corresponding Python structure

```
>>> my_vars=yaml.load (s)
```

- my_var is a Python list ! With the content of the yaml file.

```
>>> my_vars
['172.30.179.101', '172.30.179.102', '172.30.179.103',
'172.30.179.104', '172.30.179.105']
>>> type(my_vars)
<type 'list'>
```

YAML SYNTAX FOR A DICTIONARY

- this_is_a_dictionary.yml is a yaml file.
  - This is a YAML dictionary
  - Keys are separated from values by a colon + space.
  - There are 2 keys (interfaces and vlan_name)
  - The value for the first key is a list

```
more yaml_basics/this_is_a_dictionary.yml
---
interfaces:
    - ge-0/0/9
    - ge-0/0/10
    - ge-0/0/16
    - ge-0/0/18

vlan_name: v14
```

CREATE A PYTHON DICTIONNARY WITH YAML

```
>>> import yaml
>>> s=open('this_is_a_dictionary.yml').read()
>>> print s
---
interfaces:
    - ge-0/0/9
    - ge-0/0/10
    - ge-0/0/16
    - ge-0/0/18

vlan_name: v14
>>> my_vars=yaml.load (s)
>>> type(my_vars)
<type 'dict'>
>>> from pprint import pprint
>>> pprint (my_vars)
{'interfaces': ['ge-0/0/9', 'ge-0/0/10', 'ge-0/0/16', 'ge-0/0/18'],
 'vlan_name': 'v14'}
>>> my_vars['interfaces']
['ge-0/0/9', 'ge-0/0/10', 'ge-0/0/16', 'ge-0/0/18']
>>> my_vars['vlan_name']
'v14'
```

CREATE JUNOS CONFIGURATION FILES WITH JINJA2 AND YAML

JINJA2 AND YAML

- Lets use a jinja2 template and a yaml file to build the initial junos configuration files we can use with a ZTP setup to configure new devices (build phase).
- We need to provide to each new device (factory default configuration) at least the following:
  - -a root password (otherwise we can not commit the conf).
  - -a management ip @ and subnet, and a route (to be able to reach remotely the new device).
  - -allow ssh connection (in case we want to ssh it).
  - -enable netconf over ssh (to be able then to use PyEZ in the run and audit phases).
  - -an hostname.
- Only the hostname and the management ip @ are unique per device.
  - So only these 2 details are define as variables in the jinja2 template.
  - The yaml file define their values for each device.

LAB 10 - JINJA2 AND YAML

- configuration_builder/variables_build.yml is a yaml file.
  - This is a yaml list. With 3 items. Each item is a device.
  - Each item of the list is a dictionary with the device hostname and management ip @.
  - It is extremely easy to add other devices.
  - You can use another yaml structure (i.e instead of a list of dictionaries) but in that case you'll need to parse it differently from the jinja2 and python files.

```
pytraining@py-automation-master:~$ more configuration_builder/variables_build.yml
```

LAB 10 - JINJA2 AND YAML

- configuration_builder/template_build.j2 is a jinja2 template.
  - this is the template to build the initial junos configuration file.
  - It uses the variables defined in the yaml file.

```
pytraining@py-automation-master:~$ more configuration_builder/template_build.j2
```

LAB 10 - JINJA2 AND YAML

- configuration_builder/configuration_builder.py is a python script.
  - It uses the jinja2 template and the yaml file to create the initial junos configuration file for each device defined in the yaml file.
  - You can use these files with a ZTP setup to configure automatically new devices (build phase).

```
pytraining@py-automation-master:~$ more configuration_builder/configuration_builder.py
```

LAB 10 - JINJA2 AND YAML

- Use the python to generate the junos configuration file.

```
pytraining@py-automation-master:~$ python configuration_builder/configuration_builder.py
Start configuration building
generate config file for device ex4300-4 : conf_file_build_phase_ex4300-4.conf
generate config file for device ex4300-9 : conf_file_build_phase_ex4300-9.conf
generate config file for device ex4300-10 : conf_file_build_phase_ex4300-10.conf
done

pytraining@py-automation-master:~$ ls | grep build
conf_file_build_phase_ex4300-10.conf
conf_file_build_phase_ex4300-4.conf
conf_file_build_phase_ex4300-9.conf

pytraining@py-automation-master:~$ more conf_file_build_phase_ex4300-10.conf
pytraining@py-automation-master:~$ more conf_file_build_phase_ex4300-4.conf
pytraining@py-automation-master:~$ more conf_file_build_phase_ex4300-9.conf
```

JUNOS AUTOMATION WITH PYTHON PYEZ LIBRARY

PYEZ AGENDA

- PYEZ INTRODUCTION
- CONNECT TO DEVICES
- RETRIEVE FACTS
- CONFIGURATION MANAGEMENT
- NETWORK AUDIT
- SOFTWARE UPGRADE AND ASSOCIATED FUNCTIONS
- EXCEPTIONS HANDLING

