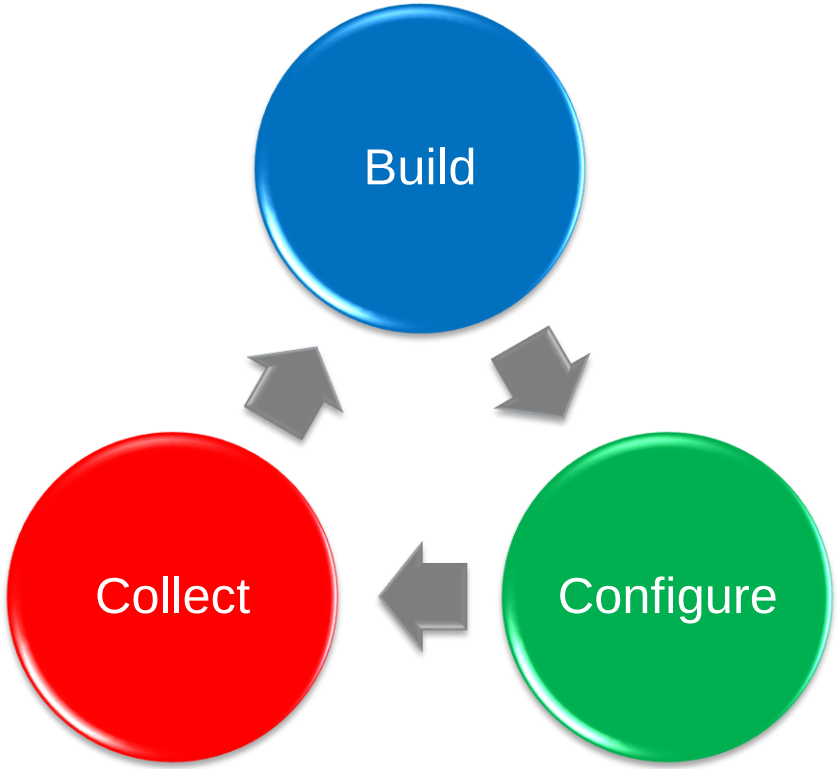
GETTING STARTED WITH PYEZ

Umberto Manferdini

NETWORK AUTOMATION

The ***Build*** phase centers around the initial design and installation of a network component



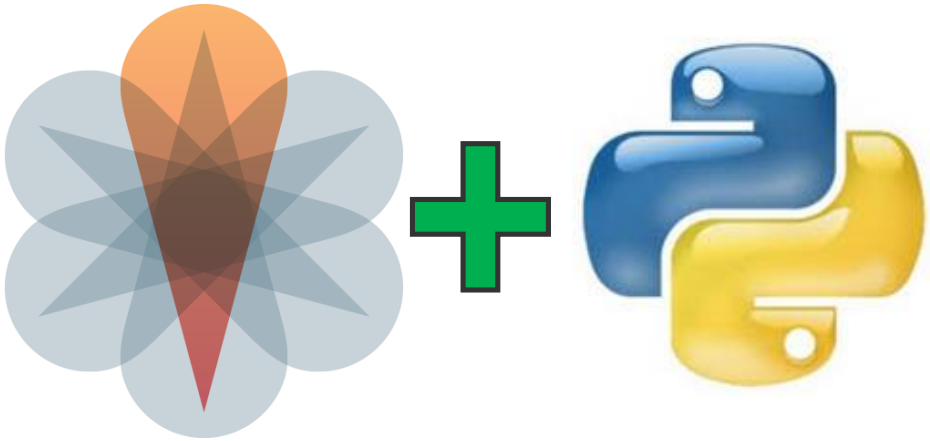
The ***Collection*** phase deals with automating the process of monitoring operational state of the platform and reacting on state conditions

The ***Configure*** phase covers methods to deploy on-demand configuration and software changes to the platform

PYTHON

- excellent for beginners, yet superb for experts
- highly scalable, suitable for large projects as well as small ones
- cross-platform
- simple

PYEZ

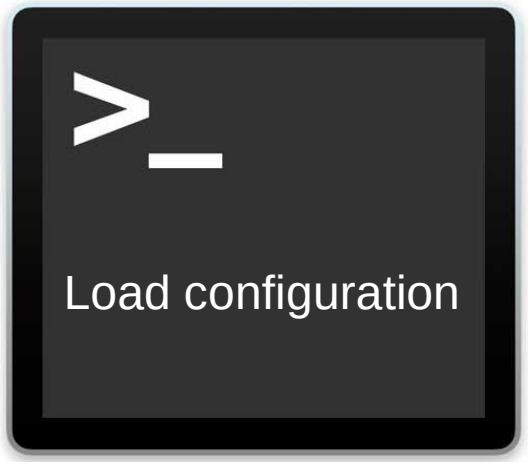


- Free Juniper Library to manage Junos Devices
- Available since 11.4
- “On box” since 16.1 release

