

Junos® OS

Junos PyEZ Developer Guide

Published
2021-06-03

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the 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

Copyright © 2021 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 <https://support.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 This Guide | xi

1

Disclaimer

Junos PyEZ Disclaimer | 2

2

Junos PyEZ Overview

Understanding Junos PyEZ | 4

Junos PyEZ Modules Overview | 8

3

Installing Junos PyEZ

Installing Junos PyEZ | 12

Installing Prerequisite Software | 13

Installing Junos PyEZ on the Configuration Management Server | 15

Installing Junos PyEZ in a Python Virtual Environment | 16

Using the Junos PyEZ Docker Image | 18

Setting Up Junos PyEZ Managed Nodes | 19

Enabling NETCONF over SSH on Devices Running Junos OS | 20

Satisfying Requirements for SSHv2 Connections | 21

Configuring Telnet Service on Devices Running Junos OS | 21

4

Connecting to and Retrieving Facts From a Device Using Junos PyEZ

Connecting to Devices Running Junos OS Using Junos PyEZ | 24

Connection Methods Overview | 24

Understanding Junos PyEZ Connection Properties | 26

Connecting to a Device Using SSH | 28

Connecting to a Device Using Outbound SSH | 32

Connecting to a Device Using Telnet | 35

Connecting to a Device Using a Serial Console Connection | 38

Authenticating Junos PyEZ Users | 41

Junos PyEZ User Authentication Overview | 41

Authenticating Junos PyEZ Users Using a Password | 42

Authenticating Junos PyEZ Users Using SSH Keys | 45

Generating and Configuring SSH Keys | 45

Referencing SSH Keys in Junos PyEZ Applications | 46

Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50

Understanding Junos PyEZ Device Facts | 50

Example: Retrieving Facts from Devices Running Junos OS | 52

Accessing the Shell on Devices Running Junos OS Using Junos PyEZ | 56

5

Using Junos PyEZ to Manage Device Operations

Using Junos PyEZ to Execute RPCs on Devices Running Junos OS | 61

Mapping Junos OS Commands to Junos PyEZ RPCs | 61

Executing RPCs as a Property of the Device Instance | 63

Specifying the Format of the RPC Output | 64

Specifying the Scope of Data to Return | 66

Specifying the RPC Timeout | 67

Normalizing the XML RPC Reply | 68

Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications | 71

Using Junos PyEZ to Reboot or Shut Down Devices Running Junos OS | 73

Performing a Reboot or Shut Down | 73

Performing a Reboot or Shut Down with a Delay or at a Specified Time | 75

Specifying the Target for the Reboot or Shutdown | 76

Rebooting a VM Host | 77

Using Junos PyEZ to Install Software on Devices Running Junos OS | 78

Supported Deployment Scenarios | 79

Specifying the Software Image Location | 79

Installation Process Overview | 81

Specifying Installation and Checksum Timeouts | 82

Logging the Installation Process | 83

Performing a VM Host Upgrade | 85

Performing a Unified ISSU or NSSU | 86

Example: Using Junos PyEZ to Install Software on Devices Running Junos OS | 87

Requirements | 88

Overview | 88

Configuration | 89

Executing the Junos PyEZ Code | 94

Verification | 94

Troubleshooting | 97

Transferring Files Using Junos PyEZ | 98

Specifying the XML Parser for a Junos PyEZ Session | 102

6

Using Junos PyEZ to Manage the Configuration

Using Junos PyEZ to Retrieve a Configuration | 105

Retrieving the Complete Candidate Configuration | 106

Specifying the Source Database for the Configuration Data | 106

Specifying the Scope of Configuration Data to Return | 108

Specifying the Format for Configuration Data to Return | 111

Retrieving Configuration Data for Standard or Custom YANG Data Models | 112

Specifying Additional RPC Options | 114

Handling Namespaces in Configuration Data | 115

Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration | 117

Using Junos PyEZ to Configure Devices Running Junos OS | 120

Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129

Configuration Process Overview | 130

Specifying the Configuration Mode	131
Specifying the Load Operation	132
Specifying the Format of the Configuration Data to Load	134
Specifying the Location of the Configuration Data	135
Loading Configuration Data from a Local or Remote File	137
Loading Configuration Data from a String	138
Loading Configuration Data Formatted as an XML Object	141
Loading Configuration Data Using Jinja2 Templates	141
Rolling Back the Configuration	144
Loading the Rescue Configuration	146
Committing the Configuration	147

Using Junos PyEZ to Commit the Configuration | 148

Committing the Candidate Configuration	148
Specifying Commit Options	150

Example: Using Junos PyEZ to Load Configuration Data from a File | 154

Requirements	154
Overview	155
Configuration	155
Executing the Junos PyEZ Code	161
Verification	161
Verifying the Configuration	162
Troubleshooting	163

Example: Using Junos PyEZ to Roll Back the Configuration | 166

Requirements	166
Overview	167
Configuration	167

Executing the Junos PyEZ Code | 171

Verification | 172

Verifying the Configuration | 172

Using Junos PyEZ to Manage the Rescue Configuration on Devices Running Junos OS | 174

Managing the Rescue Configuration | 175

Saving a Rescue Configuration | 175

Retrieving the Rescue Configuration | 175

Loading and Committing the Rescue Configuration | 176

Deleting the Rescue Configuration | 177

Example: Using Junos PyEZ to Save a Rescue Configuration | 177

Requirements | 177

Overview | 178

Configuration | 178

Executing the Junos PyEZ Code | 181

Verification | 182

Troubleshooting | 183

7

Creating and Using Junos PyEZ Tables and Views

Understanding Junos PyEZ Tables and Views | 187

Predefined Junos PyEZ Operational Tables (Structured Output) | 189

Loading Inline or External Tables and Views in Junos PyEZ Applications | 194

Importing Junos PyEZ's Predefined Tables and Views | 194

Loading Inline Tables and Views | 195

Importing External Tables and Views | 196

Using Tables and Views | 198

Defining Junos PyEZ Operational Tables for Parsing Structured Output | 200

Table Name | 202

RPC Command (rpc) | 202

RPC Default Arguments (args) | 203

RPC Optional Argument Key (args_key) | 204

Table Item (item) | 204

Table Item Key (key) | 205

Table View (view) | 209

Defining Views for Junos PyEZ Operational Tables that Parse Structured Output | 209

View Name | 211

Fields (fields) | 212

Groups (groups) and Field Groups (fields_) | 214

Using Junos PyEZ Operational Tables and Views that Parse Structured Output | 216

Retrieving Table Items | 216

Accessing Table Items | 218

Iterating Through a Table | 220

Defining Junos PyEZ Operational Tables for Parsing Unstructured Output | 221

Summary of Parameters in Op Tables for Parsing Unstructured Output | 221

Table Name | 223

Command | 223

Command Arguments (args) | 224

Target FPC (fpc) | 225

Table Item (item) | 226

Table Item Key (key) | 230

Selected Keys (key_items) | 232

Section Title (title) | 233

Field Delimiter (delimiter) | 234

Eval Expression (eval) | 235

Table View (view) | 237

Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output | 238

Summary of Parameters in Views for Parsing Unstructured Output | 238

- View Name | 239
- columns | 240
- Eval Expression (eval) | 243
- exists | 245
- fields | 246
- filters | 251
- regex | 253

Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output | 259

Defining Junos PyEZ Configuration Tables | 260

- Table Name | 263
- Configuration Scope (get or set) | 263
- Key Field (key-field) | 265
- Required Keys (required_keys) | 267
- Table View (view) | 268

Defining Views for Junos PyEZ Configuration Tables | 268

- View Name | 271
- Fields (fields) | 272
- Field Options ('get' Tables) | 274
- Field Options ('set' Tables) | 275
- Groups (groups) and Field Groups (fields_) | 277

Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279

- Retrieving Configuration Items | 280
- Specifying Inheritance and Group Options | 282
- Accessing Table Items | 284
- Iterating Through a Table | 285

Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources | 287

Creating the Structured Resource | 287

Using the Resource in a Junos PyEZ Application | 289

Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS | 290

General Configuration Process | 291

Configuring Statements Consisting of a Fixed-Form Keyword | 295

Configuring Multiple Values for the Same Statement | 296

Configuring Multiple Instances of the Same Statement | 298

Configuring Multiple Instances of the Same Resource | 300

Deleting Containers or Leaf Statements | 301

Configuring Properties Corresponding to Junos XML Attributes | 303

Using append() to Generate the Junos XML Configuration Data | 307

Viewing Your Configuration Changes | 308

Controlling the RPC Timeout Interval | 310

Saving and Loading Junos PyEZ Table XML to and from Files | 311

8

Troubleshooting Junos PyEZ

Troubleshooting jnpr.junos Import Errors | 315

Troubleshooting Junos PyEZ Connection Errors | 316

Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS | 318

Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320

Troubleshooting Timeout Errors | 320

Troubleshooting Configuration Lock Errors | 321

Troubleshooting Configuration Change Errors | 322

About This Guide

Use this guide to develop Python scripts that remotely automate and manage devices running Junos OS using the Juniper Networks Junos PyEZ Python library.

RELATED DOCUMENTATION

| *Junos PyEZ API documentation*

1

CHAPTER

Disclaimer

Junos PyEZ Disclaimer | 2

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.

2

CHAPTER

Junos PyEZ Overview

[Understanding Junos PyEZ | 4](#)

[Junos PyEZ Modules Overview | 8](#)

Understanding Junos PyEZ

IN THIS SECTION

- [Junos PyEZ Overview | 4](#)
- [Benefits of Junos PyEZ | 5](#)
- [Using Junos PyEZ in Automation Scripts | 6](#)
- [Junos PyEZ Resources | 6](#)

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 directly connect to a device using a serial console connection, telnet, or a NETCONF session over SSH. In addition, Junos PyEZ also supports connecting to the device through a telnet or SSH connection to a console server that is connected to the device's **CONSOLE** port. 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 or SSH through a console server that is directly connected to the device.

Junos PyEZ provides device, software, and file system utilities that enable you to perform common operational tasks on devices running Junos OS. You can use Junos PyEZ to:

- retrieve facts or operational information from a device
- execute remote procedure calls (RPC) available through the Junos XML API
- install or upgrade the Junos OS software
- reboot or shut down the device

- 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 configuration data
- compare configurations
- upload and commit configuration changes
- roll back the configuration
- manage the rescue configuration

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 Tables and Views enable you to both configure devices running Junos OS and extract specific operational information or configuration data from the devices. You define Tables and Views using simple YAML files that contain key:value pair mappings, so no complex coding is required to use them. You can use Tables and Views to retrieve the device configuration or the output for any Junos OS command or RPC 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.

Benefits of Junos PyEZ

- Provides an abstraction layer that enables Python programmers as well as non-programmers to easily manage and automate devices running Junos OS
- Increases operational efficiency by enabling operators to automate common tasks thereby reducing the manual configuration and management of devices
- Minimizes errors and risks by enabling structured configuration changes of targeted resources

Using Junos PyEZ in Automation Scripts

Junos OS includes the Python extensions package and the Junos PyEZ library in the software image for certain supported devices. These extensions enable you to create on-box Python scripts that use Junos PyEZ to execute RPCs and perform operational and configuration tasks on devices running Junos OS. Junos PyEZ is supported in commit, event, op, and SNMP scripts; Juniper Extension Toolkit (JET) scripts; and YANG action and translation scripts.

[Table 1 on page 6](#) summarizes the Junos PyEZ version that is available on supported devices running the given Junos OS release. For example, starting in Junos OS Release 17.4R1, an on-box Python script can leverage features in Junos PyEZ Release 2.1.4 and earlier releases.

Table 1: Junos PyEZ Version on Supported Devices Running Junos OS

Junos OS Release	Junos PyEZ Version
16.1R3 through 17.3	1.3.1
17.4R1 through 19.3	2.1.4
19.4R1 and later	2.2.0

For more information about creating onbox Python automation scripts, see *Understanding Python Automation Scripts for Devices Running Junos OS* in the *Junos OS Automation Scripting User Guide*.

Junos PyEZ Resources

Juniper Networks provides a number of Junos PyEZ resources, which are described in [Table 2 on page 6](#).

Table 2: Junos PyEZ Resources

Resource	Description	URL
API Reference	Detailed documentation for the Junos PyEZ modules.	http://junos-pyez.readthedocs.org/

Table 2: Junos PyEZ Resources (*Continued*)

Resource	Description	URL
Day One: Junos PyEZ Cookbook and script repository	Junos PyEZ network automation cookbook with a setup guide, a start-up sandbox, and a complete showcase of automation scripts that are available on GitHub.	https://www.juniper.net/documentation/en_US/day-one-books/DO_PyEZ_Cookbook.pdf https://github.com/Juniper/junosautomation/tree/master/pyez/PyEZ_Cookbook_2017
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.	https://www.juniper.net/documentation/product/en_US/junos-pyez
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.	https://github.com/Juniper/py-junos-eznc/
Google Groups forum	Forum that addresses questions and provides general support for Junos PyEZ.	http://groups.google.com/group/junos-python-ez
Sample scripts	Junos PyEZ sample scripts to get you started.	https://github.com/Juniper/junosautomation/tree/master/pyez
Stack Overflow forum	Forum that addresses questions and provides general support for Junos PyEZ.	https://stackoverflow.com/questions/tagged/pyez
Techwiki page	Juniper Networks J-Net community forum containing additional how-to articles and usage examples.	https://forums.juniper.net/t5/Automation-Scripting/Junos-PyEZ/ta-p/280496

RELATED DOCUMENTATION

[Junos PyEZ Modules Overview | 8](#)

[Installing Junos PyEZ | 12](#)

[Connecting to Devices Running Junos OS Using Junos PyEZ | 24](#)

[Using Junos PyEZ to Configure Devices Running Junos OS | 120](#)

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 3 on page 8](#) 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 Reference at <http://junos-pyez.readthedocs.org/>.

Table 3: Junos PyEZ Modules

jnpr.junos Modules	Description
device	Defines the Device class, which represents the device running Junos OS and enables you to connect to and retrieve facts from the device.
command	Includes predefined operational Tables and Views that can be used to filter unstructured output returned from CLI and vty commands and convert it to JSON.
exception	Defines exceptions encountered when accessing, configuring, and managing devices running Junos OS.
factory	Contains code pertaining to Tables and Views, including the loadyaml() function, which is used to load custom Tables and Views.
facts	A dictionary-like object of read-only facts about the device. These facts are accessed using the facts attribute of a Device object instance.

Table 3: Junos PyEZ Modules (*Continued*)

jnpr.junos Modules	Description
op	Includes predefined operational Tables and Views that can be used to filter structured output returned from RPCs.
resources	Includes predefined configuration Tables and Views representing specific configuration resources, which can be used to programmatically configure devices running Junos OS.
transport	Contains code used by the Device class to support the different connection types.
utils	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 SSH and also supports connecting to the device through a telnet or SSH connection to a console server that is connected to the device's **CONSOLE** port. All connection methods support retrieving device facts, performing operations, and executing RPCs on demand. Support for serial console connections and for telnet and SSH 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. Facts about the device are accessed using the **facts** attribute of the **Device** object instance.

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 **command**, **op**, **resources**, and **factory** modules pertain to Tables and Views. The **command** and **op** modules contain predefined operational Tables and Views that can be used to extract specific information from the output of common operational commands and RPCs 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 Tables and Views | 187

Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50

3

CHAPTER

Installing Junos PyEZ

Installing Junos PyEZ | 12

Setting Up Junos PyEZ Managed Nodes | 19

Installing Junos PyEZ

IN THIS SECTION

- [Installing Prerequisite Software | 13](#)
- [Installing Junos PyEZ on the Configuration Management Server | 15](#)
- [Installing Junos PyEZ in a Python Virtual Environment | 16](#)
- [Using the Junos PyEZ Docker Image | 18](#)

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. You have the option to install and run Junos PyEZ directly on the configuration management server, within a Python virtual environment, or as a Docker container.

As an alternative to installing Junos PyEZ directly on the server, you can install it in a virtual environment. A Python virtual environment isolates a project's Python installation and packages from those installed on the system or in other virtual environments, which prevents breaking the dependencies of other projects. You can create virtual environments when you have projects that require different versions of Python or Python packages or as an alternative to installing packages globally on the system.

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 install Junos PyEZ on the configuration management server, see the following sections:

- [No Link Title](#)
- [No Link Title](#)

To install Junos PyEZ in a Python virtual environment, see the following sections:

- [No Link Title](#)
- [No Link Title](#)

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

- *No Link Title*

Installing Prerequisite Software

Before you install the Junos PyEZ library on the configuration management server or in a virtual environment, ensure that the configuration management server has the following software installed:

- Python 2.7, or Python 3.5 or later
- All prerequisite software for the given operating system, which is outlined in *Table 1*

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

Table 4: Junos PyEZ Prerequisite Software

Operating System	Package or Library
CentOS	<ul style="list-style-type: none">• gcc• libffi-devel• libxml2-devel• libxslt-devel• openssl-devel• pip• python-devel• redhat-rpm-config

Table 4: Junos PyEZ Prerequisite Software *(Continued)*

Operating System	Package or Library
Debian	<ul style="list-style-type: none"> • libxml2-dev • libxslt1-dev • libssl-dev • pip • python-devel (required for Python 2) • python3-devel (required for Python 3)
Fedora	<ul style="list-style-type: none"> • gcc • libffi-devel • libxml2-devel • libxslt-devel • openssl-devel • pip • python-devel (required for Python 2) • python3-devel (required for Python 3) • redhat-rpm-config
FreeBSD	<ul style="list-style-type: none"> • libxml2 • libxslt • py27-pip

Table 4: Junos PyEZ Prerequisite Software (*Continued*)

Operating System	Package or Library
OSX NOTE: If Junos PyEZ does not successfully install using pip , try using easy_install to install the lxml library and then Junos PyEZ.	<ul style="list-style-type: none"> • pip • xcode
Ubuntu	<ul style="list-style-type: none"> • libffi-dev • libssl-dev • libxml2-dev • libxslt1-dev • python-dev (required for Python 2) • python3-dev (required for Python 3) • pip
Windows	<ul style="list-style-type: none"> • ecdsa • pip • pycrypto

Installing Junos PyEZ on the Configuration Management Server

After you install the prerequisite software 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 (use **sudo pip** where appropriate):

```
user@server:~$ sudo pip install junos-eznc
```

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 (use **sudo pip** where appropriate):

```
user@server:~$ sudo pip install git+https://github.com/Juniper/py-junos-eznc.git
```

NOTE: The latest code in the GitHub source repository is under active development and might not be stable.

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**.

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

Installing Junos PyEZ in a Python Virtual Environment

As an alternative to installing Python packages globally on a system, you can install the required packages and dependencies for a specific project in an isolated Python virtual environment. After you install the prerequisite software on the configuration management server, you can create a Python virtual environment for your Junos PyEZ project.