PYEZ INTRODUCTION

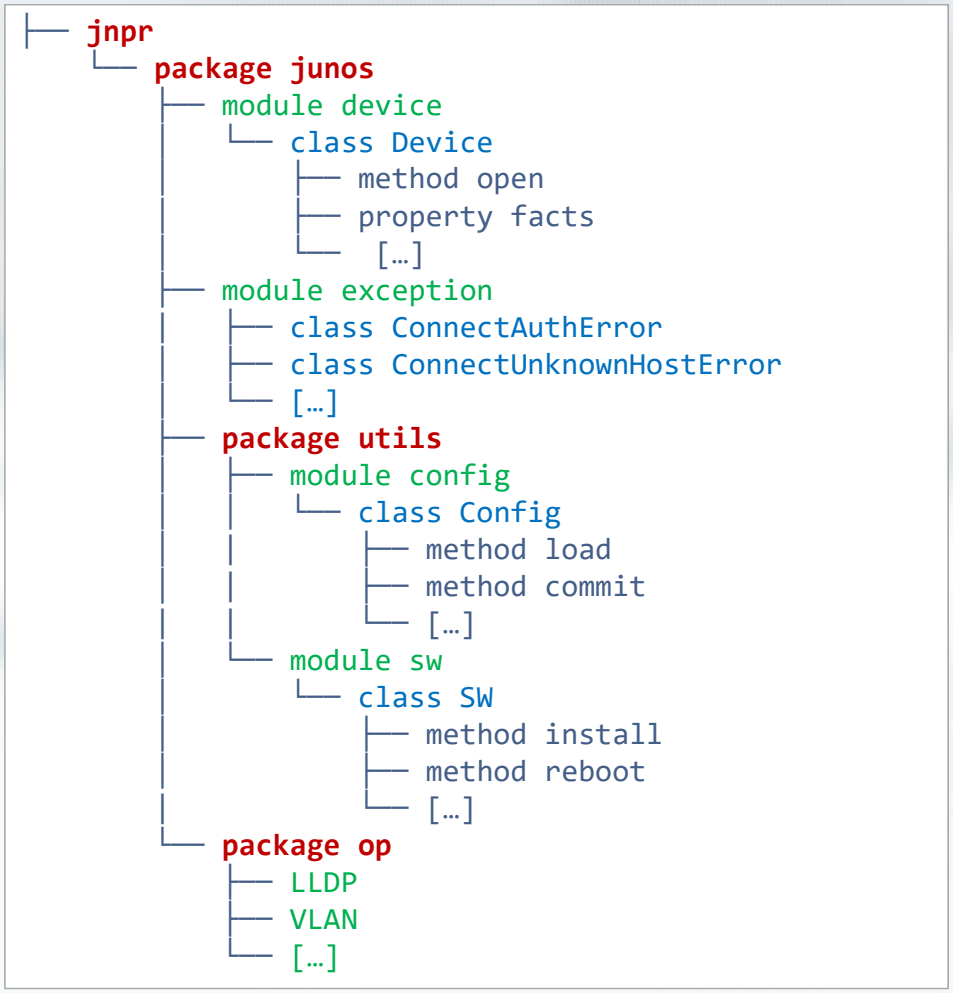
PYEZ

- Allows to manage Junos devices
- Not tied to Junos version or to Junos product.
- A Juniper package for Python
  - A package is a collection of Python modules
  - Provides classes and methods
- A Python framework
  - Provides code that is useful for larger applications.
  - Used by Ansible
- Current PyEZ version in 1.2.3
- Has been tested with Python 2.6 and 2.7.
  - Not supported with Python 3.x due to dependencies with other Python modules such as ncclient that do not yet support Python 3.x

PYEZ

- Learning PyEZ is easy
- No need to be a programmer
- It's also for network engineers
- PyEZ abstracts unnecessary details and complexity allowing us to focus on programmatically interfacing with Junos.
  - A number of low level details are handled by PyEZ such as the ssh session establishment. no need to worry about these details. Just focus on programmatically interfacing with Junos.

PYEZ ARCHITECTURE



NETCONF

NETCONF PROTOCOL

- PyEZ uses Netconf. You need to enable Netconf on your devices.
- Netconf is a Protocol to manipulate configuration of network devices
  - IETF standard (RFCs)
  - Implemented by most vendors
  - TCP transport, SSH encryption, XML encoding
  - Uses RPC (remote procedure call) over SSH
  - Client/server communication (the server is the network device)
  - Server default port must be 830 and should be configurable (RFC 6242)

NETCONF PROTOCOL

- To enable the NETCONF service on the default port (830) on your devices

```
lab@ex4200-1# set system services netconf ssh
lab@ex4200-1# commit
```

- In order to enable NETCONF using another port, use this junos command

```
lab@ex4200-1# set system services netconf ssh port port-number
```

- You might want to create another user on your devices for PyEZ (to trace PyEZ activities) (don't do it in this training).

CONNECT TO DEVICES, RETRIEVE FACTS WITH PYEZ

CONNECT TO DEVICES AND RETRIEVE FACTS

- Let me execute this python program (print_facts.py) from the server. It prints the hostname and junos version for a list of devices defined into the program :

```
python facts/print_facts.py
the device ex4200-1 is a EX4200-24T running 12.2R2.4
the device ex4200-2 is a EX4200-24T running 12.3R11.2
the device ex4200-3 is a EX4200-24T running 12.3R11.2
the device ex4200-4 is a EX4200-24T running 12.3R11.2
```

- It also write the output here

```
more my_devices_inventory.txt
```

LAB 11 - IMPORT THE CLASS DEVICE

- Import the class Device from the package PyEZ.
- The class Device is defined in the module device (device.py) in the package jnpr.junos.
- The class Device provides methods:
  - For connecting to devices
  - For retrieving facts (such as software version, serial number, …) from the devices

```
>>> from jnpr.junos import Device
# Verify that the Device class has been loaded
>>> dir()
```

LAB 11 - INSTANTIATE THE CLASS DEVICE

- Instantiate the class Device by declaring a variable (a_device) and calling the class Device passing arguments (your device credentials). This assigns the returned value (the newly created object) to the variable a_device. Example for ex4200-13

```
>>> a_device=Device (host="172.30.179.113", user="pytraining", password="Poclab123")
```

- The object a_device is an instance of the class Device

```
>>> type (a_device)
<class 'jnpr.junos.device.Device'>
>>> a_device
Device(172.30.179.113)
```

LAB 11 - LIST AVAILABLE METHODS AND PROPERTIES

- List the available methods and properties for the object a_device.
  - Some methods are open, close, …
  - Some properties are facts, …

```
>>> dir(Device)
```

LAB 11 - GET HELP WITH THE CLASS DEVICE

- Get help on the object a_device

```
>>> help(Device)
```

- Get help on a method or property of the class Device (example with the method close)

```
>>> help(Device.close)
```

LAB 11 – METHODS AND PROPERTIES IN THE CLASS DEVICE

- Use the method open to connect to the device

```
>>> a_device.open()
Device(172.30.179.113)
```

- To get the properties of the object a_device

```
>>> a_device.user
'pytraining'
>>> a_device.connected #check if the connection with your switch is still open
True
```

- Use the method close the connection to the device

```
>>> a_device.close()
>>> a_device.connected
False
```