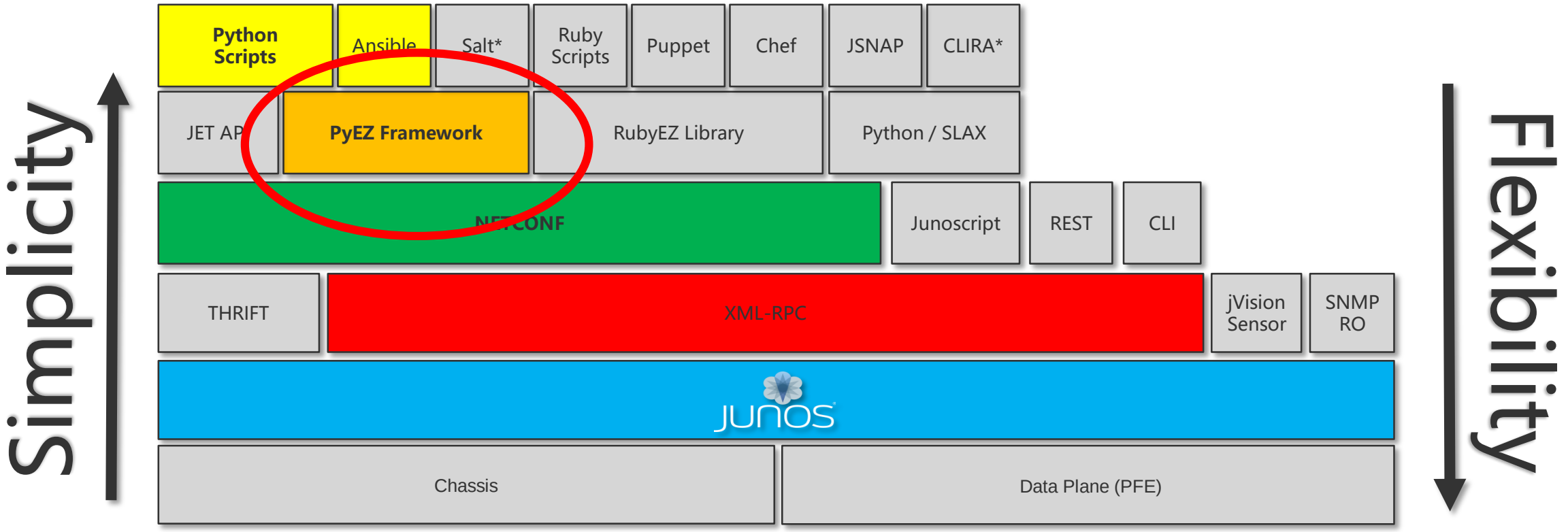
PYEZ USERS

*** Based on queries posted by their employees on Google Group**

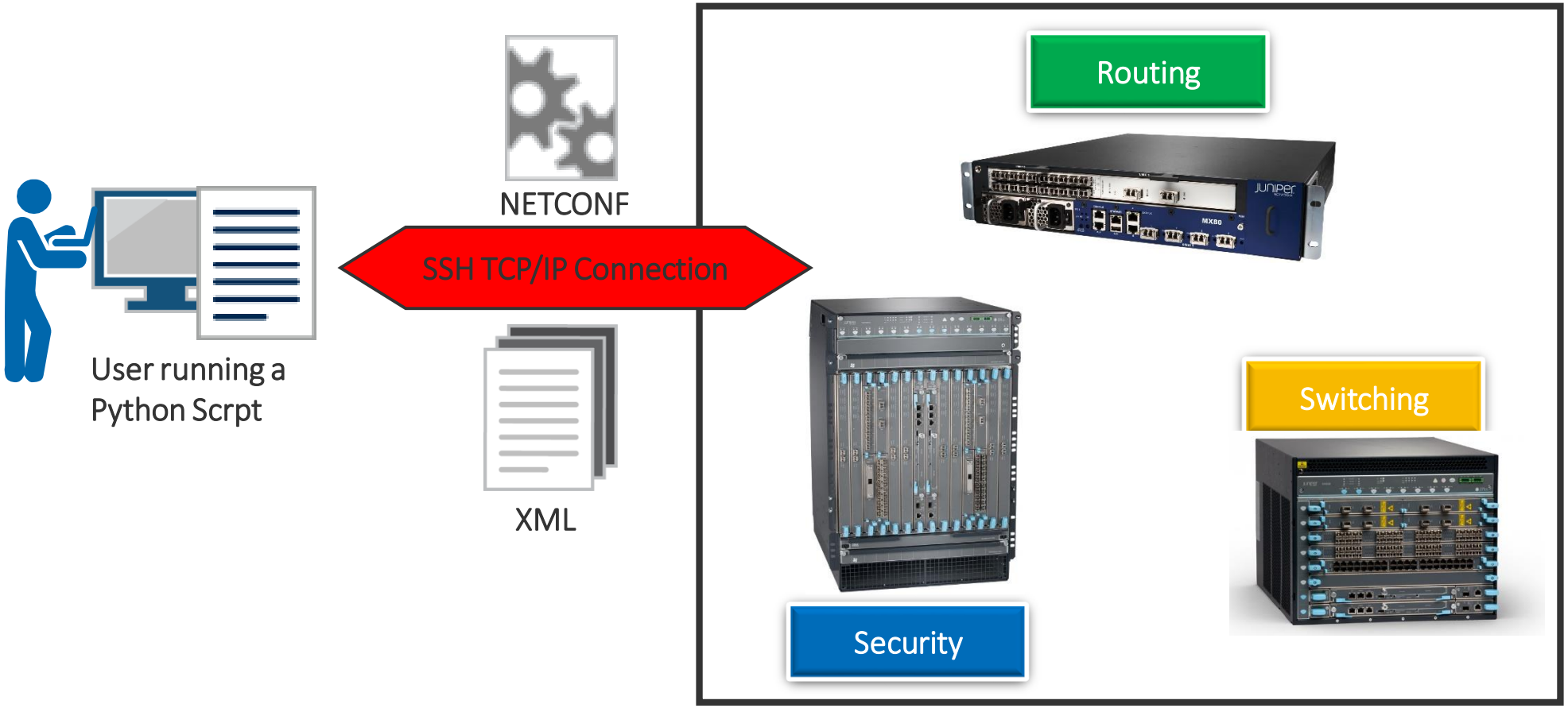
WHAT CAN I DO?