For example, to create a virtual Python 3 installation with Junos PyEZ on a Linux or macOS server:

1. Move into your existing project directory or create a new one, if none exists.

```
user@host:~$ mkdir junos-pyez
```

```
user@host:~$ cd junos-pyez
```

2. Create a virtual Python installation and specify its name, which in this case is venv.

```
user@host:~/junos-pyez$ python3 -m venv venv
```

NOTE: Ubuntu and Debian systems might require you to install the **python3-venv** package before you can create the virtual environment.

3. Activate the virtual environment by executing the script in the virtual environment's **bin** directory that is appropriate for your platform and shell.

```
user@host:~/junos-pyez$ source venv/bin/activate
```

```
(venv) user@host:~/junos-pyez$
```

4. Install Junos PyEZ.

- To install the current release of Junos PyEZ from PyPI, execute the following command:

```
(venv) user@host:~/junos-pyez$ pip install junos-eznc
```

- To install Junos PyEZ from the GitHub project master branch, execute the following command, which requires that Git is installed:

```
(venv) user@host:~/junos-pyez$ pip install git+https://github.com/Juniper/py-junos-eznc.git
```

NOTE: The latest code in the GitHub source repository is under active development and might not be stable.

5. Execute your Junos PyEZ commands or scripts within the virtual environment.
6. When you are finished working in the virtual environment, deactivate it to return to the main shell prompt.

```
(venv) user@host:~/junos-pyez$ deactivate
```

```
user@host:~/junos-pyez$
```

NOTE: For Python 2 virtual environments, use the **virtualenv** tool.

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. Starting in Junos PyEZ Release 2.1.8, the Docker images include Python 3.6. In earlier release, the 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@server:~$ 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@server:~$ 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 | 19](#)

[Understanding Junos PyEZ | 4](#)

[Junos PyEZ Modules Overview | 8](#)

[Authenticating Junos PyEZ Users | 41](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

Setting Up Junos PyEZ Managed Nodes

IN THIS SECTION

- [Enabling NETCONF over SSH on Devices Running Junos OS | 20](#)
- [Satisfying Requirements for SSHv2 Connections | 21](#)
- [Configuring Telnet Service on Devices Running Junos OS | 21](#)

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 directly to a device running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. To telnet directly to a device, you must first configure the Telnet service on the managed device. To manage devices through a NETCONF session over SSH, you must enable the SSH or NETCONF-over-SSH service on the managed device and ensure that the device meets requirements for SSHv2 connections. You do not need to configure these services if the client application connects to the device through a separate console server.

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:

Enabling NETCONF over SSH 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
```

NOTE: It is also possible to reach the NETCONF-over-SSH service on TCP port 22 by configuring the `ssh` statement at the `[edit system services]` hierarchy level, which enables SSH access to the device for all users and applications. However, we recommend configuring the `netconf ssh` statement to more easily identify and filter NETCONF traffic.

- 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 *Unresolved xref*.

Configuring Telnet Service on Devices Running Junos OS

Starting in Junos PyEZ Release 2.0, Junos PyEZ applications can telnet to a device running Junos OS, provided that the Telnet service is configured on the device. Configuring Telnet service for a device enables unencrypted, remote access to the device.

NOTE: Because telnet uses clear-text passwords (therefore creating a potential security vulnerability), we recommend that you use SSH.

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
```

RELATED DOCUMENTATION

[Installing Junos PyEZ | 12](#)

[Understanding Junos PyEZ | 4](#)

[Junos PyEZ Modules Overview | 8](#)

[Authenticating Junos PyEZ Users | 41](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

4

CHAPTER

Connecting to and Retrieving Facts From a Device Using Junos PyEZ

Connecting to Devices Running Junos OS Using Junos PyEZ | 24

Authenticating Junos PyEZ Users | 41

Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50

Accessing the Shell on Devices Running Junos OS Using Junos PyEZ | 56

Connecting to Devices Running Junos OS Using Junos PyEZ

IN THIS SECTION

- [Connection Methods Overview | 24](#)
- [Understanding Junos PyEZ Connection Properties | 26](#)
- [Connecting to a Device Using SSH | 28](#)
- [Connecting to a Device Using Outbound SSH | 32](#)
- [Connecting to a Device Using Telnet | 35](#)
- [Connecting to a Device Using a Serial Console Connection | 38](#)

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 running Junos OS using a serial console connection, telnet, or by establishing a NETCONF session over SSH. In addition, Junos PyEZ also supports connecting to the device through a telnet or SSH connection to a console server. A console server, also known as a terminal server, is a specialized device that provides a network connection to a device's out-of-band management console port.

This topic provides an overview of the connection methods supported by Junos PyEZ and explains how to use the different methods to connect to a device running Junos OS. The Junos PyEZ examples use various authentication methods, but for detailed information about authenticating a user, see ["Authenticating Junos PyEZ Users" on page 41](#).

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 or SSH to connect to the device's management interface or to a console server that is connected to the device's **CONSOLE** port. In addition, Junos PyEZ supports outbound SSH connections, in which the device running Junos OS initiates the connection with the client management application.

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 or SSH through a console server that is connected to the device.

By default, Junos PyEZ uses SSH to connect to a device . 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 1* summarizes the Junos PyEZ connection methods, their default values for certain parameters, any required Junos OS configuration, and the Junos PyEZ release in which support for that connection method was first introduced.

Table 5: Junos PyEZ Connection Modes

Connection Mode	Value of mode Argument	Default Port	Required Junos OS Configuration	First Supported Junos PyEZ Release
NETCONF over SSH (default)	–	830	<pre>[edit system services] netconf { ssh; }</pre>	1.0.0
Serial console connection	serial	/dev/ttyUSB0	–	2.0.0
Telnet to device running Junos OS	telnet	23	<pre>[edit system services] telnet;</pre>	2.0.0
Telnet through a console server	telnet	23	–	2.0.0
SSH through a console server	–	22	–	2.2.0

Table 5: Junos PyEZ Connection Modes (*Continued*)

Connection Mode	Value of mode Argument	Default Port	Required Junos OS Configuration	First Supported Junos PyEZ Release
Outbound SSH	–	–	<pre>[edit system services] outbound-ssh { ... }</pre>	2.2.0

NOTE: Before you can access a device's 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 19](#). 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 (**with ... as** syntax) 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. We recommend that you use a context manager for console connections, because the context manager automatically handles closing the connection, and failure to close the connection can lead to unpredictable results.

Understanding Junos PyEZ Connection Properties

When you connect to a device running Junos OS, Junos PyEZ stores information about the current connection as properties of the **Device** instance. *Table 2* outlines the available connection properties.

Table 6: Device Properties

Property	Description
connected	Boolean specifying the current state of the connection. Returns True when connected.
hostname	String specifying the hostname of the device to which the application is connected.
master	Boolean returning True if the Routing Engine to which the application is connected is the master Routing Engine.
port	Integer or string specifying the port used for the connection.
re_name	String specifying the Routing Engine name to which the application is connected.
timeout	Integer specifying the RPC timeout value in seconds.
uptime	Integer representing the number of seconds since the current Routing Engine was booted. This property is available starting in Junos PyEZ Release 2.1.5.
user	String specifying the user accessing the device running Junos OS.

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**.

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='router.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
```

Connecting to a Device Using SSH

The Junos PyEZ **Device** class supports using SSH to connect to a device running Junos OS. You can establish a NETCONF session over SSH with the device's management interface or you can establish an SSH connection with a console server that is directly connected to the device's **CONSOLE** port. The SSH server must be able to authenticate the user using standard SSH authentication mechanisms, as described in ["Authenticating Junos PyEZ Users" on page 41](#). To establish a NETCONF session over SSH, you must also satisfy the requirements outlined in ["Setting Up Junos PyEZ Managed Nodes" on page 19](#).

Junos PyEZ automatically queries the default SSH configuration file at `~/.ssh/config`, if one exists. When using SSH to connect to a device running Junos OS or to a console server connected to the device, Junos PyEZ first attempts SSH public key-based authentication and then tries password-based authentication. When SSH keys are in use, the supplied password is used as the passphrase for unlocking the private key. When password-based authentication is used, the supplied password is used as the device password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase, then a password is not required. 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: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS or SSH key password: ")

dev = Device(host=hostname, user=junos_username, passwd=junos_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: {}".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: ")
```



```

junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS or SSH key password: ")

dev = Device(host=hostname, user=junos_username, passwd=junos_password)
try:
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {}".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, which automatically calls the `open()` and `close()` methods. For example:

```

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

hostname = input("Device hostname: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS or SSH key password: ")

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

```

Junos PyEZ also enables a client to connect to a device running Junos OS through an SSH connection to a console server. In this case, you must specify the login credentials for the console server by including

the `cs_user` and `cs_passwd` arguments in the **Device** argument list. When SSH keys are in use, set the `cs_passwd` argument to the variable containing the passphrase for the private key.

The console server connects to the device running Junos OS through a serial connection, which can be slow. Junos PyEZ connections through a console server have a default connection timeout value of 0.5 seconds. As a result, you might need to increase the connection timeout interval by including the **Devicetimeout=seconds** argument to allow sufficient time for the client application to establish the connection.

The following Python 3 example authenticates with the console server and then the device running Junos OS. The connection timeout is set to six seconds so that the client has sufficient time to establish the connection.

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

hostname = input("Console server hostname: ")
cs_username = input("Console server username: ")
cs_password = getpass("Console server or SSH key password: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

try:
    with Device(host=hostname, user=junos_username, passwd=junos_password,
                <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cs_user=cs_username</fo:inline>, <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">cs_passwd=cs_password</
fo:inline>, timeout=6) as dev:
        print (dev.facts)
except ConnectError as err:
    print ("Cannot connect to device: {}".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@server ~]$ 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 Outbound SSH

You can configure a device running Junos OS to initiate a TCP/IP connection with a client management application that would be blocked if the client attempted to initiate the connection (for example, if the device is behind a firewall). The **outbound-ssh** configuration instructs the device to create a TCP/IP connection with the client management application and to forward the identity of the device. Once the connection is established, the management application acts as the client and initiates the SSH sequence, and the device acts as the server and authenticates the client.

NOTE: There is no initiation command with outbound SSH. Once outbound SSH is configured and committed, the device begins to initiate an outbound SSH connection based on the committed configuration. The device repeatedly attempts to create this connection until successful. If the connection between the device and the client management application is dropped, the device again attempts to create a new outbound SSH connection until successful. This connection is maintained until the outbound SSH stanza is removed from the configuration.

To configure the device running Junos OS for outbound SSH connections, include the **outbound-ssh** statement at the **[edit system services]** hierarchy level. In the following example, the device running Junos OS attempts to initiate a connection with the host at 198.51.100.101 on port 2200:

```
user@router1> show configuration system services outbound-ssh
```

```
client outbound-ssh {
  device-id router1;
  secret "$9$h1/ceWbs4UDkGD/Cpu1I-Vb"; ## SECRET-DATA
  services netconf;
  198.51.100.101 port 2200;
}
```

To establish a connection with the device running Junos OS using outbound SSH, the Junos PyEZ application sets the **sock_fd** argument in the **Device** constructor equal to the file descriptor of an existing socket and either omits the **host** argument or sets it to **None**.

The following Junos PyEZ example listens on the configured TCP port for incoming SSH sessions from devices running Junos OS. The application accepts an incoming connection and retrieves the socket's file descriptor for that connection, which is used for the value of the **sock_fd** argument. The client application establishes the SSH connection with the device, collects and prints the device facts, disconnects from the device, and waits for more connections.

```
import socket
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError
from pprint import pprint

"""
    Listen on TCP port 2200 for incoming SSH session with a device running Junos
    OS.
    Upon connecting, collect and print the devices facts,
    then disconnect from that device and wait for more connections.
"""

def launch_junos_proxy(client, addr):
    val = {
        'MSG-ID': None,
        'MSG-VER': None,
        'DEVICE-ID': None
```



```

    }
    msg = ''
    count = 3
    while len(msg) < 100 and count > 0:
        c = client.recv(1)
        if c == '\r':
            continue

        if c == '\n':
            count -= 1
            if msg.find(':'):
                (key, value) = msg.split(': ')
                val[key] = value
                msg = ''
            else:
                msg += c

    print('MSG %s %s %s' %
          (val['MSG-ID'], val['MSG-VER'], val['DEVICE-ID']))
    return client.fileno()

def main():

    PORT = 2200

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    s.bind(('', PORT))
    s.listen(5)
    print('\nListening on port %d for incoming sessions ...' % (PORT))

    sock_fd = 0
    while True:
        client, addr = s.accept()
        print('\nGot a connection from %s:%d' % (addr[0], addr[1]))
        sock_fd = launch_junos_proxy(client, addr)

        print('Logging in ...')
        try:
            with Device(host=None, sock_fd=sock_fd) as dev:
                pprint(dev.facts)

```



```

    except ConnectError as err:
        print ("Cannot connect to device: {}".format(err))

if __name__ == "__main__":
    main()

```

```

user@server:~$ python junos-pyez-outbound-ssh.py

```

```

Listening on port 2200 for incoming sessions ...
Got a connection from 10.10.0.5:57881
MSG DEVICE-CONN-INFO V1 router1
Logging in ...
{'2RE': True,
 'HOME': '/var/home/user',
 'RE0': {'last_reboot_reason': 'Router rebooted after a normal shutdown.',
         'mastership_state': 'master',
         'model': 'RE-MX-104',
         'status': 'OK',
         'up_time': '2 days, 6 hours, 22 minutes, 22 seconds'},
 'RE1': {'last_reboot_reason': 'Router rebooted after a normal shutdown.',
         'mastership_state': 'backup',
         'model': 'RE-MX-104',
         'status': 'OK',
         'up_time': '2 days, 6 hours, 22 minutes, 12 seconds'},
 'RE_hw_mi': False,
 'current_re': ['re0', 'master', 'node', 'fwdd', 'member', 'pfem'],
 'domain': 'example.com',
 'fqdn': 'router1.example.com',
 'hostname': 'router1',
 ...

```

For detailed information about configuring outbound SSH on devices running Junos OS, see *Configuring Outbound SSH Service*.

Connecting to a Device Using Telnet

The Junos PyEZ **Device** class supports connecting 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. When the application connects through a console server, specify the port through which the console server connects to the device running Junos OS.

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: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

dev = Device(host=hostname, user=junos_username, passwd=junos_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: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

dev = Device(host=hostname, user=junos_username, passwd=junos_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, which handles opening and closing the connection. For example:

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

hostname = input("Device hostname: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")
```



```

try:
    with Device(host=hostname, user=junos_username, passwd=junos_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=<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
style="italic">hostname</fo:inline>, user=<fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-style="italic">username</fo:inline>, passwd=<fo:inline
xmlns:fo="http://www.w3.org/1999/XSL/Format" font-style="italic">password</
fo:inline>, 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.

```
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

try:
    with Device(mode='serial', port='<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-style="italic">port</fo:inline>' user=junos_username,
        passwd=junos_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

junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

try:
    with Device(mode='serial', port='<fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-style="italic">port</fo:inline>', user=junos_username,
passwd=junos_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 | 19](#)

[Authenticating Junos PyEZ Users | 41](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

[Troubleshooting Junos PyEZ Connection Errors | 316](#)

Authenticating Junos PyEZ Users

IN THIS SECTION

- [Junos PyEZ User Authentication Overview | 41](#)
- [Authenticating Junos PyEZ Users Using a Password | 42](#)
- [Authenticating Junos PyEZ Users Using SSH Keys | 45](#)

Junos PyEZ User Authentication Overview

Junos PyEZ enables you to directly connect to and manage devices running Junos OS using a serial console connection, telnet, or a NETCONF session over SSH. In addition, Junos PyEZ also supports connecting to the device through a telnet or SSH connection to a console server that is connected to the device's **CONSOLE** port. The device must be able to authenticate the user using either a password or other standard SSH authentication mechanisms, depending on the connection method. When you manage devices running Junos OS through an SSH connection, the most convenient and secure way to access a device is to configure SSH keys. SSH keys enable the remote device to identify trusted users.

You can perform device operations 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`.

For SSH connections, Junos PyEZ automatically queries the default SSH configuration file at `~/.ssh/config`, if one exists, unless the `Device` argument list includes the `ssh_config` argument to specify a different configuration file. Junos PyEZ uses any relevant settings in the SSH configuration file for the given connection that are not overridden by the arguments in the `Device` argument list, such as the user or the identity file.

When the Junos PyEZ client uses SSH to connect to either the device running Junos OS or to a console server connected to the device, Junos PyEZ first attempts SSH public key-based authentication and then tries password-based authentication. When SSH keys are in use, the supplied password is used as the passphrase for unlocking the private key. When password-based authentication is used, the supplied password is used as the device password. If SSH public key-based authentication is being used and the SSH private key has an empty passphrase, then a password is not required. 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.

Authenticating Junos PyEZ Users Using a Password

To authenticate a Junos PyEZ 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 to which to connect and the username and password for the device running Junos OS and stores each value in a variable.

```
# Python 3
from jnpr.junos import Device
from getpass import getpass
import sys

hostname = input("Hostname: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS 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. If the Junos PyEZ client connects to the device through an SSH connection to a console server, include code that prompts for the console server username and password and stores each value in a variable.

```
# login credentials required for SSH connection to console server
cs_username = input("Console server username: ")
cs_password = getpass("Console server password: ")
```

4. In the **Device** constructor argument list:

- Set the **host** argument to the variable containing the hostname
- Set the **user** and **passwd** arguments to the variables containing the Junos OS login credentials
- If the Junos PyEZ client connects through a console server using SSH, set the **cs_user** and **cs_passwd** arguments to the variables containing the console server login credentials.
- Include any additional arguments required for the connection method

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

```
# Python 3
from jnpr.junos import Device
from getpass import getpass
import sys

hostname = input("Device hostname: ")
junos_username = input("Junos OS username: ")
junos_password = getpass("Junos OS password: ")

# login credentials required for SSH connection to console server
cs_username = input("Console server username: ")
cs_password = getpass("Console server password: ")

try:
    # NETCONF session over SSH
    with Device(host=hostname, user=junos_username, passwd=junos_password) as dev:

        # Telnet connection to device or console server connected to device
        #with Device(host=hostname, user=junos_username, passwd=junos_password,
        mode='telnet', port='23') as dev:
```



```

    # Serial console connection to device
    #with Device(host=hostname, user=junos_username, passwd=junos_password,
mode='serial', port='/dev/ttyUSB0') as dev:

    # SSH connection to console server connected to device
    #with Device(host=hostname, user=junos_username, passwd=junos_password,
cs_user=cs_username, cs_passwd=cs_password, timeout=5) as dev:

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

```

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.

5. Execute the Junos PyEZ code, which prompts for the hostname, the Junos OS username and password, and the console server username and password (when requested) and does not echo the password on the command line.