LAB 12 - FACTS

- By default, device facts (such as software-version, serial-number, etc.) are retrieved when the connection is established.
- Facts is a property defined in the class Device. This is a dictionary. This command returns the facts.

```
>>> from jnpr.junos import Device
>>> a_device=Device (host="172.30.179.113", user="pytraining", password="Poclab123")
>>> a_device.open()
>>> a_device.connected #check if the connection with your switch is still open
True
>>> type(a_device.facts)
<type 'dict'>
```

LAB 12 - FACTS

- Pretty print the facts with pprint

```
>>> from pprint import pprint as pp
>>> pp (a_device.facts)
{'2RE': False,
 'HOME': '/var/home/remote',
 'RE0': {'last_reboot_reason': '0x2:watchdog ',
         'mastership_state': 'master',
         'model': 'EX4200-24T, 8 POE',
         'status': 'OK',
         'up_time': '4 days, 3 minutes, 45 seconds'},
 'domain': 'poc-nl.jnpr.net',
 'fqdn': 'ex4200-1.poc-nl.jnpr.net',
 'hostname': 'ex4200-1',
 'ifd_style': 'SWITCH',
 'master': 'RE0',
 'model': 'EX4200-24T',
 'personality': 'SWITCH',
 'serialnumber': 'BM0210118154',
 'switch_style': 'VLAN',
 'vc_capable': True,
 'vc_mode': 'Enabled',
 'version': '12.2R2.4',
 'version_RE0': '12.2R2.4',
 'version_info': junos.version_info(major=(12, 2), type=R, minor=2, build=4)}
```

LAB 12 - FACTS

- Select some device facts

```
>>> a_device.facts["hostname"]
'ex4200-1'
>>> a_device.facts["version"]
'12.2R2.4'
>>> a_device.facts["version"]=="14.1R1.2"
False
```

LAB 12 – REVIEW THE FACTS PROGRAM

- The program prints the hostname and junos version for a list of devices defined in the program :

```
python facts/print_facts.py
the device ex4200-1 is a EX4200-24T running 12.2R2.4
the device ex4200-2 is a EX4200-24T running 12.3R11.2
the device ex4200-3 is a EX4200-24T running 12.3R11.2
the device ex4200-4 is a EX4200-24T running 12.3R11.2
```

- Have a look at output file.

```
more my_devices_inventory.txt
```

- Have a look at the program.

```
more facts/print_facts.py
```

CONFIGURATION MANAGEMENT WITH PYEZ

IMPORT THE CLASS CONFIG

- PyEZ provides us the necessary pieces of code to automate configuration deployment
- Import the class Config from the module config.py in the utils package

```
>>>from jnpr.junos.utils.config import Config
```

- call the dir function without argument to get the list of the names defined in the current scope. The class Config is now in the current scope.

```
>>>dir()
```

METHODS DEFINED IN THE CLASS CONFIG

- List the available methods for the class Config

```
>>> dir(Config)
```

- Some methods for the class Config:
  - Load: apply changes into the candidate conf
  - Pdiff: display conf changes between active and candidate
  - Commit-check
  - Commit: commit a candidate conf
  - Rollback
  - Lock: lock the candidate config
  - Unlock: unlock the candidate config

GET HELP WITH THE CLASS CONFIG

- Get help on the class Config

```
>>> help(Config)
```

- Get help on the Config's methods.
  - Example with method lock

```
>>> help(Config.lock)
```

INSTANTIATE THE CLASS CONFIG

- Define the candidate configuration.
  - Instantiate the class Config by declaring a variable (cfg) and calling the class passing an argument (a_device).
  - This assigns the returned value (the newly created object) to the variable cfg.
    - cfg is the candidate configuration for the device a_device

```
>>> a_device.connected #check if the connection with your switch is still open
True
>>> cfg = Config(a_device)
>>> type (cfg)
<class 'jnpr.junos.utils.config.Config'>
```

CHANGE THE CANDIDATE CONFIGURATION

- There are different ways to load changes to the candidate configuration. Lets see some of them here:

```
>>> cfg.load("set interfaces ge-0/0/23 description PyEZ", format='set')
<Element load-configuration-results at 0x7f77c8431ef0>

>>> #conf is a variable. It's a string.
>>> conf='''set vlans vlan-927 vlan-id 927
set vlans vlan-927 description "created with python"'''
>>> print conf
set vlans vlan-927 vlan-id 927
set vlans vlan-927 description "created with python"
>>> cfg.load(conf, format='set')
<Element load-configuration-results at 0x7f77c8431560>

>>> # confjunos.conf is a file with junos commands with the format set that define vlan 911
>>> cfg.load(path="configuration_management/confjunos.conf", format='set')
<Element load-configuration-results at 0x7f77c84317a0>
```

COMPARE CONFIGURATIONS

- Compare the candidate configuration and the active configuration (or a provided rollback) with the method pdiff. Examples:

```
>>> cfg.pdiff()
[edit interfaces]
+   ge-0/0/23 {
+       description PyEZ;
+   }
[edit vlans]
+   vlan-911 {
+       description "created with python";
+       vlan-id 911;
+   }
+   vlan-927 {
+       description "created with python";
+       vlan-id 927;
+   }

>>> cfg.pdiff(rb_id=1)
```

ROLLBACK THE CANDIDATE CONFIGURATION

- Rollback the candidate configuration to either the last active or a specific rollback number with the method rollback. Examples:

```
>>> cfg.rollback()
>>> cfg.rollback(rb_id=1)
```

CONFIGURATION MANAGEMENT

- Commit a candidate configuration with the method commit. Some examples:

```
>>> cfg.commit()
>>> cfg.commit(confirm=2)
>>> cfg.commit(comment="from pyez")
>>> print (a_device.cli ("show system commit"))
```

LAB 13 – CONFIGURATION MANAGEMENT

- Use PyEZ to create vlans 222, 223, 224 on your device. If you need help, have a look to the next slide.