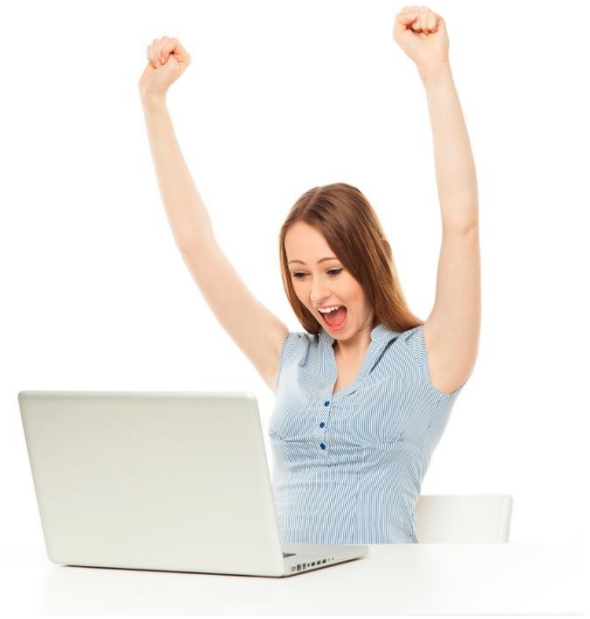
WHERE DOES PYEZ FIT?



HOW DOES IT WORK?



GETTING STARTED



Connect to a server running Linux

Download Python

Install PyEz (pip install junos-eznc)

Ready to go!

INSTALLING PYEZ ON UBUNTU

Follow these simple

```
Apt-get install python-pip python-dev libxml2-dev libxslt-dev libssl-dev libffi-dev
```

```
pip install junos-eznc
```

```
Successfully installed asn1crypto-0.24.0 bcrypt-3.1.4 cffi-1.11.5 cryptography-2.2.2
enum34-1.1.6 ipaddress-1.0.22 junos-eznc-2.1.7 lxml-4.2.1 ncclient-0.5.3 paramiko-2.4.1
pyasn1-0.4.2 pycparser-2.18 pynacl-1.2.1 pyserial-3.4 scp-0.10.2 six-1.11.0
```

Check here for other Operative Systems
https://www.juniper.net/documentation/en_US/junos-pyez/topics/task/installation/junos-pyez-server-installing.html

WORKING WITH PYEZ

PUSH configuration
UPGRADE device
READ data

…AND DEVICE MONITORING

show nat statistics
show interfaces statistics
show system hardware

Get results, parse data and use them as an input for further analysis

PREPARING THE DEVICES

Devices must be able to accept NETCONF over SSH connections

PyEz relies on it, thorugh the ncclient!

```
root@r1_re# show system services
netconf {
    ssh;
}
```

CONNECT TO A DEVICE

```
>>> from jnpr.junos import Device
>>> from pprint import pprint

>>> dev =
Device(host='10.92.35.2',user='root',password
='Embe1mpls')
>>> type(dev)
<class 'jnpr.junos.device.Device'>

>>> dev.open()
Device(10.92.35.2)

>>> dev.user
'root'
>>> dev.connected
True
```

- Need SSH credentials
- A PyEz Device object is needed

WHAT CAN A DEVICE DO?