```
bsmith@server:~$ python3 junos-pyez-pw.py
```

```

Device hostname: dc1a.example.com
Junos OS username: bsmith
Junos OS password:
Console server username: bsmith
Console server password:
{'domain': 'example.com', 'serialnumber': 'JNXXXXXXXXXX', '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'}

```


Authenticating Junos PyEZ Users Using SSH Keys

IN THIS SECTION

- [Generating and Configuring SSH Keys | 45](#)
- [Referencing SSH Keys in Junos PyEZ Applications | 46](#)

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 device to which the Junos PyEZ client will connect. To directly connect to the device running Junos OS, configure the key on that device. To connect to a device running Junos OS through a console server, configure the key on the console server. To use the keys, you must include the appropriate arguments in the **Device** argument list.

Junos PyEZ can utilize SSH keys that are actively loaded into an SSH key agent, keys that are generated in either the default location or a user-defined location, and keys that either use or forgo password protection. When connecting directly to a device running Junos OS, if the **Device** arguments do not specify a password or SSH key file, 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. When connecting to a console server, only password-protected keys are supported.

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

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@server:~$ cd ~/.ssh
user@server:~/.ssh$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
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 on each device to which the Junos PyEZ application will connect, which could include devices running Junos OS or a console server connected to the device running Junos OS.

The easiest method to configure the public key on a device running Junos OS is to load a file that contains the public key under the appropriate user account.

```
[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@server:~$ 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

IN THIS SECTION

- [Authenticating the User Using an SSH Key Agent with Actively Loaded Keys | 46](#)
- [Authenticating the User Using SSH Keys Without Password Protection | 47](#)
- [Authenticating the User Using Password-Protected SSH Key Files | 48](#)

After generating the SSH key pair and configuring the public key on the remote device, you can use the key to connect to the device 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, whether the key is actively loaded into an SSH key agent, such as `ssh-agent`, and whether the user's SSH configuration file already defines settings for that host. The following sections outline the various scenarios:

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. Junos PyEZ enables a client to connect directly to a device running Junos OS using SSH keys that are actively loaded into an SSH key agent. When connecting to a

device running Junos OS, if the **Device** arguments do not specify a password or SSH key file, 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 use SSH keys that are actively loaded into the native SSH key agent to connect directly to a device running Junos OS:

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

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

Authenticating the User Using SSH Keys Without Password Protection

Junos PyEZ enables a client to connect directly to a device running Junos OS using SSH private keys that do not have password protection, although we do not recommend using SSH private keys with an empty passphrase. Junos PyEZ does not support connecting to a console server using SSH private keys with an empty passphrase.

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='router.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='router.example.com', <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">ssh_private_key_file='/home/
user/.ssh/id_rsa_dc'</fo:inline>)
```

NOTE: If the user's SSH configuration file already specifies the local SSH private key file path for a given host, you can omit the **ssh_private_key_file** argument in the **Device** argument list.

Including the `ssh_private_key_file` argument overrides any existing `IdentityFile` value defined for a host in the user's SSH configuration file.

Authenticating the User Using Password-Protected SSH Key Files

Junos PyEZ clients can use password-protected SSH key files to connect directly to a device running Junos OS or to connect to a console server connected to the device.

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

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

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

key_password = getpass('Password for SSH private key file: ')
```

2. In the `Device` argument list, set the `passwd` argument to reference the variable containing the SSH key file password.

If the key is not in the default location and the file path is not already defined in the user's SSH configuration file, set the `ssh_private_key_file` argument to the path of the private key.

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

key_password = getpass('Password for SSH private key file: ')

dev = Device(host='router.example.com', <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">passwd=key_password</
fo:inline>, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">ssh_private_key_file='/home/user/.ssh/id_rsa_dc'</fo:inline>)
dev.open()
# ...
dev.close()
```

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

1. Include code that prompts for the login credentials for the device running Junos OS and stores each value in a variable.

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

junos_username = input('Junos OS username: ')
junos_password = getpass('Junos OS password: ')
```

2. Include code that prompts for the console server username and the SSH private key password and stores each value in a variable.

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

junos_username = input('Junos OS username: ')
junos_password = getpass('Junos OS password: ')

cs_username = input("Console server username: ")
key_password = getpass('Password for SSH private key file: ')
```

3. In the **Device** constructor argument list:

- Set the **host** argument to the console server hostname or IP address
- Set the **user** and **passwd** arguments to the variables containing the Junos OS login credentials
- Set the **cs_user** argument to the variable containing the console server username
- Set the **cs_passwd** argument to the variable containing the SSH key file password
- Set the **ssh_private_key_file** argument to the path of the private key, if the key is not in the default location and the file path is not already defined in the user's SSH configuration file

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

junos_username = input('Junos OS username: ')
junos_password = getpass('Junos OS password: ')

cs_username = input("Console server username: ")
key_password = getpass('Password for SSH private key file: ')
```



```

with Device(host='router.example.com', user=junos_username,
passwd=junos_password, cs_user=cs_username, <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">cs_passwd=key_password</
fo:inline>, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">ssh_private_key_file='/home/user/.ssh/id_rsa_dc'</fo:inline>)
as dev:
    print (dev.facts)
#     ...

```

RELATED DOCUMENTATION

[Connecting to Devices Running Junos OS Using Junos PyEZ | 24](#)

[Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS | 318](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS

IN THIS SECTION

- [Understanding Junos PyEZ Device Facts | 50](#)
- [Example: Retrieving Facts from Devices Running Junos OS | 52](#)

Understanding Junos PyEZ Device Facts

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. After connecting to a device running Junos OS, Junos PyEZ applications can retrieve facts about the device. The device facts are accessed as the **facts** attribute of the **Device** object. For detailed information about the keys that are included in the returned device facts, 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['hostname'])
    pprint (dev.facts)
```

```
user1@server:~$ python get-facts.py
```

```
'router1'
{'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 the application calls the **Deviceopen()** 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. If a fact is not supported on a given platform, or if the application encounters an issue gathering the value of a specific fact, then the fact will have the value **None**.

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 only a single fact or a set of facts, include the **keys** argument in the **facts_refresh()** method, and specify the 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.

By default, Junos PyEZ returns the device facts as a dictionary-like object. Starting in Junos PyEZ Release 2.2.1, you can view the device facts in JavaScript Object Notation (JSON). To view a JSON representation of the facts, import the **json** module, and call the **json.dumps()** function.

```
from jnpr.junos import Device
import json

with Device(host='router1.example.net') as dev:
    print (json.dumps(dev.facts))
```

Example: Retrieving Facts from Devices Running Junos OS

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. The examples use existing SSH keys for authentication.

To create a Junos PyEZ application that establishes a NETCONF session over SSH with a device running Junos OS and prints 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
from pprint import pprint
```

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: {}".format(err))
    sys.exit(1)
```

5. Print the device facts.

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

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

- 6.

Close the connection to the device.

```
dev.close()
```

7. Save and execute the program.

```
user1@server:~$ python junos-pyez-device-facts.py
```

```
'router1'
{'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'},
 ...
```

The entire program is presented here:

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

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

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



```
dev.close()
```

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

```
user1@server:~$ python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from jnpr.junos import Device
>>> from pprint import pprint
>>>
>>> dev = Device('router1.example.net')
>>> dev.open()
Device(router1.example.net)
>>>
>>> pprint(dev.facts)
{'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'},
 ...
>>>
>>> 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](#)

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

[Connecting to Devices Running Junos OS Using Junos PyEZ | 24](#)
[Authenticating Junos PyEZ Users | 41](#)
[Troubleshooting Junos PyEZ Connection Errors | 316](#)

Accessing the Shell on Devices Running Junos OS Using Junos PyEZ

The Junos OS command-line interface (CLI) has many operational mode commands that return information that is similar to the information returned by many shell commands. Thus, access to the UNIX-level shell on devices running Junos OS is usually not required. However in some cases, a user or application might need to access the shell 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.open()` and `close()` methods establish and terminate an SSH connection with the device. As a result, if the client application only requires access to the shell, it can omit the calls to the `Device.open()` and `close()` methods.

The `StartShell.run()` method executes a shell command and waits for the response. By default, the method waits for one of the default shell prompts (`%`, `#`, `>`, or `\$`) before returning the command output. If you set the `this="string"` argument 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')

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()
```

In this example, the returned tuple includes the Boolean corresponding to the exit code for the command and the command output for the **show version** command. The output in this example is truncated for brevity.

```
(False, '\r\n\r\nHostname: router1\r\nModel: mx104\r\nJunos: 17.1R8\r\nJUNOS Base OS boot [17.1R1.8]\r\n ...)
```

Instances of the **StartShell** class can also be used as context managers. In this case, you do not need to explicitly call the **StartShellopen()** and **close()** methods. For example:

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

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

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


Starting in Junos PyEZ Release 2.0.1, you can include the **StartShelltimeout** argument to specify the duration of time in seconds that the utility must wait for the expected string or pattern in the Junos OS shell before timing out. If you do not specify a timeout, the default is 30 seconds.

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

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

with StartShell(dev, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format"
font-weight="bold">timeout=60</fo:inline>) as ss:
    ss.run('cli -c "request support information | save /var/tmp/
information.txt"')
    version = ss.run('cli -c "show version"')
    print (version)
```

In certain cases, you might need to execute nonreturning shell commands, such as the **monitor traffic** command, which displays traffic that originates or terminates on the local Routing Engine. In the Junos OS CLI, the **monitor traffic** command displays the information in real time until the user sends a Ctrl+c keyboard sequence to stop the packet capture.

Starting in Junos PyEZ Release 2.1.0, you can execute nonreturning shell commands using the **StartShellrun()** method by setting the **this** parameter equal to **None**. When you include the **this=None** argument, the method waits until the specified timeout value to retrieve and return all command output from the shell. In this case, the first item of the returned tuple is **True** when the result of the executed shell command returns content, and the second item is the command output. If you omit the **this** argument or set it equal 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')

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


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.

RELATED DOCUMENTATION

Using Junos PyEZ to Execute RPCs on Devices Running Junos OS | 61

Connecting to Devices Running Junos OS Using Junos PyEZ | 24

5

CHAPTER

Using Junos PyEZ to Manage Device Operations

Using Junos PyEZ to Execute RPCs on Devices Running Junos OS | 61

Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications | 71

Using Junos PyEZ to Reboot or Shut Down Devices Running Junos OS | 73

Using Junos PyEZ to Install Software on Devices Running Junos OS | 78

Transferring Files Using Junos PyEZ | 98

Specifying the XML Parser for a Junos PyEZ Session | 102

Using Junos PyEZ to Execute RPCs on Devices Running Junos OS

IN THIS SECTION

- Mapping Junos OS Commands to Junos PyEZ RPCs | 61
- Executing RPCs as a Property of the Device Instance | 63
- Specifying the Format of the RPC Output | 64
- Specifying the Scope of Data to Return | 66
- Specifying the RPC Timeout | 67
- Normalizing the XML RPC Reply | 68

You can use Junos PyEZ to execute remote procedure calls (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 using 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 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. This topic outlines how to map CLI commands to Junos PyEZ RPCs, how to execute RPCs using Junos PyEZ, and how to customize the data returned in the RPC reply.

Mapping Junos OS Commands to Junos PyEZ RPCs

All operational commands that have Junos XML counterparts are listed in the *Junos XML API Explorer*. You can also display the Junos XML request tag element 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 tag 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@router> 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>
```

You can also display the Junos XML request tag for a command using Junos PyEZ. To display the request tag, 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='router.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='dcl1a.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>dcl1a</host-name>
...
</software-information>
```

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's argument list, change any dashes in the option name to underscores, 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's argument 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')
```

Specifying the Format of the RPC Output

By default, the RPC return value is an XML object starting at the first element under the **<rpc-reply>** tag. Starting in Junos PyEZ Release 1.2.3, you can also return the RPC output in text or JavaScript Object Notation (JSON) format by including either the **{'format':'text'}** or **{'format':'json'}** dictionary as the RPC method's first argument.

NOTE: RPC output in JSON format is only supported on devices running Junos OS Release 14.2 and later releases.

The following example returns the output of the `get_software_information()` RPC in text format, which is identical to the output emitted for the `show version` command in the CLI, except that the RPC output is enclosed within an `<output>` element.

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

with Device(host='router1.example.com') as dev:
    sw_info_text = dev.rpc.get_software_information({'format': 'text'})
    print(etree.tostring(sw_info_text))
```

```
user@server:~$ python junos-pyez-rpc-text-format.py
```

```
<output>
Hostname: router1
Model: mx104
Junos: 18.3R1.9
JUNOS Base OS boot [18.3R1.9]
JUNOS Base OS Software Suite [18.3R1.9]
JUNOS Crypto Software Suite [18.3R1.9]
JUNOS Packet Forwarding Engine Support (TRIO) [18.3R1.9]
JUNOS Web Management [18.3R1.9]
JUNOS Online Documentation [18.3R1.9]
JUNOS SDN Software Suite [18.3R1.9]
JUNOS Services Application Level Gateways [18.3R1.9]
JUNOS Services COS [18.3R1.9]
JUNOS Services Jflow Container package [18.3R1.9]
JUNOS Services Stateful Firewall [18.3R1.9]
JUNOS Services NAT [18.3R1.9]
JUNOS Services RPM [18.3R1.9]
JUNOS Services Captive Portal and Content Delivery Container package [18.3R1.9]
JUNOS Macsec Software Suite [18.3R1.9]
JUNOS Services Crypto [18.3R1.9]
JUNOS Services IPSec [18.3R1.9]
JUNOS DP Crypto Software Suite [18.3R1.9]
JUNOS py-base-powerpc [18.3R1.9]
JUNOS py-extensions-powerpc [18.3R1.9]
JUNOS jsd [powerpc-18.3R1.9-jet-1]
```



```
JUNOS Kernel Software Suite [18.3R1.9]
JUNOS Routing Software Suite [18.3R1.9]
<output>
```

The following example returns the output of the `get_software_information()` RPC in JSON format.

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

with Device(host='router1.example.com') as dev:
    sw_info_json = dev.rpc.get_software_information({'format': 'json'})
    pprint(sw_info_json)
```

```
user@server:~$ python junos-pyez-rpc-json-format.py
```

```
{u'software-information': [{u'host-name': [{u'data': u'router1'}],
                             u'junos-version': [{u'data': u'18.3R1.9'}],
                             u'package-information': [{u'comment': [{u'data':
u'JUNOS Base OS boot [18.3R1.9]'}],
                                                         u'name': [{u'data':
u'junos'}]}],
                             u'JUNOS Base OS Software Suite [18.3R1.9]'}],
                             u'name': [{u'data':
u'jbase'}]}],
...

```

Specifying the Scope of Data to Return

You can use Junos PyEZ to execute an RPC to retrieve operational information from devices running Junos OS. Starting in Junos PyEZ Release 2.3.0, when you request XML output, you can filter the reply to return only specific elements. Filtering the output is beneficial when you have extensive operational output, but you only need to work with a subset of the data.

To filter the RPC reply to return only specific tags, include the RPC method's `filter_xml` argument. The `filter_xml` parameter takes a string containing the subtree filter that selects the elements to return. The subtree filter returns the data that matches the selection criteria.

The following Junos PyEZ example executes the `<get-interface-information>` RPC and filters the output to retrieve just the `<name>` element for each `<physical-interface>` element in the reply:

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

with Device(host='router.example.com', use_filter=True) as dev:
    filter = '<interface-information><physical-interface><name/></physical-
interface></interface-information>'
    result = dev.rpc.get_interface_information(filter_xml=filter)
    print (etree.tostring(reply, encoding='unicode'))
```

When you execute the script, it displays each physical interface's name element.

```
user@server:~$ python junos-pyez-get-interface-names.py
```

```
<interface-information style="normal"><physical-interface><name>
lc-0/0/0
</name></physical-interface><physical-interface><name>
pfe-0/0/0
</name></physical-interface><physical-interface><name>
pfh-0/0/0
</name></physical-interface><physical-interface><name>
xe-0/0/0
</name></physical-interface><physical-interface><name>
xe-0/1/0
</name></physical-interface><physical-interface><name>
ge-1/0/0
</name></physical-interface>
...
</interface-information>
```

Specifying the RPC Timeout

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 Release 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 1 compares a default RPC reply to the normalized version. The default RPC reply includes many newlines that are not present in the normalized reply.

Table 7: Comparison of a Default and Normalized RPC Reply

Default RPC Reply	Normalized RPC Reply
<pre> <interface-information style="terse"> <logical-interface> <name>\nge-0/0/0.0\n</name> <admin-status>\nup\n</admin-status> <oper-status>\nup\n</oper-status> <filter-information>\n</filter- information> <address-family> <address-family-name>\nninet\n</ address-family-name> <interface-address> <ifa-local emit="emit"> \n198.51.100.1/24\n</ifa-local> </interface-address> </address-family> </logical-interface> </interface-information> </pre>	<pre> <interface-information style="terse"> <logical-interface> <name>ge-0/0/0.0</name> <admin-status>up</admin-status> <oper-status>up</oper-status> <filter-information/> <address-family> <address-family-name>inet</ address-family-name> <interface-address> <ifa-local emit="emit">198.51.100.1/24</ifa- local> </interface-address> </address-family> </logical-interface> </interface-information> </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 argument 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', <fo:inline
xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">normalize=True</fo:inline>)

# or

dev.open(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">normalize=True</fo:inline>)

```


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

```
dev.rpc.<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-  
style="italic">rpc_method</fo:inline>(<fo:inline xmlns:fo="http://www.w3.org/  
1999/XSL/Format" font-weight="bold">normalize=True</fo:inline>)
```

For example:

```
rsp = dev.rpc.get_interface_information(interface_name='ge-0/0/0.0', terse=True,  
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-  
weight="bold">normalize=True</fo:inline>)
```

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("//*[ \n  
    address-family[normalize-space(address-family-name)='inet']/ \n  
    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']/ \n  
    interface-address/ifa-local")[0].text)  
  
'198.51.100.1/24'
```

RELATED DOCUMENTATION

[Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications](#) | 71

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='router1.example.com')
dev.open()

with Config(dev, mode='exclusive') as cu:
    cu.load(path="mx-config.conf", <fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-weight="bold">ignore_warning=True</fo:inline>)
    cu.commit()

data = dev.rpc.get_configuration(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">ignore_warning=True</fo:inline>)
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

<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">commit_warnings = ['Advertisement-interval is less than four
times',
    'Chassis configuration for network services has been changed.']</fo:inline>

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

with Config(dev, mode='exclusive') as cu:
    cu.load(path="mx-config.conf")
    cu.commit(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">ignore_warning=commit_warnings</fo:inline>)

dev.close()
```

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 | 61](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320](#)

Using Junos PyEZ to Reboot or Shut Down Devices Running Junos OS

IN THIS SECTION

- [Performing a Reboot or Shut Down | 73](#)
- [Performing a Reboot or Shut Down with a Delay or at a Specified Time | 75](#)
- [Specifying the Target for the Reboot or Shutdown | 76](#)
- [Rebooting a VM Host | 77](#)

Performing a Reboot or Shut Down

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

NOTE: Starting in Junos PyEZ Release 2.1.0, the **reboot()** and **poweroff()** methods perform the requested operation on all Routing Engines in a dual Routing Engine or Virtual Chassis setup. In earlier releases, the methods only perform the operation on the 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.

```
#Python 3
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW
from jnpr.junos.exception import ConnectError
from getpass import getpass

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

try:
    with Device(host=hostname, user=username, passwd=password) as dev:
        sw = SW(dev)
        print(sw.reboot())
except ConnectError as err:
    print (err)
```

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@server:~$ python3 junos-pyez-reboot.py
```

```
Device hostname: dcl1.example.com
Device username: user1
Device password:
Shutdown NOW!
[pid 2358]
```

The following Junos PyEZ example shuts down all Routing Engines on the device, effective immediately.