LAB 13 – CONFIGURATION MANAGEMENT

```
>>> from jnpr.junos import Device #import the class Device
>>> from jnpr.junos.utils.config import Config #import the class Config
>>>
>>> # Instantiate the class Device
>>> a_device=Device (host="172.30.179.113", user="pytraining", password="Poclab123")
>>> # lets use the parameter gather_facts=False with the method open
>>> a_device.open(gather_facts=False)
>>>
>>> # instantiate the class Config. Cfg is the candidate configuration for the device a_device
>>> cfg = Config(a_device)
>>>
>>> vlans='''set vlans vlan-222 vlan-id 222
set vlans vlan-223 vlan-id 223
set vlans vlan-224 vlan-id 224'''
>>> print vlans
set vlans vlan-222 vlan-id 222
set vlans vlan-223 vlan-id 223
set vlans vlan-224 vlan-id 224
>>>
>>> cfg.load(vlans, format='set') # load change on the candidate configuration
>>> cfg.pdiff() # print the diff between the current candidate and the active configuration
>>> cfg.commit() # commit the candidate configuration
```

CONFIGURATION MANAGEMENT WITH PYEZ, YAML AND JINJA2

MERGE A YAML FILE AND A JINJA2 FILE

- We discussed previously the benefits of Yaml and Jinja2
- Execute this Python program (conf_int_with_vlan.py)

```
python configuration_management/conf_int_with_vlan.py
```

- It prompts you for a device ip@. It connects to it.
- It configures a list of interfaces with a given vlan.
  - There is no junos command in the Python code. It uses a junos template defined in the "configuration_management/template_int_vlan.j2" file.
  - There is no variable definition in the Python program. It uses the "configration_management/list_int_vlan.yml" yaml file to get the vlan details and the list of interfaces.
  - It merges the yaml file and jinja2 file and applies the change to the candidate configuration.
- It then prints the change.

MERGE A YAML FILE AND A JINJA2 FILE

- Have a look at the yaml file (list_int_vlan.yml).
  - It has the variables definition

```
more configuration_management/list_int_vlan.yml
---
host_ports:
    - ge-0/0/9
    - ge-0/0/10
    - ge-0/0/16
    - ge-0/0/18

vlan:
    name: v14
    vlan_id: 14
```

MERGE A YAML FILE AND A JINJA2 FILE

- View the jinja2 file (template_int_vlan.j2).
  - It has the junos template
  - We can use {{vlan.name }} or {{ vlan['name'] }}. You can use a dot (.) in addition to the standard Python syntax ([]): Both are valid and do the same thing

```
pytraining@py-automation-master:~$ more configuration_management/template_int_vlan.j2
set vlans {{ vlan['name'] }} vlan-id {{ vlan['vlan_id'] }}
{%- for iface in host_ports %}
set interfaces {{ iface }} unit 0 family ethernet-switching port-mode access vlan
members {{ vlan['name'] }}
{%- endfor %}
```

MERGE A YAML FILE AND A JINJA2 FILE

- This is the Python program (configuration_management/conf_int_with_vlan.py).

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
import yaml
ip=raw_input("ip address of the device:")
a_device=Device (host=ip, user="pytraining", password="Poclab123")
a_device.open()
# cfg is the candidate configuration for a_device
cfg = Config(a_device)
# rollback any pending or uncommitted change
cfg.rollback()
# s is a string with the content of the file configuration_management/list_int_vlan.yml
s=open('configuration_management/list_int_vlan.yml').read()
# yaml.load transforms a string into a structure that python can use myvars is a python dictionary
myvars=yaml.load(s)
# cfg.load merges the junos template (Jinja2) and the variables dictionary (YAML) and updates the
candidate configuration
cfg.load(template_path='configuration_management/template_int_vlan.j2', template_vars=myvars,
format='set')
cfg.pdiff()
#cfg.commit()
cfg.rollback()
```

LAB 14 –USE PYEZ AND JINJA2 AND YAML TO ENABLE LLDP

- Use PyEZ with a .j2 file and a .yml file to enable lldp on a list of several interfaces.
- If you need help, please have a look at the next slides.

LAB 14 – JINJA2 AND YAML – ENABLE LLDP

- Use a yaml file for the variables definition. Here's an example (interfaces.yml)

```
more configuration_management/interfaces.yml
---
interfaces:
    - ge-0/0/1
    - ge-0/0/2
    - ge-0/0/3
```

LAB 14 – JINJA2 AND YAML – ENABLE LLDP

- Use a jinja2 file with the junos template to enable LLDP on some interfaces. Here's an example (template_lldp.j2)

```
more configuration_management/template_lldp.j2
{%- for interface in interfaces %}
set protocols lldp interface {{ interface }}
{%- endfor %}
```

LAB 14 – JINJA2 AND YAML – ENABLE LLDP

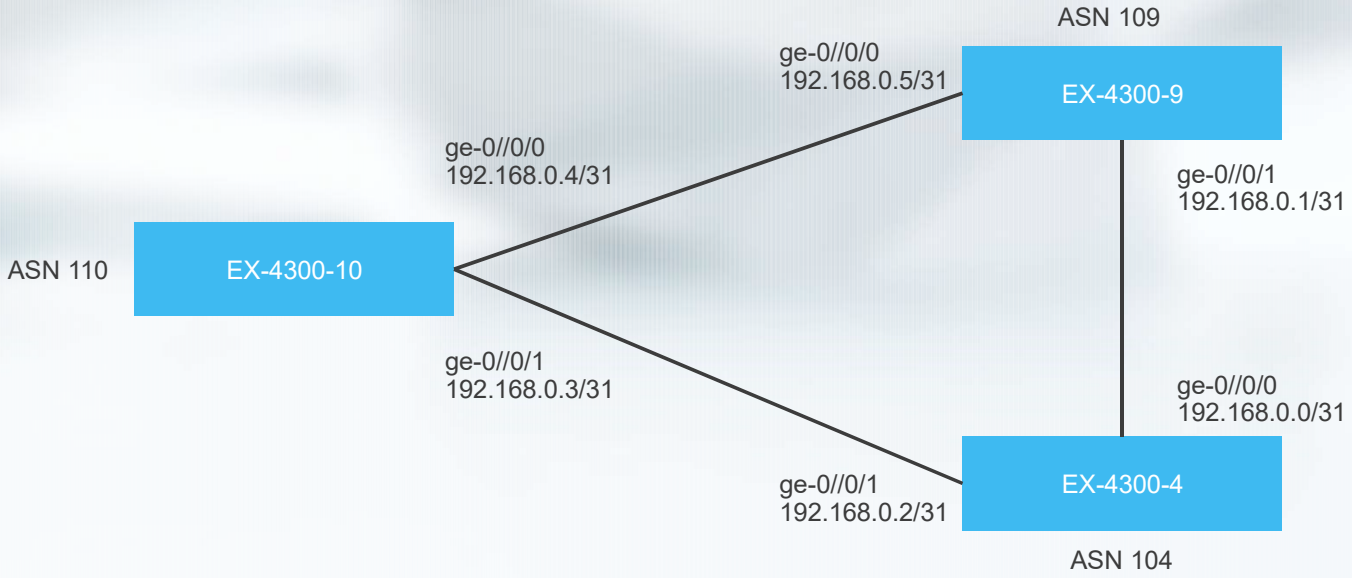
- Use a python program that merges the yaml file and jinja2 file and applies the change to the candidate configuration of your device

