YAML

YAML is extensively used for Ansible.

* **yaml.dump**(obj) => **str** type output, therefore **f.write**(str)
* **yaml.load**(fp) => **list** type output, therefore some\_list = yaml.load(fp)

|  |
| --- |
| In [**1**]: **import** **yaml**  In [**3**]: my\_list = range(8)  In [**4**]: my\_list.append("whatever")  In [**5**]: my\_list.append("Hello")  In [**7**]: my\_list.append({})  In [**8**]: my\_list[-1]  Out[**8**]: {}  In [**9**]: my\_list[-1]['ip\_addr'] = '10.10.10.1'  In [**11**]: my\_list[-1]['attribs'] = range(7)  In [**12**]: my\_list  Out[**12**]:  [0,  1,  2,  3,  4,  5,  6,  7,  'whatever',  'Hello',  {'attribs': [0, 1, 2, 3, 4, 5, 6], 'ip\_addr': '10.10.10.1'}]  In [**13**]: len(my\_list)  Out[**13**]: 11  In [**14**]: **yaml.dump**(my\_list) *# type(yaml.dump(my\_list)) =>* ***str***  Out[**14**]: '- 0\n- 1\n- 2\n- 3\n- 4\n- 5\n- 6\n- 7\n- whatever\n- Hello\n- attribs: [0, 1, 2, 3, 4, 5, 6]\n ip\_addr: 10.10.10.1\n'  In [**16**]: yaml.dump(my\_list,default\_flow\_style=True)  Out[**16**]: '[0, 1, 2, 3, 4, 5, 6, 7, whatever, Hello, {attribs: [0, 1, 2, 3, 4, 5, 6], ip\_addr: 10.10.10.1}]\n'  In [**19**]: **print** **yaml.dump**(my\_list,**default\_flow\_style=False**)  - 0  - 1  - 2  - 3  - 4  - 5  - 6  - 7  - whatever  - Hello  - attribs:  - 0  - 1  - 2  - 3  - 4  - 5  - 6  ip\_addr: 10.10.10.1  In [**20**]: **with** open("some\_file.yml", 'w') **as** f:  ...: **f.write**(**yaml.dump**(my\_list,default\_flow\_style=False))  ...: *#****f.write****(str) =>* ***yaml.dump****(my\_list,default\_flow\_style=False) is* ***str type***  In [**22**]: **with** open("some\_file.yml", 'r') **as** f:  ...: new\_list = **yaml.load(f)**  ...:  In [**24**]: new\_list  Out[**24**]:  [0,  1,  2,  3,  4,  5,  6,  7,  'whatever',  'Hello',  {'attribs': [0, 1, 2, 3, 4, 5, 6],  'ip\_addr': '10.10.10.1',  'next\_ley': 'whatever',  'some\_key': 'hello'}]  In [**25**]: new\_list[-1]  Out[**25**]:  {'attribs': [0, 1, 2, 3, 4, 5, 6],  'ip\_addr': '10.10.10.1',  'next\_ley': 'whatever',  'some\_key': 'hello'}  In [**26**]: type(new\_list[-1])  Out[**26**]: dict |

**NoneType** is the type for the None object, which is an object that indicates ***no value*.**

|  |
| --- |
| In [**71**]: **with** open("some\_file.yml") **as** fp:  ...: ne\_list=**yaml.load**(fp)  ...: **print** "yaml.loadl type:",type(yaml.load(fp))  ...: **print** "ne\_list type:",type(ne\_list)  ...:  yaml.loadl type: <type 'NoneType'>  ne\_list type: <type 'list'>  In [**72**]: **with** open("some\_file.yml") **as** fp:  ...: **print** "yaml.load type:",type(yaml.load(fp))  ...:  yaml.load type: <**type 'list'**> |

JSON

JSON is extensively used for Arista API.