```
#Python 3
from jnpr.junos import Device
from jnpr.junos.utils.sw import SW
from jnpr.junos.exception import ConnectError
from getpass import getpass
```



```

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

try:
    with Device(host=hostname, user=username, passwd=password) as dev:
        sw = SW(dev)
        print(sw.poweroff())
except ConnectError as err:
    print (err)

```

Performing a Reboot or Shut Down with a Delay or at a Specified Time

The default behavior of the **reboot()** and **poweroff()** methods is to immediately execute the operation. You can also request a delay before executing the operation, or you can schedule the operation at a particular date and time.

To delay the reboot or shutdown operation 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 or powering off. The following example requests a reboot of all Routing Engines in 2 minutes:

```

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

with Device(host='dc1a.example.com') as dev:
    sw = SW(dev)
    sw.reboot(in_min=2)

```

The target device issues messages about the impending reboot to any users logged into the system. After the specified amount of time has passed, the system reboots.

```

*** System shutdown message from user1@dc1a ***

System going down in 2 minutes

```

To schedule the reboot or shutdown operation 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 perform the operation.
- **yyymmddhhmm**—Absolute time at which to perform the operation, specified as year, month, day, hour, and minute.
- **hh:mm**—Absolute time on the current day at which to perform the operation, specified in 24-hour time.

NOTE: The **poweroff()** method supports the **at** parameter starting in Junos PyEZ Release 2.3.0.

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

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

with Device(host='dc1a.example.com') as dev:
    sw = SW(dev)
    sw.reboot(at='22:30')
```

Similarly, the following example schedules all Routing Engines to power off at 22:30 on the current day:

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

with Device(host='dc1a.example.com') as dev:
    sw = SW(dev)
    sw.poweroff(at='22:30')
```

Specifying the Target for the Reboot or Shutdown

Starting in Junos PyEZ Release 2.1.0, the **reboot()** and **poweroff()** methods reboot or shut down all Routing Engines in a dual Routing Engine or Virtual Chassis setup. You can also instruct the device to only reboot the Routing Engine to which the application is connected or to reboot or shutdown a certain node of a device running Junos OS Evolved.

To reboot only the Routing Engine to which the application is connected, include the **all_re=False** argument in the **reboot()** method.

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

with Device(host='dc1a.example.com') as dev:
    sw = SW(dev)
    sw.reboot(all_re=False)
```

To reboot or shut down a specific node of a device running Junos OS Evolved, include the **on_node** argument, and specify the node. For example:

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

with Device(host='router1.example.com') as dev:
    sw = SW(dev)
    sw.reboot(on_node='re0')
```

Rebooting a VM Host

On devices that have Routing Engines with VM host support, Junos OS runs as a virtual machine (VM) over a Linux-based host (VM host). Starting in Junos PyEZ Release 2.3.0, the **reboot()** method supports the **vmhost** argument, which enables you to reboot a VM Host. When you include the **vmhost=True** argument, the system reboots the host OS and compatible Junos OS on all Routing Engines by executing the **<request-vmhost-reboot>** RPC, which corresponds to the **request vmhost reboot** operational mode command.

The following example reboots the Routing Engines on the VM Host, which reboots both the guest Junos OS and the host OS.

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

with Device(host='switch1.example.net') as dev:
    sw = SW(dev)
    sw.reboot(vmhost=True)
```


Release History Table

Release	Description
2.1.0	Starting in Junos PyEZ Release 2.1.0, the <code>reboot()</code> and <code>poweroff()</code> methods perform the requested operation on 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 | 0

Using Junos PyEZ to Install Software on Devices Running Junos OS

IN THIS SECTION

- [Supported Deployment Scenarios | 79](#)
- [Specifying the Software Image Location | 79](#)
- [Installation Process Overview | 81](#)
- [Specifying Installation and Checksum Timeouts | 82](#)
- [Logging the Installation Process | 83](#)
- [Performing a VM Host Upgrade | 85](#)
- [Performing a Unified ISSU or NSSU | 86](#)
- [Example: Using Junos PyEZ to Install Software on Devices Running Junos OS | 87](#)

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.

This topic discusses the supported deployment scenarios, how to specify the software image location, and the general installation process and options when using Junos PyEZ to upgrade a device. It also discusses how to use Junos PyEZ to perform more specialized upgrade scenarios such as a VM host upgrade, a unified in-service software upgrade (unified ISSU), or a nonstop software upgrade (NSSU) on devices that support these features.

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
- VM host upgrades on Routing Engines with VM Host Support
- 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, an SRX Series chassis cluster, or 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 image to the configuration management server, and the `install()` method, by default, copies it to the target device before performing the installation. You can also instruct the `install()` method to install an image that already resides on the target device or resides at a URL that is reachable from the target device.

Table 1 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 8: 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 False	File path including the filename of the software package or packages on the local server running Junos PyEZ.	(Optional) Path to the directory on the target device to which the package or packages will be copied. Default is /var/tmp .
Target device	Set to True	Filename of the software package or packages.	(Optional) Path to the directory on the target device where the package or packages must already reside. Default is /var/tmp .
URL	–	URL from the perspective of the target device running Junos OS from which the software package is installed.	–

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 = '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']
```

For packages residing on the local server running Junos PyEZ, 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.

Starting in Release 2.1.5, Junos PyEZ supports installing software images from a URL. In this case, the `package` or `pkg_set` value must be a URL from the perspective of the target device running Junos OS. The package is copied over and installed from the specified URL, and the `no_copy` and `remote_path` arguments are ignored. For information about specifying the format of the URL, see *Format for Specifying Filenames and URLs in Junos OS CLI Commands*.

Installation Process Overview

To install a software image on a device running Junos OS, a Junos PyEZ application connects to the individual device or to the master device in a Virtual Chassis, creates an instance of the **SW** utility, and calls 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, checksum_algorithm='sha256')
    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 checksum of the local software package or packages using the algorithm specified in the `checksum_algorithm` argument, if the checksum is not already provided through the `checksum`

argument. Acceptable **checksum_algorithm** values are "md5", "sha1", and "sha256". The default is "md5".

- Performs a storage cleanup on the target device to create space for the software package, unless **cleanfs** is set to **False**.
- SCP or FTP copies the package to the **remote_path** directory, or if **remote_path** is not specified, to the **/var/tmp** directory, if a file with the same name and checksum does not already reside in the target location on the device.
- Computes the checksum of the remote file and compares it to the value of the local file.

After the software package is on the target device, whether downloaded there initially, copied over from the configuration management server by the **install()** method, or copied from a URL by the target device, the **install()** method performs the following operations:

- Validates the configuration against the new package if the **validate** parameter is set to **True**
- Installs the package on all Routing Engines unless **all_re** is set to **False**

NOTE: Starting in Release 2.1.5, Junos PyEZ, by default, upgrades all Routing Engines on individual devices and members in a Virtual Chassis. In earlier releases, or if **all_re=False**, Junos PyEZ only upgrades the Routing Engine to which it is connected.

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

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](#)

Specifying Installation and Checksum Timeouts

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. In some cases, the installation process or checksum calculation might exceed these time intervals.

To increase the timeout value for the installation process and the checksum calculation, include the **timeout** and **checksum_timeout** parameters, respectively, in the call to the **install()** method, and set them to appropriate values. 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, <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">timeout=2400</fo:inline>,
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">checksum_timeout=400</fo:inline>)
    if ok:
        sw.reboot()
```

NOTE: The **checksum_timeout** parameter is available starting in Junos PyEZ Release 2.1.4.

Logging the Installation Process

The Junos PyEZ install process enables you to display or log the progress of the installation by including the **progress** argument in the **install()** 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 arguments. Starting in Junos PyEZ Release 1.2.3, you can also set **progress=True** to use **sw.progress()** for basic reporting.

The following example prints the installation progress using the **myprogress** function.

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

<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-weight="bold">def
myprogress(dev, report):
    print "host: %s, report: %s" % (dev.hostname, report)
</fo:inline>
```



```
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, <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">progress=myprogress</fo:inline>)
    if ok:
        sw.reboot()
```

The progress output is in the user-defined format.

```
user@server:~$ python junos-pyez-install.py
```

```
Found package. Installing: junos-install-mx-x86-64-17.2R1.13.tgz

host: router1.example.net, report: computing checksum on local package: junos-
install-mx-x86-64-17.2R1.13.tgz
host: router1.example.net, report: cleaning filesystem ...
host: router1.example.net, report: before copy, computing checksum on remote
package: /var/tmp/junos-install-mx-x86-64-17.2R1.13.tgz
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
38682624 / 386795750 (10%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
77365248 / 386795750 (20%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
116047872 / 386795750 (30%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
154730496 / 386795750 (40%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
193413120 / 386795750 (50%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
232079360 / 386795750 (60%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
270761984 / 386795750 (70%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
309444608 / 386795750 (80%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
348127232 / 386795750 (90%)
host: router1.example.net, report: junos-install-mx-x86-64-17.2R1.13.tgz:
386795750 / 386795750 (100%)
```



```

host: router1.example.net, report: after copy, computing checksum on remote
package: /var/tmp/junos-install-mx-x86-64-17.2R1.13.tgz
host: router1.example.net, report: checksum check passed.
host: router1.example.net, report: installing software ... please be patient ...
host: router1.example.net, report: software pkgadd package-result: 0
Output:
Installing package '/var/tmp/junos-install-mx-x86-64-17.2R1.13.tgz' ...
...

```

Performing a VM Host Upgrade

On devices that have Routing Engines with VM host support, Junos OS runs as a virtual machine (VM) over a Linux-based host (VM host). A VM host upgrade, which upgrades the host OS and compatible Junos OS, requires a VM Host Installation Package (**junos-vmhost-install-x.tgz**) and is performed using the **request vmhost software add** operational mode command, which corresponds to the **<request-vmhost-package-add>** RPC.

Starting in Junos PyEZ Release 2.1.6, the **sw.install()** method supports the **vmhost=True** argument for performing a VM host upgrade. When the **vmhost=True** argument is present, the **sw.install()** method performs the installation using the **<request-vmhost-package-add>** RPC instead of the **<request-package-add>** RPC.

The following example upgrades and reboots both the Junos OS and host OS on a single Routing Engine device:

```

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

with Device(host='switch1.example.net') as dev:
    sw = SW(dev)
    ok = sw.install(package='junos-vmhost-install-qfx-x86-64-18.1R1.9.tgz',
vmhost=True, no_copy=True)
    if ok:
        sw.reboot(vmhost=True)

```

To reboot just the Junos OS software, call the **sw.reboot()** method instead.

Performing a Unified ISSU or NSSU

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. *Table 2* outlines the Junos PyEZ release in which the unified ISSU and NSSU features are first supported. For more information about unified ISSU and NSSU, see the software documentation for your product.

Table 9: Junos PyEZ Unified ISSU and NSSU Support

Junos PyEZ Release	Feature Support
2.1.0	Support for unified ISSU and NSSU on dual-Routing Engine devices running Junos OS.
2.1.6	Support for unified ISSU during a VM host upgrade for those devices with VM host support that use the request vmhost software in-service-upgrade command to perform a unified in-service software upgrade of the host OS and Junos OS.

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 on devices that support this feature, include the **issu=True** argument in the **install()** method.

In the the following example, the **install()** method upgrades Junos OS on both Routing Engines and reboots the new master Routing Engine (previously the old backup Routing Engine) as part of the installation process. If the installation is successful, the **reboot()** method then reboots the connected Routing Engine, which is the new backup Routing Engine (previously the old master Routing Engine).

```
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, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">issu=True</fo:inline>, progress=True)
    if ok:
        sw.reboot(all_re=False)
```


To perform a unified in-service software upgrade on a Routing Engine with VM host support that meets the necessary requirements and supports unified ISSU, include the **vmhost=True** and **issu=True** arguments in the **install()** method. The device upgrades from one host OS and Junos OS release to the requested release using the **<request-vmhost-package-in-service-upgrade>** RPC.

```
sw.install(package='junos-vmhost-install-qfx-x86-64-18.1R1.9.tgz', <fo:inline
xmlns:fo="http://www.w3.org/1999/XSL/Format" font-weight="bold">vmhost=True</
fo:inline>, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">issu=True</fo:inline>, progress=True)
```

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 on devices that support this feature, 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, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">nssu=True</fo:inline>, progress=True)
    if ok:
        sw.reboot(all_re=False)
```

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

IN THIS SECTION

- [Requirements | 88](#)
- [Overview | 88](#)
- [Configuration | 89](#)
- [Executing the Junos PyEZ Code | 94](#)

●	Verification 94
●	Troubleshooting 97

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

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 Junos PyEZ 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. This example assumes that the image has been downloaded to the local 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 `jnpr.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 reboot the device.

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

IN THIS SECTION

- [Creating the Junos PyEZ Program | 89](#)
- [Results | 92](#)

Creating the Junos PyEZ Program

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 = 'dcl1a.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 = 'dcl1a.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

IN THIS SECTION

- [Step-by-Step Procedure | 94](#)

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

IN THIS SECTION

- [Verifying the Installation | 95](#)

Verifying the Installation

IN THIS SECTION

- Purpose | 95
- Action | 95
- Meaning | 96

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:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
94437376 / 944211851 (10%)
2015-09-03 21:34:43,828:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
188858368 / 944211851 (20%)
2015-09-03 21:37:04,180:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
283279360 / 944211851 (30%)
2015-09-03 21:39:24,020:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
377700352 / 944211851 (40%)
2015-09-03 21:41:43,906:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
472121344 / 944211851 (50%)
2015-09-03 21:44:04,079:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
566542336 / 944211851 (60%)
2015-09-03 21:46:23,968:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
660963328 / 944211851 (70%)
2015-09-03 21:48:44,045:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
755384320 / 944211851 (80%)
2015-09-03 21:51:04,016:dcl.example.com: jinstall-13.3R1.8-domestic-signed.tgz:
849805312 / 944211851 (90%)
2015-09-03 21:53:24,058:dcl.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:dcl.example.com: computing remote checksum on: /var/tmp/
jinstall-13.3R1.8-domestic-signed.tgz
2015-09-03 21:56:01,692:dcl.example.com: checksum check passed.
2015-09-03 21:56:01,692:dcl.example.com: validating software against current
config, please be patient ...
2015-09-03 22:47:57,205:dcl.example.com: installing software ... please be
patient ...
2015-09-03 23:28:10,415:dcl.example.com: Software installation complete.
Rebooting
2015-09-03 23:28:11,525:dcl.example.com: Upgrade pending reboot cycle, please
be patient.
2015-09-03 23:28:11,525:dcl.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

IN THIS SECTION

- [Troubleshooting Timeout Errors | 97](#)

Troubleshooting Timeout Errors

Problem

The program generates an `RpcTimeoutError` message or a `TimeoutExpiredError` message and the installation fails.

```
RpcTimeoutError(host: dcl1a.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 or checksum operations that might require a longer time than the default installation and checksum timeout intervals, set the `install` method `timeout` and `checksum_timeout` parameters to appropriate values and re-run the program. For example:

```
sw.install(package=package, remote_path=remote_path, progress=update_progress,
validate=True, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">timeout=2400</fo:inline>, <fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-weight="bold">checksum_timeout=400</fo:inline>)
```


Release History Table

Release	Description
2.1.5	Starting in Release 2.1.5, Junos PyEZ, by default, upgrades all Routing Engines on individual devices and members in a Virtual Chassis.

RELATED DOCUMENTATION

[Using Junos PyEZ to Reboot or Shut Down Devices Running Junos OS | 73](#)

[Junos PyEZ Modules Overview | 8](#)

Transferring Files Using Junos PyEZ

Junos PyEZ provides utilities that enable you to perform file management tasks on devices running Junos OS. You can use the Junos PyEZ `jnpr.junos.utils.scp.SCP` class to secure copy (SCP) files between the local host and a device running Junos OS.

The `SCPopen()` and `close()` methods establish and terminate the connection with the device. As a result, if the client application only performs file copy operations, it can omit calls to the `Deviceopen()` and `close()` methods. Instances of the `SCP` class can be used as context managers, which automatically call the `open()` and `close()` methods. For example:

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

dev = Device('router1.example.com')
with SCP(dev) as scp:
    scp.put("<fo:inline xmlns:fo='http://www.w3.org/1999/XSL/Format' font-
style='italic'>local-file</fo:inline>", remote_path="<fo:inline
xmlns:fo='http://www.w3.org/1999/XSL/Format' font-style='italic'>path</
fo:inline>")
    scp.get("<fo:inline xmlns:fo='http://www.w3.org/1999/XSL/Format' font-
style='italic'>remote-file</fo:inline>", local_path="<fo:inline
xmlns:fo='http://www.w3.org/1999/XSL/Format' font-style='italic'>path</
fo:inline>")
```


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')
    msgfile = "logs/"+dev.hostname+"-messages"

    try:

        #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=msgfile)

#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=msgfile)

except Exception as err:
    print (err)
    return

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@server ~]$ ls logs
```

```
router1.example.com-messages
```

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. For example:

```
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/user1/.ssh/id_rsa_dc"
dev = Device('198.51.100.1', ssh_private_key_file=key_file)

with SCP(dev) as scp:
    scp.put("scp-test.txt", remote_path="/var/tmp/")
```

The **SCP** class also 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@server ~]$ cat ~/.ssh/config
Host 198.51.100.1
User user1
ProxyCommand ssh -l user1 198.51.100.2 nc %h 22 2>/dev/null
```

RELATED DOCUMENTATION

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

Specifying the XML Parser for a Junos PyEZ Session

Extensible Markup Language (XML) is a standard for representing and communicating information. It is a metalanguage for defining customized tags that are applied to a data set or document to describe the function of individual elements and codify the hierarchical relationships between them. Junos OS natively supports XML for the operation and configuration of devices running Junos OS.

