



---

Junos<sup>®</sup> OS

## Junos PyEZ Developer Guide

Release  
2.1.2



---

Modified: 2017-08-29

Juniper Networks, Inc.  
1133 Innovation Way  
Sunnyvale, California 94089  
USA  
408-745-2000  
[www.juniper.net](http://www.juniper.net)

Copyright © 2017 Juniper Networks, Inc. All rights reserved.

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. and/or its affiliates in the United States and other countries. All other trademarks may be property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

*Junos® OS Junos PyEZ Developer Guide*

2.1.2

Copyright © 2017 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

#### YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

#### END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at <http://www.juniper.net/support/eula/>. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

# Table of Contents

	About the Documentation . . . . .	ix
	Documentation and Release Notes . . . . .	ix
	Supported Platforms . . . . .	ix
	Documentation Conventions . . . . .	x
	Documentation Feedback . . . . .	xi
	Requesting Technical Support . . . . .	xii
	Self-Help Online Tools and Resources . . . . .	xii
	Opening a Case with JTAC . . . . .	xiii
<b>Chapter 1</b>	<b>Disclaimer . . . . .</b>	<b>15</b>
	Junos PyEZ Disclaimer . . . . .	15
<b>Chapter 2</b>	<b>Junos PyEZ Overview . . . . .</b>	<b>17</b>
	Understanding Junos PyEZ . . . . .	17
	Junos PyEZ Overview . . . . .	17
	Junos PyEZ Resources . . . . .	18
	Using Junos PyEZ in Automation Scripts . . . . .	19
	Junos PyEZ Modules Overview . . . . .	19
<b>Chapter 3</b>	<b>Installing Junos PyEZ . . . . .</b>	<b>21</b>
	Installing Junos PyEZ . . . . .	21
	Installing Junos PyEZ Dependencies . . . . .	21
	Installing the Junos PyEZ Package . . . . .	23
	Using the Junos PyEZ Docker Image . . . . .	23
	Setting Up Junos PyEZ Managed Nodes . . . . .	24
	Configuring Telnet Service on Devices Running Junos OS . . . . .	25
	Enabling NETCONF on Devices Running Junos OS . . . . .	26
	Satisfying Requirements for SSHv2 Connections . . . . .	26
<b>Chapter 4</b>	<b>Authenticating Junos PyEZ Users . . . . .</b>	<b>27</b>
	Authenticating Junos PyEZ Users Using a Password . . . . .	27
	Authenticating Junos PyEZ Users Using SSH Keys . . . . .	29
	Generating and Configuring SSH Keys . . . . .	30
	Referencing SSH Keys in Junos PyEZ Applications . . . . .	31
	Authenticating the User Using an SSH Key Agent with Actively Loaded Keys . . . . .	31
	Authenticating the User Using SSH Keys Without Password Protection . . . . .	31
	Authenticating the User Using Password-Protected SSH Key Files . . . .	32

<b>Chapter 5</b>	<b>Connecting to and Retrieving Facts From a Device Using Junos PyEZ . . . . 33</b>
	Connecting to Devices Running Junos OS Using Junos PyEZ . . . . . 33
	Connection Methods Overview . . . . . 33
	Connecting to a Device Using NETCONF over SSH . . . . . 34
	Connecting to a Device Using Telnet . . . . . 37
	Connecting to a Device Using a Serial Console Connection . . . . . 39
	Understanding Junos PyEZ Device Facts and Connection Properties . . . . . 40
	Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS . . . . . 43
	Accessing the Shell on Devices Running Junos OS Using Junos PyEZ . . . . . 46
<b>Chapter 6</b>	<b>Configuring Devices Running Junos OS . . . . . 49</b>
	Using Junos PyEZ to Configure Devices Running Junos OS . . . . . 49
	Structured vs Unstructured Configuration Changes . . . . . 49
	Configuration Process . . . . . 50
	Using the Junos PyEZ Config Utility to Load Configuration Data . . . . . 53
	Loading Configuration Data from a File . . . . . 56
	Loading Configuration Data from a String . . . . . 56
	Loading Configuration Data Formatted as an XML Object . . . . . 57
	Loading Jinja2 Templates . . . . . 58
	Example: Using Junos PyEZ to Load Configuration Data from a File . . . . . 60
	Using Junos PyEZ to Commit the Candidate Configuration . . . . . 68
	Committing the Candidate Configuration . . . . . 68
	Including Commit Options . . . . . 69
<b>Chapter 7</b>	<b>Comparing Configurations Using Junos PyEZ . . . . . 73</b>
	Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration . . . . . 73
<b>Chapter 8</b>	<b>Requesting Configuration Information Using Junos PyEZ . . . . . 77</b>
	Using Junos PyEZ to Retrieve a Configuration . . . . . 77
	Retrieving the Entire Configuration . . . . . 78
	Specifying the Source of Configuration Data to Return . . . . . 78
	Specifying the Format for Configuration Data to Return . . . . . 78
	Specifying the Scope of Configuration Data to Return . . . . . 79
	Retrieving Configuration Data for Standard or Custom YANG Data Models . . . . . 79
	Handling Namespaces in Configuration Data . . . . . 81
<b>Chapter 9</b>	<b>Rolling Back the Configuration Using Junos PyEZ . . . . . 83</b>
	Example: Using Junos PyEZ to Roll Back the Configuration . . . . . 83
<b>Chapter 10</b>	<b>Executing RPCs Using Junos PyEZ . . . . . 89</b>
	Using Junos PyEZ to Execute RPCs on Devices Running Junos OS . . . . . 89
	Mapping Junos OS Commands to Junos PyEZ RPCs . . . . . 89
	Executing RPCs as a Property of the Device Instance . . . . . 90
	Normalizing the XML RPC Reply . . . . . 92
	Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications . . . . . 93

<b>Chapter 11</b>	<b>Installing and Rebooting Software Using Junos PyEZ . . . . .</b>	<b>95</b>
	Rebooting a Device Running Junos OS Using Junos PyEZ . . . . .	95
	Using Junos PyEZ to Install Software on Devices Running Junos OS . . . . .	97
	Supported Deployment Scenarios . . . . .	97
	Specifying the Software Image Location . . . . .	98
	Installation Process Overview . . . . .	99
	ISSU and NSSU Support . . . . .	100
	Example: Using Junos PyEZ to Install Software on Devices Running Junos OS . .	101
<b>Chapter 12</b>	<b>Managing the Rescue Configuration Using Junos PyEZ . . . . .</b>	<b>109</b>
	Using Junos PyEZ to Manage the Rescue Configuration . . . . .	109
	Example: Using Junos PyEZ to Save a Rescue Configuration . . . . .	110
<b>Chapter 13</b>	<b>Transferring Files Using Junos PyEZ . . . . .</b>	<b>115</b>
	Transferring Files Using Junos PyEZ . . . . .	115
<b>Chapter 14</b>	<b>Creating and Using Junos PyEZ Tables and Views . . . . .</b>	<b>119</b>
	Understanding Junos PyEZ Tables and Views . . . . .	119
	Junos PyEZ Predefined Operational Tables and Views . . . . .	121
	Loading Inline Junos PyEZ Tables and Views . . . . .	123
	Importing External Junos PyEZ Tables and Views . . . . .	125
	Defining Junos PyEZ Operational Tables . . . . .	127
	Table Name . . . . .	129
	RPC Command (rpc) . . . . .	129
	RPC Default Arguments (args) . . . . .	129
	RPC Optional Argument Key (args_key) . . . . .	130
	Table Item (item) . . . . .	130
	Table Item Key (key) . . . . .	131
	Table View (view) . . . . .	133
	Defining Junos PyEZ Views for Operational Tables . . . . .	133
	View Name . . . . .	135
	Fields (fields) . . . . .	135
	Groups (groups) and Field Groups (fields_) . . . . .	136
	Using Junos PyEZ Operational Tables and Views . . . . .	137
	Retrieving Table Items . . . . .	138
	Accessing Table Items . . . . .	139
	Iterating Through a Table . . . . .	140
	Defining Junos PyEZ Configuration Tables . . . . .	141
	Table Name . . . . .	143
	Configuration Scope (get or set) . . . . .	143
	Key Field (key-field) . . . . .	144
	Required Keys (required_keys) . . . . .	145
	Table View (view) . . . . .	146
	Defining Junos PyEZ Views for Configuration Tables . . . . .	146
	View Name . . . . .	148
	Fields (fields) . . . . .	148
	Field Options ('get' Tables) . . . . .	150
	Field Options ('set' Tables) . . . . .	151
	Groups (groups) and Field Groups (fields_) . . . . .	152

	Using Junos PyEZ Configuration Tables to Retrieve Configuration Data . . . . .	154
	Retrieving Configuration Items . . . . .	154
	Specifying Inheritance and Group Options . . . . .	155
	Accessing Table Items . . . . .	157
	Iterating Through a Table . . . . .	158
	Overview of Using Junos PyEZ Configuration Tables to Define and Configure	
	Structured Resources . . . . .	159
	Creating the Structured Resource . . . . .	159
	Using the Resource in a Junos PyEZ Application . . . . .	160
	Using Junos PyEZ Configuration Tables to Configure Structured Resources on	
	Devices Running Junos OS . . . . .	161
	General Configuration Process . . . . .	162
	Configuring Statements Consisting of a Fixed-Form Keyword . . . . .	164
	Configuring Multiple Values for the Same Statement . . . . .	165
	Configuring Multiple Instances of the Same Statement . . . . .	166
	Configuring Multiple Instances of the Same Resource . . . . .	168
	Deleting Containers or Leaf Statements . . . . .	169
	Using append() to Generate the Junos XML Configuration Data . . . . .	170
	Viewing Your Configuration Changes . . . . .	171
	Controlling the RPC Timeout Interval . . . . .	172
	Saving and Loading Junos PyEZ Table XML to and from Files . . . . .	173
<b>Chapter 15</b>	<b>Troubleshooting Junos PyEZ . . . . .</b>	<b>175</b>
	Troubleshooting jnpr.junos Import Errors . . . . .	175
	Troubleshooting Junos PyEZ Connection Errors . . . . .	176
	Troubleshooting Junos PyEZ Authentication Errors When Managing Devices	
	Running Junos OS . . . . .	177
	Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos	
	OS . . . . .	178
	Troubleshooting Timeout Errors . . . . .	178
	Troubleshooting Configuration Lock Errors . . . . .	178
	Troubleshooting Configuration Change Errors . . . . .	179

# List of Tables

	<b>About the Documentation</b> . . . . .	<b>ix</b>
	Table 1: Notice Icons . . . . .	x
	Table 2: Text and Syntax Conventions . . . . .	x
<b>Chapter 2</b>	<b>Junos PyEZ Overview</b> . . . . .	<b>17</b>
	Table 3: Junos PyEZ Resources . . . . .	18
	Table 4: Junos PyEZ Modules . . . . .	19
<b>Chapter 3</b>	<b>Installing Junos PyEZ</b> . . . . .	<b>21</b>
	Table 5: Junos PyEZ Dependencies . . . . .	22
<b>Chapter 5</b>	<b>Connecting to and Retrieving Facts From a Device Using Junos PyEZ</b> . . . .	<b>33</b>
	Table 6: Junos PyEZ Connection Methods . . . . .	34
	Table 7: Device Properties . . . . .	42
<b>Chapter 6</b>	<b>Configuring Devices Running Junos OS</b> . . . . .	<b>49</b>
	Table 8: Parameters for Specifying the Load Operation Type in the load() and set() Methods . . . . .	51
	Table 9: Specifying the Format for Configuration Data . . . . .	54
	Table 10: Referencing Configuration Data in the load() Method . . . . .	55
	Table 11: Junos PyEZ Supported Commit Options . . . . .	69
<b>Chapter 10</b>	<b>Executing RPCs Using Junos PyEZ</b> . . . . .	<b>89</b>
	Table 12: Comparison of a Default and Normalized RPC Reply . . . . .	92
<b>Chapter 11</b>	<b>Installing and Rebooting Software Using Junos PyEZ</b> . . . . .	<b>95</b>
	Table 13: install() Method Parameter Settings for Software Package Location . . . . .	98
<b>Chapter 14</b>	<b>Creating and Using Junos PyEZ Tables and Views</b> . . . . .	<b>119</b>
	Table 14: jnpr.junos.op Modules . . . . .	121
	Table 15: Junos PyEZ Op Table Parameters . . . . .	127
	Table 16: Junos PyEZ Op View Parameters . . . . .	133
	Table 17: Junos PyEZ Configuration Table Parameters . . . . .	141
	Table 18: Junos PyEZ Configuration View Parameters . . . . .	147
	Table 19: Type Checks for 'set' Configuration Tables . . . . .	151
	Table 20: Constraint Checks for 'set' Configuration Tables . . . . .	152



# About the Documentation

- Documentation and Release Notes on page ix
- Supported Platforms on page ix
- Documentation Conventions on page x
- Documentation Feedback on page xi
- Requesting Technical Support on page xii

## Documentation and Release Notes

---

To obtain the most current version of all Juniper Networks® technical documentation, see the product documentation page on the Juniper Networks website at <http://www.juniper.net/techpubs/>.

If the information in the latest release notes differs from the information in the documentation, follow the product Release Notes.

Juniper Networks Books publishes books by Juniper Networks engineers and subject matter experts. These books go beyond the technical documentation to explore the nuances of network architecture, deployment, and administration. The current list can be viewed at <http://www.juniper.net/books>.

## Supported Platforms

---

For the features described in this document, the following platforms are supported:

- ACX Series
- EX Series
- M Series
- MX Series
- NFX Series
- PTX Series
- QFX Series
- SRX Series
- T Series

## Documentation Conventions

Table 1 on page x defines notice icons used in this guide.

Table 1: Notice Icons







Icon	Meaning	Description
	Informational note	Indicates important features or instructions.
	Caution	Indicates a situation that might result in loss of data or hardware damage.
	Warning	Alerts you to the risk of personal injury or death.
	Laser warning	Alerts you to the risk of personal injury from a laser.
	Tip	Indicates helpful information.
	Best practice	Alerts you to a recommended use or implementation.

Table 2 on page x defines the text and syntax conventions used in this guide.

Table 2: Text and Syntax Conventions

Convention	Description	Examples
<b>Bold text like this</b>	Represents text that you type.	To enter configuration mode, type the <b>configure</b> command:  user@host> <b>configure</b>
Fixed-width text like this	Represents output that appears on the terminal screen.	user@host> <b>show chassis alarms</b> No alarms currently active
<i>Italic text like this</i>	<ul style="list-style-type: none"> <li>Introduces or emphasizes important new terms.</li> <li>Identifies guide names.</li> <li>Identifies RFC and Internet draft titles.</li> </ul>	<ul style="list-style-type: none"> <li>A policy <i>term</i> is a named structure that defines match conditions and actions.</li> <li><i>Junos OS CLI User Guide</i></li> <li>RFC 1997, <i>BGP Communities Attribute</i></li> </ul>

Table 2: Text and Syntax Conventions (*continued*)

Convention	Description	Examples
<i>Italic text like this</i>	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name:  [edit] root@# <b>set system domain-name</b> <i>domain-name</i>
<b>Text like this</b>	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	<ul style="list-style-type: none"> <li>To configure a stub area, include the <b>stub</b> statement at the [edit protocols ospf area area-id] hierarchy level.</li> <li>The console port is labeled <b>CONSOLE</b>.</li> </ul>
< > (angle brackets)	Encloses optional keywords or variables.	<b>stub</b> <default-metric <i>metric</i> >;
(pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	<b>broadcast   multicast</b>  <i>(string1   string2   string3)</i>
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	<b>rsvp { # Required for dynamic MPLS only</b>
[ ] (square brackets)	Encloses a variable for which you can substitute one or more values.	<b>community name members [</b> <i>community-ids</i> <b>]</b>
Indentation and braces ( { } )	Identifies a level in the configuration hierarchy.	[edit] routing-options { static { route default { nexthop <i>address</i> ; retain; } } }
;(semicolon)	Identifies a leaf statement at a configuration hierarchy level.	
<b>GUI Conventions</b>		
<b>Bold text like this</b>	Represents graphical user interface (GUI) items you click or select.	<ul style="list-style-type: none"> <li>In the Logical Interfaces box, select <b>All Interfaces</b>.</li> <li>To cancel the configuration, click <b>Cancel</b>.</li> </ul>
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select <b>Protocols&gt;Ospf</b> .

## Documentation Feedback

We encourage you to provide feedback, comments, and suggestions so that we can improve the documentation. You can provide feedback by using either of the following methods:

- Online feedback rating system—On any page of the Juniper Networks TechLibrary site at <http://www.juniper.net/techpubs/index.html>, simply click the stars to rate the content, and use the pop-up form to provide us with information about your experience. Alternately, you can use the online feedback form at <http://www.juniper.net/techpubs/feedback/>.
- E-mail—Send your comments to [techpubs-comments@juniper.net](mailto:techpubs-comments@juniper.net). Include the document or topic name, URL or page number, and software version (if applicable).

## Requesting Technical Support

---

Technical product support is available through the Juniper Networks Technical Assistance Center (JTAC). If you are a customer with an active J-Care or Partner Support Service support contract, or are covered under warranty, and need post-sales technical support, you can access our tools and resources online or open a case with JTAC.

- JTAC policies—For a complete understanding of our JTAC procedures and policies, review the *JTAC User Guide* located at <http://www.juniper.net/us/en/local/pdf/resource-guides/7100059-en.pdf>.
- Product warranties—For product warranty information, visit <http://www.juniper.net/support/warranty/>.
- JTAC hours of operation—The JTAC centers have resources available 24 hours a day, 7 days a week, 365 days a year.

## Self-Help Online Tools and Resources

For quick and easy problem resolution, Juniper Networks has designed an online self-service portal called the Customer Support Center (CSC) that provides you with the following features:

- Find CSC offerings: <http://www.juniper.net/customers/support/>
- Search for known bugs: <http://www2.juniper.net/kb/>
- Find product documentation: <http://www.juniper.net/techpubs/>
- Find solutions and answer questions using our Knowledge Base: <http://kb.juniper.net/>
- Download the latest versions of software and review release notes: <http://www.juniper.net/customers/csc/software/>
- Search technical bulletins for relevant hardware and software notifications: <http://kb.juniper.net/InfoCenter/>
- Join and participate in the Juniper Networks Community Forum: <http://www.juniper.net/company/communities/>
- Open a case online in the CSC Case Management tool: <http://www.juniper.net/cm/>

To verify service entitlement by product serial number, use our Serial Number Entitlement (SNE) Tool: <https://tools.juniper.net/SerialNumberEntitlementSearch/>

## Opening a Case with JTAC

You can open a case with JTAC on the Web or by telephone.

- Use the Case Management tool in the CSC at <http://www.juniper.net/cm/>.
- Call 1-888-314-JTAC (1-888-314-5822 toll-free in the USA, Canada, and Mexico).

For international or direct-dial options in countries without toll-free numbers, see <http://www.juniper.net/support/requesting-support.html>.



## CHAPTER 1

# Disclaimer

- [Junos PyEZ Disclaimer on page 15](#)

### Junos PyEZ Disclaimer

---

Use of the Junos PyEZ software implies acceptance of the terms of this disclaimer, in addition to any other licenses and terms required by Juniper Networks.

Juniper Networks is willing to make the Junos PyEZ software available to you only on the condition that you accept all of the terms contained in this disclaimer. Please read the terms and conditions of this disclaimer carefully.

The Junos PyEZ software is provided *as is*. Juniper Networks makes no warranties of any kind whatsoever with respect to this software. All express or implied conditions, representations and warranties, including any warranty of non-infringement or warranty of merchantability or fitness for a particular purpose, are hereby disclaimed and excluded to the extent allowed by applicable law.

In no event will Juniper Networks be liable for any direct or indirect damages, including but not limited to lost revenue, profit or data, or for direct, special, indirect, consequential, incidental or punitive damages however caused and regardless of the theory of liability arising out of the use of or inability to use the software, even if Juniper Networks has been advised of the possibility of such damages.



## CHAPTER 2

# Junos PyEZ Overview

- [Understanding Junos PyEZ on page 17](#)
- [Junos PyEZ Modules Overview on page 19](#)

## Understanding Junos PyEZ

---

- [Junos PyEZ Overview on page 17](#)
- [Junos PyEZ Resources on page 18](#)
- [Using Junos PyEZ in Automation Scripts on page 19](#)

### Junos PyEZ Overview

Junos PyEZ is a microframework for Python that enables you to manage and automate devices running the Junos operating system (Junos OS). Junos PyEZ is designed to provide the capabilities that a user would have on the Junos OS command-line interface (CLI) in an environment built for automation tasks. Junos PyEZ does not require extensive knowledge of Junos OS or the Junos XML APIs.

Junos PyEZ enables you to manage devices running Junos OS using the familiarity of Python. However, you do not have to be an experienced programmer to use Junos PyEZ. Non-programmers can quickly execute simple commands in Python interactive mode, and more experienced programmers can opt to create more complex, robust, and reusable programs to perform tasks.

Junos PyEZ enables you to connect to devices running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. You can use Junos PyEZ to initially configure a new, or zeroized device that is not yet configured for remote access by using either a serial console connection when you are directly connected to the device or by using telnet through a console server that is directly connected to the device.

Junos PyEZ enables you to perform operational tasks on devices running Junos OS. Using Junos PyEZ, you can easily retrieve facts and operational information from a device and execute any remote procedure call (RPC) available through the Junos XML API. Junos PyEZ provides software utilities for installing the Junos OS software and rebooting or shutting down managed devices. Junos PyEZ also provides file system utilities to perform common administrative tasks such as copying files and calculating checksums.

Junos PyEZ also enables you to manage the configurations of devices running Junos OS. Junos PyEZ configuration management utilities enable you to retrieve and compare configurations, create a rescue configuration, roll back a configuration, and upload configuration changes. Junos PyEZ supports standard formats for configuration data including ASCII text, Junos XML elements, Junos OS **set** commands, and JavaScript Object Notation (JSON), and also supports using Jinja2 templates and template files for added flexibility and customization. In addition, you can use Tables and Views to define structured resources that you can use to programmatically configure a device.

Junos PyEZ provides Tables and Views to enable you to both configure devices running Junos OS and extract specific operational information or configuration data from the devices. Tables and Views are defined using simple YAML files that contain key:value pair mappings, so no complex coding is required to use them. Using Tables and Views, you can retrieve the device configuration or the output for any Junos OS command that has an RPC equivalent and then extract a customized subset of information. This is useful when you need to retrieve information from a few specific fields that are embedded in extensive command output such as for the **show route** or **show interfaces** command. In addition, starting in Junos PyEZ Release 2.0, you can use Tables and Views to define structured configuration resources. Junos PyEZ dynamically creates a configuration class for the resource, which enables you to programmatically configure the resource on a device.

## Junos PyEZ Resources

Juniper Networks provides a number of Junos PyEZ resources, which are described in [Table 3 on page 18](#).

**Table 3: Junos PyEZ Resources**

Resource	Description	URL
API Reference	Detailed documentation for the Junos PyEZ modules.	<a href="http://junos-pyez.readthedocs.org/">http://junos-pyez.readthedocs.org/</a>
Documentation	Junos PyEZ documentation containing detailed information about installing Junos PyEZ and using Junos PyEZ to perform operational and configuration tasks on devices running Junos OS.	<a href="http://www.juniper.net/documentation/en_US/release-independent/junos-pyez/information-products/pathway-pages/index.html">http://www.juniper.net/documentation/en_US/release-independent/junos-pyez/information-products/pathway-pages/index.html</a>
GitHub repository	Public repository for the Junos PyEZ project. This repository includes the most current source code, installation instructions, and release note summaries for all releases.	<a href="https://github.com/Juniper/py-junos-eznc/">https://github.com/Juniper/py-junos-eznc/</a>
Google Groups forum	Forum that addresses questions and provides general support for Junos PyEZ.	<a href="http://groups.google.com/group/junos-python-ez">http://groups.google.com/group/junos-python-ez</a>
Sample scripts	Junos PyEZ sample scripts to get you started.	<a href="https://github.com/Juniper/junosautomation/tree/master/pyez">https://github.com/Juniper/junosautomation/tree/master/pyez</a>

Table 3: Junos PyEZ Resources (*continued*)

Resource	Description	URL
Techwiki page	Juniper Networks J-Net community forum containing additional how-to articles and usage examples.	<a href="http://forums.juniper.net/t5/Automation-Scripting/Junos-PyEZ/ta-p/280496">http://forums.juniper.net/t5/Automation-Scripting/Junos-PyEZ/ta-p/280496</a>

## Using Junos PyEZ in Automation Scripts

The Junos OS automation suite of tools includes commit scripts, operation (op) scripts, SNMP scripts, and event policies and event scripts. Starting in Junos OS Release 16.1, Junos OS automation scripts can be written in Python on certain supported devices that include the Python extensions package in the software image. Python automation scripts can leverage features in Junos PyEZ Release 1.3.1 and earlier releases to execute RPCs and perform operational and configuration tasks on devices running Junos OS.

For more information about creating Python automation scripts, see [Understanding Python Automation Scripts for Devices Running Junos OS](#) in the [Junos OS Automation Scripting Feature Guide](#).

### Related Documentation

- [Junos PyEZ Modules Overview on page 19](#)
- [Installing Junos PyEZ on page 21](#)
- [Connecting to Devices Running Junos OS Using Junos PyEZ on page 33](#)
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)

## Junos PyEZ Modules Overview

Junos PyEZ is a microframework for Python that enables you to manage and automate devices running Junos OS. Junos PyEZ consists of the **jnpr.junos** package, which contains modules that handle device connectivity and provide operational and configuration utilities.

[Table 4 on page 19](#) outlines the primary Junos PyEZ modules that are used to manage devices running Junos OS. For detailed information about each module, see the Junos PyEZ API documentation at <http://junos-pyez.readthedocs.org/>.

Table 4: Junos PyEZ Modules

jnpr.junos Modules	Description
<b>device</b>	Defines the <b>Device</b> class, which represents the device running Junos OS and enables you to connect to and retrieve facts from the device.
<b>exception</b>	Defines exceptions encountered when accessing, configuring, and managing devices running Junos OS.

Table 4: Junos PyEZ Modules (*continued*)

jnpr.junos Modules	Description
<b>factory</b>	Contains code pertaining to Tables and Views, including the <code>loadyaml()</code> method, which is used to load custom Tables and Views.
<b>op</b>	Includes predefined operational Tables and Views that can be used to filter output for common operational commands.
<b>resources</b>	Includes predefined configuration Tables and Views representing specific configuration resources, which can be used to programmatically configure devices running Junos OS.
<b>transport</b>	Contains code used by the <b>Device</b> class to support different connection types such as telnet and serial console connections.
<b>utils</b>	Includes configuration utilities, file system utilities, shell utilities, software installation utilities, and secure copy utilities.

In Junos PyEZ, each device is modeled as an instance of the `jnpr.junos.device.Device` class. The **device** module provides access to devices running Junos OS through a serial console connection, telnet, or a NETCONF session over SSH. All connection methods support retrieving device facts, performing operations, and executing RPCs on demand. Support for serial console connections and for telnet connections through a console server enables you to connect to and initially configure new or zeroized devices that are not yet configured for remote access.

The **utils** module defines submodules and classes that handle software installation, file system and copy operations, and configuration management. The **exception** module defines exceptions encountered when managing devices running Junos OS.

The **op**, **resources**, and **factory** modules pertain to Tables and Views. The **op** module contains predefined operational Tables and Views that can be used to extract specific information from the output of common operational commands on devices running Junos OS. The **resources** module contains predefined configuration Tables and Views that can be used to configure specific resources on devices running Junos OS. The **factory** module contains methods that enable you to load your own custom Tables and Views in Junos PyEZ applications.

#### Related Documentation

- [Understanding Junos PyEZ on page 17](#)
- [Understanding Junos PyEZ Tables and Views on page 119](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## CHAPTER 3

# Installing Junos PyEZ

- [Installing Junos PyEZ on page 21](#)
- [Setting Up Junos PyEZ Managed Nodes on page 24](#)

## Installing Junos PyEZ

---

Junos PyEZ is a Python library that enables you to manage and automate devices running Junos OS. You can install Junos PyEZ on a UNIX-like operating system or on Windows.

Juniper Networks also provides a Junos PyEZ Docker image that enables you to run Junos PyEZ as a Docker container. The Docker container is a lightweight, self-contained system that bundles Junos PyEZ, its dependencies, and Python into a single portable container. The Docker image enables you to quickly run Junos PyEZ in interactive mode, as an executable package, or as a terminal on any platform that supports Docker.

To manually install Junos PyEZ and its dependencies, see the following sections:

- [Installing Junos PyEZ Dependencies on page 21](#)
- [Installing the Junos PyEZ Package on page 23](#)

To use the Junos PyEZ Docker image, see the following section:

- [Using the Junos PyEZ Docker Image on page 23](#)

## Installing Junos PyEZ Dependencies

Before you install the Junos PyEZ package on the configuration management server, ensure that the server has the following software installed:

- Python 2.6, Python 2.7, or Python 3.4
- All dependencies for the given operating system, which are outlined in [Table 5 on page 22](#)



**NOTE:** Python 3.x is only supported in Junos PyEZ Release 2.0 and later releases.

Table 5: Junos PyEZ Dependencies

Operating System	Dependencies
CentOS	<ul style="list-style-type: none"> <li>• pip</li> <li>• python-devel</li> <li>• libxml2-devel</li> <li>• libxslt-devel</li> <li>• gcc</li> <li>• openssl</li> <li>• libffi-devel</li> </ul>
Debian	<ul style="list-style-type: none"> <li>• python-pip</li> <li>• python-dev</li> <li>• libxml2-dev</li> <li>• libxslt-dev</li> <li>• libssl-dev</li> <li>• libffi-dev</li> </ul>
Fedora	<ul style="list-style-type: none"> <li>• python-pip</li> <li>• python-devel (required for Python 2)</li> <li>• python3-devel (required for Python 3)</li> <li>• libxml2-devel</li> <li>• libxslt-devel</li> <li>• gcc</li> <li>• openssl</li> <li>• libffi-devel</li> </ul>
FreeBSD	<ul style="list-style-type: none"> <li>• py27-pip</li> <li>• libxml2</li> <li>• libxslt</li> </ul>
OSX	<ul style="list-style-type: none"> <li>• xcode</li> <li>• pip</li> </ul> <p><b>NOTE:</b> If Junos PyEZ does not successfully install using <b>pip</b>, try using <b>easy_install</b> to install the <b>lxml</b> library and then Junos PyEZ.</p>
Ubuntu	<ul style="list-style-type: none"> <li>• python-pip</li> <li>• python-dev</li> <li>• libxml2-dev</li> <li>• libxslt-dev</li> <li>• libssl-dev</li> <li>• libffi-dev</li> </ul>
Windows	<ul style="list-style-type: none"> <li>• pip</li> <li>• pycrypto</li> <li>• ecdsa</li> </ul>

## Installing the Junos PyEZ Package

After you install the Junos PyEZ dependencies on the configuration management server, you can install the latest release of Junos PyEZ from the [Python Package Index \(PyPI\)](#). You can also download the latest version of the code from the Junos PyEZ GitHub repository. To install Junos PyEZ from GitHub, you must have Git installed on the configuration management server.

To install the current release of Junos PyEZ from PyPI, execute the following command on the command line (use **sudo pip** where appropriate):

```
[root@server]# pip install junos-eznc
```



**NOTE:** The **pip** command name might vary depending on your operating system and version of Python. Before installing Junos PyEZ using **pip**, use the **pip -V** command to display the version, and verify that the **pip** command corresponds to the version of Python that you are using for Junos PyEZ on your system. If the version is not the same as the Python version used for Junos PyEZ, then try using the **pip2** or **pip3** commands instead of **pip**.



**NOTE:** To upgrade an existing version of Junos PyEZ, include the **-U** or **--upgrade** option when executing the **pip install** command.

To install Junos PyEZ from the GitHub project master branch, execute the following command on the command line (use **sudo pip** where appropriate):

```
[root@server]# pip install git+https://github.com/Juniper/py-junos-eznc.git
```

For additional information about installing Junos PyEZ and any required dependencies, see the **INSTALL** file for your specific operating system in the Junos PyEZ GitHub repository at <https://github.com/Juniper/py-junos-eznc>.

## Using the Junos PyEZ Docker Image

Docker is a software container platform that is used to package and run an application and its dependencies in an isolated container. Juniper Networks provides a [Junos PyEZ Dockerfile](#) as well as [Junos PyEZ Docker images](#), which are automatically built for every Junos PyEZ release. The prebuilt Docker images include Python 2.7.x.

You can customize and use the Dockerfile to build your own Junos PyEZ Docker image, or you can use one of the prebuilt Docker images, which are stored on [Docker Hub](#), to run Junos PyEZ as a Docker container. You can run the container in interactive mode, as an executable package, or as a terminal.

To use a prebuilt Junos PyEZ Docker image on your configuration management server:

1. Install Docker.

See the Docker website at <https://www.docker.com> for instructions on installing and configuring Docker on your specific operating system.

2. Download the [juniper/pyez](#) Docker image from Docker Hub.

- To download the latest image, issue the following command:

```
user@host:~$ docker pull juniper/pyez
```



**NOTE:** The latest Junos PyEZ Docker image is built using the most recently committed code in the Junos PyEZ source repository, which is under active development and might not be stable.

- To download a specific image, append the appropriate release tag to the image name, for example, 2.1.2.

```
user@host:~$ docker pull juniper/pyez:tag
```

3. Move to the local directory that contains your scripts.

When you run the Docker container, the local scripts are mounted to `/scripts` in the container.

4. Run the container.

For instructions on running the container, see the official usage examples at [DOCKER-EXAMPLES.md](#).

#### Related Documentation

- [Setting Up Junos PyEZ Managed Nodes on page 24](#)
- [Understanding Junos PyEZ on page 17](#)
- [Junos PyEZ Modules Overview on page 19](#)
- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

---

## Setting Up Junos PyEZ Managed Nodes

Junos PyEZ is a Python library that enables you to manage and automate devices running Junos OS. You do not need to install any client software on the nodes in order to use Junos PyEZ to manage the devices. Also, Python is not required on the managed devices, because Junos PyEZ utilizes NETCONF and the Junos XML APIs.