See all available methods for object Device:

```
>>> dir(dev)
['ON_JUNOS', 'Template', '__class__', '__delattr__', '__dict__', '__doc__', '__enter__',
'__exit__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_auth_password', '_auth_user',
'_auto_probe', '_conf_auth_user', '_conf_ssh_private_key_file', '_conn', '_connected',
'_fact_style', '_gather_facts', '_hostname', '_j2ldr', '_manages', '_nc_transform',
'_norm_transform', '_normalize', '_ofacts', '_port', '_rpc_reply', '_ssh_config',
'_ssh_private_key_file', '_sshconf_lkup', '_sshconf_path', 'auto_probe', 'bind', 'cli',
'cli_to_rpc_string', 'close', 'connected', 'display_xml_rpc', 'execute', 'facts',
'facts_refresh', 'hostname', 'logfile', 'manages', 'master', 'ofacts', 'open',
'password', 'port', 'probe', 're_name', 'rpc', 'timeout', 'transform', 'uptime', 'user']
```

GETTING HELP!

You can get help about how to use some methods:

```
>>> help(dev.facts)
Help on _FactCache in module jnpr.junos.factcache object:

class _FactCache(_abcoll.MutableMapping)
 |  A dictionary-like object of read-only facts about the Junos device.
 |
 |  These facts are accessed as the `facts` attribute of a `Device` object
 |  instance. For example, if `dev` is an instance of a `Device` object,
 |  the hostname of the device can be accessed with::
 |
 |      dev.facts['hostname']
...
```

GATHERING FACTS

```
>>> dev.facts
{'2RE': False,
 'HOME': '/root',
 'RE0': {'last_reboot_reason': 'Router rebooted
after a normal shutdown.',
         'mastership_state': 'master',
         'model': 'RE-VMX',
         'status': 'OK',
         'up_time': '2 days, 3 hours, 52
minutes, 5 seconds'},
 'RE1': None,
 ...
 'version': '16.1R3.10',
 'version_RE0': '16.1R3.10',
 'version_RE1': None,
 'version_info': junos.version_info(major=(16,
1), type=R, minor=3, build=10),
 'virtual': True}
```

- Automatically downloaded when connecting to the device
- General information about hardware and software
- It is a python dictionary
- Optionally, you may avoid pyez to gather this information

EXPLORING FACTS

```
>>> dev.facts.keys()
['domain', 'hostname_info', 'version_RE1', 'version_RE0', 're_master', '2RE',
'serialnumber', 'vc_master', 'RE_hw_mi', 'HOME', 're_info', 'srx_cluster_id',
'hostname', 'virtual', 'version', 'master', 'vc_fabric', 'personality',
'srx_cluster_redundancy_group', 'version_info', 'srx_cluster', 'vc_mode', 'vc_capable',
'ifd_style', 'model_info', 'RE0', 'RE1', 'fqdn', 'junos_info', 'switch_style', 'model',
'current_re']

>>> dev.facts['version']
'16.1R3.10'

>>> dev.facts['re_info']['default']['default'].keys()
['status', 'last_reboot_reason', 'model', 'mastership_state']

>>> dev.facts['re_info']['default']['default']['mastership_state']
'master'
```

INTERACTING WITH DEVICES CONFIGURATION

The Config class is used to configure devices

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

>>> dir(Config)
['__class__', '__delattr__', '__dict__', '__doc__', '__enter__', '__exit__',
'__format__', '__getattribute__', '__hash__', '__init__', '__module__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'commit',
'commit_check', 'dev', 'diff', 'load', 'lock', 'pdiff', 'rescue', 'rollback',
'rpc', 'unlock']
```

It has several methods to deal with the device: commit, perform a “show | compare”, run a rpc and so on

PUSHING A CONFIGURATION

Device is initially empty:

```
root@r1_re# show interfaces
```

We load a configuration line via PyEz:

```
>>> from jnpr.junos.utils.config import Config
>>> cfg=Config(dev)
>>> cfg.load("set interfaces ge-0/0/0 description PyEz", format='set')
```

We check configuration was loaded:

```
root@r1_re# show interfaces
ge-0/0/0 {
    description PyEz;
}
```

LOAD A CONFIGURATION FILE

We can also load configuration from a file:

```
root@ubuntu:~# cat loadex.txt
set interfaces ge-0/0/1 description PyEz2

>>> cfg.load(path="loadex.txt", format='set')
<Element load-configuration-results at 0x7f706105ed88>

root@r1_re# show interfaces
ge-0/0/0 {
    description PyEz;
}
ge-0/0/1 {
    description PyEz2;
}
```

WHAT DID I CHANGE?

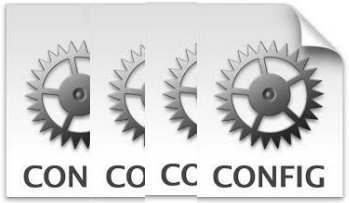
We can verify what we added by emulating what is normally done via “show | compare”:

```
>>> cfg.pdiff()

[edit]
+  interfaces {
+      ge-0/0/0 {
+          description PyEz;
+      }
+      ge-0/0/1 {
+          description PyEz2;
+      }
+  }
```

ROLLING…BACK!

Active configuration



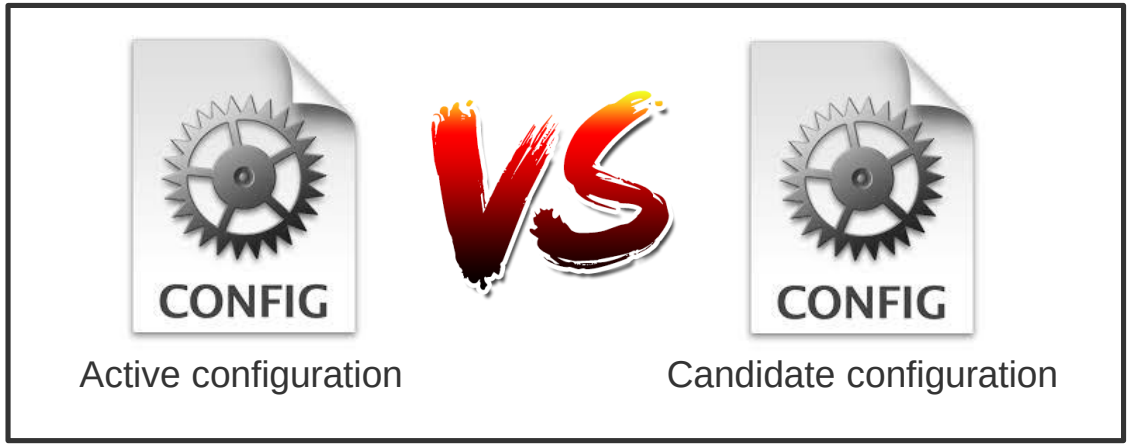
Previous configurations
Up to 20 files

```
cu = Config(dev)
cu.rollback(rb_id=1)
```

- Junos keeps up to the last 20 committed configurations
- It is possible to re-load one of those configurations
- This operation is called rollback
- PyEz has a rollback method to perform this

WHAT CHANGED?

Show differences with our candidate configuration



Show differences with a previous configuration

BACK TO THE PAST

We can rollback the configuration:

```
>>> cfg.rollback()
True
```

We re-run pdiff and this time we expect to see no differences as the active and the candidate configuration are the same:

```
>>> cfg.pdiff()
None
```

MIXING UP

We create a new configuration line:

```
>>> s='set interfaces ge-0/0/0 description newPyEz‘
```

We load it:

```
>>> cfg.load(s,format='set')
<Element load-configuration-results at 0x7f706105fcf8>
```

We check the differences:

```
>>> cfg.pdiff()

[edit]
+  interfaces {
+      ge-0/0/0 {
+          description newPyEz;
+      }
+  }
```

We commit:

```
>>> cfg.commit()
True
```

Now difference should be empty:

```
>>> cfg.pdiff()
None
```

COMMENTING A COMMIT

We restore everything:

```
>>> cfg.rollback(rb_id=1)
True
>>> cfg.pdiff()
[edit]
-  interfaces {
-      ge-0/0/0 {
-          description newPyEz;
-      }
-  }
>>> cfg.commit(comment='via pyez')
True
```

We see our commit in the device logs:

```
root@r1_re:~ # cat /var/log/messages | grep COMMIT
May  2 05:54:20  r1_re mgd[15125]: UI_COMMIT: User 'root' requested 'commit' operation
(comment: via pyez)
```

COMMIT AS YOU WISH

We have several ways to perform a commit:

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

LOAD TEXT CONFIGURATION

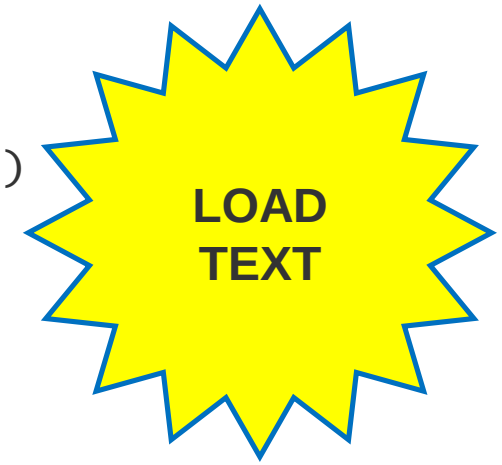
```
from jnpr.junos import Device #import the class Device
from jnpr.junos.utils.config import Config #import the class Config

a_device=Device (host="10.92.35.2", user="root", password="Embe1mpls")
a_device.open(gather_facts=False)