Client applications use XML parsers to read and work with XML documents, for example, the command output or configuration data returned as XML in the RPC reply of a device running Junos OS. XML parsers can use different approaches to parse an XML document. Document Object Model (DOM) parsers create a tree representation of the whole XML document, which is loaded in its entirety into memory. By contrast, a Simple API for XML (SAX) parser performs event-based parsing and parses each part of the XML document sequentially. As a result, SAX parsers only load a small portion of the XML data in memory at any time.

SAX parsers do not require a lot of memory to perform operations, but DOM parser memory requirements increase with the document size. In general DOM parsing is faster than SAX parsing, because the application can access the entire XML document in memory. However, as the XML document size increases, DOM parsers require more memory, and SAX parsing becomes more efficient.

Junos PyEZ uses the **ncclient** Python library, which defaults to using DOM parsing for XML processing. In this case, the parser converts the entire XML document into an **lxml** object and loads it into memory. Devices running Junos OS can return large XML documents for show command output and configuration data. If you only need to retrieve a small subset of values from the XML in these cases, DOM parsing can be inefficient.

Starting in Junos PyEZ Release 2.3.0, Junos PyEZ uses SAX parsing when possible in the following cases:

- When you use operational Tables and Views to retrieve structured output

- When you include the **use_filter=True** argument in the **Device()** instance for a given session and perform operations that request a subset of tag values from XML output

For example, the following script executes the **get_interface_information** RPC and filters the XML output to return only the **<name>** element for each physical interface. In this case, SAX parsing is used, because only a subset of the XML is requested.

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

with Device(host='router.example.com', <fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-weight="bold">use_filter=True</fo:inline>) as dev:
    sax_input = '<interface-information><physical-interface><name/></physical-
interface></interface-information>'
    result = dev.rpc.get_interface_information(filter_xml=sax_input)
    print (etree.tostring(reply, encoding='unicode'))
```




Using Junos PyEZ to Manage the Configuration

Using Junos PyEZ to Retrieve a Configuration | 105

Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration | 117

Using Junos PyEZ to Configure Devices Running Junos OS | 120

Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129

Using Junos PyEZ to Commit the Configuration | 148

Example: Using Junos PyEZ to Load Configuration Data from a File | 154

Example: Using Junos PyEZ to Roll Back the Configuration | 166

Using Junos PyEZ to Manage the Rescue Configuration on Devices Running Junos OS | 174

Using Junos PyEZ to Retrieve a Configuration

IN THIS SECTION

- [Retrieving the Complete Candidate Configuration | 106](#)
- [Specifying the Source Database for the Configuration Data | 106](#)
- [Specifying the Scope of Configuration Data to Return | 108](#)
- [Specifying the Format for Configuration Data to Return | 111](#)
- [Retrieving Configuration Data for Standard or Custom YANG Data Models | 112](#)
- [Specifying Additional RPC Options | 114](#)
- [Handling Namespaces in Configuration Data | 115](#)

Junos PyEZ applications can execute Remote Procedure Calls (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 portions of the configuration 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>*.

This topic discusses 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 260](#) and ["Using Junos PyEZ Configuration Tables to Retrieve Configuration Data" on page 279](#).

Retrieving the Complete Candidate Configuration

To retrieve the complete candidate configuration from a device running Junos OS, execute the `get_config()` RPC. 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', pretty_print=True))
```

Specifying the Source Database for the Configuration Data

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 data from the candidate configuration database. A Junos PyEZ application can also retrieve configuration data from the committed configuration database or the ephemeral configuration database.

Candidate Configuration Database

To retrieve data from the candidate configuration database, execute the `get_config()` RPC, and optionally include any additional arguments.

```
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', pretty_print=True))
```

Committed Configuration Database

To retrieve data from the committed configuration database, include the `options` argument with `'database': 'committed'` in the `get_config()` RPC call.

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



```

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">options={'database' : 'committed'}</fo:inline>)
    print (etree.tostring(data, encoding='unicode', pretty_print=True))

```

Ephemeral Configuration Database

Starting in Junos PyEZ Release 2.1.3, Junos PyEZ supports operations on the *ephemeral configuration database* on devices that support this database. When you retrieve configuration data from the shared configuration database, by default, the results do not include data from the ephemeral configuration database.

NOTE: The ephemeral database is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS. The ephemeral configuration database is an advanced feature which if used incorrectly can have a serious negative impact on the operation of the device. For more information, see *Understanding the Ephemeral Configuration Database*.

To retrieve data from the default instance of the ephemeral configuration database, first open the default ephemeral instance and then request the data. To open the default instance, use a context manager to create the **Config** instance, and include the **mode='ephemeral'** argument. For example:

```

from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError
from lxml import etree

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

try:
    dev.open()
    with <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">Config(dev, mode='ephemeral')</fo:inline> as cu:
        data = <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">dev.rpc.get_config(options={'format': 'text'})</fo:inline>
        print(etree.tostring(data, encoding='unicode'))
    dev.close()

except ConnectError as err:
    print ("Cannot connect to device: {}".format(err))

```



```
except Exception as err:
    print (err)
```

To retrieve data from a specific instance of the ephemeral configuration database, first open the ephemeral instance and then request the data. To open a user-defined instance of the ephemeral configuration database, use a context manager to create the **Config** instance, include the **mode='ephemeral'** argument, and set the **ephemeral_instance** argument to the name of the ephemeral instance.

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConnectError
from lxml import etree

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

try:
    dev.open()
    with <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">Config(dev, mode='ephemeral', ephemeral_instance='eph1')</
fo:inline> as cu:
        data = <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">dev.rpc.get_config(options={'format':'text'})</fo:inline>
        print(etree.tostring(data, encoding='unicode'))
    dev.close()

except ConnectError as err:
    print ("Cannot connect to device: {0}".format(err))
except Exception as err:
    print (err)
```

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. The **filter_xml** parameter takes a string containing the subtree filter that selects the configuration statements to return. The subtree filter returns the configuration data that matches the selection criteria.

To request multiple hierarchies, the **filter_xml** string must include the **<configuration>** root element. Otherwise, the value of **filter_xml** must represent all levels of the configuration hierarchy starting just

under the root **<configuration>** element down to the hierarchy to display. To select a subtree, include the empty tag for that hierarchy level. To return a specific object, include a content match node that defines the element and value to match.

The following Junos PyEZ application retrieves and prints the configuration at the **[edit interfaces]** and **[edit protocols]** hierarchy levels in the candidate configuration:

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

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

    filter = '<configuration><interfaces/><protocols/></configuration>'
    data = dev.rpc.get_config(filter_xml=filter)
    print (etree.tostring(data, encoding='unicode', pretty_print=True))
```

The following example 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:

    data = dev.rpc.get_config(filter_xml='<system><services/></system>')
    print (etree.tostring(data, encoding='unicode', pretty_print=True))

    data = dev.rpc.get_config(filter_xml='system/services')
    print (etree.tostring(data, encoding='unicode', pretty_print=True))

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

The following example retrieves the **<name>** element for each **<interface>** element under the **<interfaces>** hierarchy in the post-inheritance candidate configuration:

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

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



```

filter = '<interfaces><interface><name/></interface></interfaces>'
data = dev.rpc.get_config(filter_xml=filter, options={'inherit':'inherit'})
print (etree.tostring(data, encoding='unicode', pretty_print=True))

```

```
user@server:~$ python junos-pyez-get-interface-names.py
```

```

<configuration changed-seconds="1544032801" changed-localtime="2018-12-05
10:00:01 PST">
  <interfaces>
    <interface>
      <name>ge-1/0/0</name>
    </interface>
    <interface>
      <name>ge-1/0/1</name>
    </interface>
    <interface>
      <name>lo0</name>
    </interface>
    <interface>
      <name>fxp0</name>
    </interface>
  </interfaces>
</configuration>

```

The following example retrieves the subtree for the ge-1/0/1 interface:

```

from jnpr.junos import Device
from lxml import etree

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

    filter = '<interfaces><interface><name>ge-1/0/1</name></interface></
interfaces>'

```



```
data = dev.rpc.get_config(filter_xml=filter, options={'inherit':'inherit'})
print (etree.tostring(data, encoding='unicode', pretty_print=True))
```

```
user@server:~$ python junos-pyez-get-single-interface.py
```

```
<configuration changed-seconds="1544032801" changed-localtime="2018-12-05
10:00:01 PST">
  <interfaces>
    <interface>
      <name>ge-1/0/1</name>
      <description>customerA</description>
      <disable/>
      <unit>
        <name>0</name>
        <family>
          <inet>
            <address>
              <name>198.51.100.1/24</name>
            </address>
          </inet>
        </family>
      </unit>
    </interface>
  </interfaces>
</configuration>
```

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()
    print (etree.tostring(data, encoding='unicode', pretty_print=True))

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

    # Junos OS set format
    data = dev.rpc.get_config(options={'format':'set'})
    print (etree.tostring(data, encoding='unicode', pretty_print=True))

    # 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.

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 using the syntax defined for those models. When

you commit the configuration, the data model's translation scripts translate that 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. 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. By default, nonnative configuration data is not included in the `get_config()` RPC reply.

To retrieve configuration data that is defined by a nonnative YANG data model in addition to retrieving the Junos OS configuration, 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**—Retrieve configuration data that is defined by custom YANG data models. You must include the `namespace` argument when retrieving data for custom YANG data models.
- **ietf**—Retrieve configuration data that is defined by IETF YANG data models.
- **openconfig**—Retrieve configuration data that is defined by 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.

If you include the `model` argument with the value `custom`, `ietf`, or `openconfig` and also include the `filter_xml` argument to return a specific XML subtree, Junos OS only returns the matching hierarchy from the nonnative data model. If the Junos OS configuration contains a hierarchy of the same name, for example "interfaces", it is not included in the reply. The `filter_xml` option is not supported when using `model=True`.

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, encoding='unicode', pretty_print=True))
```


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, encoding='unicode', pretty_print=True))
```

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, encoding='unicode', pretty_print=True))
```

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, encoding='unicode', pretty_print=True))
```

Specifying Additional RPC Options

When you use the Junos PyEZ **get_config()** RPC to retrieve the configuration, it invokes the Junos XML protocol **<get-configuration>** operation. The RPC supports the **options** argument, which enables you to include a dictionary of key/value pairs of any attributes supported by the **<get-configuration>** operation. For the complete list of attributes supported by the Junos XML protocol **<get-configuration>** operation, see *<get-configuration>*.

For example, the **get_config()** RPC retrieves data from the pre-inheritance configuration, in which the **<groups>**, **<apply-groups>**, **<apply-groups-except>**, and **<interface-range>** tags are separate elements in the configuration output. To retrieve data from the post-inheritance configuration, which displays statements that are inherited from user-defined groups and ranges as children of the inheriting statements, you can include the **options** argument with **'inherit':'inherit'**.

For example, the following code retrieves the configuration at the **[edit system services]** hierarchy level from the post-inheritance candidate configuration. In this case, if the configuration also contains statements configured at the **[edit groups global system services]** hierarchy level, those statements

would be inherited under the **[edit system services]** hierarchy in the post-inheritance configuration and returned in the retrieved configuration data.

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

with Device(host='router1.example.net') as dev:
    data = dev.rpc.get_config(filter_xml='system/services',
    options={'inherit':'inherit'})
    print (etree.tostring(data, encoding='unicode', pretty_print=True))
```

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, encoding='unicode', pretty_print=True))
```

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>OC</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>OC</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 | 120](#)

[Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration | 117](#)

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).

NOTE: The ephemeral configuration database only stores the current version of the committed ephemeral configuration data, and as a result, it does not support comparing the current configuration to previously committed configurations.

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 between the configurations, 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 example uses the **Config** class to load configuration changes into the candidate configuration, and then calls the `pdiff()` method to print the differences between the modified candidate configuration and the active configuration before committing the changes.

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

dev = Device(host='router1.example.com').open()
with Config(dev, mode='exclusive') as cu:
    cu.load(path='configs/junos-config-mx.conf', merge=True)
    cu.pdiff()
```



```
cu.commit()

dev.close()
```

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)
print (diff)
```

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:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
```



```

dev = Device(host='router1.example.com').open()
with UserConfigTable(dev, mode='private') as userconf:
    userconf.user = 'user1'
    userconf.class_name = 'read-only'
    userconf.append()

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

dev.close()

```

The following example compares the candidate configuration to the configuration with rollback ID 5 but does not make any changes to the configuration:

```

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

with Device(host='router1.example.com') as dev:
    cu = Config(dev)
    cu.pdiff(rb_id=5)

```

RELATED DOCUMENTATION

[Using Junos PyEZ to Configure Devices Running Junos OS | 120](#)

[Example: Using Junos PyEZ to Roll Back the Configuration | 166](#)

[Junos PyEZ Modules Overview | 8](#)

Using Junos PyEZ to Configure Devices Running Junos OS

IN THIS SECTION

- [Comparing Structured and Unstructured Configuration Changes | 120](#)
- [Understanding the General Configuration Process | 123](#)
- [Specifying the Configuration Mode | 124](#)
- [Specifying the Load Operation | 126](#)
- [Creating the Config or Table Object as a Property of the Device Instance | 128](#)

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.

Comparing Structured and Unstructured Configuration Changes

Unstructured configuration changes, which consist of loading 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. In contrast, structured configuration changes use Junos PyEZ configuration Tables and Views to define specific resources to configure, for example, a Junos OS user account. 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 the **Config** utility to make unstructured configuration changes on 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. The **Config** utility also enables you to roll back to a previously committed configuration or load the existing rescue configuration.

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 which configuration statements a user can configure for the 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.

[Table 10 on page 121](#) summarizes the two methods that Junos PyEZ supports for making configuration changes.

Table 10: Junos PyEZ Structured and Unstructured Configuration Changes

Configuration Change Type	Utility	Scope	Configuration Data Format	Additional Information
Structured	Tables and Views	Limited to the configuration statements defined in the Table and View	–	Used to make targeted configuration changes Does not require knowledge of configuration formats or their syntax

Table 10: Junos PyEZ Structured and Unstructured Configuration Changes (*Continued*)

Configuration Change Type	Utility	Scope	Configuration Data Format	Additional Information
Unstructured	<code>jnpr.junos.utils.config.Config</code> class	Any part of the configuration	<ul style="list-style-type: none"> • Text • JSON • Junos OS set commands • Junos XML 	Supports: <ul style="list-style-type: none"> • loading configuration data from strings, XML objects, local or remote files, or Jinja2 Templates • loading the rescue configuration • rolling back the configuration to a previous version

This topic discusses the general configuration process and the operations and elements that are common to both configuration methods. For detailed information about performing configuration updates using either the **Config** utility or Tables and Views, see the documentation specific to that configuration method.

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

- ["Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS" on page 129](#)
- ["Using Junos PyEZ to Commit the Configuration" on page 148](#)
- ["Example: Using Junos PyEZ to Load Configuration Data from a File" on page 154](#)
- `jnpr.junos.utils.config.Config` Class

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

- ["Defining Junos PyEZ Configuration Tables" on page 260](#)

- ["Defining Views for Junos PyEZ Configuration Tables" on page 268](#)
- ["Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources" on page 287](#)
- ["Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS" on page 290](#)
- ["Using Junos PyEZ to Commit the Configuration" on page 148](#)

Understanding the General 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, depending on your preferred configuration method, and associate it with the **Device** object. For example:

Config Object

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

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

Table Object

```
from jnpr.junos import Device
from myTables.ConfigTables import ServicesConfigTable

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

By default, Junos PyEZ updates the candidate global configuration (also known as the *shared configuration database*). The basic process for making configuration changes is to lock the configuration database, load the configuration changes, commit the configuration to make it active, and then unlock the configuration database. When using the Junos PyEZ **Config** utility to make unstructured configuration changes in the shared configuration database, you can perform these actions by calling the appropriate instance methods outlined here:

1. Lock the configuration using `lock()`

2. Modify the configuration by performing one of the following actions:
 - Call `load()` when loading a new complete configuration or modifying specific portions of the configuration
 - Call `rollback()` to revert to a previously committed configuration, as described in ["Rolling Back the Configuration" on page 144](#)
 - Call `rescue()` to load the rescue configuration, as described in ["Loading the Rescue Configuration" on page 146](#)
3. Commit the configuration using `commit()`, as described in ["Committing the Configuration" on page 147](#) and ["Using Junos PyEZ to Commit the Configuration" on page 148](#)
4. Unlock the configuration using `unlock()`

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

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

Specifying the Configuration Mode

By default, Junos PyEZ updates the candidate global configuration. Starting in Junos PyEZ Release 2.0, you can specify the configuration mode to use when modifying the configuration database. To specify a mode other than the default, you must create the **Config** or Table object using a context manager (**with ... as** syntax) and set the **mode** argument to the desired mode. Supported modes include **private**, **exclusive**, **dynamic**, **batch**, and **ephemeral**.

When you specify a mode other than the default, 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. In these cases, you only need to call the `load()` and `commit()` methods to configure the device.

The following examples make configuration changes using the **configure private** mode:

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



```

dev = Device(host='dcl1a1.example.com').open()
with Config(dev, mode='private') as cu:
    cu.load('set system services netconf traceoptions file test.log',
format='set')
    cu.pdiff()
    cu.commit()

dev.close()

```

```

from jnpr.junos import Device
from myTables.ConfigTables import ServicesConfigTable

dev = Device(host='dcl1a.example.com').open()
with ServicesConfigTable(dev, mode='private') as sct:
    sct.ftp = True
    sct.ssh = True
    sct.telnet = True
    sct.append()
    sct.load()
    sct.pdiff()
    sct.commit()

dev.close()

```

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

Starting in Junos PyEZ Release 2.1.3, you can update the *ephemeral configuration database* on devices that support this database. The ephemeral database is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS.

NOTE: The ephemeral configuration database is an advanced feature which if used incorrectly can have a serious negative impact on the operation of the device. For more information, see *Understanding the Ephemeral Configuration Database*.

To open and configure the default instance of the ephemeral configuration database, include the `mode='ephemeral'` argument. For example:

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

dev = Device(host='router1.example.com').open()
with Config(dev, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">mode='ephemeral'</fo:inline>) as cu:
    cu.load('set protocols mpls label-switched-path to-hastings to 192.0.2.1',
format='set')
    cu.commit()
dev.close()
```

To open and configure a user-defined instance of the ephemeral configuration database, include the `mode='ephemeral'` argument, and set the `ephemeral_instance` argument to the name of the instance.

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

dev = Device(host='router1.example.com').open()
with Config(dev, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">mode='ephemeral'</fo:inline>, <fo:inline xmlns:fo="http://
www.w3.org/1999/XSL/Format" font-weight="bold">ephemeral_instance='eph1'</
fo:inline>) as cu:
    cu.load('set protocols mpls label-switched-path to-hastings to 192.0.2.1',
format='set')
    cu.commit()
dev.close()
```

Specifying the Load Operation

In Junos PyEZ, you can load configuration changes using a **load merge**, **load replace**, **load override**, 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 11 on page 127](#) summarizes the parameter settings required for each type of load operation.