You can use Junos PyEZ to manage devices running Junos OS using any user account that has access to the device. You can explicitly define the user when creating a new instance of the `jnpr.junos.device.Device` class, or if you do not specify a user in the parameter list, the user defaults to `$USER`. When you use Junos PyEZ to access and manage devices running Junos OS, Junos OS user account access privileges are enforced. The class configured for the Junos OS user account determines the permissions. Thus, if you use Junos PyEZ to load configuration changes onto a device, the user must have permissions to change the relevant portions of the configuration.

Junos PyEZ enables you to connect to a device running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. To use Junos PyEZ to telnet directly to a device, you must first configure the Telnet service on the managed device. To use Junos PyEZ to manage devices through a NETCONF session over SSH, you must enable the NETCONF service over SSH on the managed device and ensure that the device meets requirements for SSHv2 connections. The following sections outline the requirements and required configuration on devices running Junos OS when using Junos PyEZ to access the device using the different connection protocols:

- [Configuring Telnet Service on Devices Running Junos OS on page 25](#)
- [Enabling NETCONF on Devices Running Junos OS on page 26](#)
- [Satisfying Requirements for SSHv2 Connections on page 26](#)

## Configuring Telnet Service on Devices Running Junos OS

Starting in Junos PyEZ Release 2.0, Junos PyEZ applications can connect to a device running Junos OS using telnet. To telnet to a device running Junos OS, you must configure the Telnet service on the device. Configuring Telnet service for a device enables unencrypted, remote access to the device.

To enable Telnet service:

1. Configure the service.

```
[edit system services]
user@host# set telnet
```

2. (Optional) Configure the connection limit, rate limit, and order of authentication, as necessary.

```
[edit system services]
user@host# set telnet connection-limit connection-limit
user@host# set telnet rate-limit rate-limit
user@host# set telnet authentication-order [radius tacplus password]
```

3. Commit the configuration.

```
[edit]
user@host# commit
```

## Enabling NETCONF on Devices Running Junos OS

To enable the NETCONF over SSH service on the default port (830) on a device running Junos OS:

1. Configure the NETCONF over SSH service.

```
[edit system services]
user@host# set netconf ssh
```

2. Commit the configuration.

```
[edit]
user@host# commit
```

## Satisfying Requirements for SSHv2 Connections

The NETCONF server communicates with client applications within the context of a NETCONF session. The server and client explicitly establish a connection and session before exchanging data, and close the session and connection when they are finished. Junos PyEZ accesses the NETCONF server using the SSH protocol and standard SSH authentication mechanisms. When you use Junos PyEZ to manage devices running Junos OS, the most convenient way to access the devices is to configure SSH keys.

To establish an SSHv2 connection with a device running Junos OS, you must ensure that the following requirements are met:

- The NETCONF service over SSH is enabled on each device where a NETCONF session will be established.
- The client application has a user account and can log in to each device where a NETCONF session will be established.
- The login account used by the client application has an SSH public/private key pair or a text-based password configured.
- The client application can access the public/private keys or text-based password.

For additional information about enabling NETCONF on a device running Junos OS and satisfying the requirements for establishing an SSH session, see the *NETCONF XML Management Protocol Developer Guide*.

### Related Documentation

- [Installing Junos PyEZ on page 21](#)
- [Understanding Junos PyEZ on page 17](#)
- [Junos PyEZ Modules Overview on page 19](#)
- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## CHAPTER 4

# Authenticating Junos PyEZ Users

- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)

### Authenticating Junos PyEZ Users Using a Password

---

Junos PyEZ enables you to connect to and manage devices running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. The device must be able to authenticate the user using either a password or other standard SSH authentication mechanisms, depending on the connection method.

You can execute Junos PyEZ methods using any user account that has access to the managed device running Junos OS. You can explicitly define the user and password when creating a new instance of the `jnpr.junos.device.Device` class, or if you do not specify a user in the parameter list, the user defaults to `$USER`.

When establishing a NETCONF session over SSH, Junos PyEZ first attempts SSH public key-based authentication and then tries password-based authentication. When SSH keys are in use, the `passwd` argument is used as the passphrase for unlocking the private SSH key. When password-based authentication is used, the `passwd` argument is used as the password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase, then the `passwd` argument may be omitted. However, SSH private keys with empty passphrases are not recommended.

It is the user's responsibility to obtain the username and password authentication credentials in a secure manner appropriate for their environment. It is best practice to prompt for these authentication credentials during each invocation of the script, as shown in the following Python 3 example, rather than storing the credentials in an unencrypted format.

To authenticate a user using a password:

1. In your favorite editor, create a new file that uses the `.py` file extension.

This example uses the filename `junos-pyez-pw.py`.

2. Include code that prompts for the hostname, username, and password and stores each value in a variable.

```

from jnpr.junos import Device
from getpass import getpass
import sys

hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

```



**NOTE:** For Python 2.7, you can use the `raw_input()` function instead of `input()`, or you can install the `future` module and include the `"from builtins import input"` line in your application to make the code compatible with both Python 2 and 3.

3. In the **Device** constructor argument list, set the **host**, **user**, and **passwd** arguments to reference the appropriate variables, and include any additional arguments required for the connection method.

The following example provides sample code for each of the different connection methods:

```

from jnpr.junos import Device
from getpass import getpass
import sys

hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

# NETCONF session over SSH
dev = Device(host=hostname, user=username, passwd=password)

# Telnet connection
#dev = Device(host=hostname, user=username, passwd=password, mode='telnet',
#             port='23')

# Serial console connection
#dev = Device(host=hostname, user=username, passwd=password, mode='serial',
#             port='/dev/ttyUSB0')

try:
    dev.open()
except Exception as err:
    print (err)
    sys.exit(1)

print (dev.facts)
dev.close()

```



**NOTE:** All platforms running Junos OS have only the root user configured by default, without any password. When using Junos PyEZ to initially configure a new or zeroized device through a console connection, use `user='root'`, and omit the `passwd` parameter.

4. Execute the Junos PyEZ code, which prompts for the hostname, username, and password and does not echo the password on the command line.

```
[user@localhost]$ python junos-pyez-pw.py
Device hostname: dc1a.example.com
Device username: bsmith
Device password:
{'domain': 'example.com', 'serialnumber': 'JN1120AAAAAB', 'ifd_style':
'CLASSIC', 'version_info': junos.version_info(major=(13, 3), type=R, minor=1,
build=8), '2RE': True, 'hostname': 'dc1a', 'fqdn': 'dc1a.example.com',
'switch_style': 'NONE', 'version': '13.3R1.8', 'HOME': '/var/home/bsmith',
'model': 'MX240', 'RE0': {'status': 'OK', 'last_reboot_reason': 'Router rebooted
after a normal shutdown.', 'model': 'RE-S-1300', 'up_time': '14 days, 17
hours, 45 minutes, 8 seconds'}, 'personality': 'MX'}
```

#### Related Documentation

- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS on page 177](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## Authenticating Junos PyEZ Users Using SSH Keys

Junos PyEZ enables you to manage devices running Junos OS through a NETCONF session over SSH. When you use Junos PyEZ to manage a device running Junos OS using NETCONF over SSH, the device must be able to authenticate the user using standard SSH authentication mechanisms.

You can execute Junos PyEZ methods using any user account that has access to the managed device running Junos OS. You can explicitly define the user when creating a new instance of the `jnpr.junos.device.Device` class, or if you do not specify a user in the parameter list, the user defaults to `$USER`. When you use Junos PyEZ to manage devices running Junos OS through a NETCONF session over SSH, the most convenient and secure way to access the devices is to configure SSH keys. SSH keys enable the remote device to identify trusted users.

Junos PyEZ first attempts SSH public key-based authentication and then tries password-based authentication. When SSH keys are in use, the `passwd` argument is used as the passphrase for unlocking the private SSH key. When password-based authentication is used, the `passwd` argument is used as the password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase,

then the **passwd** argument may be omitted. However, SSH private keys with empty passphrases are not recommended.

It is the user's responsibility to obtain the username and password authentication credentials in a secure manner appropriate for their environment. It is best practice to prompt for these authentication credentials during each invocation of the script rather than storing the credentials in an unencrypted format.

To use SSH keys in a Junos PyEZ application, you must first generate the keys on the configuration management server and configure the public key on each managed device running Junos OS. You must then include the appropriate arguments in the **Device** argument list.

Junos PyEZ can utilize SSH keys that are generated in either the default location or a user-defined location and that either use or forgo password protection. Junos PyEZ also checks for keys that are actively loaded into an SSH key agent.

The following sections outline the steps for generating the SSH keys, configuring the keys on devices running Junos OS, and connecting to the device using the keys:

1. [Generating and Configuring SSH Keys on page 30](#)
2. [Referencing SSH Keys in Junos PyEZ Applications on page 31](#)

## Generating and Configuring SSH Keys

To generate SSH keys on the configuration management server and configure the public key on devices running Junos OS:

1. On the server, generate the public and private SSH key pair for the desired user, and provide any required or desired options, for example:

```
[user@localhost]$ cd ~/.ssh
[user@localhost]$ ssh-keygen -t rsa
Enter file in which to save the key (/home/user/.ssh/id_rsa): id_rsa_dc
Enter passphrase (empty for no passphrase): *****
Enter same passphrase again: *****
```

2. (Optional) Load the key into the native SSH key agent.
3. Configure the public key under the appropriate user account on all devices running Junos OS that will be managed using this key.

The easiest method is to create a file that contains the public key and then load the file into the configuration.

```
[edit]
[user@router]# set system login user username authentication load-key-file URL
[user@router]# commit
```

4. Verify that the key works by logging in to the device using the key.

```
[user@localhost]$ ssh -i ~/.ssh/id_rsa_dc router.example.com
Enter passphrase for key '/home/user/.ssh/id_rsa_dc':
```

```
user@router>
```

## Referencing SSH Keys in Junos PyEZ Applications

After generating the SSH key pair and configuring the public key on the device running Junos OS, you can connect to the device using the key by including the appropriate arguments in the **Device** constructor code. The **Device** arguments are determined by the location of the key, whether the key is password-protected, and whether the key is actively loaded into an SSH key agent, such as ssh-agent. The following sections outline the various scenarios:

- [Authenticating the User Using an SSH Key Agent with Actively Loaded Keys on page 31](#)
- [Authenticating the User Using SSH Keys Without Password Protection on page 31](#)
- [Authenticating the User Using Password-Protected SSH Key Files on page 32](#)

### Authenticating the User Using an SSH Key Agent with Actively Loaded Keys

You can use an SSH key agent to securely store private keys and avoid repeatedly retyping the passphrase for password-protected keys. If you do not provide a password or SSH key file in the arguments of the **Device** constructor, Junos PyEZ first checks the SSH keys that are actively loaded in the SSH key agent and then checks for SSH keys in the default location.

To connect to a device running Junos OS using SSH keys that are actively loaded into the native SSH key agent:

- In the **Device** argument list, you need only supply the required hostname and any desired variables.

```
dev = Device(host='dc1a.example.com')
```

### Authenticating the User Using SSH Keys Without Password Protection

To connect to a device running Junos OS using SSH keys that are in the default location and do not have password protection:

- In the **Device** argument list, you need only supply the required hostname and any desired variables.

```
dev = Device(host='dc1a.example.com')
```

Junos PyEZ first checks the SSH keys that are loaded in any active SSH key agent and then checks the SSH keys in the default location.

To connect to a device running Junos OS using SSH keys that are not in the default location and do not have password protection:

- In the **Device** argument list, set the **ssh\_private\_key\_file** argument to the path of the SSH private key.

```
dev = Device(host='dc1a.example.com',
             ssh_private_key_file='/home/user/.ssh/id_rsa_dc')
```

### Authenticating the User Using Password-Protected SSH Key Files

---

To connect to a device running Junos OS using a password-protected SSH key file:

1. Include code that prompts for the private key password and stores the value in a variable.

```
from jnpr.junos import Device
from getpass import getpass

passwd = getpass('Enter password for SSH private key file: ')
```

2. In the `Device` argument list, set the `ssh_private_key_file` argument to the path of the private key, and set the `passwd` argument to reference the password variable.

```
from jnpr.junos import Device
from getpass import getpass

host = 'dc1a.example.com'
key_file = '/home/user/.ssh/id_rsa_dc'

passwd = getpass('Enter password for SSH private key file: ')
dev = Device(host=host, passwd=passwd, ssh_private_key_file=key_file)
dev.open()
...
```

#### Related Documentation

- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS on page 177](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## CHAPTER 5

# Connecting to and Retrieving Facts From a Device Using Junos PyEZ

- [Connecting to Devices Running Junos OS Using Junos PyEZ on page 33](#)
- [Understanding Junos PyEZ Device Facts and Connection Properties on page 40](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)
- [Accessing the Shell on Devices Running Junos OS Using Junos PyEZ on page 46](#)

## Connecting to Devices Running Junos OS Using Junos PyEZ

---

Junos PyEZ is a microframework for Python that enables you to manage devices running Junos OS. Junos PyEZ models each device as an instance of the [jnpr.junos.device.Device](#) class. The **Device** class enables you to connect to a device using a serial console connection, telnet, or by establishing a NETCONF session over SSH.

The following sections provide an overview of the connection methods and details about how to use each connection type to connect to a device running Junos OS:

- [Connection Methods Overview on page 33](#)
- [Connecting to a Device Using NETCONF over SSH on page 34](#)
- [Connecting to a Device Using Telnet on page 37](#)
- [Connecting to a Device Using a Serial Console Connection on page 39](#)

## Connection Methods Overview

Junos PyEZ enables you to connect to devices running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. You must use a serial console connection when you are physically connected to the **CONSOLE** port on a device. You can use telnet to connect to the device's management interface or to a console server that is directly connected to the device's **CONSOLE** port. New or zeroized devices that have factory default configurations require access through a console connection. Thus, you can use Junos PyEZ to initially configure a device that is not yet configured for remote access by using either a serial console connection when you are directly connected to the device or by using telnet through a console server that is directly connected to the device.

By default, Junos PyEZ connects to a device and establishes a NETCONF session over SSH. To specify a different connection type, you must include the **mode** parameter in the **Device** argument list. To telnet to a device, include the **mode='telnet'** argument. To connect to a device using a serial console connection, include the **mode='serial'** argument. [Table 6 on page 34](#) summarizes the Junos PyEZ connection methods, their default values for certain parameters, and the Junos PyEZ release in which support for that connection method was first introduced.

**Table 6: Junos PyEZ Connection Methods**

Connection Mode	Value of mode Argument	Default Port	First Supported Junos PyEZ Release
NETCONF over SSH (default)	–	830	1.0
Serial console connection	serial	/dev/ttyUSB0	2.0
Telnet	telnet	23	2.0



**NOTE:** Before you can access the management interface using telnet or NETCONF over SSH, you must first enable the appropriate service at the `[edit system services]` hierarchy. For more information, see [“Setting Up Junos PyEZ Managed Nodes” on page 24](#). Because telnet uses clear-text passwords (therefore creating a potential security vulnerability), we recommend that you use SSH.



**NOTE:** It is the user's responsibility to obtain the username and password authentication credentials in a secure manner appropriate for their environment. It is best practice to prompt for these authentication credentials during each invocation of the script, rather than storing the credentials in an unencrypted format.

Junos PyEZ supports using context managers for all connection methods. When you use a context manager, Junos PyEZ automatically calls the **open()** and **close()** methods to connect to and disconnect from the device. If you do not use a context manager, you must explicitly call the **open()** and **close()** methods in your application.

## Connecting to a Device Using NETCONF over SSH

By default, Junos PyEZ connects to a device and establishes a NETCONF session over SSH. To use this connection method, you must first satisfy the requirements outlined in [“Setting Up Junos PyEZ Managed Nodes” on page 24](#). In addition, the device must be able to authenticate the user using standard SSH authentication mechanisms. For more information, see [“Authenticating Junos PyEZ Users Using a Password” on page 27](#) and [“Authenticating Junos PyEZ Users Using SSH Keys” on page 29](#).

When establishing a NETCONF session over SSH, Junos PyEZ first attempts SSH public key-based authentication and then tries password-based authentication. When SSH keys are in use, the **passwd** argument is used as the passphrase for unlocking the private SSH key. When password-based authentication is used, the **passwd** argument is used as the password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase, then the **passwd** argument may be omitted. However, SSH private keys with empty passphrases are not recommended.

To establish a NETCONF session over SSH with a device running Junos OS and print the device facts in a Junos PyEZ application using Python 3:

1. Import the **Device** class and any other modules or objects required for your tasks.

```
import sys
from getpass import getpass
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError
```

2. Create the device instance, and provide the hostname, any parameters required for authentication, and any optional parameters.

```
hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

dev = Device(host=hostname, user=username, passwd=password)
```

3. Connect to the device by calling the **open()** method, for example:

```
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    sys.exit(1)
except Exception as err:
    print (err)
    sys.exit(1)
```

4. Print the device facts.

```
print (dev.facts)
```

5. After performing any necessary tasks, close the connection to the device.

```
dev.close()
```

The sample program in its entirety is presented here:

```
import sys
from getpass import getpass
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError

hostname = input("Device hostname: ")
username = input("Device username: ")
```

```
password = getpass("Device password: ")

dev = Device(host=hostname, user=username, passwd=password)
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    sys.exit(1)
except Exception as err:
    print (err)
    sys.exit(1)

print (dev.facts)
dev.close()
```

Alternatively, you can use a context manager when connecting to the device. For example:

```
import sys
from getpass import getpass
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError

hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

try:
    with Device(host=hostname, user=username, passwd=password) as dev:
        print (dev.facts)
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    sys.exit(1)
except Exception as err:
    print (err)
    sys.exit(1)
```

Junos PyEZ automatically queries the default SSH configuration file at `~/.ssh/config`, if one exists. However, starting with Junos PyEZ Release 1.2, you can specify a different SSH configuration file when you create the device instance by including the `ssh_config` parameter in the `Device` argument list. For example:

```
ssh_config_file = "~/.ssh/config_dc"
dev = Device(host='198.51.100.1', ssh_config=ssh_config_file)
```

Also, starting in Junos PyEZ Release 1.2, Junos PyEZ provides support for ProxyCommand, which enables you to access a target device through an intermediary host that supports netcat. This is useful when you can only log in to the target device through the intermediate host.

To configure ProxyCommand, add the appropriate information to the SSH configuration file. For example:

```
[user1@localhost ~]$ cat ~/.ssh/config
Host 198.51.100.1
User user1
ProxyCommand ssh -l user1 198.51.100.2 nc %h 22 2>/dev/null
```

## Connecting to a Device Using Telnet

The Junos PyEZ **Device** class enables you to connect to a device running Junos OS using telnet, which provides unencrypted access to the network device. You can telnet to the device's management interface or to a console server that is directly connected to the device's **CONSOLE** port. You must configure the Telnet service at the **[edit system services]** hierarchy level on all devices that require access to the management interface. Accessing the device through a console server enables you to initially configure a new or zeroized device that is not yet configured for remote access.

To use Junos PyEZ to telnet to a device running Junos OS, you must include **mode='telnet'** in the **Device** argument list, and optionally include the **port** parameter to specify a port. When you include the **mode='telnet'** argument but omit the **port** parameter, the value for **port** defaults to 23.

To use Junos PyEZ to telnet to a device running Junos OS and print the device facts in a Junos PyEZ application using Python 3:

1. Import the **Device** class and any other modules or objects required for your tasks.

```
import sys
from getpass import getpass
from jnpr.junos import Device
```

2. Create the device instance with the **mode='telnet'** argument, specify the connection port if different from the default, and provide the hostname, any parameters required for authentication, and any optional parameters.

```
hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

dev = Device(host=hostname, user=username, passwd=password, mode='telnet',
port='23')
```

3. Connect to the device by calling the **open()** method.

```
try:
    dev.open()
except Exception as err:
    print (err)
    sys.exit(1)
```

4. Print the device facts.

```
print (dev.facts)
```

5. After performing any necessary tasks, close the connection to the device.

```
dev.close()
```

The sample program in its entirety is presented here:

```
import sys
from getpass import getpass
from jnpr.junos import Device

hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

dev = Device(host=hostname, user=username, passwd=password, mode='telnet',
port='23')

try:
    dev.open()
except Exception as err:
    print (err)
    sys.exit(1)

print (dev.facts)
dev.close()
```

Alternatively, you can use a context manager when connecting to the device. For example:

```
import sys
from getpass import getpass
from jnpr.junos import Device

hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

try:
    with Device(host=hostname, user=username, passwd=password, mode='telnet',
port='23') as dev:
        print (dev.facts)
except Exception as err:
    print (err)
    sys.exit(1)
```

In some cases, when you connect to a console server that emits a banner message, you might be required to press **Enter** after the message to reach the login prompt. If a Junos PyEZ application opens a Telnet session with a console server that requires the user to press **Enter** after a banner message, the application might fail to receive the login prompt, which can cause the connection to hang.

Starting in Junos PyEZ Release 2.1.0, a Junos PyEZ application can include the **console\_has\_banner=True** argument in the **Device** argument list to telnet to a console server that emits a banner message.

```
dev = Device(host=hostname, user=username, passwd=password, mode='telnet',
console_has_banner=True)
```

When you include the **console\_has\_banner=True** argument and the application does not receive a login prompt upon initial connection, the application waits for 5 seconds and then emits a newline (**\n**) character so that the console server issues the login prompt. If you omit the argument and the connection hangs, the application instead emits the **<close-session/>** RPC to terminate the connection.

## Connecting to a Device Using a Serial Console Connection

The Junos PyEZ **Device** class enables you to connect to a device running Junos OS using a serial console connection, which is useful when you must initially configure a new or zeroized device that is not yet configured for remote access. To use this connection method, you must be physically connected to the device through the **CONSOLE** port. For detailed instructions about connecting to the **CONSOLE** port on your device, see the hardware documentation for your specific device.



**NOTE:** Junos PyEZ supports using context managers for serial console connections. We recommend that you use a context manager for console connections, because the context manager automatically handles opening and closing the connection. Failure to close the connection can lead to unpredictable results.

To use Junos PyEZ to connect to a device running Junos OS through a serial console connection, you must include **mode='serial'** in the **Device** argument list, and optionally include the **port** parameter to specify a port. When you include the **mode='serial'** argument but omit the **port** parameter, the value for **port** defaults to **/dev/ttyUSB0**.

To connect to a device running Junos OS using a serial console connection and also load and commit a configuration on the device in a Junos PyEZ application using Python 3:

1. Import the **Device** class and any other modules or objects required for your tasks.

```
import sys
from getpass import getpass
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
```

2. Create the device instance with the **mode='serial'** argument, specify the connection port if different from the default, and provide any parameters required for authentication and any optional parameters.

```
username = input("Device username: ")
password = getpass("Device password: ")

try:
    with Device(mode='serial', port='port' user=username, passwd=password)
    as dev:
        print (dev.facts)
```



**NOTE:** All platforms running Junos OS have only the root user configured by default, without any password. For new or zeroized devices, use **user='root'** and omit the **passwd** parameter.

3. Load and commit the configuration on the device.

```
cu = Config(dev)
cu.lock()
cu.load(path='/tmp/config_mx.conf')
cu.commit()
cu.unlock()
```

4. Include any necessary error handling.

```
except Exception as err:
    print (err)
    sys.exit(1)
```

The sample program in its entirety is presented here:

```
import sys
from getpass import getpass
from jnpr.junos import Device
from jnpr.junos.utils.config import Config

username = input("Device username: ")
password = getpass("Device password: ")

try:
    with Device(mode='serial', port='port', user=username, passwd=password)
as dev:
    print (dev.facts)
    cu = Config(dev)
    cu.lock()
    cu.load(path='/tmp/config_mx.conf')
    cu.commit()
    cu.unlock()

except Exception as err:
    print (err)
    sys.exit(1)
```

#### Related Documentation

- [Setting Up Junos PyEZ Managed Nodes on page 24](#)
- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)
- [Troubleshooting Junos PyEZ Connection Errors on page 176](#)

---

## Understanding Junos PyEZ Device Facts and Connection Properties

After connecting to a device running Junos OS, Junos PyEZ applications can retrieve facts about the device and query for information about the current connection. The device facts are accessed as the **facts** attribute of the **Device** object instance, and information about the current connection is stored as properties of the **Device** instance. For detailed information about the keys that are included in the facts dictionary, see [jnpr.junos.facts](#).

The following example establishes a NETCONF session over SSH with the device and prints the device facts:

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

with Device(host='router1.example.net') as dev:
    pprint (dev.facts)

user1@host:~$ python get-facts.py
{'2RE': True,
 'HOME': '/var/home/user1',
 'RE0': {'last_reboot_reason': '0x200:normal shutdown',
         'mastership_state': 'master',
         'model': 'RE-MX-104',
         'status': 'OK',
         'up_time': '25 days, 8 hours, 22 minutes, 40 seconds'},
 'RE1': {'last_reboot_reason': '0x200:normal shutdown',
         'mastership_state': 'backup',
         'model': 'RE-MX-104',
         'status': 'OK',
         'up_time': '25 days, 8 hours, 23 minutes, 55 seconds'},
 ...
```

In Junos PyEZ Release 2.0.0 and earlier releases, when you call the **Device open()** method to connect to a device, Junos PyEZ automatically gathers the device facts for NETCONF-over-SSH connections and gathers the device facts for Telnet and serial console connections when you explicitly include **gather\_facts=True** in the **Device** argument list. Starting in Junos PyEZ Release 2.1.0, device facts are gathered on demand for all connection types. Each fact is gathered and cached the first time the application accesses its value or the value of a dependent fact. When you print or use device facts, previously accessed facts are served from the cache, and facts that have not yet been accessed are retrieved from the device.

Junos PyEZ caches a device fact when it first accesses the fact or a dependent fact, but it does not update the cached value upon subsequent access. To refresh the device facts, call the **facts\_refresh()** method. The **facts\_refresh()** method empties the cache of all facts, such that when the application next accesses a fact, it retrieves it from the device and stores the current value in the cache.

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

with Device(host='router1.example.net') as dev:
    pprint (dev.facts)
    dev.facts_refresh()
    pprint (dev.facts)
```

To refresh a single fact or a set of facts, include the **keys** argument in the **facts\_refresh** method, and specify the list of keys to clear from the cache. For example:

```
dev.facts_refresh(keys='hostname')
dev.facts_refresh(keys=('hostname', 'domain', 'master'))
```



**NOTE:** Starting in Junos PyEZ Release 2.0.0, exceptions that occur when gathering facts raise a warning instead of an error, which enables the script to continue running.

The Junos PyEZ **Device** class also provides a number of properties that you can query for information about the current connection. [Table 7 on page 42](#) outlines the properties. For example, after connecting to a device, you can query the **connected** property to return the current state of the connection. A **SessionListener** monitors the session and responds to transport errors by raising a **TransportError** exception and setting the **Device.connected** property to **False**.

**Table 7: Device Properties**

Property	Description
<b>connected</b>	Boolean specifying the current state of the connection. Returns <b>True</b> when connected.
<b>hostname</b>	String specifying the hostname of the device running Junos OS.
<b>master</b>	Boolean returning <b>True</b> if the Routing Engine to which the application is connected is the master Routing Engine.
<b>port</b>	String specifying the port used for the connection.
<b>re_name</b>	String specifying the Routing Engine name to which the application is connected.
<b>timeout</b>	Integer specifying the RPC timeout value in seconds.
<b>user</b>	String specifying the user accessing the device.

The following sample code prints the value of the **connected** property after connecting to a device running Junos OS and again after closing the session.

```
from jnpr.junos import Device

dev = Device(host='router1.example.net')
dev.open()
print (dev.connected)
dev.close()

print (dev.connected)
```

When you execute the program, the **connected** property returns **True** while the application is connected to the device and returns **False** after the connection is closed.

```
user@host:~$ python connect.py
True
False
```

Release History Table

Release	Description
2.1.0	Starting in Junos PyEZ Release 2.1.0, device facts are gathered on demand for all connection types.
2.0.0	Starting in Junos PyEZ Release 2.0.0, exceptions that occur when gathering facts raise a warning instead of an error, which enables the script to continue running.

#### Related Documentation

- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)
- [Connecting to Devices Running Junos OS Using Junos PyEZ on page 33](#)

## Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS

Junos PyEZ is a microframework for Python that enables you to manage and automate devices running Junos OS. Junos PyEZ models each device as an instance of the `jnpr.junos.device.Device` class. For detailed information about the `jnpr.junos.device` module, see <http://junos-pyez.readthedocs.io/en/latest/jnpr.junos.html#module-jnpr.junos.device>.

After connecting to a device running Junos OS, you can retrieve and print facts about the device. These facts are accessed as the `facts` attribute of a `Device` object instance. For detailed information about the keys that are included in the facts dictionary, see [jnpr.junos.facts](#).



**NOTE:** Starting in Junos PyEZ Release 2.1.0, device facts are gathered on demand for all connection types. Each fact is gathered and cached the first time you access its value or the value of a dependent fact. In earlier releases, when you call the `Device.open()` method to connect to a device, Junos PyEZ automatically gathers the device facts for NETCONF-over-SSH connections and gathers the device facts for Telnet and serial console connections when you explicitly include `gather_facts=True` in the `Device` argument list.

With Junos PyEZ, you can quickly execute commands in Python interactive mode, or you can create programs to perform tasks. The following example establishes a NETCONF session over SSH with a device running Junos OS and retrieves and prints facts for the device using both a simple Python program and Python interactive mode.

To create a Junos PyEZ application that establishes a NETCONF session over SSH with a device running Junos OS and prints out the device facts:

1. In your favorite editor, create a new file with a descriptive name that uses the **.py** file extension.
2. Import the **Device** class and any other modules or objects required for your tasks.

```
import sys
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError
```

3. Create the device instance and provide the hostname, any parameters required for authentication, and any optional parameters.

```
dev = Device(host='router1.example.net')
```

4. Connect to the device by calling the **open()** method.

```
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    sys.exit(1)
```

5. Print the device facts.

```
print (dev.facts)
print (dev.facts['hostname'])
```



**TIP:** To refresh the facts for a device, call the `facts_refresh()` method, for example, `dev.facts_refresh()`.

6. After performing any necessary tasks, close the connection to the device.

```
dev.close()
```

7. Save and execute the program.

```
[root@server]# python filename.py
```

The entire program is presented here:

```
import sys
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError

dev = Device(host='router1.example.net')
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
```

```

sys.exit(1)

print (dev.facts)
dev.close()

```

You can also quickly perform the same operations in Python interactive mode.

```

[root@server]# python
Python 2.7.6 (default, Jun 16 2014, 18:27:48)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from jnpr.junos import Device
>>>
>>> dev = Device('router1.example.net')
>>> dev.open()
Device(router1.example.net)
>>>
>>> print (dev.facts)
{'domain': 'example.com', 'serialnumber': 'JN1120AAAAAA', 'ifd_style': 'CLASSIC',
 'version_info': junos.version_info(major=(13, 3), type=R, minor=1, build=8),
 '2RE': True, 'hostname': 'router1', 'fqdn': 'router1.example.net', 'switch_style':
 'NONE', 'version': '13.3R1.8', 'HOME': '/root', 'model': 'MX240', 'RE0':
 {'status': 'OK', 'last_reboot_reason': 'Router rebooted after a normal shutdown.',
 'model': 'RE-S-1300', 'up_time': '14 days, 17 hours, 45 minutes, 8 seconds'},
 'personality': 'MX'}
>>>
>>> dev.close()
>>> quit()

```

The following video presents a short Python session that demonstrates how to use Junos PyEZ to connect to and retrieve facts from a device running Junos OS.



Video: [Junos PyEZ - Hello, World](#)

By default, Junos PyEZ returns the device facts as a Python dictionary. Starting in Junos PyEZ Release 1.2, you can view the device facts in YAML or JavaScript Object Notation (JSON). To view a YAML or JSON representation of the facts, import the **yaml** or **json** module, respectively, and call either the **yaml.dump()** or **json.dumps()** function, as appropriate.

```

import json
import yaml
import sys
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError

dev = Device('router1.example.net')
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {}".format(err))
    sys.exit(1)

print (yaml.dump(dev.facts))

```

```
print (json.dumps(dev.facts))
dev.close()
```

#### Release History Table

Release	Description
2.1.0	Starting in Junos PyEZ Release 2.1.0, device facts are gathered on demand for all connection types.

#### Related Documentation

- [Understanding Junos PyEZ Device Facts and Connection Properties on page 40](#)
- [Connecting to Devices Running Junos OS Using Junos PyEZ on page 33](#)
- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Troubleshooting Junos PyEZ Connection Errors on page 176](#)

## Accessing the Shell on Devices Running Junos OS Using Junos PyEZ

The Junos OS command-line interface (CLI) has many operational mode commands to replace shell commands. However in some cases, a user or application might need to access the UNIX-level shell on devices running Junos OS and execute shell commands or CLI commands from the shell. The Junos PyEZ `jnpr.junos.utils.start_shell` module defines the `StartShell` class, which enables Junos PyEZ applications to initiate an SSH connection to a device running Junos OS and access the shell. The `StartShell` methods enable the application to then execute commands over the connection and retrieve the response.

The `StartShell run()` method executes a shell command and waits for the response. If you omit the `this="string"` argument, the method waits for one of the default shell prompts (% or #) before returning the command output. If you set `this` to a specific string, the method waits for the expected string or pattern before returning the command output. The return value is a tuple, where the first item is `True` if the exit code is 0, and `False` otherwise, and the second item is the output of the command.

The following example connects to a host and executes two operational mode commands from the shell. The script first executes the `request support information` command and saves the output to a file. The script then executes the `show version` command, stores the output in the `version` variable, and then prints the contents of the variable.

```
from jnpr.junos import Device
from jnpr.junos.utils.start_shell import StartShell

dev = Device(host='router1.example.net')
dev.open()

ss = StartShell(dev)
ss.open()
ss.run('cli -c "request support information | save /var/tmp/information.txt"')
version = ss.run('cli -c "show version"')
print (version)
ss.close()
```



to a specific string or pattern, the method might return partial output for a nonreturning command if it encounters a default prompt or the specified string pattern within the command output.