```
more configuration_management/enable_lldp.py
from jnpr.junos import Device # import the class Device
from jnpr.junos.utils.config import Config # import the class Config
import yaml
s=open('configuration_management/interfaces.yml').read() #s is a string
my_variables=yaml.load(s) # my_variables is a dictionary
a_device=Device (host="172.30.179.113", user="pytraining", password="Poclab123")
a_device.open()
cfg=Config(a_device) # cfg is the candidate configuration
cfg.rollback()
cfg.load(template_path='configuration_management/template_lldp.j2', template_vars=my_variables,
format='set')
cfg.pdiff()
cfg.commit()
```

USE AUTOMATION TO APPLY COMPLEX CONFIGURATION CHANGES ACROSS A LARGE NETWORK (RUN PHASE)

DEMO JINJA2 AND YAML

- Lets use PyEZ and Jinja2 and Yaml to apply a configuration change across a list of devices.
  - In this example we will configure some external bgp neighbors and all the required other details (bgp policies, interface configurations …)



DEMO JINJA2 AND YAML

- configuration_builder/variables.yml is a yaml file.
  - It defines the variables used in a jinja2 template.
  - This is a yaml list of 3 devices.
    - Each item of the list is a dictionary with some details for a device.
    - Very easy to add other devices
    - You can use another structure (i.e instead of a list of dictionaries) but in that case you'll need to parse it differently from the jinja2 file.

```
pytraining@py-automation-master:~$ more configuration_builder/variables.yml
```

DEMO JINJA2 AND YAML

- configuration_builder/template.j2 is a jinja2 template.
  - It defines BGP neighbors and other details. It uses the variables defined in the yaml file.

```
pytraining@py-automation-master:~$ more configuration_builder/template.j2
```

DEMO JINJA2 AND YAML

- configuration_builder/configuration_builder_2.py is a python script.
  - It uses the jinja2 template and yaml file to create a junos configuration file for each device defined in the yaml file.
  - It then use PyEZ to connect to the list of devices, and load and commit the configuration change

```
pytraining@py-automation-master:~$ more configuration_builder/configuration_builder_2.py
```

DEMO JINJA2 AND YAML

```
pytraining@py-automation-master:~$ python configuration_builder/configuration_builder_2.py
Start configuration building
generate config file for device ex4300-4 : conf_file_run_phase_ex4300-4.conf
generate config file for device ex4300-9 : conf_file_run_phase_ex4300-9.conf
generate config file for device ex4300-10 : conf_file_run_phase_ex4300-10.conf
done
applying the conf to the devices ...
configuration commited on ex4300-4
configuration commited on ex4300-9
configuration commited on ex4300-10
done
pytraining@py-automation-master:~$ ls | grep run
conf_file_run_phase_ex4300-10.conf
conf_file_run_phase_ex4300-4.conf
conf_file_run_phase_ex4300-9.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_ex4300-10.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_ex4300-4.conf
pytraining@py-automation-master:~$ more conf_file_run_phase_ex4300-9.conf
```

DEMO JINJA2 AND YAML

- Connect on the devices and double check if everything is correct
  - We will see later on in this training how to audit the network with automation instead of manually …

```
pytraining@py-automation-master:~$ ssh ex4300-10
pytraining@py-automation-master:~$ ssh ex4300-4
pytraining@py-automation-master:~$ ssh ex4300-9
```

```
pytraining@ex4300-10> show configuration | compare rollback 1
pytraining@ex4300-10> show system commit
pytraining@ex4300-10> show interfaces descriptions
pytraining@ex4300-10> show lldp neighbors
pytraining@ex4300-10> show bgp summary
pytraining@ex4300-10> show bgp neighbor
```

AUDIT MANAGEMENT WITH PYEZ

CLI, XML, RPC

- CLI is optimized for humans. CLI is not optimized for programs (difficult to parse CLI output from a program)
- Junos supports also XML (Extensible Markup Language) representation.
- XML is not optimized for humans (too much markup). XML can be manipulated by programs.

CLI, XML, RPC

- When you interact with a Junos device using its command-line interface, you actually interact with:

```
pytraining@ex4200-13> show version detail | match CLI
CLI release 14.1X53-D30.3 built by builder on 2015-10-02 09:52:33 UTC
```

- Then CLI passes the equivalent XML RPC to MGD

```
pytraining@ex4200-13> show version detail | match MGD
MGD release 14.1X53-D30.3 built by builder on 2015-10-02 12:38:35 UTC
```

- Then MGD get the data
- Then MGD returns the data to CLI in the form of an XML document.
- Then CLI converts back into a human readable format for display.

CLI, XML, RPC

- To display the output of a junos CLI command in XML format, append "| display xml" option to your CLI command.
- The "| display xml rpc" option provides you the RPC to get an XML encoded response
- Example with LLDP

```
pytraining@ex4300-9> show lldp neighbors
pytraining@ex4300-9> show lldp neighbors | display xml
pytraining@ex4300-9> show lldp neighbors | display xml rpc
```

TABLES AND VIEWS

- PyEZ (the jnpr.junos.op package) allows programmatic access to junos data on the devices (so you can audit your network programmatically instead of manually)
- It uses RPCs to get the data in an XML representation
- It then parses the XML response (so you don't need to worry about this)
- It transforms the output from XML into Python data structures (tables and views, kind of list of dictionaries) that you can easily use by Python.
  - It allows the junos data to be presented using python data structures
  - No need to parse XML in your Python code
  - This enables "pythonic" access to junos data
- PyEZ uses YAML to create tables and views.

TABLES AND VIEWS

- jnpr/junos/op directory is in your packages directory (/usr/lib/python2.7/dist-packages/)
- Available for many topics (vlan information, lldp neighbors information, isis adjacencies, route table, arp table, …)
- If the one you are looking for is missing, you can easily add your own (bgp neighbors, ...)

TABLES AND VIEWS – EXAMPLE WITH LLDP

- Enter the JUNOS command “show lldp neighbors” on a device (ex4300-4 or ex4300-9 or ex4300-10).
  - Output is optimized for humans.
  - Output is difficult to handle from a program.