NOTE: Because the **load override** and **load update** operations require a complete configuration, the **overwrite=True** and **update=True** arguments must not be used when making configuration changes using Tables, which only modify specific statements in the configuration.

Table 11: Parameters for Specifying the Load Operation Type in the load() and set() Methods

Load Operation	Argument	First Supported Junos PyEZ Release	Description
load merge	merge=True	1.0	Merge the loaded configuration with the existing configuration.
load override	overwrite=True	1.0	Replace the entire configuration with the loaded configuration.
load replace (Default)	–	1.0	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the replace: tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
load update	update=True	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.

Creating the Config or Table Object as a Property of the Device Instance

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 Configure Devices Running Junos OS | 129](#)

[Example: Using Junos PyEZ to Load Configuration Data from a File | 154](#)

[Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources | 287](#)

[Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS | 290](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320](#)

Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS

IN THIS SECTION

- [Configuration Process Overview | 130](#)
- [Specifying the Configuration Mode | 131](#)
- [Specifying the Load Operation | 132](#)
- [Specifying the Format of the Configuration Data to Load | 134](#)
- [Specifying the Location of the Configuration Data | 135](#)
- [Loading Configuration Data from a Local or Remote File | 137](#)
- [Loading Configuration Data from a String | 138](#)
- [Loading Configuration Data Formatted as an XML Object | 141](#)
- [Loading Configuration Data Using Jinja2 Templates | 141](#)
- [Rolling Back the Configuration | 144](#)
- [Loading the Rescue Configuration | 146](#)
- [Committing the Configuration | 147](#)

Junos PyEZ enables you to make structured and unstructured configuration changes on devices running Junos OS. This topic discusses how to use the `jnpr.junos.utils.config.Config` utility to make 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). The **Config** utility also enables you to roll back to a previously committed configuration or revert to a rescue configuration.

Configuration Process Overview

After successfully connecting to a device running Junos OS, to configure the device using the **Config** utility, first create a **Config** object and associate it with the **Device** instance. For example:

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

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

By default, Junos PyEZ updates the candidate global configuration (also known as the *shared configuration database*). The basic process for making configuration changes is to lock the configuration database, load the configuration changes, commit the configuration to make it active, and then unlock the configuration database. When using the Junos PyEZ **Config** utility to make unstructured configuration changes in the shared configuration database, you can perform these actions by calling the appropriate instance methods outlined here:

1. Lock the configuration using `lock()`
2. Modify the configuration by performing one of the following actions:
 - Call `load()` when loading a new complete configuration or modifying specific portions of the configuration
 - Call `rollback()` to revert to a previously committed configuration, as described in ["Rolling Back the Configuration" on page 144](#)
 - Call `rescue()` to load the rescue configuration, as described in ["Loading the Rescue Configuration" on page 146](#)
3. Commit the configuration using `commit()`, as described in ["Committing the Configuration" on page 147](#) and ["Using Junos PyEZ to Commit the Configuration" on page 148](#)
4. Unlock the configuration using `unlock()`

You can also use a context manager (**with ... as** syntax) to create a **Config** object instance, and certain configuration modes require that you use one. For these modes, Junos PyEZ automatically locks and unlocks the configuration. For more information, see [No Link Title](#).

When you use the `load()` method to modify the configuration, in addition to specifying the configuration mode, you can also specify the type of load operation, the format of the configuration changes, and the source of the configuration data. The **Config** utility supports many of the same load operations and formats that are available in the Junos OS command-line interface (CLI). For more information, see:

- *No Link Title*
- *No Link Title*

You can specify the source of the configuration data as a file on the local server, a file on the client device, or a file at a URL that is reachable from the client device, or as a string, an XML object, or a Jinja2 template. For information about specifying the configuration data source, see the following sections:

- *No Link Title*
- *No Link Title*
- *No Link Title*
- *No Link Title*
- *No Link Title*

Specifying the Configuration Mode

By default, when you create a **Config** object and do not explicitly specify a configuration mode, Junos PyEZ updates the candidate global configuration. Starting in Junos PyEZ Release 2.0, you can specify the configuration mode to use when modifying the configuration database. To specify a mode other than the default, you must create the **Config** object using a context manager and set the **mode** argument to the desired mode. Supported modes include **private**, **exclusive**, **dynamic**, **batch**, and **ephemeral**.

NOTE: Starting in Junos PyEZ Release 2.1.3, you can update the *ephemeral configuration database* on devices that support this database. The ephemeral database is an alternate configuration database that provides a fast programmatic interface for performing configuration updates on devices running Junos OS. It is an advanced feature which if used incorrectly can have a serious negative impact on the operation of the device. For more information, see *Understanding the Ephemeral Configuration Database*.

When you specify a mode other than the default, 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. In these cases, you only need to call the **load()** and **commit()** methods to configure the device.

For example, the following code makes configuration changes using the **configure private** mode, which opens a private copy of the candidate configuration:

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

dev = Device(host='dcl1.example.com').open()
with Config(dev, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">mode='private'</fo:inline>) as cu:
    cu.load('set system services netconf traceoptions file test.log',
format='set')
    cu.pdiff()
    cu.commit()

dev.close()
```

For more information about the different configuration modes, see the *CLI User Guide* and ["Using Junos PyEZ to Configure Devices Running Junos OS" on page 120](#).

Specifying the Load Operation

Junos PyEZ supports loading configuration changes using a **load merge**, **load replace**, **load override**, or **load update** operation. You specify the desired load operation by including or omitting the appropriate parameters in the **Config***load()* method.

Table 1 outlines the supported load operations and the corresponding **load()** method argument. By default, Junos PyEZ performs a **load replace** operation. To perform a **load merge**, **load override**, or **load update** operation, set the **merge**, **overwrite**, or **update** parameter, respectively, to **True** in the **load()** method.

Table 12: Parameters for Specifying the Load Operation Type in the load() and set() Methods

Load Operation	Argument	First Supported Junos PyEZ Release	Description
load merge	merge=True	1.0	Merge the loaded configuration with the existing configuration.
load override	overwrite=True	1.0	Replace the entire configuration with the loaded configuration.
load replace (Default)	–	1.0	Merge the loaded configuration with the existing configuration, but replace statements in the existing configuration with those that specify the replace: tag in the loaded configuration. If there is no statement in the existing configuration, the statement in the loaded configuration is added.
load update	update=True	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.

The following example performs a **load override** operation, which replaces the entire candidate configuration with the loaded configuration, and then commits the candidate configuration to make it active.

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

config_mx = "configs/junos-config-mx.conf"
```



```
dev = Device(host='router1.example.com').open()

with Config(dev, mode='exclusive') as cu:
    cu.load(path=config_mx, <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">overwrite=True</fo:inline>)
    cu.commit()

dev.close()
```

Specifying the Format of the Configuration Data to Load

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 2 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 13: Specifying the Format for Configuration Data

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

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.

Specifying the Location of the Configuration Data

Junos PyEZ enables you to load configuration data as strings, files, XML objects, or Jinja2 Template objects. Files can contain either configuration data snippets or Jinja2 templates.

Table 3 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 14: Referencing Configuration Data in the load() Method

Parameter	Configuration Data Source	Description	format Parameter Requirements
path	Local file	Path to a file on the local configuration management server containing configuration data formatted as ASCII text, Junos XML elements, Junos OS set commands, or JSON.	You must include the format parameter when the file extension does not indicate the format of the data.
template	Jinja2 Template object	Pre-loaded Jinja2 Template object. Include the template_vars parameter in the load() method argument list to reference a dictionary containing any required Jinja2 template variables.	You must include the format parameter when the file extension does not indicate the format of the data.
template_path	Local Jinja2 template file	Path to a file on the local configuration management server containing a Jinja2 template formatted as ASCII text, Junos XML elements, Junos OS set commands, or JSON. Include the template_vars parameter in the load() method argument list to reference a dictionary containing any required Jinja2 template variables.	You must include the format parameter when the file extension does not indicate the format of the data.
url	Remote file	Path to a file located on the client device running Junos OS or at a remote URL that is reachable from the client device using an FTP or Hypertext Transfer Protocol (HTTP) URL.	You must include the format parameter when the file extension does not indicate the format of the data.

Table 14: Referencing Configuration Data in the `load()` Method (*Continued*)

Parameter	Configuration Data Source	Description	format Parameter Requirements
vargs[0]	XML object String	XML object or a string that contains configuration data formatted as ASCII text, Junos XML elements, Junos OS set commands, or JSON.	Junos PyEZ automatically detects the format of the configuration data in this case, and the format parameter is not required.

Loading Configuration Data from a Local or Remote File

Junos PyEZ enables you to load configuration data formatted as ASCII text, Junos XML elements, Junos OS **set** commands, or JSON from a file. To load configuration data from a local file on the configuration management server, set the `load()` method's **path** parameter to the absolute or relative path of the file. For example:

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

dev = Device(host='dcl1a.example.com').open()

conf_file = "configs/junos-config-interfaces.conf"
with Config(dev, mode='exclusive') as cu:
    cu.load(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">path=conf_file</fo:inline>, merge=True)
    cu.commit()

dev.close()
```

Starting in Junos PyEZ Release 2.1.5, you can load configuration data from a file located on the client device running Junos OS or at a URL that is reachable from the client device. To load configuration data from a file on the device running Junos OS or at a remote URL, set the **url** parameter to the absolute or

relative path of the file on the client device or the FTP location or Hypertext Transfer Protocol (HTTP) URL of a remote file, and include any other parameters required for the load operation. For example:

```
cu.load(url="/var/home/user/golden.conf")
```

```
cu.load(url="ftp://username@ftp.hostname.net/path/filename")
```

```
cu.load(url="http://username:password@example.com/path/filename")
```

For detailed information about specifying the URL, see the **url** attribute for the Junos XML protocol `<load-configuration>` operation.

If the file does not indicate the format of the configuration data by using one of the accepted file extensions as listed in *No Link Title*, then you must specify the format by including 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)
```

For information about loading configuration data from Jinja2 templates or template files, see *No Link Title*.

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 argument 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. In the examples, **cu** is an instance of the **Config** utility, which operates on the target device running Junos OS.

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



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

- 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>
"""
```


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 argument list, and supply any other required parameters. 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 presents an XML object and the corresponding call to the **load()** method:

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
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"),
                    )
                )
            )
        )
    )

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

with Config(dev, mode='exclusive') as cu:
    cu.load(config_xml_obj, merge=True)
    cu.commit()

dev.close()
```

Loading Configuration Data Using 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. In Jinja, blocks are delimited by '{%' and '%}' and variables are enclosed within '{{' and '}}'.

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.

```
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_vars = {
    '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.

```

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

conf_file = "configs/junos-config-interfaces-mpls.conf"
config_vars = {
    'interfaces': ['ge-1/0/1', 'ge-1/0/2', 'ge-1/0/3'],
    'description': 'MPLS interface',
    'family': 'mpls'
}

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

with Config(dev, mode='exclusive') as cu:
    cu.load(template_path=conf_file, template_vars=config_vars, merge=True)
    cu.commit()

dev.close()

```

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.

Junos PyEZ uses the Jinja2 template and dictionary of variables to render the following configuration data, which is then loaded into the candidate configuration and committed 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/>.

Rolling Back the Configuration

Devices running Junos OS store a copy of the most recently committed configuration and up to 49 previous configurations, depending on the platform. 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 Junos PyEZ `jnpr.junos.utils.config.Config` class `rollback()` method enables you to roll back the configuration on a device running Junos OS. To roll back the configuration, call the `rollback()` method and set the `rb_id` argument to the ID of the rollback configuration. Valid ID 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.

The following example prompts for the rollback ID of the configuration to restore, rolls back the configuration, prints the configuration differences, and then commits the configuration to make it the active configuration on the device.

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

rollback_id = int(input("Rollback ID of the configuration to restore: "))

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

with Config(dev, mode='exclusive') as cu:
    cu.rollback(rb_id=rollback_id)
    cu.pdiff()
    cu.commit()

dev.close()
```

```
user@server:~$ python junos-pyez-rollback.py
```

```
Rollack ID of the configuration to restore: 1

[edit interfaces]
-   ge-0/0/1 {
-       unit 0 {
-           family inet {
-               address 198.51.100.1/24;
-           }
-       }
-   }
```



```
-    }
- }
```

For a more extensive example that includes error handling, see ["Example: Using Junos PyEZ to Roll Back the Configuration" on page 166](#).

Loading 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 manage the rescue configuration on devices running Junos OS. After creating an instance of the **Config** class, you 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 to the desired operation.

To load the existing rescue configuration into the candidate configuration, specify **action="reload"**. If no rescue configuration exists, the load operation returns **False**. After loading the rescue configuration, you must commit the configuration to make it the active configuration on the device.

The following example loads and commits the rescue configuration, if one exists:

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

dev = Device(host='dcl1a.example.com').open()

with Config(dev, mode='exclusive') as cu:
    rescue = cu.rescue(action="reload")
    if rescue is False:
        print ("No existing rescue configuration.")
    else:
        cu.pdiff()
        cu.commit()

dev.close()
```


For information about creating, retrieving, or deleting the rescue configuration and for additional examples, see ["Using Junos PyEZ to Manage the Rescue Configuration on Devices Running Junos OS" on page 174](#).

Committing the Configuration

After modifying the configuration, you must commit the configuration to make it the active configuration on the device. When you use the **Config** utility to make unstructured configuration changes on a device, you commit the candidate configuration by calling the `commit()` method.

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

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

conf_file = "configs/junos-config-interfaces.conf"
with Config(dev, mode='exclusive') as cu:
    cu.load(path=conf_file, merge=True)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cu.commit()</fo:inline>

dev.close()
```

For more information about the commit operation and supported commit options in Junos PyEZ scripts, see ["Using Junos PyEZ to Commit the Configuration" on page 148](#).

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	120
Example: Using Junos PyEZ to Load Configuration Data from a File	154
Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS	320

Using Junos PyEZ to Commit the Configuration

IN THIS SECTION

- [Committing the Candidate Configuration | 148](#)
- [Specifying Commit Options | 150](#)

Junos PyEZ enables you to make structured and unstructured configuration changes on devices running Junos OS. After connecting to the device and modifying the configuration, you must commit the configuration to make it active. This topic discusses how to commit the configuration and which commit options are supported in Junos PyEZ applications.

Committing the Candidate Configuration

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

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import ConfigLoadError, CommitError

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

with Config(dev, mode='exclusive') as cu:
    try:
        cu.load(path="configs/mx_config.conf", merge=True)
        <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cu.commit()</fo:inline>
    except (ConfigLoadError, CommitError) as err:
        print (err)

dev.close()
```


To verify the syntax of the configuration without 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. For example:

```
from jnpr.junos import Device
from myTables.UserConfigTable import UserConfigTable

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

userconfig = UserConfigTable(dev)
# ...set the values for the configuration data...
userconfig.append()
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">userconfig.set(merge=True)</fo:inline>

dev.close()
```

Similarly, you can call the individual methods, as in the following example:

```
from jnpr.junos import Device
from myTables.UserConfigTable import UserConfigTable

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

userconfig = UserConfigTable(dev)
# ...set the values for the configuration data...
userconfig.append()
userconfig.lock()
userconfig.load(merge=True)
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">userconfig.commit()</fo:inline>
userconfig.unlock()
```



```
dev.close()
```

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

Specifying Commit Options

The Junos OS command-line interface (CLI) provides options for the commit operation, such as adding a commit comment or synchronizing the configuration on multiple Routing Engines. Junos PyEZ supports many of these same commit options and some additional options, which you can use in your Junos PyEZ application by including the appropriate arguments in the `commit()` or `set()` method argument list.

Table 1 outlines the supported commit options, provides the corresponding CLI command, and specifies the Junos PyEZ Release in which support for the option was first introduced.

Table 15: 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>	1.0.0
<code>confirm=(True minutes)</code>	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 True to use the default time of 10 minutes.	<code>commit confirmed <minutes></code>	1.0.0

Table 15: Junos PyEZ Supported Commit Options (*Continued*)

Commit Option Argument	Description	CLI command	Junos PyEZ Release
detail=True	Return an XML object with detailed information about the commit process.	commit display detail display xml	1.2.0
force_sync=True	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.	commit synchronize force	1.2.0
ignore_warning=True ignore_warning="string" ignore_warning=["string1", "string2"]	Ignore warnings that are raised during the commit operation. Set the argument to True to ignore all warnings, or set the argument to a string or list of strings specifying which warnings to ignore.	–	2.1.1
sync=True	Synchronize and commit the configuration on both Routing Engines.	commit synchronize	1.2.0
timeout=seconds	Wait for completion of the operation using the specified value as the timeout.	–	1.1.0

Commit Comment

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** argument 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.

Commit Confirm

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.

Commit Detail

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'))
```


Commit Synchronize

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)
```

Commit Timeout

The default time for an 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. To accommodate configuration changes that might require a commit time that is longer than the default timeout interval, include the **timeout=seconds** argument in the **commit()** or **set()** method argument list, and set the timeout interval to an appropriate value. For example:

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

Ignore Warnings

Junos PyEZ raises an **RpcError** exception when the RPC reply contains **<rpc-error>** elements with a severity of warning or higher. In cases where it is necessary or desirable to suppress the **RpcError** exceptions that are raised in response to warnings, you can include the **commit()** method's **ignore_warning** parameter. For example:

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

For more information about using the **ignore_warning** parameter, see ["Suppressing RpcError Exceptions Raised for Warnings in Junos PyEZ Applications"](#) on page 71.

RELATED DOCUMENTATION

[Using Junos PyEZ to Configure Devices Running Junos OS | 120](#)

[Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129](#)

[Example: Using Junos PyEZ to Load Configuration Data from a File | 154](#)

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

IN THIS SECTION

- [Requirements | 154](#)
- [Overview | 155](#)
- [Configuration | 155](#)
- [Executing the Junos PyEZ Code | 161](#)
- [Verification | 161](#)
- [Troubleshooting | 163](#)

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ library provides simple yet powerful methods to perform 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 local file on the configuration management server onto a device running Junos OS.

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 enable a new op script in the configuration of the specified device. The **junos-config-add-op-script.conf** file, which is located on the configuration management server, 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

IN THIS SECTION

- [Creating the Configuration Data File | 156](#)

- [Creating the Junos PyEZ Program | 156](#)
- [Results | 159](#)

Creating the Configuration Data File

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

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 = 'dcl1a.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: {}".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: {}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError:
        print ("Unable to unlock configuration: {}".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: {}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {}".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 = 'dcla.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: {}".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: {}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError:
        print ("Unable to unlock configuration: {}".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: {}".format(err))
    print ("Unlocking the configuration")
    try:
        dev.cu.unlock()
    except UnlockError as err:
        print ("Unable to unlock configuration: {}".format(err))
    dev.close()
    return

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

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

if __name__ == "__main__":
    main()