The following sample code executes the **monitor traffic interface fxp0** command, waits for 15 seconds, and then retrieves and returns the command output.

```
from jnpr.junos import Device
from jnpr.junos.utils.start_shell import StartShell

dev = Device(host='router1.example.net')
dev.open()

with StartShell(dev) as ss:
    ss.run('cli -c "monitor traffic interface fxp0"', this=None, timeout=15)

dev.close()
```

For more information about the **StartShell** class and its methods, see

[http://junos-pyez.readthedocs.io/en/latest/jnpr.junos.utils.html#module-jnpr.junos.utils.start\\_shell](http://junos-pyez.readthedocs.io/en/latest/jnpr.junos.utils.html#module-jnpr.junos.utils.start_shell).

#### Related Documentation

- [Using Junos PyEZ to Execute RPCs on Devices Running Junos OS on page 89](#)
- [Connecting to Devices Running Junos OS Using Junos PyEZ on page 33](#)

## CHAPTER 6

# Configuring Devices Running Junos OS

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Using the Junos PyEZ Config Utility to Load Configuration Data on page 53](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)
- [Using Junos PyEZ to Commit the Candidate Configuration on page 68](#)

## Using Junos PyEZ to Configure Devices Running Junos OS

---

Junos PyEZ enables you to make structured and unstructured configuration changes on devices running Junos OS. The user account that is used to make configuration changes must have permissions to change the relevant portions of the configuration on each device. If you do not define a user, the user defaults to **\$USER**.

The following sections compare structured and unstructured configuration changes and provide details about the Junos PyEZ configuration process when making unstructured configuration changes using the **Config** utility or structured configuration changes using Tables and Views.

- [Structured vs Unstructured Configuration Changes on page 49](#)
- [Configuration Process on page 50](#)

## Structured vs Unstructured Configuration Changes

Unstructured configuration changes, which consist of static or templated configuration data that is formatted as ASCII text, Junos XML elements, Junos OS **set** commands, or JSON, are performed using the **jnpr.junos.utils.config.Config** utility. In contrast, structured configuration changes use Junos PyEZ configuration Tables and Views to define specific resources to configure. When you add the Table to the Junos PyEZ framework, Junos PyEZ dynamically creates a configuration class for the resource, which enables you to programmatically configure that resource on a device.

When you use unstructured changes to modify the configuration of devices running Junos OS, you can change any portion of the configuration, but you must use one of the accepted formats for the configuration data as well as the correct syntax for that format. Users who are familiar with the supported configuration formats and want the option to modify any portion of the configuration might favor this method for configuration changes.

Structured configuration changes, on the other hand, require that you create Tables and Views to define specific resources and only enable you to configure the defined resources on the device. When you define a structured resource, you can specify the configuration statements that the user can configure for that resource, and you can also define type and constraint checks to ensure that the users supply acceptable values for the data in their Junos PyEZ application. Once a Table and View have been created, they can easily be shared and reused. A Table user can programmatically configure the resource on a device, and the user does not require any knowledge of supported configuration formats or their syntax.

For more information about using the **Config** utility to make unstructured configuration changes, see:

- [Using the Junos PyEZ Config Utility to Load Configuration Data on page 53](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)
- [jnpr.junos.utils.config.Config](#)

For more information about using configuration Tables and Views to make structured configuration changes, see:

- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)
- [Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS on page 161](#)

## Configuration Process

Junos PyEZ enables you to make configuration changes on devices running Junos OS. After successfully connecting to the device, you create a **Config** or Table object and associate it with the **Device** object. For example:

Config Object	<pre>from jnpr.junos import Device from jnpr.junos.utils.config import Config  dev = Device(host='dc1a.example.com').open() cu = Config(dev)</pre>
Table object	<pre>from jnpr.junos import Device from myTables.ConfigTables import ServicesConfigTable  dev = Device(host='dc1a.example.com').open() sct = ServicesConfigTable(dev)</pre>

The basic process for making configuration changes is to lock the configuration, load the configuration changes, commit the configuration to make it active, and then unlock the configuration. When using the Junos PyEZ **Config** utility to make unstructured configuration changes in the shared configuration database on a device, you can perform these actions by calling the appropriate instance methods outlined here:

1. Lock the configuration using **lock()**
2. Load the new configuration or configuration changes using **load()**, roll back the configuration using **rollback()**, or revert to a rescue configuration using **rescue()**
3. Commit the configuration using **commit()**
4. Unlock the configuration using **unlock()**

If you instead use Tables and Views to make structured configuration changes on a device, you can choose to call the various methods individually, or you can call the **set()** method, which calls the **lock()**, **load()**, **commit()**, and **unlock()** methods automatically.



**NOTE:** The **load()** method performs the same function for Table objects and `jnpr.junos.utils.config.Config` objects, but you supply different parameters depending on which object type calls the method.

In Junos PyEZ, you can load configuration changes using a **load merge**, **load replace**, **load overwrite**, or **load update** operation. You specify the desired load operation by including or omitting the appropriate parameters in the **set()** method when making structured configuration changes using Tables and Views, or in the **load()** method for either structured or unstructured configuration changes. [Table 8 on page 51](#) summarizes the parameter settings required for each type of load operation.

**Table 8: Parameters for Specifying the Load Operation Type in the `load()` and `set()` Methods**

Load Operation	Argument	First Supported Junos PyEZ Release	Description
<b>load merge</b>	<b>merge=True</b>	1.0	Merge the loaded configuration with the existing configuration.
<b>load overwrite</b>	<b>overwrite=True</b>	1.0	Replace the entire configuration with the loaded configuration.
<b>load replace</b> (Default)	–	1.0	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the <b>replace</b> tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
<b>load update</b> ( <b>Config load()</b> method only)	<b>update=True</b>	2.1.0	Compare the complete loaded configuration against the existing configuration. Each configuration element that is different in the loaded configuration replaces its corresponding element in the existing configuration. During the commit operation, only system processes that are affected by changed configuration elements parse the new configuration.



**NOTE:** Because the `load update` operation requires a complete configuration, the `update=True` argument must not be used when making configuration changes using Tables, which only modify specific statements in the configuration.

Starting in Junos PyEZ Release 2.0, you can specify the configuration mode to use when making structured or unstructured configuration changes. To make configuration changes in a mode other than the default configuration mode, which updates the shared configuration database, you must create the **Config** or Table object using a context manager and include the `mode` parameter in the argument list. Supported modes include **private**, **exclusive**, **dynamic**, and **batch**.

When you specify one of these modes, the context manager handles opening and locking and closing and unlocking the database. This ensures that you do not unintentionally leave the database in a locked state. Thus, you only need to call the `load()` and `commit()` methods to configure the device.

For example, the following code makes both structured and unstructured configuration changes using the **configure private** mode:

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from myTables.ConfigTables import ServicesConfigTable

dev = Device(host='host').open()

with Config(dev, mode='private') as cu:
    cu.load('set system services netconf traceoptions file test.log',
    format='set')
    cu.pdiff()
    cu.commit()

with ServicesConfigTable(dev, mode='private') as sct:
    sct.ftp = True
    sct.ssh = True
    sct.telnet = True
    sct.append()
    sct.load()
    sct.pdiff()
    sct.commit()
```



**NOTE:** The context manager handles opening and locking the configuration database in **private**, **exclusive**, **dynamic**, or **batch** mode. Thus, calling the `lock()` or `set()` methods in one of these modes results in a `LockError` exception.

The **Device** class `bind()` method enables you to attach various instances and methods to the **Device** instance. In your Junos PyEZ application, you have the option to bind the **Config** or Table object to the **Device** instance. The functionality of the methods does not change, but the method execution differs slightly. For example:

As a standalone variable:

```
dev = Device(host='dc1a.example.com').open()
cu = Config(dev)
cu.lock()
```

As a bound property:

```
dev = Device(host='dc1a.example.com').open()
dev.bind( cu=Config )
dev.cu.lock()
```

#### Related Documentation

- [Using the Junos PyEZ Config Utility to Load Configuration Data on page 53](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)
- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS on page 161](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)

## Using the Junos PyEZ Config Utility to Load Configuration Data

Junos PyEZ enables you to make structured and unstructured configuration changes on devices running Junos OS. Unstructured configuration changes, which consist of static or templated configuration data that is formatted as ASCII text, Junos XML elements, Junos OS **set** commands, or JavaScript Object Notation (JSON), are performed using the `jnpr.junos.utils.config.Config` utility. When using Junos PyEZ to make unstructured configuration changes with the **Config** utility, you can specify the configuration mode, the load operation, and the format to use for the changes.

When you create the **Config** object and do not explicitly specify a configuration mode, by default, Junos PyEZ makes updates to the shared configuration database. To make configuration changes in a mode other than the default configuration mode, you must create the **Config** object using a context manager and include the **mode** parameter in the argument list. Supported modes include **private**, **exclusive**, **dynamic**, and **batch**.

For example, the following code makes configuration changes using the **configure private** mode:

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

dev = Device(host="host").open()
with Config(dev, mode='private') as cu:
    cu.load('set system services netconf traceoptions file test.log',
    format='set')
    cu.pdiff()
    cu.commit()
```

In Junos PyEZ, you can load configuration changes using a **load merge**, **load replace**, **load overwrite**, or **load update** operation. You specify the desired load operation by including

or omitting the appropriate parameters in the **Config load()** method. By default, Junos PyEZ performs a **load replace** operation. To perform a **load merge**, **load overwrite**, or **load update** operation, set the **merge**, **overwrite**, or **update** parameter, respectively, to **True** in the **load()** method. For example:

```
cu.load(config_mx, overwrite=True)
```

For more information about specifying the type of load operation, see [“Using Junos PyEZ to Configure Devices Running Junos OS” on page 49](#).

The Junos PyEZ **Config** utility enables you to configure devices running Junos OS using one of the standard, supported formats. You can provide configuration data as strings, files, XML objects, or Jinja2 Template objects. Files can contain either configuration data snippets or Jinja2 templates. When providing configuration data within a string, file, or Jinja2 template, supported formats for the data include ASCII text, Junos XML elements, Junos OS **set** commands, and JSON. You can specify the format of the configuration data either by explicitly including the **format** parameter in the **Config** utility **load()** method or by adding the appropriate extension to the configuration data file. If you do not specify a format, the default is XML.



**NOTE:** Starting in Junos PyEZ Release 1.2, Junos PyEZ automatically detects the format when you supply the configuration data as a string.

[Table 9 on page 54](#) summarizes the supported formats for the configuration data and the corresponding value for the file extension and **format** parameter. When using Junos XML formatting for the configuration data, you must enclose the data in the top-level **<configuration>** tag.



**NOTE:** You do not need to enclose configuration data that is formatted as ASCII text, Junos OS **set** commands, or JSON in **<configuration-text>**, **<configuration-set>**, or **<configuration-json>** tags as required when configuring the device directly within a NETCONF session.

**Table 9: Specifying the Format for Configuration Data**

Configuration Data Format	File Extension	format Parameter
ASCII text	<b>.conf</b> , <b>.text</b> , <b>.txt</b>	text
JavaScript Object Notation (JSON)	<b>.json</b>	json
Junos OS <b>set</b> commands	<b>.set</b>	set
Junos XML elements	<b>.xml</b>	xml



**NOTE:** When the `overwrite` or `update` parameter is set to `True`, you cannot use the Junos OS `set` command format.



**NOTE:** Starting in Junos PyEZ Release 2.0, Junos PyEZ supports loading configuration data in JSON format on devices running Junos OS Release 16.1R1 and later releases.

Table 10 on page 55 summarizes the `load()` method parameters that you use to pass in or reference the location of the configuration data. You must always specify the format of the data by including the `format` parameter in the method call except when using strings, XML objects, or files that have the format indicated by the file extension. When using Jinja2 templates, include the `template_vars` parameter to pass in the dictionary of required template variables.

**Table 10: Referencing Configuration Data in the `load()` Method**

Parameter	Configuration Data
<code>vargs[0]</code>	XML object or a string that contains configuration data formatted as ASCII text, Junos XML elements, Junos OS <code>set</code> commands, or JSON. Junos PyEZ automatically detects the format of the configuration data in this case, and the <code>format</code> parameter is not required.
<code>path</code>	Path to a file containing configuration data formatted as ASCII text, Junos XML elements, Junos OS <code>set</code> commands, or JSON. You must also include the <code>format</code> parameter when the file extension does not indicate the format of the data.
<code>template_path</code>	Path to a file containing a Jinja2 template formatted as ASCII text, Junos XML elements, Junos OS <code>set</code> commands, or JSON. Include the <code>template_vars</code> parameter to reference a dictionary containing any required Jinja2 template variables. You must also include the <code>format</code> parameter when the file extension does not indicate the format of the data.
<code>template</code>	Pre-loaded Jinja2 Template object. Include the <code>template_vars</code> parameter to reference a dictionary containing any required Jinja2 template variables.

The following sections discuss how to construct the Junos PyEZ `load()` method to load different types of configuration data when making unstructured configuration changes with the **Config** utility. In the examples, `cu` is an instance of the **Config** utility, which operates on the target device running Junos OS.

- [Loading Configuration Data from a File on page 56](#)
- [Loading Configuration Data from a String on page 56](#)
- [Loading Configuration Data Formatted as an XML Object on page 57](#)
- [Loading Jinja2 Templates on page 58](#)

## Loading Configuration Data from a File

To load configuration data that is formatted as ASCII text, Junos XML elements, Junos OS **set** commands, or JSON from a file, set the **path** parameter to the path of the file. For example:

```
conf_file = "configs/junos-config-interfaces.conf"
cu.load(path=conf_file, merge=True)
```

If you do not use one of the accepted file extensions to indicate the format of the configuration data, then you must include the **format** parameter in the **load()** method parameter list. For example:

```
conf_file = "configs/junos-config-interfaces"
cu.load(path=conf_file, format="text", merge=True)
```

## Loading Configuration Data from a String

To load configuration data that is formatted as ASCII text, Junos XML elements, Junos OS **set** commands, or JSON from a string, include the string as the first argument in the **load()** method parameter list. Starting in Junos PyEZ Release 1.2, Junos PyEZ automatically detects the format of the configuration data in strings, so the **format** parameter is optional in this case.

The following code snippets present sample multiline strings containing configuration data in the different formats and the corresponding calls to the **load()** method. The optional **format** parameter is explicitly included in each example for clarity.

- For configuration data formatted as ASCII text:

```
config_text = """
system {
  scripts {
    op {
      file test.slax;
    }
  }
}
"""
```

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="text"**.

```
cu.load(config_text, format="text", merge=True)
```

- For configuration data formatted as Junos XML:

```
config_xml = """
<configuration>
  <system>
    <scripts>
      <op>
        <file>
          <name>test.slax</name>
        </file>
      </op>
    </scripts>
  </system>
</configuration>
"""
```

```
</configuration>
"""
```

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="xml"**.

```
cu.load(config_xml, format="xml", merge=True)
```

- For configuration data formatted as Junos OS **set** commands:

```
config_set = """
set system scripts op file test.slax
"""
```

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="set"**.

```
cu.load(config_set, format="set", merge=True)
```

- For configuration data formatted using JSON:

```
config_json = """{
  "configuration" : {
    "system" : {
      "scripts" : {
        "op" : {
          "file" : [
            {
              "name" : "test.slax"
            }
          ]
        }
      }
    }
  }
}"""
```

Load the configuration data by supplying the string as the first argument in the list, and optionally specify **format="json"**.

```
cu.load(config_json, format="json", merge=True)
```

## Loading Configuration Data Formatted as an XML Object

To load configuration data formatted as an XML object, include the object as the first argument in the **load()** method parameter list. Because the default format for configuration data is XML, you do not need to explicitly include the **format** parameter in the method call.

The following code snippets present an XML object and the corresponding call to the **load()** method:

```
from lxml.builder import E

config_xml_obj = (
    E.configuration(           # create an Element called "configuration"
        E.system(
            E.scripts(
                E.op (
                    E.file (
                        E.name("test.slax"),
                    )
                )
            )
        )
    )
```

```

    )
  )
)

```

Load the configuration data by supplying the XML object as the first argument in the list, and include any other required parameters.

```
cu.load(config_xml_obj, merge=True)
```

## Loading Jinja2 Templates

Junos PyEZ supports using Jinja2 templates to render Junos OS configuration data. Jinja is a template engine for Python that enables you to generate documents from predefined templates. The templates, which are text files in the desired language, provide flexibility through the use of expressions and variables. You can create Junos OS configuration data using Jinja2 templates in one of the supported configuration formats, which includes ASCII text, Junos XML elements, Junos OS **set** commands, and JSON. Junos PyEZ uses the Jinja2 template and a supplied dictionary of variables to render the configuration data.

Jinja2 templates provide a powerful method to generate configuration data, particularly for similar configuration stanzas. For example, rather than manually adding the same configuration statements for each interface on a device, you can create a template that iterates over a list of interfaces and creates the required configuration statements for each one. The following sample Jinja2 template generates configuration data that enables MPLS on logical unit 0 for each interface in a given list and also configures the interface under the MPLS and RSVP protocols. In Jinja, blocks are delimited by '{%' and '%}' and variables are enclosed within '{{' and '}}'.

The `configs/junos-config-interfaces-mpls.conf` file contains the following Jinja2 template:

```

interfaces {
    {% for item in interfaces %}
    {{ item }} {
        description "{{ description }}";
        unit 0 {
            family {{ family }};
        }
    } {% endfor %}
}
protocols {
    mpls {
        {% for item in interfaces %}
        interface {{ item }};
        {% endfor %}
    }
    rsvp {
        {% for item in interfaces %}
        interface {{ item }};
        {% endfor %}
    }
}

```

In the Junos PyEZ code, the corresponding dictionary of Jinja2 template variables is:

```
config = {
    'interfaces': ['ge-1/0/1', 'ge-1/0/2', 'ge-1/0/3'],
    'description': 'MPLS interface',
    'family': 'mpls'
}
```

To load the Jinja2 template in the Junos PyEZ code, set the **template\_path** parameter to the path of the template file, and set the **template\_vars** parameter to the dictionary of template variables. If you do not use one of the accepted file extensions to indicate the format of the configuration data, then you must include the **format** parameter in the **load()** method parameter list.

```
conf_file = "configs/junos-config-interfaces-mpls.conf"
cu.load(template_path=conf_file, template_vars=config, merge=True)
```



**NOTE:** If you are supplying a pre-loaded Jinja2 Template object, you must use the **template** parameter instead of the **template\_path** parameter in the **load()** method argument list.

When Junos PyEZ renders the output, it generates the following configuration data, which is loaded into the candidate configuration on the device:

```
interfaces {
  ge-1/0/1 {
    description "MPLS interface";
    unit 0 {
      family mpls;
    }
  }
  ge-1/0/2 {
    description "MPLS interface";
    unit 0 {
      family mpls;
    }
  }
  ge-1/0/3 {
    description "MPLS interface";
    unit 0 {
      family mpls;
    }
  }
}
protocols {
  mpls {
    interface ge-1/0/1;
    interface ge-1/0/2;
    interface ge-1/0/3;
  }
  rsvp {
    interface ge-1/0/1;
    interface ge-1/0/2;
    interface ge-1/0/3;
  }
}
```

The following video presents a short Python session that demonstrates how to use a Jinja2 template to configure a device running Junos OS.



Video: [Junos PyEZ - YAML, Jinja2, Template Building, Configuration Deployment, Oh My!](#)

For additional information about Jinja2, see the Jinja2 documentation at <http://jinja.pocoo.org/docs/>.

#### Release History Table

Release	Description
1.2	Starting in Junos PyEZ Release 1.2, Junos PyEZ automatically detects the format when you supply the configuration data as a string.

#### Related Documentation

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)
- [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)
- [Junos PyEZ Modules Overview on page 19](#)

### Example: Using Junos PyEZ to Load Configuration Data from a File

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ package provides simple yet powerful methods to perform certain operational and configuration tasks on devices running Junos OS. This example outlines one method for using the Junos PyEZ `jnpr.junos.utils.config.Config` utility to load configuration data from a file onto a device running Junos OS.

- [Requirements on page 60](#)
- [Overview on page 61](#)
- [Configuration on page 61](#)
- [Executing the Junos PyEZ Code on page 65](#)
- [Verification on page 65](#)
- [Troubleshooting on page 66](#)

#### Requirements

This example uses the following hardware and software components:

- Configuration management server running Python 2.6, 2.7, or 3.4 and Junos PyEZ Release 2.0 or a later release

- Device running Junos OS with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the server and device running Junos OS

## Overview

This example presents a Python program that uses the Junos PyEZ **Config** utility to enable a new op script in the configuration of the specified device. The **junos-config-add-op-script.conf** file contains the relevant configuration data formatted as ASCII text.

The Python program imports the **Device** class, which handles the connection with the device running Junos OS; the **Config** class, which is used to make unstructured configuration changes on the target device; and required exceptions from the **jnpr.junos.exception** module, which contains exceptions encountered when managing devices running Junos OS. This example binds the **Config** instance to the **Device** instance rather than creating a standalone variable for the instance of the **Config** class.

After creating the **Device** instance for the target device, the **open()** method establishes a connection and NETCONF session with the device. The **Config** utility methods then lock the candidate configuration, load the configuration changes into the candidate configuration as a **load merge** operation, commit the candidate configuration, and then unlock it.

The **load()** method **path** parameter is set to the path of the configuration file. Because the configuration file extension indicates the format of the configuration data, the **format** parameter is omitted from the argument list. Setting **merge=True** indicates that the device should perform a **load merge** operation.

After the configuration operations are complete, the NETCONF session and connection are terminated using the **close()** method. The Python program includes code for handling exceptions such as **LockError** for errors that occur when locking the configuration and **CommitError** for errors that occur during the commit operation. The program also includes code to handle any additional exceptions that might occur.

## Configuration

- [Creating the Configuration Data File on page 61](#)
- [Creating the Junos PyEZ Program on page 62](#)
- [Results on page 64](#)

### Creating the Configuration Data File

#### Step-by-Step Procedure

To create the configuration data file that is used by the Junos PyEZ program:

1. Create a new file with the appropriate extension based on the format of the configuration data, which in this example is ASCII text.

2. Include the desired configuration changes in the file, for example:

```
system {
  scripts {
    op {
      file bgp-neighbors.slax;
    }
  }
}
```

---

### Creating the Junos PyEZ Program

**Step-by-Step Procedure** To create a Python program that uses Junos PyEZ to make configuration changes on a device running Junos OS:

1. Import any required modules, classes, and objects.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError
from jnpr.junos.exception import LockError
from jnpr.junos.exception import UnlockError
from jnpr.junos.exception import ConfigLoadError
from jnpr.junos.exception import CommitError
```

2. Include any required variables, which for this example includes the hostname of the managed device and the path to the file containing the configuration data.

```
host = 'dc1a.example.com'
conf_file = 'configs/junos-config-add-op-script.conf'
```

3. Create a **main()** function definition and function call, and place the remaining statements within the definition.

```
def main():

    if __name__ == "__main__":
        main()
```

4. Create an instance of the **Device** class, and supply the hostname and any parameters required for that specific connection.

Then open a connection and establish a NETCONF session with the device.

```
# open a connection with the device and start a NETCONF session
try:
    dev = Device(host=host)
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    return
```

5. Bind the **Config** instance to the **Device** instance.

```
dev.bind(cu=Config)
```

6. Lock the configuration.

```
# Lock the configuration, load configuration changes, and commit
print ("Locking the configuration")
try:
    dev.cu.lock()
except LockError as err:
    print ("Unable to lock configuration: {0}".format(err))
    dev.close()
    return
```

7. Load the configuration changes and handle any errors.

```
print ("Loading configuration changes")
try:
    dev.cu.load(path=conf_file, merge=True)
except (ConfigLoadError, Exception) as err:
    print ("Unable to load configuration changes: {0}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError:
        print ("Unable to unlock configuration: {0}".format(err))
    dev.close()
    return
```

8. Commit the configuration.

```
print ("Committing the configuration")
try:
    dev.cu.commit(comment='Loaded by example.')
except CommitError as err:
    print ("Unable to commit configuration: {0}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {0}".format(err))
    dev.close()
    return
```

9. Unlock the configuration.

```
print ("Unlocking the configuration")
try:
    dev.cu.unlock()
except UnlockError as err:
    print ("Unable to unlock configuration: {0}".format(err))
```

10. End the NETCONF session and close the connection with the device.

```
# End the NETCONF session and close the connection  
dev.close()
```

---

## Results

On the configuration management server, review the completed program. If the program does not display the intended code, repeat the instructions in this example to correct the program.

```
from jnpr.junos import Device  
from jnpr.junos.utils.config import Config  
from jnpr.junos.exception import ConnectError  
from jnpr.junos.exception import LockError  
from jnpr.junos.exception import UnlockError  
from jnpr.junos.exception import ConfigLoadError  
from jnpr.junos.exception import CommitError  
  
host = 'dc1a.example.com'  
conf_file = 'configs/junos-config-add-op-script.conf'  
  
def main():  
    # open a connection with the device and start a NETCONF session  
    try:  
        dev = Device(host=host)  
        dev.open()  
    except ConnectError as err:  
        print ("Cannot connect to device: {0}".format(err))  
        return  
  
    dev.bind(cu=Config)  
  
    # Lock the configuration, load configuration changes, and commit  
    print ("Locking the configuration")  
    try:  
        dev.cu.lock()  
    except LockError as err:  
        print ("Unable to lock configuration: {0}".format(err))  
        dev.close()  
        return  
  
    print ("Loading configuration changes")  
    try:  
        dev.cu.load(path=conf_file, merge=True)  
    except (ConfigLoadError, Exception) as err:  
        print ("Unable to load configuration changes: {0}".format(err))  
        print ("Unlocking the configuration")  
        try:  
            dev.cu.unlock()  
        except UnlockError:  
            print ("Unable to unlock configuration: {0}".format(err))  
        dev.close()  
        return  
  
    print ("Committing the configuration")  
    try:  
        dev.cu.commit(comment='Loaded by example.')
```

```

except CommitError as err:
    print ("Unable to commit configuration: {0}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {0}".format(err))
dev.close()
return

print ("Unlocking the configuration")
try:
    dev.cu.unlock()
except UnlockError as err:
    print ("Unable to unlock configuration: {0}".format(err))

# End the NETCONF session and close the connection
dev.close()

if __name__ == "__main__":
    main()

```

## Executing the Junos PyEZ Code

**Step-by-Step Procedure** To execute the Junos PyEZ code:

- On the configuration management server, execute the program.

```

[root@server]# python junos-pyez-config.py
Locking the configuration
Loading configuration changes
Committing the configuration
Unlocking the configuration

```

## Verification

### Verifying the Configuration

**Purpose** Verify that the configuration was correctly updated on the device running Junos OS.

**Action** Log in to the device running Junos OS and view the configuration, commit history, and log files to verify the configuration and commit. For example:

```
root@dc1a> show configuration system scripts
op {
    file bgp-neighbors.slax;
}

root@dc1a> show system commit
0 2014-07-29 14:40:50 PDT by root via netconf
...

root@dc1a> show log messages
Jul 29 14:40:36 dc1a sshd[75843]: Accepted publickey for root from 198.51.100.1
    port 54811 ssh2: RSA 02:dd:53:3e:f9:97:dd:1f:d9:31:e9:7f:82:06:aa:67
Jul 29 14:40:36 dc1a sshd[75843]: subsystem request for netconf by user root
Jul 29 14:40:42 dc1a file[75846]: UI_COMMIT: User 'root' requested 'commit'
operation (comment: Loaded by example.)
Jul 29 14:40:45 dc1a mspd[75888]: mspd: No member config
Jul 29 14:40:45 dc1a mspd[75888]: mspd: Building package info
Jul 29 14:40:51 dc1a mspd[1687]: mspd: No member config
Jul 29 14:40:51 dc1a mspd[1687]: mspd: Building package info
Jul 29 14:40:51 dc1a file[75846]: UI_COMMIT_COMPLETED: commit complete
```

**Meaning** The configuration and the log file contents indicate that the correct configuration statements were successfully configured and committed on the device.

## Troubleshooting

- [Troubleshooting Timeout Errors on page 66](#)
- [Troubleshooting Configuration Lock Errors on page 67](#)
- [Troubleshooting Configuration Change Errors on page 68](#)

---

### Troubleshooting Timeout Errors

**Problem** The Junos PyEZ code generates an `RpcTimeoutError` message or a `TimeoutExpiredError` message and fails to update the device configuration.

```
RpcTimeoutError(host: dc1a.example.com, cmd: commit-configuration, timeout: 30)
```

The default time for a NETCONF RPC to time out is 30 seconds. Large configuration changes might exceed this value causing the operation to time out before the configuration can be uploaded and committed.

**Solution** To accommodate configuration changes that might require a commit time that is longer than the default timeout interval, set the timeout interval to an appropriate value and rerun the code. To configure the interval, either set the **Device timeout** property to an appropriate value, or include the **timeout=seconds** argument when you call the **commit()** method to commit the configuration data on a device. For example:

```
dev = Device(host="host")
dev.open()
dev.timeout = 300

...
dev.commit(timeout=360)
```

---

### Troubleshooting Configuration Lock Errors

---

**Problem** The Junos PyEZ code generates a `LockError` message indicating that the configuration cannot be locked. For example:

```
LockError(severity: error, bad_element: None, message: configuration database
modified)
```

A configuration lock error can occur for the following reasons:

- Another user has an exclusive lock on the configuration.
- Another user made changes to the shared configuration database but has not yet committed the changes.
- The user executing the Junos PyEZ code does not have permissions to configure the device.

**Solution** If another user has an exclusive lock on the configuration or has modified the configuration, wait until the lock is released or the changes are committed, and execute the code again. If the cause of the issue is that the user does not have permissions to configure the device, either execute the program with a user who has the necessary permissions, or if appropriate, configure the device running Junos OS to give the current user the necessary permissions to make the changes.

## Troubleshooting Configuration Change Errors

---

**Problem** The Junos PyEZ code generates a `ConfigLoadError` message indicating that the configuration cannot be modified due to a permissions issue.

```
ConfigLoadError(severity: error, bad_element: scripts, message: permission denied)
```

This error message might be generated when the user executing the Junos PyEZ code has permission to alter the configuration, but does not have permission to alter the desired portion of the configuration.

**Solution** Either execute the program with a user who has the necessary permissions, or if appropriate, configure the device running Junos OS to give the current user the necessary permissions to make the changes.

**Related Documentation**

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Using the Junos PyEZ Config Utility to Load Configuration Data on page 53](#)
- [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)
- [Junos PyEZ Modules Overview on page 19](#)

## Using Junos PyEZ to Commit the Candidate Configuration

---

Junos PyEZ enables you to make structured and unstructured configuration changes on devices running Junos OS. After successfully connecting to the device, the basic process for making changes to the shared configuration database is to lock the configuration, load the configuration changes, commit the configuration to make it active, and then unlock the configuration. When you commit the candidate configuration on a device running Junos OS, it becomes the active configuration.

The following sections detail how to commit the configuration in your Junos PyEZ application and outline the options supported for the commit operation.

- [Committing the Candidate Configuration on page 68](#)
- [Including Commit Options on page 69](#)

### Committing the Candidate Configuration

When you use the Junos PyEZ `jnpr.junos.utils.Config` utility to make unstructured configuration changes on a device, you commit the candidate configuration by calling the `Config` instance `commit()` method.

```
with Device(host='dc1a.example.com') as dev:
    cu = Config(dev)
    cu.lock()
    cu.load(path=conf_file, merge=True)
```

```
cu.commit()
```

```
cu.unlock()
```

To just verify the syntax of the configuration without actually committing it, call the `commit_check()` method in place of the `commit()` method.

```
cu.commit_check()
```

When you use Junos PyEZ configuration Tables and Views to make structured configuration changes on a device, you commit the candidate configuration by calling either the `set()` method, which automatically calls the `lock()`, `load()`, `commit()` and `unlock()` methods, or by calling the various methods individually.

```
with Device(host='dc1a.example.com') as dev:
    userconfig = UserConfigTable(dev)
    # ...set the values for the configuration data...
    userconfig.append()
```

```
userconfig.set()
```



**NOTE:** If you create the Config or Table object using a context manager and set the mode argument to private, exclusive, dynamic, or batch, you only call the `load()` and `commit()` methods to configure the device. The context manager handles opening and locking, then closing and unlocking the database, so calling the `lock()` or `set()` methods in one of these modes results in a `LockError` exception.

## Including Commit Options

The Junos OS command-line interface (CLI) includes options for the commit operation, such as adding a commit comment or synchronizing the configuration on multiple Routing Engines. You can use these same commit options in your Junos PyEZ application by including the appropriate arguments in the `commit()` or `set()` method argument list. [Table 11 on page 69](#) outlines the supported commit options, which can be used in either method, and provides the Junos PyEZ Release in which support for the option was first introduced.

**Table 11: Junos PyEZ Supported Commit Options**

Commit Option Argument	Description	CLI command	Junos PyEZ Release
<code>comment="comment"</code>	Log a comment for that commit operation in the system log file and in the device's commit history.	<code>commit comment "comment"</code>	1.0

Table 11: Junos PyEZ Supported Commit Options (*continued*)

Commit Option Argument	Description	CLI command	Junos PyEZ Release
<b>confirm=(True   minutes)</b>	Require that a commit operation be confirmed within a specified amount of time after the initial commit. Otherwise, roll back to the previously committed configuration.  Set the argument to <b>True</b> to use the default time of 10 minutes.	<b>commit confirmed &lt;minutes&gt;</b>	1.0
<b>detail=True</b>	Return an XML object with detailed information about the commit process.	<b>commit   display detail   display xml</b>	1.2
<b>force_sync=True</b>	Synchronize and commit the configuration on both Routing Engines, even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine.	<b>commit synchronize force</b>	1.2
<b>sync=True</b>	Synchronize and commit the configuration on both Routing Engines.	<b>commit synchronize</b>	1.2

When you commit the configuration, you can include a brief comment to describe the purpose of the committed changes. To log a comment describing the changes, include the **comment** parameter and a message string in the **commit()** or **set()** method argument list, as appropriate. For example:

```
cu.commit(comment='Configuring ge-0/0/0 interface')
```

Including the **comment** parameter in the argument list is equivalent to issuing the **commit comment** configuration mode command in the Junos OS CLI. The comment is logged to the system log file and included in the device's commit history, which you can view by issuing the **show system commit** command in the CLI.

To require that a commit operation be confirmed within a specified amount of time after the initial commit, include the **confirm=minutes** argument in the **commit()** or **set()** method argument list, as appropriate.