```
pytraining@ex4300-9> show lldp neighbors
Local Interface    Parent Interface    Chassis Id          Port info          System Name
ge-0/0/0           -                   4c:96:14:e6:5a:40   UPLINK to ex4300-9 ex4300-10
ge-0/0/1           -                   4c:96:14:e6:82:60   UPLINK to ex4300-9 ex4300-4
```

TABLES AND VIEWS – EXAMPLE WITH LLDP

- Use the JUNOS command “show lldp neighbors | display xml rpc” to know the equivalent RPC.
  - So the RPC to get the lldp neighbors with an XML representation is “get-lldp-neighbors-information”

```
pytraining@ex4300-9> show lldp neighbors | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/13.2X51/junos">
    <rpc>
        <get-lldp-neighbors-information>
        </get-lldp-neighbors-information>
    </rpc>
    <cli>
        <banner>{master:0}</banner>
    </cli>
</rpc-reply>
```

TABLES AND VIEWS – EXAMPLE WITH LLDP

- Display the output in XML

```
pytraining@ex4300-9> show lldp neighbors | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/13.2X51/junos">
    <lldp-neighbors-information junos:style="brief">
        <lldp-neighbor-information>
            <lldp-local-port-id>ge-0/0/0</lldp-local-port-id>
            <lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>
            <lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>
            <lldp-remote-chassis-id>4c:96:14:e6:5a:40</lldp-remote-chassis-id>
            <lldp-remote-port-description>UPLINK to ex4300-9</lldp-remote-port-description>
            <lldp-remote-system-name>ex4300-10</lldp-remote-system-name>
        </lldp-neighbor-information>
        <lldp-neighbor-information>
            <lldp-local-port-id>ge-0/0/1</lldp-local-port-id>
            <lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>
            <lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>
            <lldp-remote-chassis-id>4c:96:14:e6:82:60</lldp-remote-chassis-id>
            <lldp-remote-port-description>UPLINK to ex4300-9</lldp-remote-port-description>
            <lldp-remote-system-name>ex4300-4</lldp-remote-system-name>
        </lldp-neighbor-information>
    </lldp-neighbors-information>
    <cli>
        <banner>{master:0}</banner>
    </cli>
</rpc-reply>
```

First item

Second neighbor details

TABLES AND VIEWS – EXAMPLE WITH LLDP

- PyEZ uses the RPC get-lldp-neighbors-information to get the lldp neighbors in an XML representation.
- PyEZ parses the XML response for you. No need to take care about this. It uses the lldp Yaml file (see below) to transforms the JUNOS XML output into a structure that can be used easily by Python (much more easy than data in XML representation).
- PyEZ build a table (LLDPNeighborTable) (kind of list of dictionaries):
  - For each <lldp-neighbor-information> in the XML response (so for each item/neighbor), PyEZ build a LLDPNeighborView with the <lldp-local-port-id> and <lldp-remote-system-name>.

```
pytraining@py-automation-master:~$ more /usr/local/lib/python2.7/dist-packages/jnpr/junos/op/lldp.yml
---
LLDPNeighborTable:
  rpc: get-lldp-neighbors-information
  item: lldp-neighbor-information
  key: lldp-local-interface | lldp-local-port-id
  view: LLDPNeighborView

LLDPNeighborView:
  fields:
    local_int: lldp-local-interface | lldp-local-port-id
    remote_sysname: lldp-remote-system-name
```

PyEZ uses this RPC to get the data in XML

PyEZ parses the XML response. It searches this value in the XML tags. The XML element (defined with this tag) is an item

Each item needs to be uniquely identified. PyEZ uses this tag from the XML response for this.

For each item, PyEZ creates this view.

The View defines fields. We use friendly names for the fields.

PyEZ uses these tags to create the View.

LAB 15 – TABLES AND VIEWS – LLDP

- Please compare the switch output:
  - –show lldp neighbors | display xml
  - –show lldp neighbors | display xml rpc
- And the lldp Yaml file:
  - –more /usr/local/lib/python2.7/dist-packages/jnpr/junos/op/lldp.yml
- Please locate the following elements on both the switch output and the yaml file:
  - get-lldp-neighbors-information (this is the rpc)
  - lldp-neighbor-information (this is the details for one item/neighbor)
  - lldp-local-port-id
  - lldp-remote-system-name
- We need a key (lldp-local-port-id) to uniquely identify each item
- The View defines 'friendly' names for the fields of the item (local_int, remote_sysname)

TABLES AND VIEWS: DEMO WITH LLDP

- Let me execute this Python program.
  - It uses the op package from PyEZ for LLDP.

```
python tables_and_views/lldp_neighbor_status.py

LLDP neighbors of device 172.30.179.65 (hostname is ex4300-4):
interface me0 has this neighbor: mgmt-13
interface ge-0/0/0 has this neighbor: ex4300-9
interface ge-0/0/1 has this neighbor: ex4300-10

LLDP neighbors of device 172.30.179.95 (hostname is ex4300-9):
interface ge-0/0/0 has this neighbor: ex4300-10
interface ge-0/0/1 has this neighbor: ex4300-4

LLDP neighbors of device 172.30.179.96 (hostname is ex4300-10):
interface me0 has this neighbor: mgmt-13
interface ge-0/0/0 has this neighbor: ex4300-9
interface ge-0/0/1 has this neighbor: ex4300-4
```

TABLES AND VIEWS: DEMO WITH LLDP

- Import the class LLDPNeighborTable from the module jnpr.junos.op.lldp :

```
>>>from jnpr.junos.op.lldp import LLDPNeighborTable
```

- Instantiate the class LLDPNeighborTable by declaring a variable (lldp_neighbors) and calling the class LLDPNeighborTable passing an argument (a_device is a Device instance). This assigns the returned value (the newly created object) to the variable lldp_neighbors.