```


Executing the Junos PyEZ Code

IN THIS SECTION

- [Step-by-Step Procedure | 161](#)

Step-by-Step Procedure

To execute the Junos PyEZ code:

- On the configuration management server, execute the program.

```
user@server:~$ python junos-pyez-config.py
```

```
Locking the configuration
Loading configuration changes
Committing the configuration
Unlocking the configuration
```

Verification

IN THIS SECTION

- [Verifying the Configuration | 162](#)

Verifying the Configuration

IN THIS SECTION

- Purpose | 162
- Action | 162
- Meaning | 163

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:

```
user@dc1a> show configuration system scripts
```

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

```
user@dc1a> show system commit
```

```
0    2014-07-29 14:40:50 PDT by user via netconf  
...
```

```
user@dc1a> show log messages
```

```
Jul 29 14:40:36  dc1a sshd[75843]: Accepted publickey for user 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 dcl1a sshd[75843]: subsystem request for netconf by user user
Jul 29 14:40:42 dcl1a file[75846]: UI_COMMIT: User 'user' requested 'commit'
operation (comment: Loaded by example.)
Jul 29 14:40:45 dcl1a mspd[75888]: mspd: No member config
Jul 29 14:40:45 dcl1a mspd[75888]: mspd: Building package info
Jul 29 14:40:51 dcl1a mspd[1687]: mspd: No member config
Jul 29 14:40:51 dcl1a mspd[1687]: mspd: Building package info
Jul 29 14:40:51 dcl1a 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

IN THIS SECTION

- [Troubleshooting Timeout Errors | 163](#)
- [Troubleshooting Configuration Lock Errors | 164](#)
- [Troubleshooting Configuration Change Errors | 165](#)

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: dcl1a.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 **Devicetimeout** 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="<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
style="italic">host</fo:inline>")
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 | 120](#)

[Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129](#)

[Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration | 117](#)

[Junos PyEZ Modules Overview | 8](#)

Example: Using Junos PyEZ to Roll Back the Configuration

IN THIS SECTION

- [Requirements | 166](#)
- [Overview | 167](#)
- [Configuration | 167](#)
- [Executing the Junos PyEZ Code | 171](#)
- [Verification | 172](#)

Juniper Networks provides support for using Python to manage devices running Junos OS. The Junos PyEZ library provides simple yet powerful methods to perform 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

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 `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. The `rollback()` method loads the configuration into the candidate configuration, which is then committed to make it active by calling the `commit()` method.

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

IN THIS SECTION

- [Creating the Junos PyEZ Program | 168](#)
- [Results | 170](#)

Creating the Junos PyEZ Program

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 = '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: {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 roll back 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 = 'dcl1a.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: {}".format(err))
    except RpcError as err:
        print ("Unable to roll back configuration changes: {}".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

IN THIS SECTION

- [Step-by-Step Procedure | 171](#)

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

IN THIS SECTION

- [Verifying the Configuration | 172](#)

Verifying the Configuration

IN THIS SECTION

- [Purpose | 172](#)
- [Action | 173](#)
- [Meaning | 174](#)

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 the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129](#)

[Using Junos PyEZ to Compare the Candidate Configuration and a Previously Committed Configuration | 117](#)

[Junos PyEZ Modules Overview | 8](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320](#)

Using Junos PyEZ to Manage the Rescue Configuration on Devices Running Junos OS

IN THIS SECTION

- [Managing the Rescue Configuration | 175](#)
- [Example: Using Junos PyEZ to Save a Rescue Configuration | 177](#)

The Junos PyEZ `jnpr.junos.utils.config.Config` utility enables you to manage the rescue configuration on devices running Junos OS. 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.

Managing the Rescue Configuration

IN THIS SECTION

- [Saving a Rescue Configuration | 175](#)
- [Retrieving the Rescue Configuration | 175](#)
- [Loading and Committing the Rescue Configuration | 176](#)
- [Deleting the Rescue Configuration | 177](#)

The `jnpr.junos.utils.config.Config` utility enables you to save, retrieve, load, and delete the rescue configuration on a device running Junos OS. After creating an instance of the **Config** class, you use the `rescue()` method to manage the rescue configuration. Specify the action to perform on the rescue configuration by setting the `rescue()` method **action** parameter to the desired operation. Valid action values include **"save"**, **"get"**, **"reload"**, and **"delete"**. The following examples illustrate the method call for each `rescue()` method action.

Saving a Rescue Configuration

When you create a rescue configuration, the device saves the most recently committed configuration as the rescue configuration. To save the active configuration as the rescue configuration, specify **action="save"** in the `rescue()` method argument list. This operation overwrites any existing rescue configuration. For example:

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

with Device(host='dc1a.example.com') as dev:
    cu = Config(dev)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cu.rescue(action="save")</fo:inline>
```

Retrieving the Rescue Configuration

To retrieve an existing rescue configuration, specify **action="get"**, and optionally specify the format as **"json"**, **"text"** or **"xml"**. If you do not specify a format, the default format is text. If the device does not have an existing rescue configuration, the `rescue()` method returns **None**.

The following example retrieves and prints the rescue configuration, if one exists.

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

with Device(host='dcl1.example.com') as dev:
    cu = Config(dev)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">rescue = cu.rescue(action="get", format="xml")</fo:inline>
    if rescue is None:
        print ("No existing rescue configuration.")
    else:
        print (etree.tostring(rescue, encoding='unicode'))
```

Loading and Committing the Rescue Configuration

To load the existing rescue configuration into the candidate configuration, specify **action="reload"**. If no rescue configuration exists, the load operation returns **False**. After loading the rescue configuration, you must commit the configuration to make it the active configuration on the device.

The following example attempts to load the rescue configuration, and if one exists, commits it to make it the active configuration.

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

with Device(host='dcl1.example.com') as dev:

    with Config(dev, mode='exclusive') as cu:
        rescue = <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cu.rescue(action="reload")</fo:inline>
        if rescue is False:
            print ("No existing rescue configuration.")
        else:
            cu.pdiff()
            <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">cu.commit()</fo:inline>
```


Deleting the Rescue Configuration

To delete the existing rescue configuration, specify `action="delete"`.

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

with Device(host='dc1a.example.com') as dev:
    cu = Config(dev)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-weight="bold">cu.rescue(action="delete")</fo:inline>
```

Example: Using Junos PyEZ to Save a Rescue Configuration

IN THIS SECTION

- [Requirements | 177](#)
- [Overview | 178](#)
- [Configuration | 178](#)
- [Executing the Junos PyEZ Code | 181](#)
- [Verification | 182](#)
- [Troubleshooting | 183](#)

Juniper Networks provides support for using Python to manage 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, if one does not already exist.

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
- 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 instructs the device to 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

IN THIS SECTION

- [Creating the Junos PyEZ Program | 178](#)
- [Results | 180](#)

Creating the Junos PyEZ Program

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 jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.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 = 'dcl1a.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

    # 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

IN THIS SECTION

- [Step-by-Step Procedure | 181](#)

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

IN THIS SECTION

- [Verifying the Configuration | 182](#)

Verifying the Configuration

IN THIS SECTION

- [Purpose | 182](#)
- [Action | 183](#)

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@dcla> show system configuration rescue
```

```
## Last changed: 2014-07-31 17:59:04 PDT
version 13.3R1.8;
groups {
    re0 {
        system {
            host-name dcla;
        }
    }
    ...
}
[output truncated]
```

Troubleshooting

IN THIS SECTION

- [Troubleshooting Unsupported Action Errors | 183](#)

Troubleshooting Unsupported Action 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 the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129](#)

[Junos PyEZ Modules Overview | 8](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320](#)

7

CHAPTER

Creating and Using Junos PyEZ Tables and Views

[Understanding Junos PyEZ Tables and Views | 187](#)

[Predefined Junos PyEZ Operational Tables \(Structured Output\) | 189](#)

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Defining Junos PyEZ Operational Tables for Parsing Structured Output | 200](#)

[Defining Views for Junos PyEZ Operational Tables that Parse Structured Output | 209](#)

[Using Junos PyEZ Operational Tables and Views that Parse Structured Output | 216](#)

[Defining Junos PyEZ Operational Tables for Parsing Unstructured Output | 221](#)

[Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output | 238](#)

[Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output | 259](#)

[Defining Junos PyEZ Configuration Tables | 260](#)

[Defining Views for Junos PyEZ Configuration Tables | 268](#)

[Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279](#)

[Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources | 287](#)

[Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS | 290](#)

[Saving and Loading Junos PyEZ Table XML to and from Files | 311](#)

Understanding Junos PyEZ Tables and Views

Junos PyEZ Tables and Views are simple YAML definitions that enable you to:

- extract operational information from devices running Junos OS
- retrieve configuration data from devices running Junos OS
- configure devices running Junos OS

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. Tables and Views are defined using YAML, so no complex coding is required to create your own. To extract information, you use predefined or custom Tables to map command output or configuration data to a table, which consists of a collection of items. Each Table item represents a record of data and has a unique key. A Table also references a specific View, which is used to map the tag names or fields in the data to the variable names in the Python data structure.

[Table 16 on page 187](#) outlines the different types of Tables and notes the Junos PyEZ release in which each type was first supported. Junos PyEZ operational (op) Tables select items from operational command output. Op Tables can execute RPCs and parse structured output, or they can execute commands and parse unstructured output. Junos PyEZ configuration Tables define structured configuration resources that select or configure statements in specified hierarchies of the given configuration database.

Configuration tables that define the **get** property can only retrieve configuration data. Configuration Tables that define the **set** property can both retrieve as well as modify the configuration statements defined in the corresponding View. When you add the configuration Table to the Junos PyEZ framework, Junos PyEZ dynamically creates a configuration class for the resource, which enables you to programmatically configure the resource on a device.

Table 16: Junos PyEZ Table Types

Table Type	Subset	Description	Junos PyEZ Release
Operational Table	RPC with structured output	Execute an RPC on a device and return structured XML output	1.0

Table 16: Junos PyEZ Table Types (*Continued*)

Table Type	Subset	Description	Junos PyEZ Release
	Command with unstructured output	Execute a CLI command on a device or execute a vty command on an FPC and return unstructured CLI-formatted output	2.3
Configuration Table	get	Retrieve configuration data	1.2
	set	Retrieve configuration data or configure statements defined in the corresponding View	2.0

For example, the following op Table retrieves output for the **get-arp-table-information** RPC with the **no-resolve** option, which corresponds to the **show arp no-resolve** command in the Junos OS CLI. The Table extracts **arp-table-entry** elements from the XML output. 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. In this case, **mac-address**, **ip-address**, and **interface-name** are child elements of **arp-table-entry**.

```

---
ArpTable:
  rpc: get-arp-table-information
  args:
    no-resolve: True
  item: arp-table-entry
  key: mac-address
  view: ArpView

ArpView:
  fields:
    mac_address: mac-address
    ip_address: ip-address
    interface_name: interface-name

```

For information about creating and using operational Tables and Views, see the following topics:

- ["Defining Junos PyEZ Operational Tables for Parsing Structured Output" on page 200](#)
- ["Defining Views for Junos PyEZ Operational Tables that Parse Structured Output" on page 209](#)
- ["Using Junos PyEZ Operational Tables and Views that Parse Structured Output" on page 216](#)
- ["Predefined Junos PyEZ Operational Tables \(Structured Output\)" on page 189](#)
- ["Defining Junos PyEZ Operational Tables for Parsing Unstructured Output" on page 221](#)
- ["Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output" on page 238](#)
- ["Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output" on page 259](#)

For information about creating and using configuration Tables and Views, see the following topics:

- ["Defining Junos PyEZ Configuration Tables" on page 260](#)
- ["Defining Views for Junos PyEZ Configuration Tables" on page 268](#)
- ["Using Junos PyEZ Configuration Tables to Retrieve Configuration Data" on page 279](#)
- ["Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources" on page 287](#)
- ["Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS" on page 290](#)

For information about loading or importing custom Tables and Views in your Junos PyEZ application or about saving data to files, see the following topics:

- ["Loading Inline or External Tables and Views in Junos PyEZ Applications" on page 194](#)
- ["Saving and Loading Junos PyEZ Table XML to and from Files" on page 311](#)

Predefined Junos PyEZ Operational Tables (Structured Output)

Junos PyEZ operational (op) Tables for structured output provide a simple and efficient way to extract information from the XML output of an RPC. The Junos PyEZ `jnpr.junos.op` module provides predefined Table and View definitions for RPCs corresponding to some common operational commands. [Table 17 on page 190](#) lists each of the modules, the Table names defined in that module, and the general 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 your own custom Table and View definitions.

Table 17: 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
elsethernetswitchingtable	ElsEthernetSwitchingTable	get-ethernet-switching-table-information	show ethernet-switching table
ethernetswitchingtable	EthernetSwitchingTable	get-ethernet-switching-table-information	show ethernet-switching table
ethport	EthPortTable	get-interface-information (filtered for Ethernet interfaces)	show interfaces media
fpc	FpcHwTable FpcMiReHwTable	get-chassis-inventory	show chassis hardware

Table 17: jnpr.junos.op Modules *(Continued)*

Module	Table	RPC	CLI Command
	FpcInfoTable FpcMiRelInfoTable	get-fpc-information	show chassis fpc
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-lldp-neighbors-information	show lldp neighbors
nd	NdTable	get-ipv6-nd-information	show ipv6 neighbors

Table 17: jnpr.junos.op Modules *(Continued)*

Module	Table	RPC	CLI Command
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
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

Table 17: jnpr.junos.op Modules *(Continued)*

Module	Table	RPC	CLI Command
	RouteSummaryTable	get-route-summary-information	show route summary
securityzone	SecurityZoneTable	get-zones-information	show security zones
teddb	TedTable TedSummaryTable	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 demonstrates how to use the **RouteTable** Table.



Video: [Junos PyEZ - Tables](#)

RELATED DOCUMENTATION

[Loading Inline or External Tables and Views in Junos PyEZ Applications](#) | 194

[Using Junos PyEZ Operational Tables and Views that Parse Structured Output](#) | 216

Loading Inline or External Tables and Views in Junos PyEZ Applications

IN THIS SECTION

- [Importing Junos PyEZ's Predefined Tables and Views | 194](#)
- [Loading Inline Tables and Views | 195](#)
- [Importing External Tables and Views | 196](#)
- [Using Tables and Views | 198](#)

Junos PyEZ Tables and Views provide a simple and efficient way to configure devices running Junos OS or extract specific information from operational command output or configuration data for a device. Junos PyEZ provides a set of predefined operational Tables and Views that you can use in applications, or you can create your own custom operational or configuration Tables and Views.

You can create quick inline Tables and Views as a multiline string directly in the Python application, or you can create one or more Table and View definitions in external files and import the Tables into your Python application. Inline Tables and Views are simpler to use, but using external files enables you to create a central, reusable library.

To use Junos PyEZ's predefined Tables and Views in your Python application, you must import the Table into your application. To use custom Tables and Views, you must create the Table and View definitions, and then either load or import the definitions into the application, depending on whether they are internal or external to the module. The following sections outline this process for Tables and Views that are both internal and external to the module.

Importing Junos PyEZ's Predefined Tables and Views

The Junos PyEZ `jnpr.junos.op` module and `jnpr.junos.command` module provide predefined Table and View definitions for some common operational RPCs and commands. To use Junos PyEZ's predefined Tables and Views in your Python application, you must include the appropriate import statements in your application. Along with importing the Junos PyEZ **Device** class, you must also import any required Tables.

The following example imports a predefined operational Table, **EthPortTable**, from the `jnpr.junos.op.ethport` module:

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable
```

After you import the Table and View definitions, you can use them as described in *No Link Title*. The following example retrieves the data for the RPC defined in the Table and then prints the interface name and operational status.

```
from jnpr.junos import Device
from jnpr.junos.op.ethport import EthPortTable

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

    for item in eth:
        print("{}: {}".format(item.name, item.oper))
```

For more information about Junos PyEZ's predefined Tables and Views, see ["Predefined Junos PyEZ Operational Tables \(Structured Output\)" on page 189](#).

Loading Inline Tables and Views

To create, load, and use custom inline Tables and Views in your Junos PyEZ application:

1. Import the following classes and libraries in your module:

```
from jnpr.junos import Device
from jnpr.junos.factory.factory_loader import FactoryLoader
import yaml
```

2. Define one or more Tables and Views in YAML as a multiline string.

```
myYAML = """
---
UserTable:
```



```

get: system/login/user
view: UserView
UserView:
  fields:
    username: name
    userclass: class
"""

```

3. Load the Table and View definitions by including the following statement, where *string-name* is the identifier for the multiline string that contains the Table/View definition:

```

globals().update(FactoryLoader().load(yaml.load(<fo:inline
xmlns:fo="http://www.w3.org/1999/XSL/Format" font-style="italic">string-
name</fo:inline>)))

```

4. Connect to the device and use the Table to retrieve information, configure the device, or both, depending on the type of Table, for example:

```

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.get()

    for account in users:
        print("Username is {} \n User class is {}".format(account.username,
account.userclass))

```

After the Table and View definitions are loaded, there is no difference in how you use inline or external Tables in your module. For additional information, see *No Link Title*.

Importing External Tables and Views

External Table and View definitions are placed in files that are external to your Junos PyEZ application. To create external custom Tables and Views and import them into your Junos PyEZ application:

1. Define one or more Tables and Views in YAML, and save them to a file that has a **.yaml** extension.

```
[user@server]$ 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
```

2. Create a file that has the same base name as your Table file but uses a **.py** extension, and include the following four lines of code in the file.

```
[user@server]$ cat myTables/ConfigTables.py
```

```
from jnpr.junos.factory import loadyaml
from os.path import splitext
_YAML_ = splitext(__file__)[0] + '.yaml'
globals().update(loadyaml(_YAML_))
```


3. If the `.yaml` and `.py` files are located in a subdirectory, include an `__init__.py` file in that subdirectory, so that Python checks this directory when it processes the Table import statements in your application.

```
[user@server]$ ls myTables
```

```
__init__.py  ConfigTables.py  ConfigTables.yaml
```

4. In the Junos PyEZ application, import the **Device** class and any required Tables.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable
```

5. Connect to the device and use the Table to retrieve information, configure the device, or both, depending on the type of Table, for example:

```
with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.get()

    for account in users:
        print("Username is {} \nUser class is {}".format(account.username,
        account.userclass))
```

After the Table and View definitions are loaded, there is no difference in how you use inline or external Tables in your module. For additional information, see *No Link Title*.