```
cu.commit(confirm=15)
```

If the commit is not confirmed within the given time limit, the configuration automatically rolls back to the previously committed configuration and a broadcast message is sent to all logged-in users. The allowed range is 1 through 65,535 minutes. You can also specify **confirm=True** to use the default rollback time of 10 minutes. To confirm the commit operation, call either the **commit()** or **commit\_check()** method.

The confirmed commit operation is useful for verifying that a configuration change works correctly and does not prevent management access to the device. If the change prevents access or causes other errors, the automatic rollback to the previous configuration enables access to the device after the rollback deadline passes. If you lose connectivity to the device, you must issue the Junos PyEZ **open()** method to restore connectivity.

If the device has dual Routing Engines, you can synchronize and commit the configuration on both Routing Engines by including the **sync=True** argument in the **commit()** or **set()** method argument list.

```
cu.commit(sync=True)
```

When you include the **sync=True** argument, the device copies the candidate configuration stored on the local Routing Engine to the other Routing Engine, verifies the candidate's syntactic correctness, and commits it on both Routing Engines. To force the **commit synchronize** operation to succeed even if there are open configuration sessions or uncommitted configuration changes on the other Routing Engine, use the **force\_sync=True** argument, which causes the device to terminate any configuration sessions on the other Routing Engine before synchronizing and committing the configuration.

```
cu.commit(force_sync=True)
```

You can review the details of the entire commit operation by including the **detail=True** argument in the **commit()** or **set()** method argument list. When you include this argument, the method returns an XML object with detailed information about the commit process. The return value is equivalent to the contents enclosed by the **<commit-results>** element in the output of the **commit | display detail | display xml** command in the CLI.

```
from lxml import etree
...
commit_detail = cu.commit(detail=True)
print(etree.tostring(commit_detail, encoding='unicode'))
```

#### Related Documentation

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)



## CHAPTER 7

# Comparing Configurations Using Junos PyEZ

- Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73

## Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration

---

Devices running Junos OS store a copy of the most recently committed configuration and up to 49 previous configurations. The Junos PyEZ `jnpr.junos.utils.config.Config` utility enables you to compare the candidate configuration to a previously committed configuration and print or return the difference. The `pdiff()` method prints the difference directly to standard output, whereas the `diff()` method returns the difference as an object. The methods are equivalent to issuing the `show | compare rollback n` configuration mode command in the Junos OS command-line interface (CLI).

The `diff()` and `pdiff()` methods retrieve the difference between the candidate configuration and a previously committed configuration, which is referenced by the rollback ID parameter, `rb_id`, in the method call. If the parameter is omitted, the rollback ID defaults to 0, which corresponds to the active or most recently committed configuration.

The difference is returned in patch format. Statements that exist only in the candidate configuration are prefixed with a plus sign (+), and statements that exist only in the comparison configuration and not in the candidate configuration are prefixed with a minus sign (-). If there is no difference, the methods return or print **None**.

In a Junos PyEZ application, after establishing a connection with the device, you can call the `diff()` or `pdiff()` method for a `Config` or `Table` object to compare the candidate and rollback configurations. The following code snippet uses the `Config` class to load unstructured configuration changes into the candidate configuration and then calls the `pdiff()` method to print the difference between the modified candidate configuration and the active configuration.

```
dev = Device(host="dc1a.example.com")
dev.open()

cu = Config(dev)

cu.lock()
cu.load(config_xml, format="xml", merge=True)
```

```
cu.pdiff()
...
```

When you execute the code, it prints the differences to standard output. For example:

```
[edit system scripts op]
+   file bgp-neighbors.slax;
[edit interfaces]
+   ge-1/0/0 {
+       unit 0 {
+           family inet {
+               address 198.51.100.1/26;
+           }
+       }
+   }
-   ge-1/1/0 {
-       unit 0 {
-           family inet {
-               address 198.51.100.65/26;
-           }
-       }
-   }
```

To retrieve the difference between the configurations as an object for further manipulation, call the **diff()** method instead of the **pdiff()** method, and store the output in a variable. For example:

```
diff = cu.diff(rb_id=2)
```

When you use Junos PyEZ configuration Tables and Views to make structured configuration changes on a device, you can load and commit the configuration data either by calling the **lock()**, **load()**, **commit()** and **unlock()** methods individually, or by calling the **set()** method, which calls all of these methods automatically. If you use configuration Tables to configure a device, and you want to compare the updated candidate configuration to a previously committed configuration using the **diff()** or **pdiff()** methods in your application, you must use the **load()** and **commit()** methods instead of the **set()** method. Doing this enables you to retrieve the differences after the configuration data is loaded into the candidate configuration but before it is committed. For example:

```
with Device(host="dc1a.example.com") as dev:

    userconf = UserConfigTable(dev)
    userconf.user = 'user1'
    userconf.class_name = 'read-only'
    userconf.password = '$ABC123'
    userconf.append()

    userconf.lock()
    userconf.load(merge=True)
    userconf.pdiff()

    userconf.commit()
    userconf.unlock()
    ...
```

#### Related Documentation

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Example: Using Junos PyEZ to Roll Back the Configuration on page 83](#)

- [Using Junos PyEZ to Manage the Rescue Configuration on page 109](#)
- [Junos PyEZ Modules Overview on page 19](#)



## CHAPTER 8

# Requesting Configuration Information Using Junos PyEZ

- [Using Junos PyEZ to Retrieve a Configuration on page 77](#)

## Using Junos PyEZ to Retrieve a Configuration

---

Junos PyEZ applications can execute RPCs on demand on devices running Junos OS. After creating an instance of the **Device** class, an application can execute RPCs as a property of the **Device** instance. Junos PyEZ applications can use the **get\_config()** RPC to request the complete configuration or selected configuration hierarchies for both the native Junos OS configuration as well as for configuration data corresponding to standard (IETF, OpenConfig) or custom YANG data models that have been added to the device.



**NOTE:** The Junos PyEZ **get\_config** RPC invokes the Junos XML protocol **<get-configuration>** operation. For more information about the **<get-configuration>** operation and its options, see [<get-configuration>](#).

The following sections discuss how to retrieve the configuration by using the Junos PyEZ **get\_config()** RPC. For information about using Tables and Views to retrieve configuration data, see [“Defining Junos PyEZ Configuration Tables” on page 141](#) and [“Understanding Junos PyEZ Tables and Views” on page 119](#).

- [Retrieving the Entire Configuration on page 78](#)
- [Specifying the Source of Configuration Data to Return on page 78](#)
- [Specifying the Format for Configuration Data to Return on page 78](#)
- [Specifying the Scope of Configuration Data to Return on page 79](#)
- [Retrieving Configuration Data for Standard or Custom YANG Data Models on page 79](#)
- [Handling Namespaces in Configuration Data on page 81](#)

## Retrieving the Entire Configuration

To retrieve the entire candidate configuration from a device running Junos OS, execute the `get_config()` RPC, and print the result. The default output format is XML. For example:

```
from jnpr.junos import Device
from lxml import etree

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config()
    print(etree.tostring(data, encoding='unicode'))
```

## Specifying the Source of Configuration Data to Return

When a Junos PyEZ application uses the `get_config()` RPC to retrieve configuration information from a device running Junos OS, by default, the server returns configuration data from the candidate configuration. A Junos PyEZ application can specify that the `get_config()` RPC return configuration data from the committed configuration database by including the `options` argument with `'database':'committed'` in the RPC call.

```
from jnpr.junos import Device
from lxml import etree

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config(options={'database' : 'committed'})
    print(etree.tostring(data, encoding='unicode'))
```

## Specifying the Format for Configuration Data to Return

The Junos PyEZ `get_config()` RPC invokes the Junos XML protocol `<get-configuration>` operation, which can return Junos OS configuration data as Junos XML elements, CLI configuration statements, Junos OS `set` commands, or JavaScript Object Notation (JSON). By default, the `get_config()` RPC returns configuration data as XML.

To specify the format in which to return the configuration data, the Junos PyEZ application includes the `options` dictionary with `'format':'format'` in the `get_config()` argument list. To request CLI configuration statements, Junos OS `set` commands, or JSON format, set the `format` value to `text`, `set`, or `json`, respectively.

As in NETCONF and Junos XML protocol sessions, Junos PyEZ returns the configuration data in the expected format enclosed within the appropriate XML element for that format. The RPC reply encloses configuration data in XML, text, or `set` command formats in `<configuration>`, `<configuration-text>`, and `<configuration-set>` elements, respectively.

```
from jnpr.junos import Device
from lxml import etree
from pprint import pprint

with Device(host='router1.example.net') as dev:

    # XML format (default)
    data = dev.rpc.get_config()
    pprint(etree.tostring(data, encoding='unicode'))

    # Text format
    data = dev.rpc.get_config(options={'format':'text'})
    print(etree.tostring(data))

    # Junos OS set format
```

```
data = dev.rpc.get_config(options={'format':'set'})
print (etree.tostring(data))

# JSON format
data = dev.rpc.get_config(options={'format':'json'})
pprint (data)
```



**NOTE:** Depending on the version of Python and the format of the output, you might need to modify the print statement to display more human-readable output.

## Specifying the Scope of Configuration Data to Return

In addition to retrieving the complete Junos OS configuration, a Junos PyEZ application can retrieve specific portions of the configuration by invoking the `get_config()` RPC with the `filter_xml` argument, which defines the configuration elements to return. The `filter_xml` parameter can take a string with the Junos XML elements or XPath representation of the configuration statements to return, or it can take an `lxml.etree.Element` with the Junos XML elements to return. The value of `filter_xml` must represent all levels of the configuration hierarchy starting just under the root (represented by the `<configuration>` element) down to each element to display.

The following Junos PyEZ application retrieves and prints the configuration at the `[edit system services]` hierarchy level using different but equivalent values for the `filter_xml` argument.

```
from jnpr.junos import Device
from lxml import etree

with Device(host='router1.example.net') as dev:

    # Junos XML elements
    data = dev.rpc.get_config(filter_xml='<system><services/></system>')
    print(etree.tostring(data, encoding='unicode'))

    # XPath expression
    data = dev.rpc.get_config(filter_xml='system/services')
    print(etree.tostring(data, encoding='unicode'))

    # lxml.etree.Element
    filter = etree.XML('<system><services/></system>')
    data = dev.rpc.get_config(filter_xml=filter)
    print(etree.tostring(data, encoding='unicode'))
```

## Retrieving Configuration Data for Standard or Custom YANG Data Models

You can load standardized or custom YANG modules onto devices running Junos OS to add data models that are not natively supported by Junos OS but can be supported by translation. You configure nonnative data models in the candidate configuration by using the syntax defined for those models. When you commit the configuration, the translation scripts translate those portions of the data and commit the corresponding Junos OS configuration as a transient change in the checkout configuration.

The candidate and active configurations contain the configuration data for nonnative YANG data models in the syntax defined by those models. However, because the translated configuration data is committed as a transient change, the candidate and active configurations do not explicitly display the translated data in the Junos OS syntax when you view the configuration by issuing the **show** or **show configuration** command.

Starting in Junos PyEZ Release 2.1.0, Junos PyEZ applications can retrieve configuration data for standard (IETF, OpenConfig) and custom YANG data models in addition to retrieving the native Junos OS configuration by including the appropriate arguments in the `get_config()` RPC.

To retrieve configuration data that is defined by a nonnative YANG data model, execute the `get_config()` RPC with the **model** argument, and include the **namespace** argument when appropriate. The **model** argument takes one of the following values:

- **custom**—Specify that the configuration data be defined in custom YANG data models. You must include the **namespace** argument when retrieving data for custom YANG data models.
- **ietf**—Specify that the configuration data be defined in IETF YANG data models.
- **openconfig**—Specify that the configuration data be defined in the OpenConfig YANG data models.
- **True**—Retrieve all configuration data including the complete Junos OS configuration and data from any YANG data models.

If you specify the **ietf** or **openconfig** value for the **model** argument, Junos PyEZ automatically uses the appropriate namespace. If you retrieve data for a custom YANG data model by using **model='custom'**, you must also include the **namespace** argument with the corresponding namespace.

In the following example, the `get_config()` RPC retrieves the OpenConfig **bgp** configuration hierarchy from the candidate configuration on the device. If you omit the **filter\_xml** argument, the RPC returns the complete Junos OS and OpenConfig candidate configurations.

```
from jnpr.junos import Device
from lxml import etree

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config(filter_xml='bgp', model='openconfig')
    print(etree.tostring(data))
```

The following RPC retrieves the **interfaces** configuration hierarchy from the candidate configuration for an IETF YANG data model:

```
data = dev.rpc.get_config(filter_xml='interfaces', model='ietf')
print(etree.tostring(data))
```

The following RPC retrieves the **l2vpn** configuration hierarchy from the candidate configuration for a custom YANG data model with the given namespace:

```
data = dev.rpc.get_config(filter_xml='l2vpn', model='custom',
namespace="http://yang.juniper.net/customyang/demo/l2vpn")
print(etree.tostring(data))
```

The following RPC retrieves the complete Junos OS candidate configuration as well as the configuration data for other YANG data models that have been added to the device:

```
data = dev.rpc.get_config(model=True)
print (etree.tostring(data))
```

## Handling Namespaces in Configuration Data

The Junos PyEZ `get_config()` RPC, by default, strips out any namespaces in the returned configuration data. Junos PyEZ applications can retain the namespace in the returned configuration data, which enables you to load the data back onto a device, such as when you want to quickly modify the existing configuration.

To retain namespaces in the configuration data, include the `remove_ns=False` argument in the `get_config()` argument list. For example:

```
from jnpr.junos import Device
from lxml import etree

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config(filter_xml='bgp', model='openconfig',
                             remove_ns=False)
    print (etree.tostring(data))
```

In the following truncated output, the `<bgp>` element retains the `xmlns` attribute that defines the namespace:

```
<bgp xmlns="http://openconfig.net/yang/bgp">
  <neighbors>
    <neighbor>
      <neighbor-address>198.51.100.1</neighbor-address>
      <config>
        <peer-group>0C</peer-group>
        <neighbor-address>198.51.100.1</neighbor-address>
        <enabled>true</enabled>
        <peer-as>64496</peer-as>
      </config>
    </neighbor>
  </neighbors>
  ...
```

If the `get_config()` `remove_ns=False` argument is omitted, the namespace is not included in the output.

```
<bgp>
  <neighbors>
    <neighbor>
      <neighbor-address>198.51.100.1</neighbor-address>
      <config>
        <peer-group>0C</peer-group>
        <neighbor-address>198.51.100.1</neighbor-address>
        <enabled>true</enabled>
        <peer-as>64496</peer-as>
      </config>
    </neighbor>
  </neighbors>
  ...
```

- Related Documentation**
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
  - [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)

## CHAPTER 9

# Rolling Back the Configuration Using Junos PyEZ

- [Example: Using Junos PyEZ to Roll Back the Configuration on page 83](#)

### Example: Using Junos PyEZ to Roll Back the Configuration

---

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ package provides simple yet powerful methods to perform certain operational and configuration tasks on devices running Junos OS. This example outlines how to use the Junos PyEZ `jnpr.junos.utils.config.Config` utility to roll back the configuration on a device running Junos OS.

- [Requirements on page 83](#)
- [Overview on page 83](#)
- [Configuration on page 84](#)
- [Executing the Junos PyEZ Code on page 87](#)
- [Verification on page 87](#)

### Requirements

This example uses the following hardware and software components:

- Configuration management server running Python 2.7 or 3.4 and Junos PyEZ Release 2.0 or a later release
- Device running Junos OS with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the server and device running Junos OS

### Overview

This example presents a Python program that uses the Junos PyEZ **Config** utility to roll back the configuration on the specified device. Devices running Junos OS store a copy of the most recently committed configuration and up to 49 previous configurations. You can roll back to any of the stored configurations. This is useful when configuration changes cause undesirable results, and you want to revert back to a known working configuration.

Rolling back the configuration is similar to the process for making configuration changes on the device, but instead of loading configuration data, you perform a rollback, which replaces the entire candidate configuration with a previously committed configuration.

The Python program imports the **Device** class, which handles the connection with the device running Junos OS; the **Config** class, which is used to perform configuration mode commands on the target device; and required exceptions from the `jnpr.junos.exception` module, which contains exceptions encountered when managing devices running Junos OS.

After creating the **Device** instance for the target device, the `open()` method establishes a connection and NETCONF session with the device. The **Config** utility methods then lock, roll back, commit, and unlock the candidate configuration.

The `rollback()` method has a single parameter, `rb_id`, which is the rollback ID specifying the stored configuration to load. Valid values are 0 (zero, for the most recently committed configuration) through one less than the number of stored previous configurations (maximum is 49). If you omit this parameter in the method call, it defaults to 0. This example loads the configuration with rollback ID 1, which is the configuration committed just prior to the active configuration. Note that the `rollback()` method loads the configuration into the candidate configuration, which is then committed by calling the `commit()` method to make it active.

After rolling back and committing the configuration, the NETCONF session and connection are terminated using the `close()` method. The Python program includes code for handling exceptions such as **LockError** for errors that occur when locking the configuration and **CommitError** for errors that occur during the commit operation. The program also includes code to handle any additional exceptions that might occur.

## Configuration

---

### Creating the Junos PyEZ Program

---

#### Step-by-Step Procedure

To create a Python program that uses Junos PyEZ to roll back the configuration on a device running Junos OS:

1. Import any required modules, classes, and objects.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError
from jnpr.junos.exception import LockError
from jnpr.junos.exception import RpcError
from jnpr.junos.exception import CommitError
from jnpr.junos.exception import UnlockError
```

2. Include any required variables, which for this example includes the hostname of the managed device.

```
host = 'dc1a.example.com'
```

3. Create a `main()` function definition and function call, and place the remaining statements within the definition.

```
def main():

    if __name__ == "__main__":
        main()
```

4. Create an instance of the `Device` class, and supply the hostname and any parameters required for that specific connection.

```
dev = Device(host=host)
```

5. Open a connection and establish a NETCONF session with the device.

```
# open a connection with the device and start a NETCONF session
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
    return
```

6. Create an instance of the `Config` utility.

```
# Set up config object
cu = Config(dev)
```

7. Lock the configuration.

```
# Lock the configuration
print ("Locking the configuration")
try:
    cu.lock()
except LockError as err:
    print ("Unable to lock configuration: {0}".format(err))
    dev.close()
    return
```

8. Roll back and commit the configuration, and handle any errors.

```
try:
    print ("Rolling back the configuration")
    cu.rollback(rb_id=1)
    print ("Committing the configuration")
    cu.commit()
except CommitError as err:
    print ("Error: Unable to commit configuration: {0}".format(err))

except RpcError as err:
    print ("Unable to rollback configuration changes: {0}".format(err))
```

9. Unlock the configuration, and then end the NETCONF session and close the connection with the device.

```
finally:
    print ("Unlocking the configuration")
    try:
        cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {0}".format(err))
    dev.close()
    return
```

---

## Results

On the configuration management server, review the completed program. If the program does not display the intended code, repeat the instructions in this example to correct the program.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError
from jnpr.junos.exception import LockError
from jnpr.junos.exception import RpcError
from jnpr.junos.exception import CommitError
from jnpr.junos.exception import UnlockError

host = 'dc1a.example.com'

def main():
    dev = Device(host=host)
    # open a connection with the device and start a NETCONF session
    try:
        dev.open()
    except ConnectError as err:
        print ("Cannot connect to device: {0}".format(err))
        return

    # Set up config object
    cu = Config(dev)

    # Lock the configuration
    print ("Locking the configuration")
    try:
        cu.lock()
    except LockError as err:
        print ("Unable to lock configuration: {0}".format(err))
        dev.close()
        return
    try:
        print ("Rolling back the configuration")
        cu.rollback(rb_id=1)
        print ("Committing the configuration")
        cu.commit()
    except CommitError as err:
        print ("Error: Unable to commit configuration: {0}".format(err))
    except RpcError as err:
        print ("Unable to rollback configuration changes: {0}".format(err))
```

```
finally:
    print ("Unlocking the configuration")
    try:
        cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {}".format(err))
    dev.close()
    return

if __name__ == "__main__":
    main()
```

## Executing the Junos PyEZ Code

**Step-by-Step Procedure** To execute the Junos PyEZ code:

- On the configuration management server, execute the program.

```
[user@server]$ python junos-pyez-config-rollback.py
Locking the configuration
Rolling back the configuration
Committing the configuration
Unlocking the configuration
```

## Verification

### Verifying the Configuration

---

**Purpose** Verify that the configuration was correctly rolled back on the device running Junos OS.

**Action** Log in to the device running Junos OS and view the configuration or configuration differences and the log file. For example:

```
user@dc1a> show configuration | compare rollback 1
[edit system scripts op]
-   file bgp-neighbors.slax;
[edit interfaces]
- ge-1/0/0 {
-     unit 0 {
-       family inet {
-         address 198.51.100.1/26;
-       }
-     }
-   }
+ ge-1/1/0 {
+   unit 0 {
+     family inet {
+       address 198.51.100.65/26;
+     }
+   }
+ }

user@dc1a> show log messages
Sep 19 12:42:06 dc1a sshd[5838]: Accepted publickey for user from 198.51.100.1
port 58663 ssh2: RSA 02:dd:53:3e:f9:97:dd:1f:d9:31:e9:7f:82:06:aa:67
Sep 19 12:42:10 dc1a file[5841]: UI_LOAD_EVENT: User 'user' is performing a
'rollback 1'
Sep 19 12:42:11 dc1a file[5841]: UI_COMMIT: User 'user' requested 'commit'
operation (comment: none)
Sep 19 12:42:26 dc1a file[5841]: UI_COMMIT_COMPLETED: commit complete
```

**Meaning** The configuration differences and the log file contents indicate that the configuration was successfully rolled back and committed on the device.

- Related Documentation**
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
  - [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)
  - [Using Junos PyEZ to Manage the Rescue Configuration on page 109](#)
  - [Example: Using Junos PyEZ to Roll Back the Configuration on page 83](#)
  - [Junos PyEZ Modules Overview on page 19](#)
  - [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)

## CHAPTER 10

# Executing RPCs Using Junos PyEZ

- [Using Junos PyEZ to Execute RPCs on Devices Running Junos OS on page 89](#)
- [Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications on page 93](#)

## Using Junos PyEZ to Execute RPCs on Devices Running Junos OS

---

You can use Junos PyEZ to execute RPCs on demand on devices running Junos OS. After creating an instance of the **Device** class, you can execute RPCs as a property of the **Device** instance. You can perform most of the same operational commands in Junos PyEZ that you can execute in the CLI.

The Junos XML API is an XML representation of Junos OS configuration statements and operational mode commands. It defines an XML equivalent for all statements in the Junos OS configuration hierarchy and many of the commands that you issue in CLI operational mode. Each operational mode command with a Junos XML counterpart maps to a request tag element and, if necessary, a response tag element. Request tags are used in remote procedure calls (RPCs) within NETCONF or Junos XML protocol sessions to request information from a device running Junos OS. The server returns the response using Junos XML elements enclosed within the response tag element.

When you use Junos PyEZ to execute RPCs, you map the request tag name to a method name. The following sections outline how to map CLI commands to Junos PyEZ RPCs, how to execute RPCs using Junos PyEZ, and how to normalize the RPC reply:

- [Mapping Junos OS Commands to Junos PyEZ RPCs on page 89](#)
- [Executing RPCs as a Property of the Device Instance on page 90](#)
- [Normalizing the XML RPC Reply on page 92](#)

## Mapping Junos OS Commands to Junos PyEZ RPCs

All operational commands that have Junos XML counterparts are listed in the *Junos XML API Operational Developer Reference*. You can also display the Junos XML request tag elements for any operational mode command that has a Junos XML counterpart either

on the CLI or using Junos PyEZ. Once you obtain the request tag, you can map it to the Junos PyEZ RPC method name.

To display the Junos XML request tags for a command in the CLI, include the **| display xml rpc** option after the command. The following example displays the request tag for the **show route** command:

```
user@host> show route | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <rpc>
    <get-route-information>
    </get-route-information>
  </rpc>
</rpc-reply>
```

Starting in Junos PyEZ Release 1.2, you can also display the Junos XML request tags for a command using Junos PyEZ. To display the request tags, call the **Device** instance **display\_xml\_rpc()** method, and include the command string and **format='text'** as arguments. For example:

```
from jnpr.junos import Device

with Device(host='dc1a.example.com') as dev:
    print (dev.display_xml_rpc('show route', format='text'))
```

Executing the program returns the request tag for the **show route** command.

```
<get-route-information>
</get-route-information>
```

You can map the request tags for an operational command to a Junos PyEZ RPC method name. To derive the RPC method name, replace any hyphens in the request tag with underscores (\_) and remove the enclosing angle brackets. For example, the **<get-route-information>** request tag maps to the **get\_route\_information()** method name.

## Executing RPCs as a Property of the Device Instance

Each instance of **Device** has an **rpc** property that enables you to execute any RPC available through the Junos XML API. In a Junos PyEZ application, after establishing a connection with the device, you can execute the RPC by appending the **rpc** property and RPC method name to the device instance as shown in the following example:

```
from jnpr.junos import Device
from lxml import etree

with Device(host='dc1a.example.com') as dev:
    #invoke the RPC equivalent to "show version"
    sw = dev.rpc.get_software_information()
    print(etree.tostring(sw, encoding='unicode'))
```

The return value is an XML object starting at the first element under the **<rpc-reply>** tag. In this case, the **get\_software\_information()** RPC returns the **<software-information>** element.

```
<software-information>
<host-name>dc1a</host-name>
```

```
...
</software-information>
```

Starting in Junos PyEZ Release 1.2.3, you can also return the RPC output in JSON format or in text format by including either the `{'format':'json'}` or `{'format':'text'}` dictionary as the first argument in the list. For example:

```
sw = dev.rpc.get_software_information({'format':'json'})
print(sw)
```



**NOTE:** RPC output in text format is enclosed within an `<output>` element. RPC output in JSON format is only supported on devices running Junos OS Release 14.2 and later releases.

Junos OS commands can have fixed-form options that do not have a value. For example, the Junos XML equivalent for the `show interfaces terse` command indicates that `terse` is an empty element.

```
user@router> show interfaces terse | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos">
  <rpc>
    <get-interface-information>
      <terse/>
    </get-interface-information>
  </rpc>
</rpc-reply>
```

To execute an RPC and include a command option that does not take a value, add the option to the RPC method parameter list and set it equal to `True`. The following code executes the Junos PyEZ RPC equivalent of the `show interfaces terse` command:

```
rsp = dev.rpc.get_interface_information(terse=True)
```

Junos OS commands can also have options that require a value. For example, in the following output, the `interface-name` element requires a value, which is the name of the interface for which you want to return information:

```
user@router> show interfaces ge-0/0/0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos">
  <rpc>
    <get-interface-information>
      <interface-name>ge-0/0/0</interface-name>
    </get-interface-information>
  </rpc>
</rpc-reply>
```

To execute an RPC and include a command option that requires a value, add the option to the RPC method parameter list, change any dashes in the option name to underscores, and then set it equal to the appropriate value. The following example executes the Junos PyEZ RPC equivalent of the `show interfaces ge-0/0/0` command:

```
rsp = dev.rpc.get_interface_information(interface_name='ge-0/0/0')
```

RPC execution time can vary considerably depending on the RPC and the device. By default, NETCONF RPCs time out after 30 seconds. Starting in Junos PyEZ Release 1.2, you can extend the timeout value by including the **dev\_timeout=seconds** argument when you execute the RPC to ensure that the RPC does not time out during execution. **dev\_timeout** adjusts the device timeout only for that single RPC operation.

```
dev.rpc.get_route_information(table='inet.0', dev_timeout=55)
```

## Normalizing the XML RPC Reply

When you execute an RPC, the RPC reply can include data that is wrapped in newlines or contains other superfluous whitespace. Unnecessary whitespace can make it difficult to parse the XML and find information using text-based searches. Starting in Junos PyEZ 1.2, you can normalize an RPC reply, which strips out all leading and trailing whitespace and replaces sequences of internal whitespace characters with a single space.

[Table 12 on page 92](#) compares a default RPC reply to the normalized version.

**Table 12: Comparison of a Default and Normalized RPC Reply**

Default RPC Reply	Normalized RPC Reply
<pre>&lt;interface-information style="terse"&gt; &lt;logical-interface&gt; &lt;name&gt;\nge-0/0/0.0\n&lt;/name&gt; &lt;admin-status&gt;\nup\n&lt;/admin-status&gt; &lt;oper-status&gt;\nup\n&lt;/oper-status&gt; &lt;filter-information&gt;\n&lt;/filter-information&gt; &lt;address-family&gt; &lt;address-family-name&gt;\ninet\n&lt;/address-family-name&gt; &lt;interface-address&gt; &lt;ifa-local emit="emit"&gt;\n198.51.100.1/24\n&lt;/ifa-local&gt; &lt;/interface-address&gt; &lt;/address-family&gt; &lt;/logical-interface&gt; &lt;/interface-information&gt;</pre>	<pre>&lt;interface-information style="terse"&gt; &lt;logical-interface&gt; &lt;name&gt;ge-0/0/0.0&lt;/name&gt; &lt;admin-status&gt;up&lt;/admin-status&gt; &lt;oper-status&gt;up&lt;/oper-status&gt; &lt;filter-information/&gt; &lt;address-family&gt; &lt;address-family-name&gt;inet&lt;/address-family-name&gt; &lt;interface-address&gt; &lt;ifa-local emit="emit"&gt;198.51.100.1/24&lt;/ifa-local&gt; &lt;/interface-address&gt; &lt;/address-family&gt; &lt;/logical-interface&gt; &lt;/interface-information&gt;</pre>

You can enable normalization for the duration of a session with a device, or you can normalize an individual RPC reply when you execute the RPC. To enable normalization for the entire device session, include **normalize=True** in the parameter list either when you create the device instance or when you connect to the device using the **open()** method.

```
dev = Device(host='router1.example.com', user='root', normalize=True)
```

*# or*

```
dev.open(normalize=True)
```

To normalize an individual RPC reply, include **normalize=True** in the argument list for that RPC method.

```
dev.rpc.rpc_method(normalize=True)
```

For example:

```
rsp = dev.rpc.get_interface_information(interface_name='ge-0/0/0.0', terse=True,
normalize=True)
```

If you do not normalize the RPC reply, you must account for any whitespace when using XPath expressions that reference a specific node or value. The following example selects the IPv4 address for a logical interface. In the XPath expression, the predicate specifying the `inet` family must account for the additional whitespace in order for the search to succeed. The resulting value includes leading and trailing newlines.

```
rsp = dev.rpc.get_interface_information(interface_name='ge-0/0/0.0', terse=True)
print (rsp.xpath("//*[ \
    address-family[normalize-space(address-family-name)='inet']/ \
    interface-address/ifa-local")[0].text)
'\n198.51.100.1/24\n'
```

When you normalize the RPC reply, any leading and trailing whitespace is removed, which makes text-based searches much more straightforward.

```
rsp = dev.rpc.get_interface_information(interface_name='ge-0/0/0.0', terse=True,
    normalize=True)
print (rsp.xpath("//*[address-family[address-family-name='inet']/ \
    interface-address/ifa-local")[0].text)
'198.51.100.1/24'
```

#### Related Documentation

- [Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications on page 93](#)
- [Accessing the Shell on Devices Running Junos OS Using Junos PyEZ on page 46](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications

Junos PyEZ enables you to perform operational and configuration tasks on devices running Junos OS. In a Junos PyEZ application, when you call specific methods or execute on-demand RPCs, Junos PyEZ sends the appropriate RPCs to the device to perform the operation or retrieve the requested information. If the RPC reply contains `<rpc-error>` elements with a severity of warning or higher, the Junos PyEZ application raises an **RpcError** exception.

In certain cases, it might be necessary or desirable to suppress the **RpcError** exceptions that are raised in response to warnings. You can instruct a Junos PyEZ application to suppress **RpcError** exceptions that are raised for warnings by including the `ignore_warning` argument in the method call or RPC invocation. The `ignore_warning` argument takes a Boolean, a string, or a list of strings.

To instruct the application to ignore all warnings for an operation or RPC, include the `ignore_warning=True` argument in the method call or RPC invocation. The following example ignores all warnings for the `load()` method and the `get_configuration()` RPC:

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

dev = Device(host="host")
dev.open()
with Config(dev) as cu:
    cu.load(path="mx-config.conf", ignore_warning=True)
```

```
cu.commit()

data = dev.rpc.get_configuration(ignore_warning=True)
print(etree.tostring(data, encoding='unicode'))

dev.close()
```

If you include `ignore_warning=True` and all of the `<rpc-error>` elements have a severity of warning, the application ignores all warnings and does not raise an `RpcError` exception. However, any `<rpc-error>` elements with higher severity levels will still raise exceptions.