```
>>> lldp_neighbors=LLDPNeighborTable(a_device)
>>> type(lldp_neighbors)
<class 'jnpr.junos.factory.OpTable.LLDPNeighborTable'>
```

TABLES AND VIEWS: DEMO WITH LLDP

- Use the method get to retrieve the lldp information (XML encoded) from the Device instance (a_device) and build the LLDPNeighborTable instance (lldp_neighbors).
- The LLDPNeighborTable instance (lldp_neighbors) is a structure that can be used easily by Python.

```
>>> lldp_neighbors.get()
LLDPNeighborTable:ex4300-4: 3 items
```

TABLES AND VIEWS: DEMO WITH LLDP

- The LLDPNeighborTable instance (lldp_neighbors) is a structure that can be used easily by Python.
  - It becomes easy to manipulate LLDP data :

```
>>> lldp_neighbors[1].local_int
'ge-0/0/0'
>>> lldp_neighbors[1].remote_sysname
'ex4300-9'
>>>
>>> for lldp_neighbor in lldp_neighbors:
     print (lldp_neighbor.remote_sysname)

mgmt-13
ex4300-9
ex4300-10
```

TABLES AND VIEWS: DEMO WITH LLDP

```
more tables_and_views/lldp_neighbor_status.py
from jnpr.junos import Device
from jnpr.junos.op.lldp import LLDPNeighborTable
import yaml

my_list_of_devices=open('tables_and_views/devices.yml').read()
my_list_of_switches=yaml.load (my_list_of_devices)

for host in my_list_of_switches:
        switch = Device(host=host, user='pytraining', password='Poclab123')
        switch.open()
        print "\nLLDP neighbors of device " + host + " (hostname is " + switch.facts["hostname"]+ "):"
        lldp_neighbors=LLDPNeighborTable(switch)
        lldp_neighbors.get()
        for neighbor in lldp_neighbors:
                print "interface " + neighbor.local_int + " has this neighbor: " + neighbor.remote_sysname
```

LAB 16 – TABLES AND VIEWS WITH BGP

- Use PyEZ to create a script that prints BGP connection state.
- If you need help, you can review the previous example.
- the table and view definition for BGP is here:
  /usr/local/lib/python2.7/dist-packages/jnpr/junos/op/bgp.yml
- If you still need help, please have a look at the next slides.

LAB 16 – TABLES AND VIEWS WITH BGP

- For each device in a device list, this program prints:
  - The list of its BGP neighbors
  - The status of its BGP connections

```
python tables_and_views/bgp_states.py

status of BGP neighbors of device 172.30.179.65 (hostname is ex4300-4):
External BGP neighbor 192.168.0.1+57665 is Established (flap count is: 0)
External BGP neighbor 192.168.0.3+58699 is Established (flap count is: 0)

status of BGP neighbors of device 172.30.179.95 (hostname is ex4300-9):
External BGP neighbor 192.168.0.0+179 is Established (flap count is: 0)
External BGP neighbor 192.168.0.4+51736 is Established (flap count is: 0)

status of BGP neighbors of device 172.30.179.96 (hostname is ex4300-10):
External BGP neighbor 192.168.0.2+179 is Established (flap count is: 0)
External BGP neighbor 192.168.0.5+179 is Established (flap count is: 0)
```

LAB 16 - TABLES AND VIEWS WITH BGP

```
more tables_and_views/bgp_states.py
from jnpr.junos import Device
from jnpr.junos.op.bgp import *
import yaml

my_list_of_devices=open('tables_and_views/devices.yml').read()
my_list_of_switches=yaml.load (my_list_of_devices)

for element in my_list_of_switches:
    switch = Device(host=element, user='pytraining', password='Poclab123')
    switch.open()
    bgp=BGPNeighborTable (switch)
    bgp.get()
    print "\nswitch.facts["hostname"] + ":"
    for item in bgp:
        print item.type + "_" + item.neighbor + ": " + item.state + " (flaps: " + item.flap_count +")"
```

LAB 17 - OTHER TABLES AND VIEWS SCRIPTS

- There are more scripts available in the tables_and_views directory:

```
python tables_and_views/uplinks_and_downlinks_last_flap.py
more tables_and_views/uplinks_and_downlinks_last_flap.py

python tables_and_views/search_an_lldp_neighbor.py
more tables_and_views/search_an_lldp_neighbor.py

python tables_and_views/lldp_neighbor_status.py
more tables_and_views/lldp_neighbor_status.py