* **json.dump**(obj,fp) => *#Serialize obj as a JSON formatted stream to fp, => create a file named “my\_file.json” with my\_list*
* **json.load**(fp) => *#Deserialize f to a Python* ***list***

|  |
| --- |
| In [**1**]: **import** **json**  In [**2**]: my\_list = range(8)  In [**3**]: my\_list.append('whatever')  In [**4**]: my\_list.append('hello')  In [**5**]: my\_list.append({})  In [**6**]: my\_list[-1]  Out[**6**]: {}  In [**8**]: my\_list[-1]['ip\_addr'] = '10.10.10.1'  In [**9**]: my\_list[-1]['attribs'] = range(5)  In [**10**]: my\_list  Out[**10**]:  [0,  1,  2,  3,  4,  5,  6,  7,  'whatever',  'hello',  {'attribs': [0, 1, 2, 3, 4], 'ip\_addr': '10.10.10.1'}]  In [**11**]: **json.dumps**(my\_list) *#convert python object(list,dict,string…) into JSON string => Serialize obj to a JSON formatted* [*str*](https://docs.python.org/3/library/stdtypes.html#str)  Out[**11**]: '[0, 1, 2, 3, 4, 5, 6, 7, "whatever", "hello", {"attribs": [0, 1, 2, 3, 4], "ip\_addr": "10.10.10.1"}]'  In [**12**]: **type**(json.dumps(my\_list))  Out[**12**]: **str**  In [**14**]: **with open('my\_file.json','w') as f:**  **...: json.dump(my\_list,f)** *#Serialize obj as a JSON formatted stream to fp, => create a file named “my\_file.json” with my\_list*  ...:  ...:  In [**28**]: **with open("my\_file.json") as f:**  **...: new\_list = json.load(f)** *#Deserialize f to a Python object*  ...:  In [**29**]: new\_list  Out[**29**]:  [0,  1,  2,  3,  4,  5,  6,  7,  u'whatever',  u'hello',  {u'attribs': [0, 1, 2, 3, 4], u'ip\_addr': u'10.10.10.1'}]  In [**30**]: **from pprint import pprint as pp**  In [**31**]: pp(new\_list)  [0,  1,  2,  3,  4,  5,  6,  7,  u'whatever',  u'hello',  {u'attribs': [0, 1, 2, 3, 4], u'ip\_addr': u'10.10.10.1'}]  In [**32**]: type(json.dumps(my\_list))  Out[**32**]: str |

CiscoConfParse

* Deal with cisco’s hierarchical output parsing
* Easier parsing indented output

|  |
| --- |
| **from ciscoconfparse import CiscoConfParse**  ### elements(<class 'ciscoconfparse.models\_cisco.IOSCfgLine'>) of list can have attributes such as .text, .children, and .all\_children  **cisco\_cfg = CiscoConfParse("/Users/pkim/tmp/cr1.sjc2.txt")**  **intf = cisco\_cfg.find\_objects(r"^interface")** # the type of intf is list, list of interface elements with IOSCfgLine.  **po1 = intf[0]** #pick only first element of the list which is "<IOSCfgLine # 72 'interface Port-Channel1'>""  **for item in po1.children: #the children of the element**  **print item.text** #item shows <IOSCfgLine # xx ' blah blah... >, item.text shows actual contents only  **nat\_pmap = cisco\_cfg.find\_objects(r"^policy-map type pbr NAT-PMAP")**  **for child in nat\_pmap[0].all\_children:** #show all the indented lines under nat\_pmap[0]  **print child.text**  **check\_child = child.is\_child** #check whether this child is a child of a parent?  **print check\_child**  **check\_parent = child.is\_parent** #check whether this child is parent?  **print check\_parent**  **print child.parent.text** # print all the direct parent  **print "\*\*\*\*\*",type(child.parent)**  **print child.all\_parents** # print all the parents on each level  **print "===",type(child.all\_parents)**  **check\_dscp = cisco\_cfg.find\_objects\_w\_child(parentspec = r"^interface",childspec = r"dscp")** *#parent spec 에도 맞고 child spec에도 맞는 걸 찾아낸다.*  **for children in range(len(check\_dscp)):**  **print check\_dscp[children].text**  **print "========================="**  **check\_non\_dscp = cisco\_cfg.find\_objects\_wo\_child(parentspec = r"^interface",childspec = r"whatever")** *#parent spec 에는 맞고 child spec에는 없는것을 찾아낸다.*  **for children in range(len(check\_non\_dscp)):**  **print check\_non\_dscp[children].text**  === output ===  description "200G Aggregated Ethernet cr1<->cr2"  switchport trunk allowed vlan 201,400,403,405-410,415,425,430,435,450,455,465,470,475,490,499,501-502,550,950,4094  switchport mode trunk  switchport trunk group mlagpeer  10 class NO-NAT-CMAP  20 class NAT-CMAP  set nexthop 69.194.242.244  True  False  20 class NAT-CMAP  \*\*\*\*\* <class 'ciscoconfparse.models\_cisco.IOSCfgLine'>  [<IOSCfgLine # 315 'policy-map type pbr NAT-PMAP'>, <IOSCfgLine # 317 ' 20 class NAT-CMAP' (parent is # 315)>]  === <type 'list'>  interface Ethernet3/7  interface Ethernet3/10  interface Ethernet3/11  interface Ethernet3/12  interface Ethernet3/13  interface Ethernet3/15  interface Ethernet3/16  interface Ethernet3/19  interface Ethernet3/20  interface Ethernet3/21  =========================  interface Port-Channel1  interface Port-Channel2  interface Ethernet3/7  interface Ethernet3/8  interface Ethernet3/9  interface Ethernet3/10  interface Ethernet3/30  interface Ethernet3/31  interface Vlan550  interface Vlan950  interface Vlan4094 |