To instruct the application to ignore specific warnings, set the `ignore_warning` argument to a string or a list of strings containing the warnings to ignore. For example, the following Junos PyEZ application ignores two specific warnings during the commit operation:

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

commit_warnings = ['Advertisement-interval is less than four times',
                   'Chassis configuration for network services has been changed.']

dev = Device(host="host").open()
with Config(dev) as cu:
    cu.load(path="mx-config.conf")
    cu.commit(ignore_warning=commit_warnings)
```

The Junos PyEZ application suppresses `RpcError` exceptions if all of the `<rpc-error>` elements have a severity of warning and each warning in the response matches one or more of the specified strings. When `ignore_warning` is set to a string or list of strings, the string is used as a case-insensitive regular expression. If a string contains only alphanumeric characters, it results in a case-insensitive substring match. However, you can include any regular expression pattern supported by the `re` library to match warnings.

#### Related Documentation

- [Using Junos PyEZ to Execute RPCs on Devices Running Junos OS on page 89](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)

# Installing and Rebooting Software Using Junos PyEZ

- [Rebooting a Device Running Junos OS Using Junos PyEZ on page 95](#)
- [Using Junos PyEZ to Install Software on Devices Running Junos OS on page 97](#)
- [Example: Using Junos PyEZ to Install Software on Devices Running Junos OS on page 101](#)

## Rebooting a Device Running Junos OS Using Junos PyEZ

---

The Junos PyEZ `jnpr.junos.utils.sw.SW` utility enables you to reboot a device running Junos OS by executing the `reboot()` method. You can perform an immediate system reboot, request a reboot with an optional delay, or you can schedule a reboot at a specified date and time. The `reboot()` method, by default, executes the reboot immediately and reboots all Routing Engines, if in a dual Routing Engine or Virtual Chassis setup.



**NOTE:** Starting in Junos PyEZ 2.1.0, the `reboot()` method reboots all Routing Engines in a dual Routing Engine or Virtual Chassis setup. In earlier releases, the `reboot()` method reboots only that Routing Engine to which the application is connected.



**NOTE:** If a Junos PyEZ application reboots a device from a NETCONF-over-SSH session or from a Telnet session through the management interface, the application loses connectivity to the device when it reboots. If the application requires access to the device after the reboot, it must issue the Junos PyEZ `open()` method to restore connectivity.

The following Junos PyEZ example, which uses Python 3, establishes a NETCONF session over SSH with a device running Junos OS and reboots all Routing Engines, effective immediately.

```
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW
from jnpr.junos.exception import ConnectError
from getpass import getpass
import sys
```

```
hostname = input("Device hostname: ")
username = input("Device username: ")
password = getpass("Device password: ")

dev = Device(host=hostname, user=username, passwd=password)

try:
    dev.open()
except ConnectError as err:
    print (err)
    sys.exit(1)

sw = SW(dev)
print(sw.reboot())

dev.close()
```

The application prompts for the device hostname and user credentials. After requesting the system reboot, the application displays the reboot message and the process ID for the process on the connected Routing Engine.

```
user1@host:~$ python3 junos-pyez-reboot.py
Device hostname: dc1a-console.example.com
Device username: user1
Device password:
Shutdown NOW!
[pid 2358]
```

To delay the reboot by a specified number of minutes, set the optional **in\_min** parameter to the amount of time in minutes that the system should wait before rebooting. The following example requests a reboot of all Routing Engines in 2 minutes:

```
sw.reboot(in_min=2)
```

To schedule the reboot at a specific time, include the **at** parameter, which takes a string that can be specified in one of the following ways:

- **now**—Stop or reboot the software immediately.
- **+minutes**—Number of minutes from now to reboot the software.
- **yymmddhhmm**—Absolute time at which to reboot the software, specified as year, month, day, hour, and minute.
- **hh:mm**—Absolute time on the current day at which to reboot the software, specified in 24-hour time.

The following example schedules a system reboot of all Routing Engines at 22:30 on the current day:

```
sw.reboot(at="22:30")
```

Starting in Junos PyEZ 2.1.0, the default for the **reboot()** method is to reboot all Routing Engines in a dual Routing Engine or Virtual Chassis setup. To reboot only the Routing Engine to which the application is connected, include the **all\_re=False** argument in the method.

```
sw.reboot(all_re=False)
```

For information about the **reboot()** method, see

<http://junos-pyez.readthedocs.io/en/latest/jnpr.junos.utils.html#jnpr.junos.utils.sw.SW.reboot>.

#### Release History Table

Release	Description
2.1.0	Starting in Junos PyEZ 2.1.0, the <b>reboot()</b> method reboots all Routing Engines in a dual Routing Engine or Virtual Chassis setup.

#### Related Documentation

- [Example: Using Junos PyEZ to Install Software on Devices Running Junos OS on page 101](#)

## Using Junos PyEZ to Install Software on Devices Running Junos OS

The Junos PyEZ [jnpr.junos.utils.sw.SW](#) utility enables you to install or upgrade the software image on devices running Junos OS. The **install()** method installs the specified software package.

The following sections discuss the supported deployment scenarios, how to specify the software image location, the general installation process, and support for unified in-service software upgrades (unified ISSU) and nonstop software upgrades (NSSU):

- [Supported Deployment Scenarios on page 97](#)
- [Specifying the Software Image Location on page 98](#)
- [Installation Process Overview on page 99](#)
- [ISSU and NSSU Support on page 100](#)

### Supported Deployment Scenarios

The Junos PyEZ [jnpr.junos.utils.sw.SW](#) utility enables you to install or upgrade the software image on an individual device running Junos OS or on the members in a mixed or non-mixed Virtual Chassis. The following scenarios are supported:

- Standalone devices with a single Routing Engine
- Standalone devices equipped with dual Routing Engines
- EX Series Virtual Chassis in mixed and non-mixed-mode configurations
- QFX Series Virtual Chassis in mixed and non-mixed-mode configurations
- Mixed EX Series and QFX Series Virtual Chassis
- Deployment configurations that have some form of *in-service* features enabled, such as unified ISSU or NSSU



**NOTE:** The [jnpr.junos.utils.sw.SW](#) utility does not support upgrading devices in an MX Series Virtual Chassis, SRX Series chassis cluster, or in a Virtual Chassis Fabric (VCF).

## Specifying the Software Image Location

When you use Junos PyEZ to install software on devices running Junos OS, you can download the software package to the configuration management server, and the `install()` method, by default, copies it to the target device before performing the installation. You can also download the software package directly to the target device and instruct the `install()` method to install it from there.

Table 13 on page 98 outlines the `install()` method parameters that you must set depending on the software package location. You must always include either the **package** or **pkg\_set** parameter in the `install()` method invocation.

**Table 13: `install()` Method Parameter Settings for Software Package Location**

Software Package Location	no_copy Parameter	package or pkg_set Parameter	remote_path Parameter
Configuration management server	Omitted or set to <b>False</b>	File path including the filename of the software package or packages on the server.	(Optional) Path to the directory on the target device to which the package or packages will be copied. Default is <b>/var/tmp</b> .
Remote device	Set to <b>True</b>	Filename of the software package or packages.	(Optional) Path to the directory on the target device where the package or packages already reside. Default is <b>/var/tmp</b> .

The **package** argument is used to install software on a single device running Junos OS or on members in a non-mixed Virtual Chassis. The **package** argument is a string that specifies a single software image. For example:

```
package = '/tmp/jinstall-13.3R1.8-domestic-signed.tgz'
```

The **pkg\_set** argument is used to install software on the members in a mixed Virtual Chassis. It contains a list or tuple of strings that specify the necessary software images, in no particular order, for the various Virtual Chassis members. For example:

```
pkg_set=['jinstall-qfx-5-13.2X51-D35.3-domestic-signed.tgz',
        'jinstall-ex-4300-13.2X51-D35.3-domestic-signed.tgz']
```

When you omit the **no\_copy** argument or set it to **False**, the server copies the specified software package to the device. Including the **package** argument causes the server to copy the package to the target device (individual device or master router or switch in a non-mixed Virtual Chassis), and including the **pkg\_set** argument causes the server to copy all packages in the list to the master router or switch in a mixed Virtual Chassis. By default, software images are placed in the **/var/tmp** directory unless the **remote\_path** argument specifies a different directory.

If you set the **no\_copy** argument to **True**, the necessary software packages must already exist on the target device or Virtual Chassis master device before the installation begins.

The packages must reside either in the directory specified by the `remote_path` argument, or if `remote_path` is omitted, in the default `/var/tmp` directory.

## Installation Process Overview

To install a software package on a device running Junos OS, connect to the individual device or to the master device in a Virtual Chassis, create an instance of the **SW** utility, and call the `install()` method with any required or optional arguments. For example:

```
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW

pkg = 'junos-install-mx-x86-64-17.2R1.13.tgz'

with Device(host='router1.example.net') as dev:
    sw = SW(dev)
    ok = sw.install(package=pkg, validate=True)
    if ok:
        sw.reboot()
```

For the current list of `install()` method parameters, see

<http://junos-pyez.readthedocs.io/en/latest/jnpr.junos.utils.html#jnpr.junos.utils.sw.SW.install>.

If the software package is located on the configuration management server, and the `no_copy` parameter is omitted or set to `False`, the `install()` method performs the following operations before installing the software:

- Computes the local MD5 checksum on the software package if it is not provided in the `checksum` argument
- Performs a storage cleanup on the device to create space for the software package, unless `cleanfs` is set to `False`
- SCP copies the package to the remote directory specified in the `remote_path` argument or to the `/var/tmp` directory, if `remote_path` is not specified
- Computes the remote MD5 checksum and matches it to the local value

Once the software package is on the target device, whether downloaded there initially or copied over by the `install()` method, the `install()` method then performs the following operations:

- Validates the configuration against the new package if the `validate` parameter is set to `True`
- Installs the package



**NOTE:** By default, Junos PyEZ upgrades only the master Routing Engine on each device running Junos OS.

The `install()` method returns `True` if the installation is successful. The `install()` method does not reboot the device. To reboot or shut down the device after the installation is complete, call the `reboot()` or `shutdown()` method, respectively.

Junos PyEZ performs operations over a NETCONF session. The default time for a NETCONF RPC to time out is 30 seconds. During the installation process, Junos PyEZ increases the RPC timeout interval to 1800 seconds (30 minutes) when copying and installing the package on the device and to 300 seconds (5 minutes) when computing the checksum. If necessary, you can increase the timeout value for the installation by including the **timeout** parameter in the call to the **install()** method.

```
ok = sw.install(package=pkg, validate=True, timeout=2400)
```

The Junos PyEZ install process enables you to log the progress of the installation by including the **progress** argument in the method call. The argument is set to a callback function, which must have a function prototype defined that includes the **Device** instance and report string. Starting in Junos PyEZ 1.2.3, you can also set **progress=True** to use **sw.progress()** for basic reporting.

```
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW

def update_progress(dev, report):
    print (dev.hostname, '> ', report)

pkg = 'junos-install-mx-x86-64-17.2R1.13.tgz'

with Device(host='router1.example.net') as dev:
    sw = SW(dev)
    ok = sw.install(package=pkg, validate=True, progress=update_progress)
    if ok:
        sw.reboot()
```

The following video presents a short Python session that demonstrates how to use Junos PyEZ to install Junos OS.



[Video: Junos PyEZ - Software Upgrading Device](#)

---

## ISSU and NSSU Support

Starting in Junos PyEZ Release 2.1.0, Junos PyEZ provides support for performing a unified in-service software upgrade (unified ISSU) or a nonstop software upgrade (NSSU) on devices that support the feature and meet the necessary requirements. For more information about the unified ISSU and NSSU features, see the software documentation for your product.

The unified ISSU feature enables you to upgrade between two different Junos OS releases with no disruption on the control plane and with minimal disruption of traffic. To perform a unified in-service software upgrade, include the **issu=True** argument in the **install()** method. For example:

```
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW

pkg = 'junos-install-mx-x86-64-17.2R1.13.tgz'
with Device(host='router1.example.net') as dev:
    sw = SW(dev)
    ok = sw.install(package=pkg, issu=True, progress=True)
    if ok:
        sw.reboot()
```

The NSSU feature enables you to upgrade the Junos OS software running on a switch or Virtual Chassis with redundant Routing Engines with minimal disruption to network traffic. To perform a nonstop software upgrade, include the `nssu=True` argument in the `install()` method. For example:

```
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW

pkg = 'jinstall-ex-4300-14.1X53-D44.3-domestic-signed.tgz'
with Device(host='switch1.example.net') as dev:
    sw = SW(dev)
    ok = sw.install(package=pkg, nssu=True, progress=True)
    if ok:
        sw.reboot()
```

#### Related Documentation

- [Rebooting a Device Running Junos OS Using Junos PyEZ on page 95](#)
- [Example: Using Junos PyEZ to Install Software on Devices Running Junos OS on page 101](#)
- [Junos PyEZ Modules Overview on page 19](#)

---

## Example: Using Junos PyEZ to Install Software on Devices Running Junos OS

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ package provides simple yet powerful methods to perform certain operational and configuration tasks on devices running Junos OS. This example outlines how to use the Junos PyEZ `jnpr.junos.utils.sw.SW` utility to install or upgrade the software image on a device running Junos OS.

- [Requirements on page 101](#)
- [Overview on page 102](#)
- [Configuration on page 103](#)
- [Executing the Junos PyEZ Code on page 106](#)
- [Verification on page 106](#)
- [Troubleshooting on page 107](#)

### Requirements

This example uses the following hardware and software components:

- Configuration management server running Python 2.7 or 3.4 and Junos PyEZ Release 2.0 or later release
- Device running Junos OS with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the server and device running Junos OS

## Overview

This example presents a Python program that uses the Junos PyEZ **SW** utility to upgrade Junos OS on the specified device. Before using Junos PyEZ to install the software, you must download the software image either to the configuration management server or to the managed device. This example assumes that the image has been downloaded to the server.

The program imports the Junos PyEZ **Device** class, which handles the connection with the device running Junos OS; the **SW** class, which is used to perform the software installation operations on the target device; and required exceptions from the **junos.junos.exception** module, which contains exceptions encountered when managing devices running Junos OS. The program also imports the **os**, **sys**, and **logging** Python modules for verifying the existence of the software package and performing basic logging functions.

The program defines the **update\_progress()** method, which is used by the **install()** method to report on the progress of the installation. By logging the installation process, you can more readily identify the point where any failures occur. In this example, progress messages are sent to standard output and also logged in a separate file.

Before connecting to the device and proceeding with the installation, the program first verifies that the software package exists. If the file cannot be found, the program exits with an error message. If the file exists, the program creates the **Device** instance for the target device and calls the **open()** method to establish a connection and NETCONF session with the device.

The program creates an instance of the **SW** utility and uses the **install()** method to install the Junos OS software image on the target device. The **package** variable defines the path on the local server to the new Junos OS image. Because the **no\_copy** parameter defaults to False, the installation process copies the software image from the local server to the target device. The **remote\_path** variable defines the path on the target device to which the software package is copied. The default is **/var/tmp**. Although not required, this example explicitly configures the parameter for clarity.

When the **install()** method is called, the program calculates the local MD5 checksum, performs a storage cleanup and copies the software image to the target device, computes the remote MD5 checksum and compares it to the local value, validates the configuration against the new image, and then installs the package. If the installation is successful, the program then calls the **reboot()** method to request that the device reboot.

After performing the installation, the NETCONF session and connection are terminated using the **close()** method. The program includes code for handling any exceptions that might occur when connecting to the device or performing the installation.

## Configuration

### Creating the Junos PyEZ Program

**Step-by-Step Procedure** To create a Python program that uses Junos PyEZ to install a software image on a device running Junos OS:

1. Import any required modules, classes, and objects.

```
import os, sys, logging
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW
from jnpr.junos.exception import ConnectError
```

2. Include any required variables, which for this example includes the hostname of the managed device, the software package path, and the log file.

```
host = 'dc1a.example.com'
package = '/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz'
remote_path = '/var/tmp'
validate = True
logfile = '/var/log/junos-pyez/install.log'
```

3. Define the logging method used within the program and by the `install()` method.

```
def update_progress(dev, report):
    # log the progress of the installing process
    logging.info(report)
```



**NOTE:** Starting in Junos PyEZ 1.2.3, you can set `progress=True` to use `sw.progress()` for basic reporting.

4. Create a `main()` function definition and function call, and place the remaining statements within the definition.

```
def main():
    if __name__ == "__main__":
        main()
```

5. Initialize the logger instance.

```
# initialize logging
logging.basicConfig(filename=logfile, level=logging.INFO,
                    format='%(asctime)s: %(name)s: %(message)s')
logging.getLogger().name = host
logging.getLogger().addHandler(logging.StreamHandler())
logging.info('Information logged in {0}'.format(logfile))
```

6. (Optional) Add code that verifies the existence of the software package.

```
# verify package exists
if not (os.path.isfile(package)):
    msg = 'Software package does not exist: {0}'.format(package)
    logging.error(msg)
    sys.exit()
```

7. Create an instance of the **Device** class, and supply the hostname and any parameters required for that specific connection.

Then open a connection and establish a NETCONF session with the device.

```
# open a connection with the device and start a NETCONF session
dev = Device(host=host)
try:
    dev.open()
except ConnectError as err:
    logging.error('Cannot connect to device: {0}\n'.format(err))
    return
```

8. Create an instance of the **SW** utility.

```
# Create an instance of SW
sw = SW(dev)
```

9. Include code to install the software package and to reboot the device if the installation succeeds.

```
try:
    logging.info('Starting the software upgrade process: {0}' \
        .format(package))
    ok = sw.install(package=package, remote_path=remote_path,
        progress=update_progress, validate=validate)
except Exception as err:
    msg = 'Unable to install software, {0}'.format(err)
    logging.error(msg)
    ok = False

if ok is True:
    logging.info('Software installation complete. Rebooting')
    rsp = sw.reboot()
    logging.info('Upgrade pending reboot cycle, please be patient.')

    logging.info(rsp)
else:
    msg = 'Unable to install software, {0}'.format(ok)
    logging.error(msg)
```

10. End the NETCONF session and close the connection with the device.

```
# End the NETCONF session and close the connection
dev.close()
```

## Results

On the configuration management server, review the completed program. If the program does not display the intended code, repeat the instructions in this example to correct the program.

```
import os, sys, logging
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW
from jnpr.junos.exception import ConnectError

host = 'dc1a.example.com'
package = '/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz'
remote_path = '/var/tmp'
validate = True
logfile = '/var/log/junos-pyez/install.log'

def update_progress(dev, report):
    # log the progress of the installing process
    logging.info(report)

def main():
    # initialize logging
    logging.basicConfig(filename=logfile, level=logging.INFO,
                        format='%(asctime)s: %(name)s: %(message)s')
    logging.getLogger().name = host
    logging.getLogger().addHandler(logging.StreamHandler())
    logging.info('Information logged in {0}'.format(logfile))

    # verify package exists
    if not (os.path.isfile(package)):
        msg = 'Software package does not exist: {0}'.format(package)
        logging.error(msg)
        sys.exit()

    dev = Device(host=host)
    try:
        dev.open()
    except ConnectError as err:
        logging.error('Cannot connect to device: {0}\n'.format(err))
        return

    # Create an instance of SW
    sw = SW(dev)

    try:
        logging.info('Starting the software upgrade process: {0} \
                    .format(package))
        ok = sw.install(package=package, remote_path=remote_path,
                        progress=update_progress, validate=validate)
    except Exception as err:
        msg = 'Unable to install software, {0}'.format(err)
        logging.error(msg)
        ok = False

    if ok is True:
        logging.info('Software installation complete. Rebooting')
```

```
        rsp = sw.reboot()
        logging.info('Upgrade pending reboot cycle, please be patient.')
        logging.info(rsp)
    else:
        msg = 'Unable to install software, {0}'.format(ok)
        logging.error(msg)

    # End the NETCONF session and close the connection
    dev.close()

if __name__ == "__main__":
    main()
```

## Executing the Junos PyEZ Code

**Step-by-Step Procedure** To execute the Junos PyEZ code:

- On the configuration management server, execute the program.

```
[user@server]$ python junos-pyez-install.py
Information logged in /var/log/junos-pyez/install.log
Starting the software upgrade process:
/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz
computing local checksum on:
/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz
cleaning filesystem ...
starting thread (client mode): 0x282d4110L
Connected (version 2.0, client OpenSSH_6.7)
...
```

## Verification

### Verifying the Installation

---

**Purpose** Verify that the software installation was successful.

**Action** Review the progress messages, which are sent to both standard output and the log file that is defined in the program, for details about the installation. Sample log file output is shown here. Some output has been omitted for brevity.

```
[user@server]$ cat /var/log/junos-pyez/install.log
2015-09-03 21:29:20,795:dc1a.example.com: Information logged in
/var/log/junos-pyez/install.log
2015-09-03 21:29:35,257:dc1a.example.com: Starting the software upgrade process:
/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz
2015-09-03 21:29:35,257:dc1a.example.com: computing local checksum on:
/var/tmp/junos-install/jinstall-13.3R1.8-domestic-signed.tgz
2015-09-03 21:29:47,025:dc1a.example.com: cleaning filesystem ...
2015-09-03 21:30:00,870:paramiko.transport: starting thread (client mode):
0x282d4110L
2015-09-03 21:30:01,006:paramiko.transport: Connected (version 2.0, client
OpenSSH_6.7)
...
2015-09-03 21:30:01,533:paramiko.transport: userauth is OK
2015-09-03 21:30:04,002:paramiko.transport: Authentication (public key) successful!
2015-09-03 21:30:04,003:paramiko.transport: [chan 0] Max packet in: 32768 bytes
```

```

2015-09-03 21:30:04,029:paramiko.transport: [chan 0] Max packet out: 32768 bytes
2015-09-03 21:30:04,029:paramiko.transport: Secsh channel 0 opened.
2015-09-03 21:30:04,076:paramiko.transport: [chan 0] Sesch channel 0 request ok
2015-09-03 21:32:23,684:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
94437376 / 944211851 (10%)
2015-09-03 21:34:43,828:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
188858368 / 944211851 (20%)
2015-09-03 21:37:04,180:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
283279360 / 944211851 (30%)
2015-09-03 21:39:24,020:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
377700352 / 944211851 (40%)
2015-09-03 21:41:43,906:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
472121344 / 944211851 (50%)
2015-09-03 21:44:04,079:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
566542336 / 944211851 (60%)
2015-09-03 21:46:23,968:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
660963328 / 944211851 (70%)
2015-09-03 21:48:44,045:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
755384320 / 944211851 (80%)
2015-09-03 21:51:04,016:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
849805312 / 944211851 (90%)
2015-09-03 21:53:24,058:dc1a.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
944211851 / 944211851 (100%)
2015-09-03 21:53:24,389:paramiko.transport: [chan 0] EOF sent (0)
2015-09-03 21:53:24,466:paramiko.transport: EOF in transport thread
2015-09-03 21:53:24,514:dc1a.example.com: computing remote checksum on:
/var/tmp/jinstall-13.3R1.8-domestic-signed.tgz
2015-09-03 21:56:01,692:dc1a.example.com: checksum check passed.
2015-09-03 21:56:01,692:dc1a.example.com: validating software against current
config, please be patient ...
2015-09-03 22:47:57,205:dc1a.example.com: installing software ... please be patient
...
2015-09-03 23:28:10,415:dc1a.example.com: Software installation complete. Rebooting
2015-09-03 23:28:11,525:dc1a.example.com: Upgrade pending reboot cycle, please
be patient.
2015-09-03 23:28:11,525:dc1a.example.com: Shutdown NOW!
[pid 55494]

```

**Meaning** The log file contents indicate that the image was successfully copied to and installed on the target device.

## Troubleshooting

- [Troubleshooting Timeout Errors on page 108](#)

## Troubleshooting Timeout Errors

---

**Problem** The program generates an `RpcTimeoutError` message or a `TimeoutExpiredError` message and the installation fails.

```
RpcTimeoutError(host: dc1a.example.com, cmd: request-package-validate, timeout: 1800)
```

Long operations might exceed the RPC timeout interval, particularly on slower devices, causing the RPC to time out before the operation can be completed. The default time for a NETCONF RPC to time out is 30 seconds. During the installation process, Junos PyEZ increases the RPC timeout interval to 300 seconds (5 minutes) when computing the checksum and to 1800 seconds (30 minutes) when copying and installing the package on the device.

**Solution** To accommodate install operations that might require a longer time than the default installation timeout interval of 1800 seconds, set the `install` method `timeout` parameter to an appropriate value and re-run the program. For example:

```
sw.install(package=package, remote_path=remote_path,
           progress=update_progress, validate=True, timeout=2400)
```

To accommodate checksum operations that might require a longer time than the default timeout interval of 300 seconds, set the `Device` method `timeout` variable to an appropriate value and re-run the program. For example:

```
dev = Device(host=host)
dev.open()
dev.timeout = 600
```

**Related Documentation**

- [Using Junos PyEZ to Install Software on Devices Running Junos OS on page 97](#)
- [Rebooting a Device Running Junos OS Using Junos PyEZ on page 95](#)
- [Junos PyEZ Modules Overview on page 19](#)
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)

## CHAPTER 12

# Managing the Rescue Configuration Using Junos PyEZ

- [Using Junos PyEZ to Manage the Rescue Configuration on page 109](#)
- [Example: Using Junos PyEZ to Save a Rescue Configuration on page 110](#)

### Using Junos PyEZ to Manage the Rescue Configuration

---

A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore at any time. You use the rescue configuration when you need to revert to a known configuration or as a last resort if your router or switch configuration and the backup configuration files become damaged beyond repair. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration.

The Junos PyEZ `jnpr.junos.utils.config.Config` utility enables you to save, retrieve, load, and delete a rescue configuration on devices running Junos OS. After creating an instance of the `Config` class, you can use the `rescue()` method to manage the rescue configuration. You specify the action to perform on the rescue configuration by setting the `rescue()` method `action` parameter.

The following examples illustrate the method call for each `rescue()` method action. `cu` is an instance of the `Config` utility, which operates on the target device running Junos OS.

- To save the active configuration as a rescue configuration, specify `action="save"`. This overwrites any existing rescue configuration.

```
cu.rescue(action="save")
```

- To retrieve an existing rescue configuration, specify `action="get"`, and optionally define the format as either "text" or "xml".

```
rescue = cu.rescue(action="get", format="xml")
```

- To load an existing rescue configuration into the candidate configuration, specify `action="reload"`. Note that you must commit the candidate configuration in order to make it the active configuration on the device.

```
cu.rescue(action="reload")
cu.commit()
```

- To delete an existing rescue configuration, specify **action="delete"**.

```
cu.rescue(action="delete")
```

#### Related Documentation

- [Example: Using Junos PyEZ to Save a Rescue Configuration on page 110](#)
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)
- [Example: Using Junos PyEZ to Roll Back the Configuration on page 83](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)
- [Junos PyEZ Modules Overview on page 19](#)

---

## Example: Using Junos PyEZ to Save a Rescue Configuration

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ package provides simple yet powerful methods to perform certain operational and configuration tasks on devices running Junos OS. This example outlines how to use the Junos PyEZ `jnpr.junos.utils.config.Config` utility to save a rescue configuration on a device running Junos OS.

- [Requirements on page 110](#)
- [Overview on page 110](#)
- [Configuration on page 111](#)
- [Executing the Junos PyEZ Code on page 113](#)
- [Verification on page 113](#)
- [Troubleshooting on page 114](#)

### Requirements

This example uses the following hardware and software components:

- Configuration management server running Python 2.7 or 3.4 and Junos PyEZ Release 2.0 or later release
- Device running Junos OS with NETCONF enabled and a user account configured with appropriate permissions
- SSH public/private key pair configured for the appropriate user on the server and device running Junos OS

### Overview

This example presents a Python program that uses the Junos PyEZ **Config** utility to save a rescue configuration on the specified device. A rescue configuration allows you to define a known working configuration or a configuration with a known state that you can restore

at any time. When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration.

The Python program imports the **Device** class, which handles the connection with the device running Junos OS; the **Config** class, which is used to perform the rescue configuration operations on the target device; and required exceptions from the **jnpr.junos.exception** module, which contains exceptions encountered when managing devices running Junos OS. After creating the **Device** instance for the target device, the **open()** method establishes a connection and NETCONF session with the device.

The program first determines if there is an existing rescue configuration on the target device. If a rescue configuration exists, it is printed to standard output. If there is no existing rescue configuration, the program requests that the device create one. The **rescue()** method **action** parameter is set to "get" to retrieve the existing rescue configuration and to "save" to create a rescue configuration if one does not exist.

After performing the rescue configuration operations, the NETCONF session and connection are terminated using the **close()** method. The Python program includes code for handling exceptions such as **ConnectError** for errors that occur when connecting to the device. The program also includes code to handle any additional exceptions that might occur.

## Configuration

### Creating the Junos PyEZ Program

#### Step-by-Step Procedure

To create a Python program that uses Junos PyEZ to save a rescue configuration if one does not already exist on the device running Junos OS:

1. Import any required modules, classes, and objects.

```
from junpr.junos import Device
from junpr.junos.utils.config import Config
from junpr.junos.exception import ConnectError
```

2. Include any required variables, which for this example includes the hostname of the managed device.

```
host = 'dcla.example.com'
```

3. Create a **main()** function definition and function call, and place the remaining statements within the definition.

```
def main():
    if __name__ == "__main__":
        main()
```

4. Create an instance of the **Device** class, and supply the hostname and any parameters required for that specific connection.

```
dev = Device(host=host)
```

5. Open a connection and establish a NETCONF session with the device.

```
# open a connection with the device and start a NETCONF session
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {}".format(err))
    return
```

6. Create an instance of the **Config** utility.

```
# Create an instance of Config
cu = Config(dev)
```

7. Print the existing rescue configuration or save one if none exists.

```
# Print existing rescue configuration or save one if none exists
try:
    rescue = cu.rescue(action="get", format="text")
    if rescue is None:
        print ("No existing rescue configuration.")
        print ("Saving rescue configuration.")
        cu.rescue(action="save")
    else:
        print ("Rescue configuration found:")
        print (rescue)
except Exception as err:
    print (err)
```

8. End the NETCONF session and close the connection with the device.

```
# End the NETCONF session and close the connection
dev.close()
```

---

## Results

On the configuration management server, review the completed program. If the program does not display the intended code, repeat the instructions in this example to correct the program.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError

host = 'dc1a.example.com'

def main():

    dev = Device(host=host)

    # open a connection with the device and start a NETCONF session
    try:
        dev.open()
    except ConnectError as err:
        print ("Cannot connect to device: {}".format(err))
        return
```

```
# Create an instance of Config
cu = Config(dev)

# Print existing rescue configuration or save one if none exists
try:
    rescue = cu.rescue(action="get", format="text")
    if rescue is None:
        print ("No existing rescue configuration.")
        print ("Saving rescue configuration.")
        cu.rescue(action="save")
    else:
        print ("Rescue configuration found:")
        print (rescue)
except Exception as err:
    print (err)

# End the NETCONF session and close the connection
dev.close()

if __name__ == "__main__":
    main()
```

## Executing the Junos PyEZ Code

### Step-by-Step Procedure

To execute the Junos PyEZ code:

- On the configuration management server, execute the program.

```
[user@server]$ python junos-pyez-config-rescue-create.py
No existing rescue configuration.
Saving rescue configuration.
```

In this example, the target device does not have an existing rescue configuration, so the device saves one. If you execute the program a second time, it outputs the rescue configuration that was saved during the initial execution.

## Verification

### Verifying the Configuration

---

**Purpose** Verify that the rescue configuration exists on the device running Junos OS.

**Action** Log in to the device running Junos OS and view the rescue configuration. For example:

```
user@dc1a> show system configuration rescue
## Last changed: 2014-07-31 17:59:04 PDT
version 13.3R1.8;
groups {
  re0 {
    system {
      host-name dc1a;
    }
  }
}
...
[output truncated]
```

## Troubleshooting

- [Troubleshooting Configuration Change Errors on page 114](#)

---

### Troubleshooting Configuration Change Errors

**Problem** The Junos PyEZ code generates an error message indicating an unsupported action.

unsupported action:

This error message is generated when the **rescue()** method **action** argument contains an invalid value.

**Solution** Set the **rescue()** method **action** argument to a valid action, which includes "save", "get", "reload", and "delete".

#### Related Documentation

- [Using Junos PyEZ to Manage the Rescue Configuration on page 109](#)
- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration on page 73](#)
- [Example: Using Junos PyEZ to Roll Back the Configuration on page 83](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)
- [Junos PyEZ Modules Overview on page 19](#)

# Transferring Files Using Junos PyEZ

- Transferring Files Using Junos PyEZ on page 115

## Transferring Files Using Junos PyEZ

---

Junos PyEZ provides utilities that enable you to perform file management tasks on devices running Junos OS. The Junos PyEZ `jnpr.junos.utils.scp.SCP` class can be used to secure copy (SCP) files between the local host and a device running Junos OS. Instances of the `SCP` class can be used as context managers. For example:

```
with SCP(dev) as scp:
    scp.put("local-file", remote_path="path")
    scp.get("remote-file", local_path="path")
```

Starting in Junos PyEZ Release 1.2.3, `SCP` introduces new ways to enable you to track the progress of transfers using the `progress` parameter. By default, `SCP` does not print progress messages. Set `progress=True` to print default progress messages at transfer completion intervals of 10 percent or greater.

```
with SCP(dev, progress=True) as scp:
```

Alternatively, you can define a custom function to print progress messages, and then set the `progress` parameter equal to the name of the function. The function definition should include two parameters corresponding to the device instance and the progress message. For example:

```
def log(dev, report):
    print (dev.hostname, ': ', report)

def main():
    ....
    with SCP(dev, progress=log) as scp:
```

The following sample program transfers the `scp-test1.txt` and `scp-test2.txt` files from the local host to the `/var/tmp` directory on the target device, and then transfers the messages log file from the target device to a `logs` directory on the local host. The messages log is renamed to append the device hostname to the filename. The example uses SSH keys, which are already configured on the local host and the device, for authentication.

For comparison purposes, the program uses both the default progress messages as well as custom messages, which are defined in the function named `log`, to track the progress of the transfers.