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'''

cfg.load(vlans, format='set')
cfg.pdiff()
cfg.commit()
```



LOAD XML CONFIGURATION

```
...
data = """<policy-options>
          <policy-statement action="delete">
            <name>F5-in</name>
            <term>
                <name>test</name>
                <then>
                    <accept/>
                </then>
            </term>
            <from>
                <protocol>mpls</protocol>
            </from>
        </policy-statement>
        </policy-options>"""
cu.load(data)
cu.commit()
```

TEMPLATES

```
interfaces {
    {% for item in interfaces %}
    {{ item }} {
        description "{{ description }}";
        unit 0 {
            family {{ family }};
        }
    } {% endfor %}
}
```

JINJA

```
data= {
    'interfaces': ['ge-1/0/1', 'ge-1/0/2'],
    'description': 'MPLS interface',
    'family': 'mpls'
}
```

```
interfaces:
  - ge-1/0/1
  - ge-1/0/2
description: 'MPLS interface'
family: mpls
```

YAML

```
interfaces {
    ge-1/0/1{
        description “MPLS interface”;
        unit 0 {
            family mpls;
        }
    }
 ge-1/0/2{
        description “MPLS interface”;
        unit 0 {
            family mpls;
        }
    }
}
```

AUTOMATING VPN PROVISION: YAML & JINJA2

```
root@ubuntu:~# cat vpn.yaml
instances:
    - name: VPN1
      type: vrf
      rd: 192.168.100.1:21
      rt: target:64555:601
    - name: VPN2
      type: vrf
      rd: 192.168.100.1:22
      rt: target:64555:602

root@ubuntu:~# cat vpn.j2
{%- for x in instances %}
set routing-instances {{ x.name }} instance-type {{ x.type }}
set routing-instances {{ x.name }} route-distinguisher {{ x.rd }}
set routing-instances {{ x.name }} vrf-target {{ x.rt }}
{%- endfor %}
```

BASELINE EXAMPLE

```
#! /usr/bin/python

#Import Required Modules
from jnpr.junos import Device
from jnpr.junos.exception import *
from jnpr.junos.utils.config import Config
import sys
import yaml

a_device=Device (host='10.92.35.2', user="root", password="Embe1mpls")
a_device.open()

cfg = Config(a_device)

yfile = "vpn.yaml"
s=open(yfile).read()
myvars=yaml.load(s)

jfile = "vpn.j2"
cfg.load(template_path=str(jfile), template_vars=myvars, format='set')

cfg.pdiff()
cfg.commit()
```

We run the script and check VRFs are provisioned:

```
root@ubuntu:~/vpn_example_yaml_jinja# python load.py

[edit]
+  routing-instances {
+      VPN1 {
+          instance-type vrf;
+          route-distinguisher 192.168.100.1:21;
+          vrf-target target:64555:601;
+      }
+      VPN2 {
+          instance-type vrf;
+          route-distinguisher 192.168.100.1:22;
+          vrf-target target:64555:602;
+      }
+  }
```

MIXING ALL TOGETHER!

```
jfile = “example.j2“
yfile = “example.yaml

dev = Device(host=address, user="root", password="Embe1mpls")
dev.open()
cfg = Config(dev)

s=open(yfile).read()
myvars=yaml.load(s)
cfg.load(template_path=str(jfile), template_vars=myvars, format='set')

if cfg.commit_check():
     if cfg.commit:
         cfg.commit(timeout=300)
         print 'Successfully Committed'
     else:
         print 'Commit Failed'
     else:
         print 'Commit Check Failed'
dev.close()
```

Have your jinja and yaml files

Connect to the device and load the configuration

Load the YAML data structure and merges it with the jinja template to create the configuration

Commit the configuration

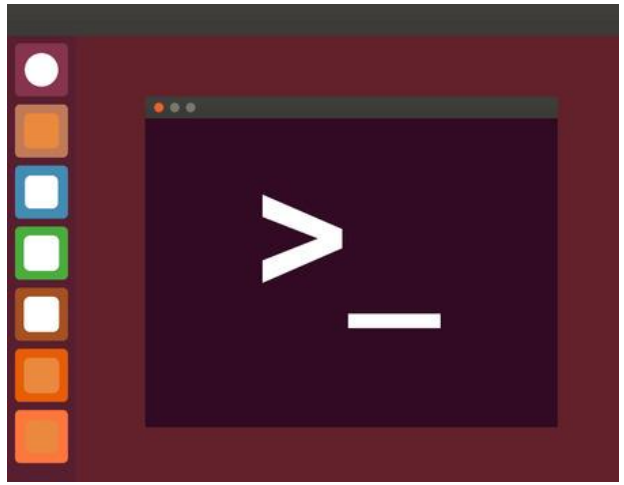
CLI COMMANDS