python tables_and_views/bgp_states.py
more tables_and_views/bgp_states.py

python tables_and_views/bgp_non_established.py
more tables_and_views/bgp_non_established.py
```

REST CALLS WITH PYTHON REQUESTS PACKAGE

REST APIs

- Many systems have REST APIs : JUNOS, Junos Space, Contrail, Openstack, NSX-V …
- You first need to have the REST API documentation for your system.
- Then you can use a graphical REST Client (browser add-on: REST Easy, RESTClient, Postman) to start playing with REST APIs and learn more about REST APIs.
  - Graphical REST clients are for humans.
  - If you need automation and programmatic access, you have to use a command line REST client.
- You can then use Python as a REST Client to handle REST Calls. It is easy to parse the REST servers answers if they use a json format (json format is a dictionary).

REST APIs ON JUNOS 15.1

- JUNOS 15.1 supports REST API to submit RPCs
  - You can read the database
  - You can use HTTP get and post methods to submit RPCs to the REST Server.
  - You can retrieve data in XML or JSON
- The documentation is here:
  - https://www.juniper.net/documentation/en_US/junos15.1/information-products/pathway-pages/rest-api/rest-api.pdf
- REST configuration is under “system services” (default port is 3000)
- REST Explorer is an optional tool (GUI) for testing
- JUNOS CLI ouput with “| display json” is also available

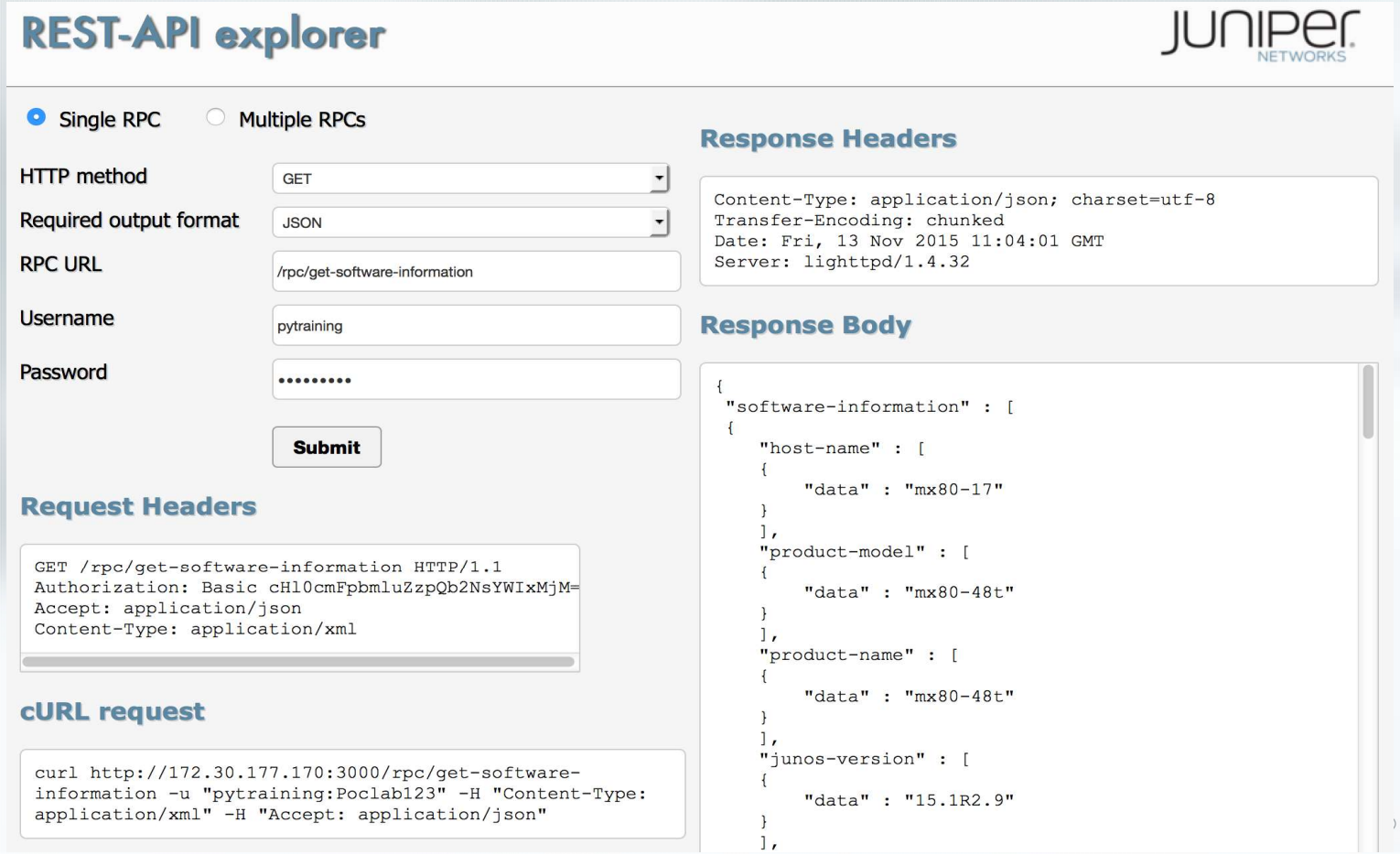
LAB 18 - REST APIs ON JUNOS 15.1

- We will use an MX running JUNOS 15.
  - 172.30.177.170/pytraining/Poclab123
  - Rest service with HTTP is enabled on the default port (3000).
  - Rest explorer is also enabled.

```
set system services rest http
set system services rest http rest-explorer
```

LAB 18 - REST EXPLORER ON JUNOS 15.1

- Use the REST explorer (http://172.30.177.170:3000)



LAB 18 - REST CALLS TO JUNOS WITH PYTHON

- The python program rest_basics/get_mx_software_information.py uses the JUNOS REST APIs to get some details regarding the device

```
pytraining@py-automation-master:~$ python rest_basics/get_mx_software_information.py
Software version: 15.1R2.9
Host-name: mx80-17
Product name: mx80-48t
```

REST CALLS TO JUNOS WITH PYTHON

- Authentication
  - The REST API uses HTTP Basic Authentication. all requests require a base 64 encoded username and password included in the Authorization header.

```
>>> # import the requests library
>>> import requests
>>> # import the class HTTPBasicAuth from module requests.auth. This class attaches HTTP
Basic Authentication to a request.
>>> from requests.auth import HTTPBasicAuth
>>>
>>> r=requests.get('http://172.30.177.170:3000/rpc/get-software-information',
auth=HTTPBasicAuth('pytraining', 'Poclab123'))
>>> r.status_code
200
>>> r.headers['Content-type']
'application/xml; charset=utf-8‘
>>>
```

REST CALLS TO JUNOS WITH PYTHON

- Lets add the HTTP request header Accept: application/json to have the REST server answer with a JSON representation instead of XML.

```
>>> my_headers = { 'Accept': 'application/json' }
>>> r = requests.get('http://172.30.177.170:3000/rpc/get-software-information',
auth=HTTPBasicAuth('pytraining', 'Poclab123'),headers=my_headers)
>>>
>>> r.headers['Content-type']
'application/json; charset=utf-8‘
>>> type(r.json())
<type 'dict'>
>>> from pprint import pprint as pp
>>> pp(r.json())
…
>>> print r.json()['software-information'][0]['product-name'][0]['data']
mx80-48t
>>> print r.json()['software-information'][0]['host-name'][0]['data']
mx80-17
>>> print r.json()['software-information'][0]['junos-version'][0]['data']
15.1R2.9
>>> '15.1R2.9' in r.content
True
```


JUNIPer
NETWORKS
THANK YOU!