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

```

def log(dev, report):
    print (dev.hostname + ': ' + report)

def main():
    dev = Device('router1.example.com')
    try:
        dev.open()

        #Default progress messages
        with SCP(dev, progress=True) as scp1:
            scp1.put("scp-test1.txt", remote_path="/var/tmp/")
            scp1.get("/var/log/messages",
local_path="logs/"+dev.hostname+"-messages")

        #Custom progress messages
        with SCP(dev, progress=log) as scp2:
            scp2.put("scp-test2.txt", remote_path="/var/tmp/")
            scp2.get("/var/log/messages",
local_path="logs/"+dev.hostname+"-messages")

    except Exception as err:
        print (err)
        return
    else:
        dev.close()

if __name__ == "__main__":
    main()

```

The progress of the transfers is sent to standard output. The default output (**progress=True**) includes the device name, the file being transferred, and the progress of the transfer in both bytes and as a percentage.

```

router1.example.com: scp-test1.txt: 8 / 8 (100%)
router1.example.com: logs/router1.example.com-messages: 0 / 229513 (0%)
router1.example.com: logs/router1.example.com-messages: 24576 / 229513 (10%)
router1.example.com: logs/router1.example.com-messages: 139264 / 229513 (60%)
router1.example.com: logs/router1.example.com-messages: 229513 / 229513 (100%)

```

The custom function produces similar output in this case.

```

router1.example.com : scp-test2.txt: 1 / 1 (100%)
router1.example.com : logs/router1.example.com-messages: 0 / 526493 (0%)
router1.example.com : logs/router1.example.com-messages: 57344 / 526493 (10%)
router1.example.com : logs/router1.example.com-messages: 106496 / 526493 (20%)
router1.example.com : logs/router1.example.com-messages: 212992 / 526493 (40%)
router1.example.com : logs/router1.example.com-messages: 319488 / 526493 (60%)
router1.example.com : logs/router1.example.com-messages: 368640 / 526493 (70%)
router1.example.com : logs/router1.example.com-messages: 475136 / 526493 (90%)
router1.example.com : logs/router1.example.com-messages: 526493 / 526493 (100%)

```

After executing the program, you can verify that the **scp-test1.txt** and **scp-test2.txt** files were copied to the correct directory on the target device by issuing the **file list** command on the device.

```

user1@router1> file list /var/tmp/scp-test*

```

```
/var/tmp/scp-test1.txt
/var/tmp/scp-test2.txt
```

On the local host, the **messages** log file, which is renamed to include the device hostname, should be present in the **logs** directory.

```
[user1@localhost ~]$ ls logs
router1.example.com-messages
```

The **SCP** class provides support for ProxyCommand, which enables you to transfer files from the local host to the target device through an intermediary host that supports netcat. This is useful when you can only log in to the target device through the intermediate host. To configure ProxyCommand, add the appropriate information to the SSH configuration file. For example:

```
[user1@localhost ~]$ cat ~/.ssh/config
Host 198.51.100.1
User user1
ProxyCommand ssh -l user1 198.51.100.2 nc %h 22 2>/dev/null
```

By default, Junos PyEZ queries the default SSH configuration file at `~/.ssh/config`, if one exists. However, you can specify a different SSH configuration file when you create the device instance by including the **ssh\_config** parameter in the **Device** argument list.

```
ssh_config_file = "~/.ssh/config_dc"
dev = Device('198.51.100.1', ssh_config=ssh_config_file)
```

Starting in Junos PyEZ Release 2.0.1, when you include the **ssh\_private\_key\_file** parameter in the **Device** argument list to define a specific SSH private key file for authentication, the **SCP** instance uses the same key file for authentication when transferring files.

```
key_file="/home/user/.ssh/id_rsa_private"
dev = Device('198.51.100.1', ssh_private_key_file=key_file)
dev.open()

with SCP(dev) as scp:
    scp.put("scp-test.txt", remote_path="/var/tmp/")
```

- Related Documentation**
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)



## CHAPTER 14

# Creating and Using Junos PyEZ Tables and Views

- [Understanding Junos PyEZ Tables and Views on page 119](#)
- [Junos PyEZ Predefined Operational Tables and Views on page 121](#)
- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
- [Importing External Junos PyEZ Tables and Views on page 125](#)
- [Defining Junos PyEZ Operational Tables on page 127](#)
- [Defining Junos PyEZ Views for Operational Tables on page 133](#)
- [Using Junos PyEZ Operational Tables and Views on page 137](#)
- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)
- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS on page 161](#)
- [Saving and Loading Junos PyEZ Table XML to and from Files on page 173](#)

## Understanding Junos PyEZ Tables and Views

---

Junos PyEZ Tables and Views enable you to extract operational information and configuration data from devices running Junos OS as well as configure devices running Junos OS. To extract information, you use the Table widget to map command output or configuration data to a table, which consists of a collection of items that can then be examined as a View. Starting in Junos PyEZ Release 2.0, you can also use Tables and Views to define structured configuration resources. When you add the Table to the Junos PyEZ framework, Junos PyEZ dynamically creates a configuration class for the resource, which you can use to programmatically configure the resource on a device. Tables and Views are defined using YAML, so no complex coding is required to create your own custom Tables and Views.

Tables and Views provide a simple and efficient way to extract information from complex operational command output or configuration data and map it to a Python data structure.

Operational (op) Tables select items from the RPC reply of an operational command, and configuration Tables select data from specific hierarchies in the selected configuration database. Each Table item represents a record of data and has a unique key. A Table also references a Table View, which is used to map the tag names in the data to the variable property names used within the Python module.

For example, the following op Table retrieves output for the **get-arp-table-information** RPC, which corresponds to the **show arp** command in the CLI. The Table extracts **<arp-table-entry>** elements from the output, and the corresponding View selects three fields from each **arp-table-entry** item by mapping the user-defined field name to the XPath expression that corresponds to the location of that data in the Junos XML output.

```
---
ArpTable:
  rpc: get-arp-table-information
  item: arp-table-entry
  key: mac-address
  view: ArpView

ArpView:
  fields:
    mac_address: mac-address
    ip_address: ip-address
    interface_name: interface-name
```

Junos PyEZ configuration Tables can be used to both retrieve and modify configuration data on devices running Junos OS. Configuration tables that define the **get** property can only retrieve configuration data. Configuration Tables that define the **set** property can both retrieve and configure the statements defined in the corresponding View.

For information about creating and using operational Tables and Views, see the following topics:

- [Defining Junos PyEZ Operational Tables on page 127](#)
- [Defining Junos PyEZ Views for Operational Tables on page 133](#)
- [Using Junos PyEZ Operational Tables and Views on page 137](#)
- [Junos PyEZ Predefined Operational Tables and Views on page 121](#)

For information about creating and using configuration Tables and Views, see the following topics:

- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)
- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS on page 161](#)

For information about loading or importing custom Tables and Views in your Junos PyEZ application, see the following topics:

- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
  - [Importing External Junos PyEZ Tables and Views on page 125](#)
  - [Saving and Loading Junos PyEZ Table XML to and from Files on page 173](#)
- Related Documentation**
- [Defining Junos PyEZ Operational Tables on page 127](#)
  - [Defining Junos PyEZ Configuration Tables on page 141](#)
  - [Junos PyEZ Predefined Operational Tables and Views on page 121](#)
  - [Using Junos PyEZ Operational Tables and Views on page 137](#)
  - [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)

## Junos PyEZ Predefined Operational Tables and Views

Junos PyEZ operational (op) Tables and Views provide a simple and efficient way to extract information from complex operational command output. The Junos PyEZ `jnpr.junos.op` module contains Table and View definitions for some common operational commands. [Table 14 on page 121](#) lists each of the modules, the Table names defined in that module, and the RPC and corresponding CLI command for each Table. For information about the command options provided to the RPC, the key for each item, and the fields selected by the corresponding View, review the Table and View definitions in the `.yaml` file for that module.

For the most current list of Table and View definitions, see the Junos PyEZ GitHub repository at <https://github.com/Juniper/py-junos-eznc/>. You can also create custom Table and View definitions, which are loaded using the `jnpr.junos.factory.loadyaml()` method.

**Table 14: `jnpr.junos.op` Modules**

Module	Table	RPC	CLI Command
arp	ArpTable	get-arp-table-information	show arp
bfd	BfdSessionTable	get-bfd-session-information	show bfd session extensive
bgp	bgpTable	get-bgp-neighbor-information	show bgp neighbor
ccc	CCCTable	get-ccc-information	show connections
ethernetswitchingtable	EthernetSwitchingTable	get-ethernet-switching-table-information	show ethernet-switching table

Table 14: jnpr.junos.op Modules (*continued*)

Module	Table	RPC	CLI Command
ethport	EthPortTable	get-interface-information (filtered for Ethernet interfaces)	show interfaces media
fpc	FpcHwTable	get-chassis-inventory	show chassis hardware
	FpcMiReHwTable		
	FpcInfoTable	get-fpc-information	show chassis fpc
	FpcMiReInfoTable		
idpattacks	IDPAttackTable	get-idp-attack-table-information	show security idp attack table
intopticdiag	PhyPortDiagTable	get-interface-optics-diagnostics-information	show interfaces diagnostics optics
inventory	ModuleTable	get-chassis-inventory	show chassis hardware
isis	IsisAdjacencyTable	get-isis-adjacency-information	show isis adjacency
l2circuit	L2CircuitConnectionTable	get-l2ckt-connection-information	show l2circuit connections
lACP	LACPPortTable	get-lACP-interface-information	show lACP interfaces
ldp	LdpNeighborTable	get-ldp-neighbor-information	show ldp neighbor extensive
lldp	LLDPNeighborTable	get-lddp-neighbors-information	show lldp neighbors
nd	NdTable	get-ipv6-nd-information	show ipv6 neighbors
ospf	OspfNeighborTable	get-ospf-neighbor-information	show ospf neighbor
	OspfInterfaceTable	get-ospf-interface-information	show ospf interface
	ospfTable	get-ospf-overview-information	show ospf overview
	OspfRoutesTable	get-ospf-route-information	show ospf route

Table 14: jnpr.junos.op Modules (*continued*)

Module	Table	RPC	CLI Command
phyport	PhyPortTable	get-interface-information (filtered for Ethernet interfaces)	show interfaces
	PhyPortStatsTable	get-interface-information (filtered for Ethernet interfaces)	show interfaces extensive
	PhyPortErrorTable	get-interface-information (filtered for Ethernet interfaces)	show interfaces extensive
routes	RouteTable	get-route-information	show route
	RouteSummaryTable	get-route-summary-information	show route summary
teddb	TedTable	get-ted-database-information	show ted database
vlan	VlanTable	get-vlan-information	show vlans
xcvr	XcvrTable	get-chassis-inventory	show chassis hardware

The following video presents a short Python session that demonstrates how to use the RouteTable widget.



Video: Junos PyEZ - Tables

#### Related Documentation

- [Using Junos PyEZ Operational Tables and Views on page 137](#)

## Loading Inline Junos PyEZ Tables and Views

Junos PyEZ Tables and Views provide a simple and efficient way to configure devices running Junos OS as well as extract specific information from operational command output and configuration data on the device. You can create quick inline Tables and Views within Python modules as multiline strings, or you can create one or more Table and View definitions in external files and import the required Tables into your Python modules. Inline Tables and Views are simpler to use, but using external files enables you to create a central, reusable library.

To use inline Tables and Views in your modules, you must create the Table and View definitions, and then load the definitions in the module. In Junos PyEZ, you define inline Tables and Views in YAML as a multiline string. For example:

```
myYAML = """
---
UserTable:
```

```
    get: system/login/user
    view: UserView
UserView:
    fields:
        username: name
        userclass: class
"""
```

To use inline Tables and Views, you must include several import statements in your module. Along with importing the Junos PyEZ **Device** class, you must also import the **yaml** package and the Junos PyEZ **jnpr.junos.factory.factory\_loader.FactoryLoader** class.

```
from jnpr.junos import Device
from jnpr.junos.factory.factory_loader import FactoryLoader
import yaml
```

The **yaml** package enables you to work with YAML, and the **FactoryLoader** class is used to load the Table and View definitions.

To load the Table and View definitions, you must include the following statement in your Python module, where *string-name* is the string that contains the definition:

```
globals().update(FactoryLoader().load(yaml.load(string-name)))
```

In this statement, the **yaml.load()** method maps the YAML-formatted Table definition to Python. The **FactoryLoader().load()** method then loads the Table and View definitions as if you had imported them from a module so that you can manipulate them as a class. Finally, the **globals().update()** call imports the definitions directly into your namespace.

After the Table and View definitions are loaded, there is no difference in how you use inline or external Tables in your module. To use a Table, first create the **Device** instance and open a connection to the target device. Then create the Table instance and associate it with the **Device** instance. You can then use the Table to retrieve information, configure the device, or both, depending on the type of Table.

```
dev = Device(host='router.example.com').open()
users = UserTable(dev)
```

In this example, **UserTable** retrieves **user** objects from the **[edit system login]** hierarchy level. To retrieve the Table information, call the **get()** method with any desired arguments.

```
users.get()
```

You can now iterate over and manipulate the resulting object to extract the required information. For example:

```
for account in users:
    print("Username is {}\nUser class is {}".format(account.username,
    account.userclass))
```

The sample Junos PyEZ program in its entirety is presented here:

```
from jnpr.junos import Device
from jnpr.junos.factory.factory_loader import FactoryLoader
import yaml

myYAML = """
---
UserTable:
    get: system/login/user
    view: UserView
```

```

UserView:
  fields:
    username: name
    userclass: class
"""

globals().update(FactoryLoader().load(yaml.load(myYAML)))

dev = Device(host='router.example.com').open()
users = UserTable(dev)
users.get()

for account in users:
    print("Username is {}\nUser class is {}".format(account.username,
    account.userclass))

dev.close()

```

- Related Documentation**
- [Understanding Junos PyEZ Tables and Views on page 119](#)
  - [Importing External Junos PyEZ Tables and Views on page 125](#)
  - [Defining Junos PyEZ Operational Tables on page 127](#)
  - [Defining Junos PyEZ Configuration Tables on page 141](#)

## Importing External Junos PyEZ Tables and Views

Junos PyEZ Tables and Views provide a simple and efficient way to configure devices running Junos OS as well as extract specific information from operational command output and configuration data on the device. You can create quick inline Tables and Views within Python modules as multiline strings, or you can create one or more Table and View definitions in external files and import the required Tables into your Python modules. Inline Tables and Views are simpler to use, but using external files enables you to create a central, reusable library.

To use external Tables and Views in your modules, you must create the Table and View definitions, and then import the definitions into the module. Tables and Views are defined using YAML.

External Table and View definitions are placed in files that are external to the Python module. You must create two files with the same base name, one of which has a **.yaml** extension, and the other of which has a **.py** extension. The **.yaml** file contains one or more Table/View definitions. For example:

```

[user@localhost]$ cat myTables/ConfigTables.yaml
---
UserTable:
  get: system/login/user
  view: UserView

UserView:
  fields:
    username: name
    userclass: class

```

```
ExtendedUserTable:
    get: system/login/user
    view: ExtendedUserView
```

```
ExtendedUserView:
    fields:
        username: name
        userclass: class
        userid: uid
```

The **.py** file must always contain the following four lines of code, which cause the main script to load the Table and View definitions as if you had imported them from a module so that you can manipulate them as a class within the script:

```
[user@localhost]$ cat myTables/ConfigTables.py
from jnpr.junos.factory import loadyaml
from os.path import splitext
__YAML__ = splitext(__file__)[0] + '.yaml'
globals().update(loadyaml(__YAML__))
```

If the **.yaml** and **.py** files are located in a subdirectory, you must also add an **\_\_init\_\_.py** file to that subdirectory, so that Python checks this directory when it processes the Table import statements in your module.

```
[user@localhost]$ ls myTables
__init__.py  ConfigTables.py  ConfigTables.yaml
```

To use external Tables and Views, you must include several import statements in your module. Along with importing the Junos PyEZ **Device** class, you must also import any required Tables that will be used in the script.

In the following sample code, the **myTables.ConfigTables** construct references the **ConfigTables.py** file within the **myTables** subdirectory and imports **UserTable** as defined in the **ConfigTables.yaml** file:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable
```

After the Table and View definitions are loaded, there is no difference in how you use inline or external Tables in your module. To use a Table, first create the **Device** instance and open a connection to the target device. Then create the Table instance and associate it with the **Device** instance. You can then use the Table to retrieve information, configure the device, or both, depending on the type of Table.

```
dev = Device(host='router.example.com').open()
users = UserTable(dev)
```

In this example, **UserTable** retrieves **user** objects from the **[edit system login]** hierarchy level. To retrieve the Table information, call the **get()** method with any desired arguments.

```
users.get()
```

You can now iterate over and manipulate the resulting object to extract the required information. For example:

```
for account in users:
    print("Username is {}\nUser class is {}".format(account.username,
        account.userclass))
```

The sample Junos PyEZ program in its entirety is presented here:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router.example.com').open()
users = UserTable(dev)
users.get()

for account in users:
    print("Username is {}\nUser class is {}".format(account.username,
        account.userclass))

dev.close()
```

#### Related Documentation

- [Understanding Junos PyEZ Tables and Views on page 119](#)
- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
- [Defining Junos PyEZ Operational Tables on page 127](#)
- [Defining Junos PyEZ Configuration Tables on page 141](#)

## Defining Junos PyEZ Operational Tables

You define Junos PyEZ operational (op) Tables to extract specific data from the RPC reply of an operational command executed on a device running Junos OS. This enables you to quickly retrieve and review the relevant operational state information for the device.

Junos PyEZ Tables are formatted using YAML. Op Table definitions can include a number of required and optional parameters, which are summarized in [Table 15 on page 127](#).

**Table 15: Junos PyEZ Op Table Parameters**

Table Parameter Name	Table Parameter	Description
Table name	—	User-defined Table identifier.
RPC command	<b>rpc</b>	Request tag name of the RPC for which to return operational data.
RPC default arguments	<b>args</b>	(Optional) Default command options and arguments for the RPC.

Table 15: Junos PyEZ Op Table Parameters (*continued*)

Table Parameter Name	Table Parameter	Description
RPC optional argument key	<b>args_key</b>	<p>(Optional) Reference to a command's optional first argument when that argument does not require a specific keyword.</p> <p>If you do not include this property, you must specify the keyword, or option name, for optional arguments that are included in the <code>get()</code> method argument list when you retrieve the operational data.</p>
Table item	<b>item</b>	<p>XPath expression relative to the top-level element within the <code>&lt;rpc-reply&gt;</code> element that selects the items to extract from the data.</p> <p>These items become the reference for the associated View.</p>
Table item key	<b>key</b>	<p>(Optional) One or more XPath expressions selecting the tag or tags whose values uniquely identify the Table item for items that do not use the <code>&lt;name&gt;</code> element as an identifier or for cases where composite keys are required.</p> <p>If the item uses the <code>&lt;name&gt;</code> element for the identifier, you can omit this property.</p>
Table View	<b>view</b>	View that is used to extract field data from the Table items.

Consider the following Junos PyEZ op Table, which extracts operational state information for Ethernet interfaces on the target device:

```

---
EthPortTable:
  rpc: get-interface-information
  args:
    media: True
    interface_name: '[afgx]e*'
  args_key: interface_name
  item: physical-interface
  view: EthPortView

```

The following sections discuss the different components of the Table:

- [Table Name on page 129](#)
- [RPC Command \(rpc\) on page 129](#)
- [RPC Default Arguments \(args\) on page 129](#)
- [RPC Optional Argument Key \(args\\_key\) on page 130](#)
- [Table Item \(item\) on page 130](#)
- [Table Item Key \(key\) on page 131](#)
- [Table View \(view\) on page 133](#)

## Table Name

The Table name is a user-defined identifier for the Table. The YAML file or string can contain one or more Tables. The start of the YAML document must be left justified. For example:

```
---
EthPortTable:
  # Table definition
```

## RPC Command (rpc)

A Junos PyEZ op Table is used to extract specific information from operational command output. You must include the **rpc** property in the op Table definition to specify the operational command for which to retrieve data.

The **rpc** value is the Junos XML request tag for a command. For example, the request tag name for the **show interfaces** command is **get-interface-information**.

```
rpc: get-interface-information
```

The request tag can be found in the *Junos XML API Operational Developer Reference*, displayed in the CLI by including the **| display xml rpc** option after the command, or displayed using the Junos PyEZ **Device** instance **display\_xml\_rpc()** method.

## RPC Default Arguments (args)

The optional **args** property defines the default command options and arguments for the RPC. These are listed as key-value pairs that are indented under **args**. A default argument is used when you call the **get()** method in your script and do not provide an argument that overrides that default.

If an option is just a flag that does not require a specific value, you can include it in the argument list by setting the option value to **True** in the Table definition. For example, the **show interfaces media** command maps to the **get-interface-information** request tag with the argument **media: True**. If an option requires a value, set the argument value to the value you want to use as the default.

```
rpc: get-interface-information
args:
  media: True
  interface_name: '[afgx]e*'
```



**NOTE:** If the option name in the Junos OS command-line interface (CLI) is hyphenated, you must change any dashes in the name to underscores.

By default, Junos PyEZ normalizes all Table keys and values, which strips out all leading and trailing whitespace and replaces sequences of internal whitespace characters with a single space. To disable normalization for a Table, include **normalize: False** in the argument list.

```
args:
  normalize: False
```

## RPC Optional Argument Key (args\_key)

You use the optional **args\_key** property in cases where CLI commands take an optional first argument that does not require you to explicitly specify an option name or keyword. In the following example, the **show interfaces** command takes an interface name as an optional argument:

```
user@router> show interfaces ?
Possible completions:
<[Enter]> Execute this command
<interface-name> Name of physical or logical interface
ge-0/0/0
ge-0/0/0.0
```

The **args\_key** property enables you to use this optional argument when retrieving operational data without having to explicitly specify the keyword or option name.

```
args_key: interface_name
```

If you include the **args\_key** property in your Table definition, you can specify the argument value but omit the option name when you retrieve the data.

```
EthPortTable(dev).get('ge-0/3/0')
```

If you omit the **args\_key** property in your Table definition, you must explicitly specify the option name if you want to include this parameter when you retrieve the data.

```
EthPortTable(dev).get(interface_name='ge-0/3/0')
```

## Table Item (item)

The Table **item** property, which is required in all op Table definitions, identifies the data to extract from the RPC output. The **item** value is an XPath expression relative to the top-level element within the **<rpc-reply>** tag that selects the desired elements. These items become the reference for the associated View.

The following example shows sample, truncated CLI command output:

```
user@router> show interfaces media "[afgx]e*" | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1R1/junos">
  <interface-information
    xmlns="http://xml.juniper.net/junos/15.1R1/junos-interface" junos:style="normal">
    <physical-interface>
      <name>ge-0/0/0</name>
      <admin-status junos:format="Enabled">up</admin-status>
      <oper-status>up</oper-status>
      ...
    </physical-interface>
    <physical-interface>
      <name>ge-0/0/1</name>
      <admin-status junos:format="Enabled">up</admin-status>
      <oper-status>up</oper-status>
      ...
    </physical-interface>
  </interface-information>
</rpc-reply>
```

To select the **<physical-interface>** elements from this output, you include the **item** property with the XPath expression to the element. In this case, the **<physical-interface>** element is a direct child of the top-level **<interface-information>** element, and the XPath expression for the **item** value is just the element name.

```
item: physical-interface
```

These items become the reference for the associated View.

## Table Item Key (key)

The optional **key** property is an XPath expression or a list of XPath expressions that selects which tag or tags are used to uniquely identify a Table item for those items that do not use the **<name>** element as an identifier or for cases where you want to use composite keys.

In the following command output, each **<physical-interface>** element is uniquely identified by its **<name>** child element:

```
user@router> show interfaces media "[afgx]e*" | display xml
<rpc-reply>
  <interface-information>
    <physical-interface>
      <name>ge-0/0/0</name>
      ...
    </physical-interface>
    <physical-interface>
      <name>ge-0/0/1</name>
      ...
    </physical-interface>
  </interface-information>
</rpc-reply>
```

If the Table **item** property selects the **<physical-interface>** elements, you can omit the **key** property from the Table definition if you want to use the **<name>** element as the key by default.

In contrast, consider the following **show route brief** command output:

```
<rpc-reply>
  <route-information>
    <route-table>
      ...
      <rt junos:style="brief">
        <rt-destination>10.0.0.0/24</rt-destination>
        <rt-entry>
          <active-tag>*</active-tag>
          <current-active/>
          <last-active/>
          <protocol-name>Static</protocol-name>
          <preference>5</preference>
          <age junos:seconds="9450374">15w4d 09:06:14</age>
          ...
        </rt-entry>
      </rt>
      <rt junos:style="brief">
        <rt-destination>10.0.10.0/24</rt-destination>
        <rt-entry>
```

```

        <active-tag>*</active-tag>
        <current-active/>
        <last-active/>
        <protocol-name>Direct</protocol-name>
        <preference>0</preference>
        <age junos:seconds="9450380">15w4d 09:06:20</age>
        ...
    </rt-entry>
</rt>
</route-table>
</route-information>
</rpc-reply>

```

When selecting the **route-table/rt** elements, there is no corresponding **<name>** element to uniquely identify each route entry. When the **<name>** identifier is absent, the **key** property must specify which tag or tags uniquely identify each item. In this case, you can uniquely identify each **route-table/rt** item by using **<rt-destination>** as the key.

```

item: route-table/rt
key: rt-destination

```

Table items can be defined by a key consisting of a single element or multiple elements. Single-element keys use a simple XPath expression for the value of the **key** property. Composite keys are defined by a list of XPath expressions. Consider the following Table definition:

```

PicHwTable:
  rpc: get-chassis-inventory
  item: ../name[starts-with(., 'PIC')]/parent::*
  key:
    - ancestor::*[starts-with(name, 'FPC')]/name
    - ancestor::*[starts-with(name, 'MIC')]/name
    - name
  view: PicHwView

```

The composite key for this Table definition might be similar to the following:

```
('FPC 2', 'MIC 0', 'PIC 0')
```

If a composite key references a missing element, Junos PyEZ replaces the value in the key with **None**.

```
('FPC 0', None, 'PIC 0')
```

The **key** property also supports the XPath union (|) operator. For example, the **LLDPNeighborTable**, which is shown here for reference, can select the **lldp-local-interface** and **lldp-local-port-id** elements as keys:

```

---
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
    local_parent: lldp-local-parent-interface-name
    remote_type: lldp-remote-chassis-id-subtype
    remote_chassis_id: lldp-remote-chassis-id

```

```
remote_port_desc: lldp-remote-port-description
remote_sysname: lldp-remote-system-name
```

When the **key** property uses the union operator, each key present in the RPC reply is added to the list of keys. The union operator can be used to specify an implicit "or" and is useful in situations where different tag names are present for different types of configurations or releases. For example, if the RPC returns **lldp-local-interface** as the identifier for one device, and the same RPC returns **lldp-local-port-id** as the identifier for another device, the Table automatically selects the appropriate key.

## Table View (view)

The **view** property associates the Table definition with a particular View. A View maps your user-defined field names to elements in the selected Table items using XPath expressions. You can customize the View to only select the necessary elements from the Table items.

### Related Documentation

- [Defining Junos PyEZ Views for Operational Tables on page 133](#)
- [Using Junos PyEZ Operational Tables and Views on page 137](#)
- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
- [Importing External Junos PyEZ Tables and Views on page 125](#)
- [Defining Junos PyEZ Configuration Tables on page 141](#)

## Defining Junos PyEZ Views for Operational Tables

Junos PyEZ operational (op) Tables select specific data from the RPC reply of an operational command for devices running Junos OS. A Table is associated with a View, which is used to access fields in the Table items. You associate a Table with a particular View by including the **view** property in the Table definition, which takes the View name as its argument.

A View maps your user-defined field names to XML elements in the selected Table items. A View enables you to access specific fields in the output as variables with properties that can be manipulated in Python. Junos PyEZ handles the extraction of the data into Python as well as any type conversion or data normalization.

Junos PyEZ Views, like Tables, are formatted using YAML. View definitions that are associated with op Tables can include a number of parameters, which are summarized in [Table 16 on page 133](#).

**Table 16: Junos PyEZ Op View Parameters**

View Parameter Name	View Parameter	Description
View name	–	User-defined View identifier.

Table 16: Junos PyEZ Op View Parameters (*continued*)

View Parameter Name	View Parameter	Description
Field items	<b>fields</b>	Associative array, or dictionary, of key-value pairs that map user-defined field names to XPath expressions that select elements in the Table items.
Field groups	<b>fields_group</b>	Associative array, or dictionary, of key-value pairs that map user-defined field names to XPath expressions that select elements in the Table items. The XPath expressions are relative to the context set by the corresponding groups parameter.
Groups	<b>groups</b>	Associative array, or dictionary, of key-value pairs that map a user-defined group name to an XPath expression (relative to the Table item context) that sets the XPath context for fields in that group.

Consider the following Junos PyEZ op Table and View, which extract operational state information for Ethernet interfaces on the target device:

```

---
EthPortTable:
  rpc: get-interface-information
  args:
    media: True
    interface_name: '[afgx]e*'
  args_key: interface_name
  item: physical-interface
  view: EthPortView

EthPortView:
  groups:
    mac_stats: ethernet-mac-statistics
    flags: if-device-flags
  fields:
    oper: oper-status
    admin: admin-status
    description: description
    mtu: { mtu : int }
    link_mode: link-mode
    macaddr: current-physical-address
  fields_mac_stats:
    rx_bytes: input-bytes
    rx_packets: input-packets
    tx_bytes: output-bytes
    tx_packets: output-packets
  fields_flags:
    running: { ifdf-running: flag }
    present: { ifdf-present: flag }

```

The following sections discuss the different components of the View:

- [View Name on page 135](#)
- [Fields \(fields\) on page 135](#)
- [Groups \(groups\) and Field Groups \(fields\\_\) on page 136](#)

## View Name

The View name is a user-defined identifier for the View. You associate a Table with a particular View by including the **view** property in the Table definition and providing the View name as its argument. For example:

```
---
EthPortTable:
  # Table definition
  view: EthPortView

EthPortView:
  # View definition
```

## Fields (fields)

You customize Views so that they only reference the necessary elements from the selected Table items. To do this you include the **fields** property and an associative array containing the mapping of user-defined field names to XPath expressions that select the desired elements from the Table item. The XPath expressions are relative to the Table item context.

Consider the following sample RPC output:

```
<rpc-reply>
  <interface-information>
    <physical-interface>
      <name>ge-0/3/0</name>
      <admin-status junos:format="Enabled">up</admin-status>
      <oper-status>down</oper-status>
      <local-index>135</local-index>
      <snmp-index>530</snmp-index>
      <link-level-type>Ethernet</link-level-type>
      <mtu>1514</mtu>
      ...
    </physical-interface>
  </interface-information>
</rpc-reply>
```

If the Table **item** parameter selects **<physical-interface>** elements from the output, the XPath expression for each field in the View definition is relative to that context. The following View definition maps each user-defined field name to a child element of the **<physical-interface>** element:

```
EthPortView:
  fields:
    oper: oper-status
    admin: admin-status
    mtu: { mtu : int }
```

In the Python script, you can then access a View item as a variable property. By default, each View item has a **name** property that references the key that uniquely identifies that item.

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable

dev = Device(host='router.example.com').open()
```

```
eths = EthPortTable(dev).get()

for item in eths:
    print (item.name)
    print (item.oper)
    print (item.admin)
    print (item.mtu)

dev.close()
```

The field format determines the type for a field's value. By default, field values are stored as strings. You can specify a different type for the field value in the field mapping. The following example defines the value of the **mtu** element to be an integer:

```
EthPortView:
  fields:
    mtu: { mtu : int }
```

In the RPC output, some Junos XML elements are just empty elements that act as flags. You can explicitly indicate that a field is a flag in the field mapping. The field item value for a flag is True if the element is present in the output and False if the element is absent. The following example defines the **ifdf-running** element as a flag:

```
EthPortView:
  fields:
    mtu: { mtu : int }
    running: { if-device-flags/ifdf-running : flag }
```

You can also set the field item value to a Boolean by using the following syntax:

```
fieldname: { element-name: (True | False)=regex(expression) }
```

The element's value is evaluated against the regular expression passed to **regex()**. If the element's value matches the expression, the field item value is set to the Boolean defined in the format. In the following example, the **oper\_status\_down** field is set to True if the value of the **oper-status** element contains 'down':

```
oper_status_down: { oper-status: True=regex(down) }
```

You can also use more complex regular expressions and match against multiple values. The following field item is set to True if the address in the **rt-destination** element starts with '198.51.':

```
dc1_route: { rt-destination: True=regex(^198\.51\.) }
```

The following field item is set to True if the **no-refresh** element contains either value in the regular expression.

```
no_refresh: { no-refresh: 'True=regex(Session ID: 0x0|no-refresh)' }
```

## Groups (groups) and Field Groups (fields\_)

Groups provide a shortcut method to select and reference elements within a specific node-set in a Table item.

In the following RPC output, the **<if-device-flags>** element contains multiple child elements corresponding to values displayed in the **Device flags** field in the CLI output:

```
<rpc-reply>
  <interface-information>
    <physical-interface>
      <name>ge-0/3/0</name>
```

```

...
<if-device-flags>
  <ifdf-present/>
  <ifdf-running/>
</if-device-flags>
...
</physical-interface>
</interface-information>
</rpc-reply>

```

Within the View definition, you can use the **fields** property to access the child elements by providing the full XPath expression to each element relative to the selected Table item. For example, if the EthPortTable definition selects **<physical-interface>** items, the field item mapping would use the following XPath expressions:

```

EthPortView:
  fields:
    present: if-device-flags/ifdf-present
    running: if-device-flags/ifdf-running

```

Alternatively, you can create a group that sets the context to the **<if-device-flags>** element and then define field group items that just provide the XPath expression relative to that context. You can define any number of groups within a View definition.

To create a group, include the **groups** property and map a user-defined group name to the XPath expression that defines the new context. Then define a field group whose name is **fields\_** followed by the group name. The field group is an associative array containing the mapping of user-defined field names to XPath expressions that now are relative to the context set within **groups**.

The following example defines the group **flags** and the corresponding field group **fields\_flags**. The **flags** group sets the context to the **physical-interface/if-device-flags** hierarchy level, and the **present** and **running** fields access the values of the **ifdf-present** and **ifdf-running** elements, respectively.