```
>>> from jnpr.junos import Device

>>> dev =
Device(host='10.92.35.2',user='root
',password='Embe1mpls')
>>> dev.open()
Device(10.92.35.2)

>>> print dev.cli("show version",
warning=False)
Hostname: r1_re
Model: mx960
Junos: 16.1R3.10
JUNOS OS Kernel 64-bit
[20160927.337663_builder_stable_10]
...
```

- We can run CLI commands directly
- Output returned as a string
- Not all the CLI commands will work
  - Try and see!



FROM A COMMAND TO A RPC

In Junos, any operational command has a corresponding RPC

https://apps.juniper.net/xmlapi/operTags.jsp

https://apps.juniper.net/xmlapi/

The RPC can then be used in PyEz to retrieve data

Data is obtained by default in XML format.

Optionally, we can get result in JSON format.

RPCS AND ARGUMENTS

```
root@r1_re> show interfaces terse ge-0/0/0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R3/junos">
    <rpc>
        <get-interface-information>
                <terse/>
        </get-interface-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

get-interface-information

Usage

```
<usage>
   <rpc>
     <get-interface-information>
       <routing-instance>routing-instance</routing-instance>
       <satellite-device>satellite-device</satellite-device>
       <ifl-class>ifl-class</ifl-class>
       <aggregation-device></aggregation-device>
       <zone></zone>
       <extensive></extensive>
       <statistics></statistics>
       <media></media>
       <detail></detail>
       <terse></terse>
       <brief></brief>
       <descriptions></descriptions>
       <snmp-index>snmp-index</snmp-index>
       <switch-port>switch-port</switch-port>
       <interface-name>interface-name</interface-name>
     </get-interface-information>
   </rpc>
 </usage>
```

RUNNING A RPC

```
>>>
ifs=dev.rpc.get_interface_information(interface_n
ame='ge-0/0/0', terse=True)

>>> print etree.tostring(ifs)
<interface-information style="terse">
<physical-interface>
<name>
ge-0/0/0
</name>
<admin-status>
up
</admin-status>
<oper-status>
up
</oper-status>
</physical-interface>
</interface-information>


root@r1_re# run show interfaces terse ge-0/0/0
Interface               Admin Link Proto    Local
ge-0/0/0                up    up
```

- Run the RPC with required arguments
- Result is returned in XML format
- Content is identical to the corresponding CLI operational command
  - It is simply XML formatted

ROUTING TABLE

```
>>> from jnpr.junos.op.routes import RouteTable

>>> tbl = RouteTable(dev)
>>> tbl.get()
RouteTable:10.92.35.2: 9 items

>>> tbl.keys()
['10.0.0.0/8', '10.92.32.0/19', '10.92.35.2/32',
'128.92.35.2/32', '172.16.0.0/12']

>>> tbl['172.16.0.0/12'].keys()
['nexthop', 'age', 'via', 'protocol']
>>> tbl['172.16.0.0/12']['protocol']
'Static'
>>> tbl['172.16.0.0/12']['via']
'fxp0.0'
>>> tbl['172.16.0.0/12']['nexthop']
'10.92.63.254'
>>> tbl['172.16.0.0/12']['age']
190388
```

- You can access the routing table
- Use RouteTable object
- Allows you to see much information for each route
  - Protocol
  - Next-hop
  - Output interface
  - Age
- Relies on PyEz tables & views
  - You an build your own in order to include your desired information

GET CONFIG

```
from lxml import etree

dev = Device(host='xxxx', user='demo',
password='demo123', gather_facts=False)
dev.open()

cnf = dev.rpc.get_config()
print etree.tostring(cnf)
```

- Get full configuration
- XML format

COPYING FILES

```
from jnpr.junos.utils.scp import SCP

dev = Device(host='xxxx', user='demo',
password='demo123')
dev.open()

with SCP(dev, progress=True) as scp:
     scp.get('/var/tmp/nitin.log','info.txt')
dev.close()
```

- PyEz supports SCP
- Useful to copy or retrieve files from devices
- Might be used to load Junos images
- Could allow us to download log files

IT IS UPGRADE TIME!

New Junos 17.1

Junos 14.3

Junos 17.1

Junos 15.1

Junos 15.2

PYEZ APPROACH TO RELEASE UPGRADE

Junos upgrade package

STEP 1: check if device needs upgrade

STEP 2: copy upgrade package to device

STEP 3: install new Junos version

UPGRADE MANY DEVICES, ONLY WHEN NEEDED!

```
for x in devices:

    dev = Device(host=hosts[x], user="root", password="Embe1mpls")
    dev.open()

    if dev.facts['version']!=args.ver:

        sw = SW(dev)
        try:
            ok = sw.install(package=package, remote_path=remote_path, progress=True, validate=validate)
        except Exception as err:
            msg = 'Unable to install software, {0}'.format(err)
            ok = False

        if ok is True:
            sw.reboot()
        else:
            print 'Unable to install software.'
    else:
        print "Device already has the target version, No need for upgrade!"

    dev.close()
```

This command copied the package into the device and starts the upgrade procedure

If there is no need, I do not upgrade!