**\*Execute cmds over telnet**

|  |
| --- |
| #!/usr/bin/env python  '''  Convert the code from exercise2 to a class-based solution  '''  import telnetlib  import time  import socket  import sys  import getpass  TELNET\_PORT = 23  TELNET\_TIMEOUT = 6  class TelnetConn(object):  '''  Establish and manage telnet connection to network devices  '''  def \_\_init\_\_(self, ip\_addr, username, password*): #initialize all the necessary variables and environment ready for further use, no return is expected.*  self.ip\_addr = ip\_addr  self.username = username  self.password = password  try:  self.remote\_conn = telnetlib.Telnet(self.ip\_addr, TELNET\_PORT, TELNET\_TIMEOUT) *# pre-stage the telnet session connection*  except **socket.timeout:**  **sys.exit**("Connection timed-out")  def login(self):  '''  Login to network device  '''  self.remote\_conn.**read\_until**("sername:", TELNET\_TIMEOUT) *#wait until “sername:” is coming up or dedicated time(=TELNET\_TIMEOUT(6sec)) without no “sername:” coming up.*  self.remote\_conn.**write**(self.username + '\n')  self.remote\_conn.**read\_until**("ssword:", TELNET\_TIMEOUT)  self.remote\_conn.**write**(self.password + '\n')  time.sleep(1)  def send\_command(self, cmd, sleep\_time=1): *# default values(=sleep\_time)* ***should come*** *after the received variable(=cmd). #default value(=paging\_cmd) isn't counted as an argument*  '''  Send a command down the telnet channel  Return the response  '''  cmd = cmd.rstrip()  self.remote\_conn.write(cmd + '\n')  time.sleep(sleep\_time)  **return** self.remote\_conn.read\_very\_eager() *# the only return in this script!!*  def disable\_paging(self, paging\_cmd='terminal length 0'): *#default value(=paging\_cmd) isn't counted as an argument*  '''  Disable the paging of output  '''  self.send\_command(paging\_cmd)  def close\_conn(self):  '''  Close telnet connection  '''  self.remote\_conn.close()  def main():  '''  Convert the code from exercise2 to a class-based solution  '''  ip\_addr = raw\_input("IP address: ")  ip\_addr = ip\_addr.strip()  username = 'pyclass'  password = getpass.getpass()  my\_conn = TelnetConn(ip\_addr, username, password)  my\_conn.login()  my\_conn.disable\_paging() *#default value(=paging\_cmd) isn't counted as an argument*  output = my\_conn.send\_command('show ip int brief') *#the type of output is "str"*  print "\n\n"  print output  print "\n\n"  my\_conn.close\_conn()  if \_\_name\_\_ == "\_\_main\_\_":  main() |