```

EthPortView:
  groups:
    flags: if-device-flags
  fields_flags:
    present: ifdf-present
    running: ifdf-running

```

Whether you use fields or field groups, you access the value in the same manner within the Junos PyEZ script by using the user-defined field names.

- Related Documentation**
- [Defining Junos PyEZ Operational Tables on page 127](#)
  - [Using Junos PyEZ Operational Tables and Views on page 137](#)

## Using Junos PyEZ Operational Tables and Views

Junos PyEZ operational (op) Tables and Views provide a simple and efficient way to extract specific information from complex operational command output for devices running Junos OS. After loading or importing the Table definition into your Python module, you can retrieve the Table items and extract and manipulate the data.

To retrieve information from a specific device, you must create a Table instance and associate it with the **Device** object representing the target device. For example:

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable

dev = Device(host='router.example.com').open()
eths = EthPortTable(dev)
```

The following sections discuss how to then retrieve and manipulate the data:

- [Retrieving Table Items on page 138](#)
- [Accessing Table Items on page 139](#)
- [Iterating Through a Table on page 140](#)

## Retrieving Table Items

The Table **item** property determines which items are extracted from the operational command output. For example, the Junos PyEZ EthPortTable definition, which is included here for reference, executes the **show interfaces media "[afgx]e"** command by default and extracts the **physical-interface** items from the output.

```
---
EthPortTable:
  rpc: get-interface-information
  args:
    media: True
    interface_name: '[afgx]e*'
  args_key: interface_name
  item: physical-interface
  view: EthPortView
```

You retrieve the Table items in your Python script by calling the **get()** method and supplying any desired arguments. If the Table definition includes default arguments in the **args** property, the executed RPC automatically includes these arguments when you call **get()** unless you override them in your argument list.

To retrieve all Table items, call the **get()** method with an empty argument list.

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable

dev = Device(host='router.example.com').open()
eths = EthPortTable(dev).get()
```

You can also retrieve specific Table items by passing command options as arguments to the **get()** method. If the command option is a flag that does not take a value, set the option equal to True in the argument list. Otherwise, include the argument and desired value as a key-value pair in the argument list. You can review the possible arguments for operational commands in the Junos OS CLI.

By default, EthPortTable returns information for Ethernet interfaces that have names matching the expression **"[afgx]e"**. To retrieve the Table item for the ge-0/3/0 interface only, include **interface\_name='ge-0/3/0'** as an argument to **get()**.

```
eths = EthPortTable(dev).get(interface_name='ge-0/3/0')
```



**NOTE:** If the option name in the Junos OS command-line interface (CLI) is hyphenated, you must change any dashes in the name to underscores. The argument value, however, is a string and as such can contain hyphens.

If the CLI command takes an optional first argument that does not require you to explicitly specify an option name or keyword, you can omit the option name in the `get()` method argument list provided that the Table `args_key` property references this argument. In the following example, the `show interfaces` command takes an interface name as an optional argument:

```
user@router> show interfaces ?
Possible completions:
<[Enter]> Execute this command
<interface-name> Name of physical or logical interface
ge-0/0/0
ge-0/0/0.0
```

The `EthPortTable` definition `args_key` property defines the optional argument as `interface_name`, which enables you to use this argument without having to explicitly specify the option name in the `get()` method argument list.

```
EthPortTable(dev).get('ge-0/3/0')
```

By default, Junos PyEZ normalizes all op Table keys and values, which strips out all leading and trailing whitespace and replaces sequences of internal whitespace characters with a single space. To disable normalization, include `normalize=False` as an argument to the `get()` method.

```
eths = EthPortTable(dev).get(interface_name='ge-0/3/0', normalize=False)
```

## Accessing Table Items

After you retrieve the Table items, you can treat them like a Python dictionary, which enables you to use methods in the standard Python library to access and manipulate the items.

To view the list of dictionary keys corresponding to the Table item names, call the `keys()` method.

```
eths = EthPortTable(dev).get(interface_name='ge-0/3/0')
print (eths.keys())
```

In this case, there is only a single key.

```
['ge-0/3/0']
```

You can verify that a specific key is present in the Table items by using the Python `in` operator.

```
'ge-0/3/0' in eths
```

To view a list of the fields, or values, associated with each key, call the **values()** method. The **values()** method returns a list of tuples with the name-value pairs for each field that was defined in the View.

```
print (eths.values())

[('oper', 'down'), ('rx_packets', '0'), ('macaddr', '00:00:5E:00:53:01'),
 ('description', None), ('rx_bytes', '0'), ('admin', 'up'), ('mtu', 1514),
 ('running', True), ('link_mode', None), ('tx_bytes', '0'), ('tx_packets', '0'),
 ('present', True)]
```

To view the complete list of items, including both keys and values, call the **items()** method.

```
print (eths.items())

[('ge-0/3/0', [('oper', 'down'), ('rx_packets', '0'), ('macaddr',
'00:00:5E:00:53:01'), ('description', None), ('rx_bytes', '0'), ('admin', 'up'),
('mtu', 1514), ('running', True), ('link_mode', None), ('tx_bytes', '0'),
('tx_packets', '0'), ('present', True)])]
```

## Iterating Through a Table

Tables support iteration, which enables you to loop through each Table item in the same way that you would loop through a list or dictionary. This makes it easy to quickly format and print desired fields.

The EthPortTable definition, which is included in the **jnpr.junos.op** module, executes the **show interfaces media "[afgx]e"** command and extracts the **physical-interface** items from the output. The following code loops through the **physical-interface** items and prints the name and operational status of each Ethernet port:

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable

dev = Device(host='router.example.com').open()
eths = EthPortTable(dev).get()

for port in eths:
    print ("{}: {}".format(port.name, port.oper))
```

The **oper** field, which is defined in EthPortView, corresponds to the value of the **oper-status** element in the output. The EthPortView definition does not define a **name** field. By default, each View item has a **name** property that references the key that uniquely identifies that item.

The output includes the interface name and operational status.

```
ge-0/3/0: up
ge-0/3/1: up
ge-0/3/2: up
ge-0/3/3: up
```

### Related Documentation

- [Defining Junos PyEZ Operational Tables on page 127](#)
- [Defining Junos PyEZ Views for Operational Tables on page 133](#)

## Defining Junos PyEZ Configuration Tables

You define Junos PyEZ configuration Tables to extract specific data from the selected configuration database of a device running Junos OS or to create structured resources that can be used to programmatically configure a device running Junos OS. Thus, you can quickly retrieve or modify specific configuration objects on the device.

Junos PyEZ Tables are formatted using YAML. When you define a Junos PyEZ configuration Table, you must specify the configuration scope using either **get** or **set**. Tables that include the **get** property can only retrieve the specified configuration data from a device. Tables that include the **set** property define configuration resources that you can use to configure the device as well as to retrieve configuration data from the device. Thus, they are a superset and include all of the functionality of Tables that specify **get**.

Configuration Table definitions can include a number of required and optional parameters, which are summarized in [Table 17 on page 141](#). [Table 17 on page 141](#) also specifies whether the parameter can be used in Tables that solely retrieve configuration data from the device (**get**) or in Tables that can also configure the device (**set**).

**Table 17: Junos PyEZ Configuration Table Parameters**

Table Parameter Name	Table Parameter	Table Type	Description
Table name	—	<b>get</b> or <b>set</b>	User-defined Table identifier.
Configuration scope	<b>get</b> or <b>set</b>	—	<p>XPath expression relative to the top-level <b>&lt;configuration&gt;</b> element that identifies the configuration hierarchy level at which to select or configure objects, depending on the Table type.</p> <p>Specify <b>get</b> to retrieve configuration objects or specify <b>set</b> to both configure and retrieve objects.</p> <p>These objects become the reference for the associated View.</p>
Configuration resource key field	<b>key-field</b>	<b>set</b>	<p>String or list of strings that references any field names defined in the View that map to identifier elements and can be used to uniquely identify the configuration object. For example, you might specify the field name that corresponds to the <b>&lt;name&gt;</b> element for an object.</p> <p>You must always define at least one key field in the Table, and users must declare values for all keys when configuring the resource in their application.</p>

Table 17: Junos PyEZ Configuration Table Parameters (*continued*)

Table Parameter Name	Table Parameter	Table Type	Description
Required keys	<b>required_keys</b>	<b>get or set</b>	<p>(Optional) Associative array, or dictionary, of key-value pairs that map a hierarchy level in the configuration scope to the element that uniquely identifies the object at that hierarchy level, for example, the <b>&lt;name&gt;</b> element.</p> <p>Users must include all required keys as arguments to the <b>get()</b> method when retrieving the configuration data in their application.</p>
Table View	<b>view</b>	<b>get or set</b>	View associated with the Table.

Consider the following Junos PyEZ configuration Tables and their associated Views.

**UserTable**, which includes the **get** property, extracts configuration data for user accounts on the target device. **UserConfigTable**, which includes the **set** property, defines a structured configuration resource that can be used to configure user accounts on the target device as well as retrieve configuration data for user accounts.

```

---
UserTable:
  get: system/login/user
  required_keys:
    user: name
  view: UserView
UserView:
  fields:
    username: name
    userclass: class
    uid: uid

UserConfigTable:
  set: system/login/user
  key-field:
    username
  required_keys:
    user: name
  view: UserConfigView
UserConfigView:
  fields:
    username: name
    userclass: class
    uid: uid
    password: authentication/encrypted-password
    fullname: full-name

```

The following sections discuss the different components of the Tables:

- [Table Name on page 143](#)
- [Configuration Scope \(get or set\) on page 143](#)

- [Key Field \(key-field\) on page 144](#)
- [Required Keys \(required\\_keys\) on page 145](#)
- [Table View \(view\) on page 146](#)

## Table Name

The Table name is a user-defined identifier for the Table. The YAML file or string can contain one or more Tables. The start of the YAML document must be left justified. For example:

```
---
UserTable:
  # Table definition
```

## Configuration Scope (get or set)

The configuration scope property, which is required in all configuration Table definitions, identifies the configuration hierarchy level at which to retrieve or configure objects, depending on the Table type. Junos PyEZ configuration Tables can be used to both retrieve and modify configuration data on a device running Junos OS. Configuration tables that specify the **get** property can only retrieve configuration data. Configuration Tables that specify the **set** property can both configure and retrieve data.

The value for **get** or **set** is an XPath expression relative to the top-level **<configuration>** element that identifies the hierarchy level at which to retrieve or set the configuration data. This data becomes the reference for the associated View.

Consider the following sample configuration hierarchy:

```
user@router> show configuration system login | display xml
<rpc-reply>
  <configuration>
    <system>
      <login>
        ...
        <user>
          <name>user1</name>
          <uid>2001</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>...</encrypted-password>
          </authentication>
        </user>
        <user>
          <name>readonly</name>
          <uid>3001</uid>
          <class>read-only</class>
          <authentication>
            <encrypted-password>...</encrypted-password>
          </authentication>
        </user>
      </login>
    </system>
  </configuration>
</rpc-reply>
```

To retrieve or configure the **user** elements at the **[edit system login]** hierarchy level, the value for the **get** or **set** property would use the following XPath expression:

```
system/login/user
```



**NOTE:** Do not include a slash ( / ) at the end of the XPath expression, because the script will generate an error.

For example, to define a Table that can only be used to retrieve **user** objects, use **get**.

```
get: system/login/user
```

To define a Table that can be used to configure **user** objects in addition to retrieving them, use **set**.

```
set: system/login/user
```

By default, Junos PyEZ configuration Tables retrieve data from the candidate configuration database. When you call the **get()** method in the Python script to retrieve the Table data, you can specify that the method should instead return data from the committed configuration database by passing in the **options** argument and including the **'database':'committed'** item in the **options** dictionary. For example:

```
table_object.get(options={'database': 'committed'})
```

## Key Field (key-field)

In the Junos OS configuration, each instance of a configuration object, for example, an interface or a user account, must have a unique identifier. In many cases, the **<name>** element, which is explicitly displayed in the Junos XML output, uniquely identifies each instance of the object. However, in some cases, a different element or a combination of elements is used. For example, a logical interface is uniquely identified by the combination of the physical interface name and the logical unit number.

Configuration Tables that specify the **set** property to define a configuration resource must indicate which element or combination of elements uniquely identifies the resource. The **key-field** property, which is a string or list of strings, serves this function and is required for all **set** configuration Tables.

The View for a **set** Table must explicitly define fields for all identifier elements for the configuration resource. The **key-field** property must then reference all of the field names for the identifier elements in the Table definition. When using the Table to configure the resource, a Junos PyEZ application must supply values for all key fields.

For example, the following Table defines a structured resource that can be used to configure user accounts at the **[edit system login]** hierarchy level. The View explicitly defines the **username** field and maps it to the **name** element at the **[edit system login user]** hierarchy level. The **key-field** property references this field to indicate that the **name** element uniquely identifies instances of that object.

```
UserConfigTable:
  set: system/login/user
  key-field:
```

```

    username
    required_keys:
        user: name
    view: UserConfigView
UserConfigView:
    fields:
        username: name
        userclass: class
        uid: uid
        password: authentication/encrypted-password
        fullname: full-name

```

When the Junos PyEZ application configures instances of the **UserConfigTable** resource on the device, it must define a value for the **username** key for each instance. For example:

```

from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable

dev = Device(host='router1.example.com').open()
users = UserConfigTable(dev)
users.username = 'admin'
users.userclass = 'super-user'
...

```

If the configuration Table defines statements in multiple hierarchy levels that have identifiers at each level, the **key-field** property must include all of the identifiers. For example, if the Table configures a logical unit on an interface, the **key-field** property must include both the interface name and the logical unit number as keys.

## Required Keys (required\_keys)

You include the optional **required\_keys** property in your configuration Table definition to require that the Table users provide values for one or more keys when they retrieve the data in their application. Each key must map a hierarchy level in the configuration scope defined by the **get** or **set** parameter to the **<name>** identifier at that level. You can only define one key per hierarchy level.

In the following example, **UserTable** requires that the Junos PyEZ application specify the value of a **name** element at the **[edit system login user]** hierarchy level when retrieving the data:

```

UserTable:
    get: system/login/user
    required_keys:
        user: name
    view: UserView

```

In the corresponding Junos PyEZ script, you must include the required keys in the **get()** method argument list. The following example requests the configuration data for the user named 'readonly':

```

from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router1.example.com').open()
users = UserTable(dev).get(user='readonly')

```

You can only require keys at hierarchy levels in the configuration scope defined by the **get** or **set** parameter. Consider the following definition for **get**:

```
get: interfaces/interface/unit
```

In this case, you can request that the user provide values for the interface name and the unit number as shown in the following sample code, but you cannot define a required key for the interface address, which is at a lower hierarchy level:

```
required_keys:
  interface: name
  unit: name
```

## Table View (view)

The **view** property associates the Table definition with a particular View. A View maps your user-defined field names to elements in the selected Table items using XPath expressions. You can customize the View to only select certain elements to retrieve or configure, depending on the Table type and operation.

For more information about defining Views for configuration Tables, see [“Defining Junos PyEZ Views for Configuration Tables” on page 146](#).

### Related Documentation

- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)
- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
- [Importing External Junos PyEZ Tables and Views on page 125](#)
- [Defining Junos PyEZ Operational Tables on page 127](#)

---

## Defining Junos PyEZ Views for Configuration Tables

Junos PyEZ configuration Tables can extract specific data from the selected configuration database of a device running Junos OS, or they can define structured resources that can be used to programmatically configure a device running Junos OS. A Table is associated with a View, which is used to select and reference elements in the Table data. You associate a Table with a particular View by including the **view** property in the Table definition, which takes the View name as its argument.

A *View* maps your user-defined field names to XML elements in the selected Table items. A View enables you to access specific fields in the data as variables with properties that can be manipulated in Python. Junos PyEZ handles the extraction of the data into Python as well as any type conversion or data normalization.

When retrieving configuration data using Tables that have the **get** or the **set** property, the View fields specify which configuration data the application should retrieve for the object. For Tables that include the **set** property and define resources that you can configure on a device, the fields defined in the View restrict which statements that you can configure for that resource.

Junos PyEZ Views, like Tables, are formatted using YAML. View definitions that are associated with configuration Tables can include a number of parameters, which are summarized in [Table 18 on page 147](#).

**Table 18: Junos PyEZ Configuration View Parameters**

View Parameter Name	View Parameter	Table Type	Description
View name	–	<b>get</b> or <b>set</b>	User-defined View identifier.
Field items	<b>fields</b>	<b>get</b> or <b>set</b>	<p>Associative array, or dictionary, of key-value pairs that map user-defined field names to XPath expressions that select elements in the configuration data. The XPath expressions are relative to the context defined by the <b>get</b> or <b>set</b> property for that Table.</p> <p>When the Table scope uses <b>get</b>, the fields identify the data to extract from the configuration. When the Table scope uses <b>set</b>, the fields identify the elements that you can configure or retrieve, depending on the operation.</p>
Field groups	<b>fields_group</b>	<b>get</b> or <b>set</b>	<p>Associative array, or dictionary, of key-value pairs that map user-defined field names to XPath expressions that select elements in the configuration data. The XPath expressions are relative to the context set by the corresponding <b>groups</b> parameter.</p> <p>When the Table scope uses <b>get</b>, the fields identify the data to extract from the configuration. When the Table scope uses <b>set</b>, the fields identify the elements that you can configure or retrieve, depending on the operation.</p>
Groups	<b>groups</b>	<b>get</b> or <b>set</b>	Associative array, or dictionary, of key-value pairs that map a user-defined group name to an XPath expression that sets the XPath context for fields in that group. The Xpath expression is relative to the context defined by the <b>get</b> or <b>set</b> property for that Table.

Consider the following Junos PyEZ configuration Tables and Views. **UserTable**, which includes the **get** property, extracts configuration data for user accounts on the target device. **UserConfigTable**, which includes the **set** property, defines a structured configuration resource that can be used to configure user accounts on the target device as well as retrieve configuration data for user accounts.

```
---
UserTable:
  get: system/login/user
  view: UserView
UserView:
  groups:
    auth: authentication
  fields:
    username: name
    userclass: class
    uid: uid
```

```
        uidgroup: { uid: group }
        fullgroup: { full-name: group }
    fields_auth:
        password: encrypted-password

---
UserConfigTable:
    set: system/login/user
    key-field:
        username
    view: UserConfigView
UserConfigView:
    groups:
        auth: authentication
    fields:
        username: name
        userclass: class
        uid: uid
    fields_auth:
        password: encrypted-password
```

The following sections discuss the different components of the View:

- [View Name on page 148](#)
- [Fields \(fields\) on page 148](#)
- [Field Options \('get' Tables\) on page 150](#)
- [Field Options \('set' Tables\) on page 151](#)
- [Groups \(groups\) and Field Groups \(fields\\_\) on page 152](#)

## View Name

The View name is a user-defined identifier for the View. You associate a Table with a particular View by including the **view** property in the Table definition and providing the View name as its argument. For example:

```
---
UserTable:
    # Table definition
    view: UserView

UserView:
    # View definition
```

## Fields (fields)

You customize Views so that they only reference the necessary elements in the selected configuration data. To do this you include the **fields** property and an associative array containing the mapping of user-defined field names to the XPath expressions that select the desired elements from the configuration Table items. The XPath expressions are relative to the configuration scope defined by the **get** or **set** property in the Table definition.

When retrieving configuration data using Tables that include either the **get** or the **set** property, the fields defined in the View identify the statements to extract from the configuration. For Tables that include the **set** property and define resources that you can configure on a device, the fields identify the statements that you can configure for that resource. You must explicitly define fields for all identifier elements for a configuration

resource. These identifier fields are then referenced in the **key-field** property in the corresponding Table definition.

Consider the following sample configuration hierarchy:

```
user@router> show configuration system login | display xml
<rpc-reply>
  <configuration>
    <system>
      <login>
        ...
        <user>
          <name>user1</name>
          <uid>2001</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>...</encrypted-password>
          </authentication>
        </user>
        <user>
          <name>readonly</name>
          <uid>3001</uid>
          <class>read-only</class>
          <authentication>
            <encrypted-password>...</encrypted-password>
          </authentication>
        </user>
      </login>
    </system>
  </configuration>
</rpc-reply>
```

If the Table **get** or **set** parameter defines the scope as **system/login/user**, the XPath expression for each field in the View definition is relative to that context. The following View definition maps the user-defined field names **username**, **userclass**, and **uid** to child elements of the **<user>** element:

```
UserTable:
  get: system/login/user
  ...

UIView:
  fields:
    username: name
    userclass: class
    uid: uid
```

If the Table definition includes the **set** property, you must explicitly define fields for any identifier elements that uniquely identify the object, which in this case is **<name>**. The Table's **key-field** property must reference all View fields that map to identifier elements for an object. You must always define at least one identifier element in the **fields** and **key-field** properties in **set** Tables.

In the Python script, you can then access a View item as a variable property.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router.example.com').open()
```

```

users = UserTable(dev).get()

for account in users:
    print("Username is {}\nUser class is {}".format(account.name,
account.userclass))

dev.close()

```



**NOTE:** When retrieving configuration data, each object that has a <name> element in the data has a default `name` property that you can use to access the value for that element.

View fields can include different options depending on the type of the Table that references that View. Tables that define structured configuration resources (**set**) can include type and constraint checks for each field to ensure that the Junos PyEZ application provides valid data when configuring the resource on a device. Tables that retrieve configuration data (**get**) can include options that return attribute values for specific elements or that specify the data type to use in the application. “[Field Options \('get' Tables\)](#)” on page 150 and “[Field Options \('set' Tables\)](#)” on page 151 outline the options that can be included when using **get** and **set** Tables, respectively.

## Field Options ('get' Tables)

Tables that include the **get** property and solely retrieve configuration data from a device can define a number of options or operators for fields in the associated View. This section outlines the various options.

The field format determines the type for a field's value. By default, field values are stored as strings. You can specify a different type for the value in the field mapping. The following example defines the value of the **uid** element to be an integer:

```

UIView:
  fields:
    username: name
    userclass: class
    uid: { uid : int }

```

You can also set the field item's value to a Boolean by using the following syntax:

```

fieldname: { element-name: (True | False)=regex(expression) }

```

The element's value is evaluated against the regular expression passed to **regex()**. If the element's value matches the expression, the field item's value is set to the Boolean defined in the format. In the following example, the **superuser** field is set to True if the value of the **class** element contains 'super-user':

```

superuser: { class : True=regex(super-user) }

```

Junos PyEZ also provides the **group** operator for fields in configuration Views. The operator enables you to access the value of the **junos:group** attribute for elements that are inherited from user-defined groups. This value indicates the group from which that element was inherited.

You include the **group** operator in the field mapping to reference the value of the **junos:group** attribute instead of the value of the element. The following example defines the **uidgroup** and **fullgroup** fields with the **group** operator. When you access these field names in a script, the field references the value of the **junos:group** attribute associated with the **uid** or **full-name** element.

```
UserView:
  groups:
    auth: authentication
  fields:
    username: name
    userclass: class
    uid: uid
    uidgroup: { uid: group }
    fullgroup: { full-name: group }
  fields_auth:
    password: encrypted-password
```

## Field Options ('set' Tables)

Tables that define structured configuration resources (**set**) can include type and constraint checks for each field in the associated View to ensure that the Junos PyEZ application provides valid data when configuring the resource on a device. Type checks ensure that the Junos PyEZ application supplies the correct data type when it configures the statements for a specific resource. Constraint checks enable you to define a default value for statements and ensure that the application supplies values that are in the correct range for those statements. The supported type and constraint checks, which are included as options for the fields of the associated View, are outlined in this section.

[Table 19 on page 151](#) and [Table 20 on page 152](#) summarize the type and constraint checks, respectively, that you can define for fields in the View of a **set** configuration Table. Type checks are mutually exclusive, but multiple constraint checks can be defined for each field.

**Table 19: Type Checks for 'set' Configuration Tables**

type Value	Description	Example
<b>bool</b>	Field only accepts Boolean values of <b>True</b> or <b>False</b>	<code>enable: { 'enable': { 'type': 'bool' } }</code>
<b>enum</b>	Field only accepts one of the values defined in the <b>enum</b> list	<code>enc: { 'encapsulation': { 'type': { 'enum': ['vlan-ccc', 'vlan-vpls'] } } }</code>
<b>float</b>	Field only accepts floating point values	<code>drift: { 'clock-drift': { 'type': 'float' } }</code>
<b>int</b>	Field only accepts integer values	<code>uid: { 'uid': { 'type': 'int' } }</code>
<b>str</b>	Field only accepts string values	<code>name: { 'name': { 'type': 'str' } }</code>

Table 20: Constraint Checks for 'set' Configuration Tables

Constraint Check Name	Description	Example
<b>default</b>	Default value for a field.  A field uses its default value when the user does not explicitly configure the field. If the user calls the <b>reset()</b> method to reset field values in the application, fields that have a defined default are set to that value.	<code>native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'default' : 501 } }</code>
<b>maxValue</b>	Maximum value for a field, which is interpreted based on the field <b>type</b> .	<code>native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'minValue' : 0, 'maxValue' : 4094 } }</code>
<b>minValue</b>	Minimum value for a field, which is interpreted based on the field <b>type</b> .	<code>native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'minValue' : 0, 'maxValue' : 4094 } }</code>

You can only define a single type check for a field, but you can define multiple constraint checks. Thus a field could include a **default** value, a minimum value (**minValue**), and a maximum value (**maxValue**).

```
native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'default' : 501,  
    'minValue' : 0, 'maxValue' : 4094 } }
```

The **minValue** and **maxValue** options are interpreted based on the value for the **type** option. By default, field values are strings. For strings, **minValue** and **maxValue** are the minimum and maximum lengths for the string. For integers and floats, the values are the minimum and maximum values for that type.

If you include type or constraint checks for a field, and the user supplies configuration data that fails the checks, the Junos PyEZ application raises the appropriate **TypeError** or **ValueError** exception with a message that describes the error.

## Groups (groups) and Field Groups (fields\_)

Groups provide a shortcut method to select and reference elements within a specific node-set in a Table item.

In the following configuration data, the **<authentication>** element contains a child element corresponding to the user's authentication:

```
<configuration>
  <system>
    <login>
      ...
      <user>
        <name>user1</name>
        <uid>2001</uid>
        <class>super-user</class>
        <authentication>
          <encrypted-password>...</encrypted-password>
        </authentication>
      </user>
    </system>
  </configuration>
```

```

        <uid>3001</uid>
        <class>read-only</class>
        <authentication>
            <encrypted-password>...</encrypted-password>
        </authentication>
    </user>
</login>
</system>
</configuration>

```

Within the View definition, you can use the **fields** property to access the child elements by providing the full XPath expression to each element relative to the selected configuration hierarchy. For example, if the Table **get** or **set** property selects **<user>** elements at the **[edit system login]** hierarchy level, the field item mapping would use the following XPath expression:

```

UserConfigTable:
    set: system/login/user
    ...

UserConfigView:
    fields:
        password: authentication/encrypted-password

```

Alternatively, you can create a group that sets the XPath context to the **<authentication>** element and then define field group items that just provide the XPath expression relative to that context. You can define any number of groups within a View definition.

To create a group, include the **groups** property and map a user-defined group name to the XPath expression that defines the new context. Then define a field group whose name is **fields\_** followed by the group name. The field group is an associative array containing the mapping of user-defined field names to XPath expressions that now are relative to the context set within **groups**.

The following example defines the group **auth** and the corresponding field group **fields\_auth**. The **auth** group sets the context to the **system/login/user/authentication** hierarchy level, and the **password** field references the value of the **encrypted-password** element.

```

UserConfigTable:
    set: system/login/user
    ...

UserConfigView:
    groups:
        auth: authentication
    fields_auth:
        password: encrypted-password
    ...

```

Whether you use fields or field groups, you access the value in the same manner within the Junos PyEZ script by using the user-defined field names.

#### Related Documentation

- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)

## Using Junos PyEZ Configuration Tables to Retrieve Configuration Data

---

Junos PyEZ configuration Tables and Views provide a simple and efficient way to extract specific information from the selected configuration database of a device running Junos OS. After loading or importing the Table definition into your Python module, you can retrieve the configuration data.

Junos PyEZ configuration Tables that specify the **get** property can only retrieve configuration data. Tables that specify the **set** property can configure resources on devices running Junos OS as well as retrieve data in the same manner as Tables that specify the **get** property.

To retrieve information from a specific device, you must create a Table instance and associate it with the **Device** object representing the target device. For example:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router.example.com').open()
users = UserTable(dev)
```

The following sections discuss how to then retrieve and manipulate the data:

- [Retrieving Configuration Items on page 154](#)
- [Specifying Inheritance and Group Options on page 155](#)
- [Accessing Table Items on page 157](#)
- [Iterating Through a Table on page 158](#)

### Retrieving Configuration Items

The configuration Table **get** or **set** property identifies the data to extract from the configuration. For example, the following sample Table definition extracts **user** elements at the **[edit system login]** configuration hierarchy level:

```
UserTable:
    get: system/login/user
    view: UserView

UserView:
    fields:
        username: name
        userclass: class
```

You retrieve the configuration data in your Python script by calling the **get()** method and supplying any desired arguments.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router.example.com').open()
users = UserTable(dev).get()
```

If the Table definition includes the **required\_keys** parameter, you must include key-value pairs for each required key in the **get()** method argument list. The following Table definition requires that the **get()** method argument list include a **user** argument with a value that

corresponds to the value of a **name** element at the **[edit system login user]** hierarchy level:

```
UserTable:
  get: system/login/user
  required_keys:
    user: name
  view: UserView
```

In the **get()** method, you must include the required key in the argument list; otherwise, the code throws a **ValueError** exception. The following example requests the configuration data for the user named 'operator':

```
users = UserTable(dev).get(user='operator')
```



**NOTE:** If the argument name is hyphenated, you must change any dashes in the name to underscores. The argument value, however, is a string and as such can contain hyphens.

You can include the **get()** method **namesonly=True** argument to return configuration data that contains only name keys at the hierarchy level specified in the **get** or **set** property of your Table definition.

```
from jnpr.junos import Device
from myTables.ConfigTables import InterfaceTable

dev = Device(host='router.example.com').open()
interfaces = InterfaceTable(dev).get(namesonly=True)
```

For example, if **get** is defined to retrieve configuration data at the **interfaces/interface** hierarchy level, and you include the **namesonly=True** argument in the **get()** method when you retrieve the data in your Junos PyEZ script, the method only returns the values in the **<name>** elements that are direct children of the **interfaces/interface** hierarchy level. Information in elements that are siblings of the **<name>** element is not returned, and data for **<name>** elements at lower levels in the hierarchy is not returned.

By default, Junos PyEZ configuration Tables retrieve data from the candidate configuration database. When you call the **get()** method in the Python script to retrieve the Table data, you can specify that the method should instead return data from the committed configuration database by passing in the **options** argument and including the **'database':'committed'** item in the **options** dictionary.

```
table_options = {'inherit':'inherit', 'groups':'groups', 'database':'committed'}

dev = Device(host='router.example.com').open()
users = UserTable(dev).get(options = table_options)
```

## Specifying Inheritance and Group Options

You can control inheritance and group options when you retrieve configuration data by using the **options** argument in the **get()** method argument list. The **options** argument takes a dictionary, and by default is set to the following value, which applies inheritance and groups for the returned configuration data:

```
options = {'inherit': 'inherit', 'groups': 'groups'}
```

If you do not redefine the **options** argument in your Python script, it automatically uses the default.

The **inherit** option specifies how the configuration data displays statements that are defined in configuration groups and interface ranges. By default, the **'inherit':'inherit'** option is included, and the configuration data encloses tag elements inherited from user-defined groups or interface ranges within the inheriting tag elements rather than display the **<groups>**, **<apply-groups>**, **<apply-groups-except>**, or **<interface-range>** elements separately. To apply inheritance but also include tag elements for statements defined in the **junos-defaults** group, use **'inherit':'defaults'** in the **options** argument.

To disable inheritance, set the dictionary value to an empty string.

```
{'inherit':''}
```

Including both the **'inherit':'inherit'** and **'groups':'groups'** options returns configuration data that also indicates the configuration group from which elements are inherited. An element that is inherited from a particular group includes the **junos:group="source-group"** attribute in its opening tag, as shown in the following example:

```
<configuration>
  <interfaces>
    <interface junos:group="re0">
      <name junos:group="re0">fxp0</name>
      <unit junos:group="re0">
        <name junos:group="re0">0</name>
        <family junos:group="re0">
          <inet junos:group="re0">
            <address junos:group="re0">
              <name junos:group="re0">198.51.100.1/24</name>
            </address>
          </inet>
        </family>
      </unit>
    </interface>
  </interfaces>
  ...
</configuration>
```

To provide access to the attributes in the View definition, you can include the appropriate XPath syntax in the field mapping. The following example defines the **ifgroup** field and maps it to the **junos:group** attribute of the interface's **<name>** element:

```
InterfaceTable:
  get: interfaces/interface
  view: InterfaceView
InterfaceView:
  fields:
    ifname: name
    ifaddress: unit/family/inet/address/name
    ifgroup: name/@group
```

Junos PyEZ also provides the **group** operator, which is a shortcut method for accessing the **junos:group** attribute of an element. The following example defines the **ifgroup** field, which is mapped to the **name** element with the **group** operator. When you access **ifgroup** within your script, it references the value for the **junos:group** attribute associated with the interface's **<name>** element.

```
InterfaceTable:
  get: interfaces/interface
  view: InterfaceView
InterfaceView:
  fields:
    ifname: name
    ifaddress: unit/family/inet/address/name
    ifgroup: { name : group }
```

If an element is not inherited from a group, the value of a field that references the **group** attribute is **None**.

## Accessing Table Items

After you retrieve the configuration items, you can treat them like a Python dictionary, which enables you to use methods in the standard Python library to access and manipulate the items.

To view the list of dictionary keys corresponding to the configuration item names, call the **keys()** method.

```
users = UserTable(dev).get()
print (users.keys())

['user1', 'readonly']
```

You can verify that a specific key is present in the Table items by using the Python **in** operator.