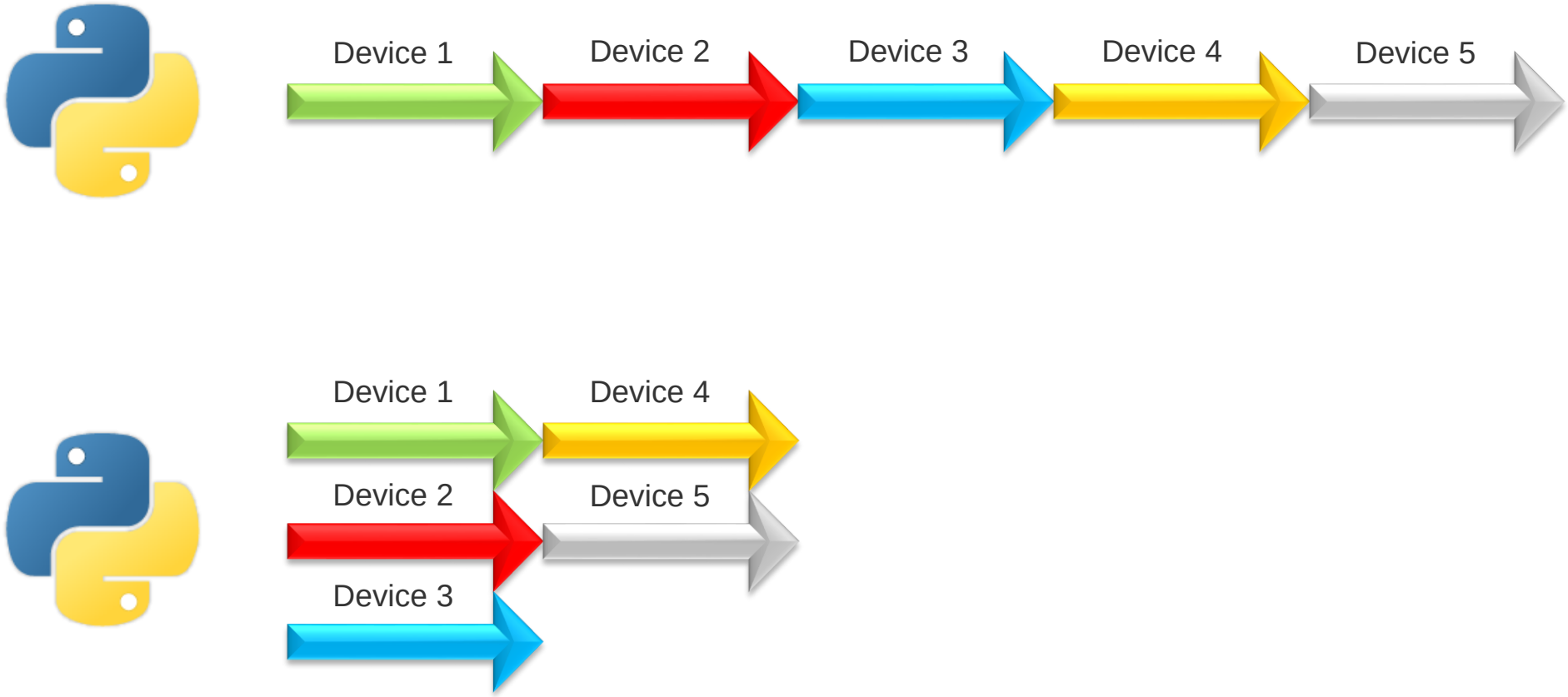
WHAT IF SOMETHING GOES WRONG?

PyEz is ready to handle errors and unexpected behaviors:

- Unable to connect to the device
- Unable to lock the configuration
- Unable to commit the configuration

Everything is managed by PyEz Python exceptions!

SEQUENTIAL EXECUTION VS PARALLEL EXECUTION



CODING MULTIPROCESSING

- Python supports multiprocessing
- Can run the same script on different devices in parallel
- Overall running time dramatically decreases
- Internal tests showed a 3.5 decrease factor
- Need to understand “how much to parallelize”
  - Not an exact science
  - Needs tests
  - Varies depending on the server used to run the script

```
#! /usr/bin/python

from multiprocessing import Pool

def f(x):
    #our function
    #connecting to devices
    #retrieving options
    return 1

if __name__ == '__main__':
    p = Pool(10)
    l = […] #arguments
    p.map(f, l)
```

everywhere