```
'readonly' in users
```

To view a list of the fields, or values, associated with each key, call the **values()** method. The **values()** method returns a list of tuples with the name-value pairs for each field that was defined in the View.

```
print (users.values())

[('username', 'user1'), ('userclass', 'super-user')], [('username', 'readonly'), ('userclass', 'read-only')]
```

To view the complete list of items, including both keys and values, call the **items()** method.

```
print (users.items())

['user1', [('username', 'user1'), ('userclass', 'super-user')]], ('readonly', [('username', 'readonly'), ('userclass', 'read-only')])]
```

## Iterating Through a Table

Tables support iteration, which enables you to loop through each configuration item in the same way that you would loop through a list or dictionary. This makes it easy to quickly format and print desired fields.

The following Table definition extracts the **system/login/user** items from the configuration data:

```
UserTable:
  get: system/login/user
  view: UserView

UserView:
  fields:
    username: name
    userclass: class
```

The Junos PyEZ code presented here loops through the **user** items and prints the name and class of each user:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

dev = Device(host='router.example.com').open()
users = UserTable(dev).get()

for user in users:
    print("Username is {}\nUser class is {}".format(user.username,
    user.userclass))

dev.close()
```

The **username** and **userclass** fields, which are defined in **UserView**, correspond to the values of the **name** and **class** elements, respectively, in the configuration data. The output includes the user's name and class.

```
Username is user1
User class is super-user
Username is readonly
User class is read-only
```

Although **UserView** defines a **username** field that maps to the **name** element, by default, each View item has a **name** property that references the key that uniquely identifies that item. Thus, you could also use **user.name** in this example to reference the value of the **name** element.

### Related Documentation

- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)

## Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources

Starting in Junos PyEZ Release 2.0, Junos PyEZ enables you to use Tables and Views to configure devices running Junos OS. Tables and Views are defined using simple YAML files that contain key and value pair mappings, so no complex coding is required to create them. You can create Tables and Views that define structured configuration resources. When you add the Table to the Junos PyEZ framework, Junos PyEZ dynamically creates a configuration class for the resource, which you can use to programmatically configure the resource on a device.

To configure devices running Junos OS using configuration Tables and Views, you must identify the resource to model, create the Table and View definitions for that resource, and then use those definitions to configure the resource in your Junos PyEZ application. The general steps are outlined in the following sections:

1. [Creating the Structured Resource on page 159](#)
2. [Using the Resource in a Junos PyEZ Application on page 160](#)

### Creating the Structured Resource

To create the structured resource:

1. Identify the Junos OS configuration for which you want to define a structured resource; for example, a **user** object at the **[edit system login]** hierarchy level.

```
user@host> show configuration system login | display xml
<rpc-reply>
  <configuration>
    <system>
      <login>
        <user>
          <name>jsmith</name>
          <uid>555</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>$ABC123</encrypted-password>
          </authentication>
        </user>
      </login>
    </system>
  </configuration>
  ...
</rpc-reply>
```

2. Create the Table and View definitions for the structured resource.

For detailed information about creating configuration Tables and Views, see “[Defining Junos PyEZ Configuration Tables](#)” on page 141 and “[Defining Junos PyEZ Views for Configuration Tables](#)” on page 146.

```
UserConfigTable:
  set: system/login/user
  key-field:
    username
```

```
view: UserConfigView

UserConfigView:
  groups:
    auth: authentication
  fields:
    username: name
    userclass: { class : { 'default' : 'unauthorized' }}
    uid: { uid: { 'type': 'int', 'default':1001, 'minValue':100,
'maxValue':64000 }}
    fullname: full-name
    fields_auth:
      password: encrypted-password
```

3. Add the structured resource to the Junos PyEZ framework either as an inline string or as an external file, as discussed in the following topics:

- [Loading Inline Junos PyEZ Tables and Views on page 123](#)
- [Importing External Junos PyEZ Tables and Views on page 125](#)

## Using the Resource in a Junos PyEZ Application

To configure the resource in your Junos PyEZ application:

1. Create a **Device** object and connect to the device. For example:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable

dev = Device(host='router.example.com').open()
```

2. Create a Table object and associate it with the device.

```
uc = UserConfigTable(dev)
```

3. Configure the resource by defining values for the necessary fields, including all key fields that are defined in the Table's **key-field** property.

For detailed information about configuring the resource, see [“Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS” on page 161](#).

```
uc.username = 'user1'
uc.userclass = 'operator'
uc.password = '$ABC123'
```

4. Call the **append()** method to build the Junos XML configuration that contains the configuration changes.

```
uc.append()
```



**NOTE:** After you call `append()`, the value for each field is reset to its default value or to `None`, if the View does not define a default. If you configure another resource, the initial values for that resource are the reset values rather than the values that were configured for the previous resource.

5. Repeat Step 3 and Step 4 for each additional resource to configure.
6. Load and commit the configuration changes to the shared configuration database on the device by using one of the following approaches:
  - Call the `set()` method, which automatically calls the `lock()`, `load()`, `commit()`, and `unlock()` methods.

```
uc.set(merge=True, comment="Junos PyEZ commit")
```

- Call the individual `lock()`, `load()`, `commit()`, and `unlock()` methods.

```
uc.lock()
uc.load(merge=True)
uc.commit(comment="Junos PyEZ commit")
uc.unlock()
```

For more information about the using the different methods to load and commit the configuration data, see “Using Junos PyEZ to Configure Devices Running Junos OS” on page 49 and “Using Junos PyEZ to Commit the Candidate Configuration” on page 68.

#### Related Documentation

- [Defining Junos PyEZ Configuration Tables on page 141](#)
- [Defining Junos PyEZ Views for Configuration Tables on page 146](#)
- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS on page 161](#)

## Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS

Junos PyEZ configuration Tables and Views that specify the `set` property enable you to define structured resources that can be used to programmatically configure devices running Junos OS. After loading or importing the Table definition for your structured resource into your Junos PyEZ application, the application can configure the resource on your devices.

The following sections discuss the general process and some specific tasks for using Junos PyEZ configuration Tables and Views to configure structured resources on a device:

- [General Configuration Process on page 162](#)
- [Configuring Statements Consisting of a Fixed-Form Keyword on page 164](#)

- [Configuring Multiple Values for the Same Statement on page 165](#)
- [Configuring Multiple Instances of the Same Statement on page 166](#)
- [Configuring Multiple Instances of the Same Resource on page 168](#)
- [Deleting Containers or Leaf Statements on page 169](#)
- [Using `append\(\)` to Generate the Junos XML Configuration Data on page 170](#)
- [Viewing Your Configuration Changes on page 171](#)
- [Controlling the RPC Timeout Interval on page 172](#)

## General Configuration Process

The configuration Table **set** property identifies the configuration hierarchy level at which a resource is configured and sets the XPath context for fields in the View. For example, the following Table defines a **user** resource at the **[edit system login]** hierarchy level:

```
UserConfigTable:
  set: system/login/user
  key-field:
    username
  view: UserConfigView

UserConfigView:
  groups:
    auth: authentication
  fields:
    username: name
    userclass: { class : { 'default' : 'unauthorized' }}
    uid: { uid: { 'type': 'int', 'default':1001, 'minValue':100,
'maxValue':64000 }}
    fullname: full-name
  fields_auth:
    password: encrypted-password
```

The fields that are included in the View define which leaf statements the user can configure for that resource. A field can define a default value as well as type and constraint checks.

To configure a structured resource on a device, you must load or import the Table into your application. You then create a Table object and associate it with the **Device** object representing the target device. For example:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree

dev = Device(host='router.example.com').open()
uc = UserConfigTable(dev)
```

To define values for a resource's configuration statements, set the corresponding field names as defined in the View equal to the desired values.

```
tableobject.fieldname = value
```

The default value for a field is **None** unless the View explicitly defines a default for that field. If the View defines a type or constraint check for a field, the application must supply the correct data type and value for that field and ideally handle any errors that might be

raised in the event that the check fails. You must always define values for any key fields that are declared in the Table's **key-field** property.

The following code imports **UserConfigTable** and configures the **username**, **userclass**, and **password** fields:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree

dev = Device(host='router.example.com').open()
uc = UserConfigTable(dev)

uc.username = 'user1'
uc.userclass = 'operator'
uc.password = '$ABC123'
```

For detailed information about more specific configuration tasks, such as configuring statements with fixed-form keywords or multiple values, configuring multiple instances of a statement or resource, or deleting a leaf or container statement, see the following sections:

- [Configuring Statements Consisting of a Fixed-Form Keyword on page 164](#)
- [Configuring Multiple Values for the Same Statement on page 165](#)
- [Configuring Multiple Instances of the Same Resource on page 168](#)
- [Configuring Multiple Instances of the Same Statement on page 166](#)
- [Deleting Containers or Leaf Statements on page 169](#)

After you configure an object, you must call the **append()** method to build the corresponding Junos XML configuration and add it to the **lxml** object that stores the master set of configuration changes for that Table object.

```
uc.append()
```

After building the XML, the **append()** method also resets all fields to the their default values or to **None** if the View does not define a default for that field. This enables you to configure multiple objects in the same application and ensures that you do not unintentionally use a value defined for one resource when you configure subsequent resources. Each time you configure a new resource, you must call **append()** to add the configuration changes to the master set of changes. For more information about the **append()** method, see ["Using append\(\) to Generate the Junos XML Configuration Data" on page 170](#).

If necessary, you can also manually reset all fields for a Table object by calling the **reset()** method.

```
uc.reset()
```

The **reset()** method restores all fields back to their default values or to **None** if the View does not define a default. The **reset()** method only resets the current values of the fields. It does not affect the XML containing the configuration changes that has been constructed up to that point by any calls to the **append()** method.

You can retrieve the XML configuration representing your changes at any point in the application by calling the `get_table_xml()` method, which is discussed in detail in [“Viewing Your Configuration Changes” on page 171](#).

```
configXML = uc.get_table_xml()
if (configXML is not None):
    print (etree.tostring(configXML, encoding='unicode', pretty_print=True))
```

After configuring all necessary objects and calling `append()`, you can load your configuration changes into the shared configuration database on the device by using one of two methods:

- Call the `set()` method, which automatically calls the `lock()`, `load()`, `commit()`, and `unlock()` methods
- Call the `lock()`, `load()`, `commit()`, and `unlock()` methods individually

Using the single `set()` method provides simplicity, but calling the individual methods provides additional flexibility such as when you need to call other methods after loading the configuration data, but before committing it. For example, you might want to call the `diff()` or `pdiff()` methods to review the configuration differences after you load the data but before you commit it. Or you might need to call the `rollback()` method to reset the candidate configuration back to the active configuration instead of committing it. For more information about using the different methods to load and commit the configuration data, see [“Using Junos PyEZ to Configure Devices Running Junos OS” on page 49](#) and [“Using Junos PyEZ to Commit the Candidate Configuration” on page 68](#).

In the case of large load and commit operations that might time out, you can adjust the RPC timeout interval by including the `timeout` parameter in the `set()` or `commit()` method argument list. For more information, see [“Controlling the RPC Timeout Interval” on page 172](#).

A configuration table that specifies the `set` parameter is a superset and has all the features of a configuration table that specifies the `get` parameter. You can retrieve configuration data in the same way in your Junos PyEZ application whether the Table specifies `set` or `get`. For information about using configuration Tables to retrieve configuration data, see [“Using Junos PyEZ Configuration Tables to Retrieve Configuration Data” on page 154](#).

## Configuring Statements Consisting of a Fixed-Form Keyword

A *leaf statement* is a CLI configuration statement that does not contain any other statements. Most leaf statements define a value for one characteristic of a configuration object and have the following form:

**keyword value;**

Some leaf statements consist of a fixed-form keyword only, without an associated variable-form value. For example, the `ftp` statement at the `[edit system services]` hierarchy level is an example of a fixed-form keyword.

```
system {
  services {
    ftp;
  }
}
```

```
}
```

The Junos XML API represents such statements with an empty tag.

```
<configuration>
  <system>
    <services>
      <ftp>
      </ftp>
      ...
    </services>
  </system>
</configuration>
```

To configure a fixed-form keyword in your Junos PyEZ application, such as the **ftp** statement under **[edit system services]**, set the value of the corresponding field name as defined in the View equal to the Boolean value **True**.

Consider the following View, which defines the **ftp** field with a type constraint to ensure that the value for the field is a Boolean:

```
ServicesView:
  fields:
    ftp: { 'ftp' : { 'type': 'bool' } }
    ...
```

To configure the **ftp** field in your Junos PyEZ application, set the field equal to **True**.

```
from jnpr.junos import Device
from myTables.ConfigTables import ServicesConfigTable

dev = Device(host='router.example.com').open()
sc = ServicesConfigTable(dev)

sc.ftp = True
sc.append()
sc.set()
```

## Configuring Multiple Values for the Same Statement

Some Junos OS leaf statements accept multiple values, which might be either user-defined or drawn from a set of predefined values. CLI notation uses square brackets to enclose all values in a single statement, as in the following:

```
keyword [ value1 value2 value3 ...];
```

For example, you might need to configure a VLAN ID list for a trunk interface, as in the following configuration:

```
interfaces {
  ge-0/0/1 {
    native-vlan-id 510;
    unit 0 {
      family bridge {
        interface-mode trunk;
        vlan-id-list [ 510 520 530 ];
      }
    }
  }
}
```

```

    }
  }
}

```

To configure a leaf statement with multiple values in your Junos PyEZ application, set the value of the corresponding field, as defined in the View, equal to a Python list containing the desired values. In the following example, the **vlan-list** field maps to the **vlan-id-list** statement in the CLI. To configure the statement with multiple VLAN IDs, set the field name equal to the list of IDs.

```

from jnpr.junos import Device
from myTables.ConfigTables import InterfacesConfigTable

dev = Device(host='router.example.com').open()
intf = InterfacesConfigTable(dev)

intf.name = 'ge-0/0/1'
intf.mode = 'trunk'
intf.native_vlan = 510
intf.vlan_list = [510, 520, 530]

```



**NOTE:** The Python list that you use for the value of a field in your Junos PyEZ application is a comma-delimited list of values. This is different from the space-delimited list that you would configure in the CLI.

## Configuring Multiple Instances of the Same Statement

In certain situations, the Junos OS configuration enables you to configure multiple instances of the same statement. For example, you might configure multiple addresses under the same protocol family for a logical interface. In the following configuration snippet, the loopback interface has multiple addresses configured at the **[edit interfaces lo0 unit 0 family inet]** hierarchy level:

```

interfaces {
  lo0 {
    unit 0 {
      family inet {
        address 192.168.100.1/32;
        address 192.168.100.2/32;
      }
    }
  }
}

```

The Junos XML representation of the configuration is as follows:

```

<configuration>
  <interfaces>
    <interface>
      <name>lo0</name>
      <unit>
        <name>0</name>
        <family>
          <inet>
            <address>
              <name>192.168.100.1/32</name>

```

```

        </address>
        <address>
            <name>192.168.100.2/32</name>
        </address>
    </inet>
</family>
</unit>
</interface>
</interfaces>
</configuration>

```

When you use Junos PyEZ configuration Tables to manage structured resources on devices running Junos OS, you define values for configuration statements by setting the corresponding field names equal to the desired values. However, you cannot define the same field twice in your Junos PyEZ application, because the second value will overwrite the first value. Instead, you must set the field equal to a list of values, and Junos PyEZ handles the conversion to XML.

Consider the following Table and View:

```

InterfaceTable:
  set: interfaces/interface
  key-field:
    - name
    - unit_name
  view: InterfaceView

InterfaceView:
  fields:
    name: name
    desc: description
    unit_name: unit/name
    ip_address: unit/family/inet/address

```

The following sample code illustrates how to configure multiple addresses for the loopback interface on a device running Junos OS. In this case, you set the `ip_address` field equal to a list of addresses.

```

lo0_addresses = ['192.168.100.1/32', '192.168.100.2/32']

...
intf = InterfaceTable(dev)

intf.name='lo0'
intf.unit_name = 0
intf.ip_address = lo0_addresses
intf.append()
intf.set()
...

```

The resulting configuration is:

```

[edit interfaces]
+ lo0 {
+   unit 0 {
+     family inet {
+       address 192.168.100.1/32;
+       address 192.168.100.2/32;
+     }
+   }
+ }

```

```
+     }  
+ }
```

## Configuring Multiple Instances of the Same Resource

When you use Junos PyEZ configuration Tables to configure structured resources on devices running Junos OS, you might need to configure multiple objects, or records, for the same structured resource. For example, you might configure multiple interfaces or users at the same time. To configure multiple objects for the same structured resource in a Junos PyEZ application, you must define the values for one object's fields, call the **append()** method, and then repeat this process for each subsequent object.

For example, to configure multiple users, define the field values for the first user, and call the **append()** method. Then define the field values for the second user and call the **append()** method. The **append()** method builds the Junos XML data for the configuration change and adds it to the **xml** object storing the master set of configuration changes. The method also automatically resets all of the fields back to their default values, as defined in the View, or to **None** if a field does not have a defined default.

The following sample code configures two user objects:

```
uc = UserConfigTable(dev)  
  
uc.username = 'user1'  
uc.userclass = 'operator'  
uc.uid = 1005  
uc.password = '$ABC123'  
uc.append()  
  
uc.username = 'user2'  
uc.userclass = 'operator'  
uc.uid = 1006  
uc.password = '$ABC456'  
uc.append()  
  
uc.set()
```



**NOTE:** If you do not call the **append()** method after configuring one of multiple objects for the same resource, the field values for the second object will overwrite the field values for the first object.

---

The following sample code configures the same two users using a more compact syntax:

```
users = ['user1', 'user2']
uids = ['1005', '1006']
passwd = ['$ABC123', '$ABC456']

uc = UserConfigTable(dev)
for user, uid, passwd in zip(users, uids, passwd):
    uc.username = user
    uc.userclass = 'operator'
    uc.uid = uid
    uc.password = passwd
    uc.append()

uc.set()
```

## Deleting Containers or Leaf Statements

In some cases, you might need to delete container or leaf statements in the configuration. When you use Junos PyEZ configuration Tables to manage structured resources, you can perform this operation in your application by setting the appropriate field value to **{'operation': 'delete'}**. You must always define values for all key fields when deleting a container or leaf statement to indicate to which object the deletion applies.

Consider the following Junos PyEZ configuration Table and View:

```
---
UserConfigTable2:
  set: system/login
  key-field:
    - username
  view: UserConfigView2

UserConfigView2:
  groups:
    auth: authentication
  fields:
    user: user
    username: user/name
    classname: { user/class : { 'type' : { 'enum' : ['operator', 'read-only',
'super-user'] } } }
    uid: { user/uid : { 'type' : 'int', 'minValue' : 100, 'maxValue' : 64000
} }
  fields_auth:
    password: user/encrypted-password
```

To delete a leaf statement for the resource defined in the Table and View, set the value of the field corresponding to that statement to **{'operation': 'delete'}**. The following example deletes the **uid** statement for user **jsmith**:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable2

dev = Device(host='router.example.com').open()
uc = UserConfigTable2(dev)
uc.username = 'jsmith'
uc.uid = { 'operation' : 'delete' }
uc.append()
uc.set()
```

To delete a container from the configuration, the View must define a field for that container. In the example Table and View, the configuration scope defined by the **set** property is **system/login**. The View defines the field **'user'**, which maps to the **system/login/user** container. This definition enables you to delete user objects, if necessary. If you do not define a field for the container, you can only delete statements within the container, but you cannot delete the container itself.

To delete a container in the Junos PyEZ application, set the value of the field corresponding to the container to **{'operation': 'delete'}**, and define the key field to indicate the object to delete. The following example deletes the user **jsmith** from the configuration:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable2
from lxml import etree

dev = Device(host='router.example.com').open()
uc = UserConfigTable2(dev)

uc.user = { 'operation' : 'delete' }
uc.username = 'jsmith'
uc.append()
print (etree.tostring(uc.get_table_xml(), encoding='unicode',
pretty_print=True))
uc.set()
```

The application prints the Junos XML configuration data returned by the **get\_table\_xml()** method. The user element with identifier 'jsmith' includes the **operation="delete"** attribute to instruct Junos OS to remove that object from the configuration.

```
<configuration>
  <system>
    <login>
      <user operation="delete">
        <name>jsmith</name>
      </user>
    </login>
  </system>
</configuration>
```

## Using **append()** to Generate the Junos XML Configuration Data

When you use Junos PyEZ configuration Tables to configure structured resources on devices running Junos OS, you define the values for a resource's fields and then call the **append()** method. Each call to the **append()** method generates the Junos XML configuration data for the current set of changes and adds it to the **lxml** object that stores the master set of configuration changes.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable

dev = Device(host='router.example.com').open()
uc = UserConfigTable(dev)
uc.username = 'user1'
uc.userclass = 'operator'
uc.password = '$ABC123'

uc.append()
...
```

Calling the **append()** method generates the Junos XML configuration data for your resource. The configuration changes only include those fields that have either a default value defined in the View or a user-configured value. Fields that retain their initial value of **None** are ignored.

After building the XML, the **append()** method also resets all fields back to their default values, as defined in the View, or to **None** if a field does not have a defined default. Resetting the fields ensures that when you configure multiple objects in the same application, you do not set a field value for one object and then unintentionally use that value in subsequent calls to **append()** for a different object. Thus, you must define new values for all **key-field** fields for each call to **append()**.



**NOTE:** Once you append nodes to the master set of configuration changes, you cannot undo the operation.

The **append()** method only adds the new changes to the **lxml** object containing the master set of configuration changes. You must explicitly call the **set()** method or the **load()** and **commit()** methods to load and commit the changes on the device.

## Viewing Your Configuration Changes

When you use Junos PyEZ configuration Tables to configure structured resources on devices running Junos OS, you define the values for a resource's fields and then call the **append()** method. Each call to the **append()** method generates the Junos XML configuration data for the current set of changes and adds it to the **lxml** object that stores the master set of configuration changes. At times, you might need to review the configuration data that has been constructed up to a certain point in the application, or you might want to view the differences between the candidate and active configurations after you load your configuration changes onto the device.

To retrieve the Junos XML configuration data containing your changes, call the Table object's **get\_table\_xml()** method. The **get\_table\_xml()** method returns the XML configuration that has been constructed up to that point in the application. When you call the **set()** method or the **load()** and **commit()** methods, the application loads and commits this Junos XML configuration data on the device.

The following example calls the **get\_table\_xml()** method to retrieve the configuration changes and then stores them in the **configXML** variable. Prior to calling the **append()** method, the **get\_table\_xml()** method returns **None**. Thus, the application only serializes and prints the XML configuration data if the returned value is not **None**.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree

dev = Device(host='router.example.com').open()
uc = UserConfigTable(dev)
uc.username = 'user1'
uc.userclass = 'operator'
uc.password = '$ABC123'
uc.append()
```

```
configXML = uc.get_table_xml()
if (configXML is not None):
    print (etree.tostring(configXML, encoding='unicode', pretty_print=True))
else:
    print (configXML)

uc.set()
```

The `get_table_xml()` method only returns the Junos XML data for your configuration changes. You might also want to compare the candidate and active configurations after loading the configuration changes onto the device to review the differences before you commit the changes.

To retrieve the differences, you can call the `lock()`, `load()`, `commit()`, and `unlock()` methods separately and view your configuration differences by calling the `pdiff()` method after you load the data but before you commit it. The `pdiff()` method with an empty argument list compares the candidate configuration to the active configuration and prints the difference in patch format directly to standard output.

```
...
uc.append()
uc.lock()
uc.load()
uc.pdiff()
...
uc.commit()
uc.unlock()
```

## Controlling the RPC Timeout Interval

When you use Junos PyEZ configuration Tables to configure structured resources on devices running Junos OS, you can load and commit your configuration changes by calling the `set()` method or the `load()` and `commit()` methods. The `set()` and `commit()` methods use the RPC timeout value as defined in the `device` module. If you do not configure a new value for the **Device timeout** property, Junos PyEZ uses the default value of 30 seconds.

Large configuration changes might exceed the default or configured timeout value, causing the operation to time out before the configuration can be uploaded and committed on the device. To accommodate certain configuration changes that might require load and commit times that are longer than the default or configured timeout interval, set the **timeout=seconds** argument to an appropriate value when you call the `set()` or `commit()` method in your application. For example:

```
uc = UserConfigTable(dev)

uc.username = 'user1'
uc.userclass = 'operator'
uc.uid = 1005
uc.password = '$ABC123'
uc.append()

uc.set(timeout=300)
```

- Related Documentation**
- [Defining Junos PyEZ Configuration Tables on page 141](#)
  - [Defining Junos PyEZ Views for Configuration Tables on page 146](#)

- [Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources on page 159](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)

## Saving and Loading Junos PyEZ Table XML to and from Files

Junos PyEZ Tables and Views enable you to extract targeted data from operational command output or the selected configuration database on a device running Junos OS. You can export Table data as XML, which enables you to retrieve information for one or more devices and process it at a later time. Junos PyEZ provides the `savexml()` method for this purpose.

The `savexml()` method enables you to specify a destination file path for the exported data, and optionally include the device hostname and activity timestamp in the filename. You can control the format of the timestamp using the standard `strftime` format.

For example, suppose that you want to loop through a list of devices and collect transceiver data using the `XcvrTable` definition in the `jnpr.junos.op.xcvr` module. The following code defines a list of device hostnames, prompts the user for a username and password, and then loops through and makes a connection to each device:

```
import sys
from getpass import getpass
from junpr.junos import Device
from junpr.junos.op.xcvr import XcvrTable

devlist = ['router1.example.com', 'router2.example.com']
user = raw_input('username: ')
passwd = getpass('password: ')

for host in devlist:
    sys.stdout.write('connecting to %s ... ' % host)
    sys.stdout.flush()

    dev = Device(host, user=user, password=passwd)
    dev.open()
    print('ok.')

    # log data

    dev.close()
```

At this point, the program does not yet retrieve any transceiver data. Running the program results in the following output:

```
[user1@server]$ python xcvr_demo.py
username: user1
password:
connecting to router1.example.com ... ok.
connecting to router2.example.com ... ok.
```

To collect and log the transceiver data, you associate the Table with each target device, retrieve the data, and save it to a file using the `savexml()` method. You can include `hostname=True` and `timestamp=True` in the `savexml()` argument list to append the hostname and timestamp to the output filename. If you retrieve data for multiple devices in this manner, you must differentiate the output filename for each device with the hostname, timestamp, or both to prevent the data for one device from overwriting the data for the previous device in the same file.

```
# log data
xcvrs = XcvrTable(dev).get()
xcvrs.savexml(path='/var/tmp/xcvrs/xcvr.xml', hostname=True, timestamp=True)
```



**NOTE:** The path argument assumes that the target directory exists on your local file system.

---

After adding the additional code to the device loop in the program and then executing the program, you can examine the contents of the target directory. In this example, the hostname and timestamp values are embedded in the filenames.

```
[user1@server]$ ls /var/tmp/xcvrs
xcvr_router1.example.com_20131226093921.xml
xcvr_router2.example.com_20131226093939.xml
```

You can import the XML data back into a Table widget at a later time for post processing. To import the data, associate the Table with the XML file instead of a target device. For example:

```
xmlpath = '/var/tmp/xcvrs/xcvr_router1.example.com_20131226093921.xml'
xcvrs = XcvrTable(path=xmlpath)
xcvrs.get()
```

**Related  
Documentation**

- [Using Junos PyEZ Operational Tables and Views on page 137](#)
- [Using Junos PyEZ Configuration Tables to Retrieve Configuration Data on page 154](#)

# Troubleshooting Junos PyEZ

- [Troubleshooting jnpr.junos Import Errors on page 175](#)
- [Troubleshooting Junos PyEZ Connection Errors on page 176](#)
- [Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS on page 177](#)
- [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)

## Troubleshooting jnpr.junos Import Errors

---

**Problem**    **Description:**

Python generates an error message that the **jnpr.junos** module was not found. For example:

```
from jnpr.junos import Device
ImportError: No module named junos
```

**Cause**    The Juniper Networks Junos PyEZ Python library must be installed before importing the package and using it to perform operations on devices running Junos OS.

**Solution**    Install Junos PyEZ on the configuration management server and update any necessary environment variables. For installation instructions, see [“Installing Junos PyEZ” on page 21](#).

To verify that Junos PyEZ is successfully installed, start Python and import the **jnpr.junos** package.

```
[user@localhost ~]$ python
>>>
>>> import jnpr.junos
>>> jnpr.junos.__version__
'1.0.0'
```

If the **jnpr.junos** package is successfully imported and there is no error message, then Junos PyEZ is correctly installed.

- Related Documentation**
- [Installing Junos PyEZ on page 21](#)
  - [Junos PyEZ Modules Overview on page 19](#)
  - [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

---

## Troubleshooting Junos PyEZ Connection Errors

---

**Problem Description:**

When using Junos PyEZ to manage devices running Junos OS, the code generates an error that the connection was refused. For example:

```
jnpr.junos.exception.ConnectRefusedError
```

**Cause** NETCONF is not enabled on the device or the number of connections exceeds the limit.

The most likely cause for a refused connection error is that NETCONF over SSH is not enabled on the device running Junos OS. To quickly test whether NETCONF is enabled, verify that the user account can successfully start a NETCONF session with the device.

```
[user@server]$ ssh user@R1.example.com -p 830 -s netconf
```

**Solution** If NETCONF is not enabled on the device running Junos OS, enable NETCONF.

```
[edit]
user@R1# set system services netconf ssh
user@R1# commit
```

If the number of NETCONF sessions exceeds the limit, increase the maximum number of permitted sessions up to 250. The default is 75.

```
[edit]
user@R1# set system services netconf ssh connection-limit limit
user@R1# commit
```

- Related Documentation**
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)
  - [Troubleshooting jnpr.junos Import Errors on page 175](#)
  - [Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS on page 177](#)
  - [Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS on page 178](#)

## Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS

### Problem Description:

Junos PyEZ generates an error regarding failed authentication. For example:

```
unable to connect to dc1a.example.com: ConnectAuthError(dc1a.example.com)
```

### Cause The device running Junos OS might fail to authenticate the user for the following reasons:

- The user does not have an account on the device running Junos OS.
- The user has an account with a text-based password configured on the device running Junos OS, but the wrong password or no password is supplied for the user when creating the **Device** instance.
- The user has an account on the device running Junos OS with SSH keys configured, but the SSH keys are inaccessible on either the device or the configuration management server.



**NOTE:** If you do not specify a user when creating the Device instance, the user defaults to \$USER.

**Solution** Ensure that the user executing the Junos PyEZ code has a login account on all target devices running Junos OS and that an SSH public/private key pair or text-based password is configured for the account. If SSH keys are configured, verify that the user can access them. Also, confirm that the correct parameters are supplied when creating the **Device** instance.

### Related Documentation

- [Authenticating Junos PyEZ Users Using a Password on page 27](#)
- [Authenticating Junos PyEZ Users Using SSH Keys on page 29](#)
- [Using Junos PyEZ to Connect to and Retrieve Facts from Devices Running Junos OS on page 43](#)

## Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS

The following sections outline errors that you might encounter when using Junos PyEZ to configure devices running Junos OS. These sections also present potential causes and solutions for each error.

- [Troubleshooting Timeout Errors on page 178](#)
- [Troubleshooting Configuration Lock Errors on page 178](#)
- [Troubleshooting Configuration Change Errors on page 179](#)

### Troubleshooting Timeout Errors

Problem	Description:
---------	--------------

	The Junos PyEZ code generates an <code>RpcTimeoutError</code> message or a <code>TimeoutExpiredError</code> message and fails to update the device configuration.
--	---

	<code>RpcTimeoutError(host: dc1a.example.com, cmd: commit-configuration, timeout: 30)</code>
--	--

Cause	The default time for a NETCONF RPC to time out is 30 seconds. Large configuration changes might exceed this value causing the operation to time out before the configuration can be uploaded and committed.
-------	---

Solution	To accommodate configuration changes that might require a commit time that is longer than the default timeout interval, set the timeout interval to an appropriate value and rerun the code. To configure the interval, either set the <b>Device timeout</b> property to an appropriate value, or include the <b>timeout=seconds</b> argument in the <b>commit()</b> or <b>set()</b> method when you load and commit configuration data on a device. For example:
----------	---

```
dev = Device(host="host")
dev.open()
dev.timeout = 300

cu = Config(dev)
cu.lock()
cu.load(path="junos-config.conf", merge=True)
cu.commit(timeout=360)
cu.unlock()
```

### Troubleshooting Configuration Lock Errors

Problem	Description:
---------	--------------

	The Junos PyEZ code generates a <code>LockError</code> message indicating that the configuration cannot be locked.
--	--

	<code>LockError(severity: error, bad_element: None, message: configuration database modified)</code>
--	--

**Cause** A configuration lock error can occur for the following reasons:

- Another user has an exclusive lock on the configuration.
- Another user made changes to the shared configuration database but has not yet committed the changes.
- The user executing the Junos PyEZ code does not have permissions to configure the device.

**Solution** If another user has an exclusive lock on the configuration or has modified the configuration, wait until the lock is released or the changes are committed, and execute the code again. If the cause of the issue is that the user does not have permissions to configure the device, either execute the program with a user who has the necessary permissions, or if appropriate, configure the device running Junos OS to give the current user the necessary permissions to make the changes.

## Troubleshooting Configuration Change Errors

**Problem** Description:

The Junos PyEZ code generates a `ConfigLoadError` message indicating that the configuration cannot be modified due to a permissions issue.

```
ConfigLoadError(severity: error, bad_element: scripts, message: permission denied)
```

**Cause** This error message might be generated when the user executing the Junos PyEZ code has permission to alter the configuration, but does not have permission to alter the desired portion of the configuration.

**Solution** Either execute the program with a user who has the necessary permissions, or if appropriate, configure the device running Junos OS to give the current user the necessary permissions to make the changes.

**Related Documentation**

- [Using Junos PyEZ to Configure Devices Running Junos OS on page 49](#)
- [Using the Junos PyEZ Config Utility to Load Configuration Data on page 53](#)
- [Example: Using Junos PyEZ to Load Configuration Data from a File on page 60](#)