Using Tables and Views

After you load or import the Table and View definitions, you can use the predefined, inline, or custom external Tables in the same manner.

To use a Table:

1. Create the **Device** instance and open a connection to the target device:

```
with Device(host='router.example.com') as dev:
```


2. Create the Table instance and associate it with the **Device** instance.

```
users = UserTable(dev)
```

3. Use the Table to retrieve information, configure the device, or both, depending on the type of Table.

```
users.get()
```

4. Iterate over and manipulate the resulting object to extract the required information.

```
for account in users:
    print("Username is {} \nUser class is {}".format(account.username,
account.userclass))
```

The following example imports a custom external Table, **UserTable**. The application connects to the device and calls the Table's **get()** method to retrieve **user** objects from the **[edit system login]** hierarchy level. The application then prints each username and its corresponding login class.

```
from jnpr.junos import Device
from myTables.ConfigTables import UserTable

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.get()

    for account in users:
        print("Username is {} \nUser class is {}".format(account.username,
account.userclass))
```

For more information about using Junos PyEZ Tables, see the following topics:

- ["Using Junos PyEZ Operational Tables and Views that Parse Structured Output" on page 216](#)
- ["Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output" on page 259](#)
- ["Using Junos PyEZ Configuration Tables to Retrieve Configuration Data" on page 279](#)
- ["Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS" on page 290](#)

RELATED DOCUMENTATION

Understanding Junos PyEZ Tables and Views 187
Predefined Junos PyEZ Operational Tables (Structured Output) 189

Defining Junos PyEZ Operational Tables for Parsing Structured Output

IN THIS SECTION

- [Table Name | 202](#)
- [RPC Command \(rpc\) | 202](#)
- [RPC Default Arguments \(args\) | 203](#)
- [RPC Optional Argument Key \(args_key\) | 204](#)
- [Table Item \(item\) | 204](#)
- [Table Item Key \(key\) | 205](#)
- [Table View \(view\) | 209](#)

Junos PyEZ operational (op) Tables for structured output extract specific data from the XML output of an RPC 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 18 on page 200](#).

Table 18: Parameters in Junos PyEZ Op Tables for Structured Output

Table Parameter Name	Table Parameter	Description
Table name	-	User-defined identifier for the Table.

Table 18: Parameters in Junos PyEZ Op Tables for Structured Output (*Continued*)

Table Parameter Name	Table Parameter	Description
RPC command	rpc	Request tag name of the RPC to execute.
RPC default arguments	args	(Optional) Default command options and arguments for the RPC.
RPC optional argument key	args_key	<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 get() method argument list when you retrieve the operational data.</p>
Table item	item	<p>XPath expression relative to the top-level element within the <rpc-reply> 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	key	<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 <name> element as an identifier or for cases where composite keys are required.</p> <p>If the item uses the <name> element for the identifier, you can omit this property.</p>
Table View	view	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: '[afgxe][et]-*'
args_key: interface_name
item: physical-interface
view: EthPortView

```

The following sections discuss the different components of the Table:

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)

Junos PyEZ op Tables for structured output extract specific information from the XML output of an RPC. You must include the **rpc** property in the op Table definition to specify the RPC to execute on a device.

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 by using one the following methods:

- appending the | **display xml rpc** option after the command in the Junos OS CLI
- executing the Junos PyEZ **Device** instance **display_xml_rpc('command', format='text')** method
- finding the command and corresponding tag in the *Junos XML API Explorer*
- finding the command and corresponding tag in the *Junos XML API Operational Developer Reference*

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: '[afgxe][et]-*'
```

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 a CLI command takes 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.

```
eths = 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.

```
eths = 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 "[afgxe][et]-*" | 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 "[afgxe][et]-*" | 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 can 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

```

In addition, starting in Junos OS Release 2.3.0, a Table can define **key: Null** to indicate that a key is not required. In this case, the Table and View return the data as a simple dictionary rather than a nested one with keys.

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 Views for Junos PyEZ Operational Tables that Parse Structured Output | 209](#)

[Using Junos PyEZ Operational Tables and Views that Parse Structured Output | 216](#)

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Defining Junos PyEZ Configuration Tables | 260](#)

Defining Views for Junos PyEZ Operational Tables that Parse Structured Output

IN THIS SECTION

- [View Name | 211](#)
- [Fields \(fields\) | 212](#)
- [Groups \(groups\) and Field Groups \(fields_\) | 214](#)

Junos PyEZ operational (op) Tables for structured output select specific data from the XML output of an RPC executed on a device running Junos OS. A Table is associated with a View, which is used to access fields in the Table items and map them to user-defined Python variables. 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 variables 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. The keys defined in the View must be valid Python variable names.

Junos PyEZ Views, like Tables, are formatted using YAML. Views that parse structured output can include a number of parameters, which are summarized in [Table 19 on page 210](#).

Table 19: Parameters in Views for Junos PyEZ Op Tables for Structured Output

View Parameter Name	View Parameter	Description
View name	-	User-defined identifier for the View.
Field items	fields	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 field names must be valid Python variable names.
Field groups	fields_group	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. The field names must be valid Python variable names.
Groups	groups	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: '[afgxe][et]-*'
  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

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: EthPortViewEthPortView:
  # 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 field names must be valid Python variable names. 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
```



```

with Device(host='router.example.com') as dev:
    eths = EthPortTable(dev)
    eths.get()
    for item in eths:
        print (item.name)
        print (item.oper)
        print (item.admin)
        print (item.mtu)

```

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 field names must be valid Python variable names.

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 for Parsing Structured Output | 200](#)

[Using Junos PyEZ Operational Tables and Views that Parse Structured Output | 216](#)

Using Junos PyEZ Operational Tables and Views that Parse Structured Output

IN THIS SECTION

- [Retrieving Table Items | 216](#)
- [Accessing Table Items | 218](#)
- [Iterating Through a Table | 220](#)

Junos PyEZ operational (op) Tables for structured output extract specific data from the XML output of an RPC executed on a device 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

with Device(host='router.example.com') as dev:
    eths = EthPortTable(dev)
```

The following sections discuss how to then retrieve and manipulate the data:

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 "[afgxe][et]-*" media** command by default and extracts the **physical-interface** items from the output.

```
---
EthPortTable:
```



```
rpc: get-interface-information
args:
  media: True
  interface_name: '[afgxe][et]-*'
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

with Device(host='router.example.com') as dev:
    eths = EthPortTable(dev)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">eths.get()</fo:inline>
```

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 "[afgxe][et]-*". 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)
eths.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.

```
eths = EthPortTable(dev)
eths.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)
eths.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.

```
if '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 "[afgxe][et]-*" media` 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 junpr.junos import Device
from junpr.junos.op.ethport import EthPortTable

with Device(host='router.example.com') as dev:
    eths = EthPortTable(dev)
    eths.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 for Parsing Structured Output | 200](#)

[Defining Views for Junos PyEZ Operational Tables that Parse Structured Output | 209](#)

Defining Junos PyEZ Operational Tables for Parsing Unstructured Output

IN THIS SECTION

- [Summary of Parameters in Op Tables for Parsing Unstructured Output | 221](#)
- [Table Name | 223](#)
- [Command | 223](#)
- [Command Arguments \(args\) | 224](#)
- [Target FPC \(fpc\) | 225](#)
- [Table Item \(item\) | 226](#)
- [Table Item Key \(key\) | 230](#)
- [Selected Keys \(key_items\) | 232](#)
- [Section Title \(title\) | 233](#)
- [Field Delimiter \(delimiter\) | 234](#)
- [Eval Expression \(eval\) | 235](#)
- [Table View \(view\) | 237](#)

Junos PyEZ operational (op) Tables for unstructured output extract data from the text output of a CLI command executed on a device running Junos OS or a vty command executed on a given Flexible PIC Concentrator (FPC). The extracted data is then converted to JSON. This enables you to quickly retrieve and analyze operational state information for the device. Junos PyEZ op Tables for unstructured output are particularly useful when you need to parse command output that cannot be returned in a structured format such as XML.

This topic discusses the different components of the Table.

Summary of Parameters in Op Tables for Parsing Unstructured Output

Junos PyEZ Tables are formatted using YAML. Op Table definitions can include a number of required and optional parameters, which are summarized in *Table 1*.

Table 20: Parameters in Junos PyEZ Op Tables for Unstructured Output

Table Parameter Name	Table Parameter	Description
Table name	–	User-defined identifier for the Table.
Command	command	CLI or vty command to execute.
Command arguments	args	(Optional) When you define the command as a Jinja template, args is an associative array, or dictionary, of key-value pairs that map the variables in the command template to the default values used when the template is rendered.
Target FPC	target	(Optional) Flexible PIC Concentrator (FPC) on which to execute a vty command.
Table item	item	(Optional) String or regular expression that defines how to split the output into sections. These sections become the iterable reference for the associated View. Specify '*' to extract and match against the whole string rather than each line.
Table item key	key	(Optional) String or list of strings that define one or more keys that uniquely identify each Table item.
Selected keys	key_items	(Optional) List of one or more Table item keys for which to return data.
Section title	title	(Optional) String that selects the section of output containing the data to parse.
Field delimiter	delimiter	(Optional) Delimiter that defines how to split the data in command output comprised of key-value pairs. The extracted data is stored as key-value pairs in a dictionary.

Table 20: Parameters in Junos PyEZ Op Tables for Unstructured Output (*Continued*)

Table Parameter Name	Table Parameter	Description
Eval expression	eval	(Optional) Associative array, or dictionary, of one or more key-value pairs that map a user-defined key to a string containing a mathematical expression. When you retrieve the data, the expression is evaluated using the Python eval function. The key and the calculated value are added to the final data returned by the Table and View.
Table View	view	(Optional) View that is used to extract field data from the Table items.

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. The following example defines a Table named **ChassisFanTable**:

```
---
ChassisFanTable:
  command: show chassis fan
  key: fan-name
  view: ChassisFanView
```

Command

A Junos PyEZ op Table for unstructured output extracts data from the text output of a CLI or vty command. You must include the **command** property in the Table definition to specify the CLI command to execute on a device or the vty command to execute on a given FPC. You can define the command as a simple string or a Jinja template.

For example, the following Table executes the **show chassis fan** command on the device.

```
---
ChassisFanTable:
    command: show chassis fan
    key: fan-name
    view: ChassisFanView
```

The following Table executes the **show cmerror module brief** vty command on the target FPC.

```
---
CMEErrorTable:
    command: show cmerror module brief
    target: fpc1
    key: module
    view: CMEErrorView
```

When you define the command as a Jinja template, you must also supply the **args** parameter with a dictionary of key-value pairs that map the variables in the template to the values used when the template is rendered. For information about defining the command as a Jinja template, see *No Link Title*.

Command Arguments (args)

You can define the CLI or vty command for the **command** parameter using a Jinja template and substitute variables for the command arguments. When you use a Jinja template, you must also define the **args** parameter, which is a dictionary of key-value pairs that map the variable names in the Jinja template to the values used when the template is rendered. You can provide default values for the template variables in the Table definition, and you can also define the values in the Junos PyEZ application.

To define default values for the template variables, include the **args** parameter in the Table definition and map each template variable to its default value. The following Table defines a command using a Jinja template that has one variable, **protocol**. The **args** parameter defines the default value for **protocol**, which is used when you call the **get()** method in your script and do not provide an argument that overrides that default.

```
---
DdosPolicerStatsTable:
    command: show ddos policer stats {{ protocol }}args:
```



```

    protocol: ospf
    target: Null
    title: "DDOS Policer Statistics:"
    key: location
    view: DdosPolicerStatsView

```

Additionally, you can define the **args** argument as a dictionary in the Table's **get()** method to:

- define a value for any template variable that does not have a default value defined in the Table
- override the default value defined in the Table for one or more template variables

The following example executes the command in the previous Table using protocol 'bgp' instead of the default value of 'ospf'.

```

from jnpr.junos import Device
from jnpr.junos.command.pfe_ddos_policer import DdosPolicerStatsTable
from pprint import pprint
import json

with Device(host='router1.example.com') as dev:
    stats = DdosPolicerStatsTable(dev)
    stats.get(target='fpc0', <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/
Format" font-weight="bold">args={'protocol': 'bgp'}</fo:inline>)
    pprint(json.loads(stats.to_json()))

```

Target FPC (fpc)

Junos PyEZ op Tables can execute vty commands on a specific Flexible PIC Concentrator (FPC). When you use a vty command, the Table must include the **target** parameter to define the target FPC. You can set **target** to **Null** and force the user to specify the target FPC in the application, or you can set **target** to a default FPC, and the user can optionally override this value in the application.

The following Table executes the **show memory** vty command, but sets **target: Null**, which requires that the user supply the target FPC in the Junos PyEZ application:

```

---
FpcMemory:
    command: show memory
    target: Null

```



```

key: ID
key_items:
  - 0
  - 1
view: FPCMemoryView

```

The following Table executes the **show memory** vty command on FPC 1, unless the user overrides this value in the Junos PyEZ application.

```

---
FpcMemory:
  command: show memory
  target: fpc1
  key: ID
  key_items:
    - 0
    - 1
  view: FPCMemoryView

```

In the Junos PyEZ application, to define the target FPC or override the default FPC defined in the Table, set the **target** argument in the Table's **get()** method to the desired FPC, for example:

```

from jnpr.junos import Device
from jnpr.junos.command.fpc_memory import FpcMemory
from pprint import pprint
import json

with Device(host='router.example.com') as dev:
    stats = FpcMemory(dev)
    stats.get(target='fpc0')
    pprint(json.loads(stats.to_json()))

```

Table Item (item)

The optional Table **item** property is a string or regular expression that defines how to split the command output for parsing. If the output has similar, repeating sets of data, you can define **item** to match on and extract each iteration of the data. For example, **show interfaces** returns a similar set of data for many

interfaces. Alternatively, you can define **item: '*'** when you need to extract the data as a single block of text.

Consider the following output for the **show devices local vty** command:

```
TSEC Ethernet Device Driver: .le1, Control 0x4296c218, (1000Mbit)
HW reg base 0xff724000
  [0]: TxBD base 0x7853ce20, RxBD Base 0x7853d640
  [1]: TxBD base 0x7853d860, RxBD Base 0x7853e080
  [2]: TxBD base 0x7853e2a0, RxBD Base 0x785422c0
  [3]: TxBD base 0x785426e0, RxBD Base 0x78544700
Receive:
  185584608 bytes, 2250212 packets, 0 FCS errors, 0 multicast packets
  107271 broadcast packets, 0 control frame packets
  0 PAUSE frame packets, 0 unknown OP codes
  0 alignment errors, 0 frame length errors
  0 code errors, 0 carrier sense errors
  0 undersize packets, 0 oversize packets
  0 fragments, 0 jabbers, 0 drops
Receive per queue:
  [0]: 0 bytes, 0 packets, 0 dropped
        0 jumbo, 0 truncated jumbo
  [1]: 0 bytes, 0 packets, 0 dropped
        0 jumbo, 0 truncated jumbo
  [2]: 0 bytes, 0 packets, 0 dropped
        0 jumbo, 0 truncated jumbo
  [3]: 203586808 bytes, 2250219 packets, 0 dropped
        0 jumbo, 0 truncated jumbo
Transmit:
  288184646 bytes, 2038370 packets, 0 multicast packets
  106531 broadcast packets, 0 PAUSE control frames
  0 deferral packets, 0 excessive deferral packets
  0 single collision packets, 0 multiple collision packets
  0 late collision packets, 0 excessive collision packets
  0 total collisions, 0 drop frames, 0 jabber frames
  0 FCS errors, 0 control frames, 0 oversize frames
  0 undersize frames, 0 fragments frames
Transmit per queue:
  [0]: 10300254 bytes,          72537 packets
        0 dropped,              0 fifo errors
  [1]: 4474302 bytes,          106531 packets
        0 dropped,              0 fifo errors
```



```

[2]: 260203538 bytes,      1857137 packets
      0 dropped,          0 fifo errors
[3]:  199334 bytes,      2179 packets
      0 dropped,          0 fifo errors
TSEC status counters:
kernel_dropped:0, rx_large:0 rx_short: 0
rx_nonoctet: 0, rx_crcerr: 0, rx_overrun: 0
rx_bsy: 0,rx_babr:0, rx_trunc: 0
rx_length_errors: 0, rx_frame_errors: 0 rx_crc_errors: 0
rx_errors: 0, rx_ints: 2250110, collisions: 0
eberr:0, tx_babt: 0, tx_underrun: 0
tx_timeout: 0, tx_timeout: 0,tx_window_errors: 0
tx_aborted_errors: 0, tx_ints: 2038385, resets: 1

TSEC Ethernet Device Driver: .le3, Control 0x42979220, (1000Mbit)
HW reg base 0xff726000
[0]: TxBD base 0x78545720, RxBD Base 0x78545f40
[1]: TxBD base 0x78546160, RxBD Base 0x78546980
[2]: TxBD base 0x78546ba0, RxBD Base 0x7854abc0
[3]: TxBD base 0x7854afe0, RxBD Base 0x7854d000
Receive:
  0 bytes, 0 packets, 0 FCS errors, 0 multicast packets
  0 broadcast packets, 0 control frame packets
  0 PAUSE frame packets, 0 unknown OP codes
  0 alignment errors, 0 frame length errors
  0 code errors, 0 carrier sense errors
  0 undersize packets, 0 oversize packets
  0 fragments, 0 jabbers, 0 drops
Receive per queue:
[0]: 0 bytes, 0 packets, 0 dropped
      0 jumbo, 0 truncated jumbo
[1]: 0 bytes, 0 packets, 0 dropped
      0 jumbo, 0 truncated jumbo
[2]: 0 bytes, 0 packets, 0 dropped
      0 jumbo, 0 truncated jumbo
[3]: 0 bytes, 0 packets, 0 dropped
      0 jumbo, 0 truncated jumbo
Transmit:
  6817984 bytes, 106531 packets, 0 multicast packets
  106531 broadcast packets, 0 PAUSE control frames
  0 deferral packets, 0 excessive deferral packets
  0 single collision packets, 0 multiple collision packets

```



```

    0 late collision packets, 0 excessive collision packets
    0 total collisions, 0 drop frames, 0 jabber frames
    0 FCS errors, 0 control frames, 0 oversize frames
    0 undersize frames, 0 fragments frames
Transmit per queue:
  [0]:          0 bytes,          0 packets
        0 dropped,          0 fifo errors
  [1]:  4474302 bytes,    106531 packets
        0 dropped,          0 fifo errors
  [2]:          0 bytes,          0 packets
        0 dropped,          0 fifo errors
  [3]:          0 bytes,          0 packets
        0 dropped,          0 fifo errors
TSEC status counters:
kernel_dropped:0, rx_large:0 rx_short: 0
rx_nonoctet: 0, rx_crcerr: 0, rx_overrun: 0
rx_bsy: 0,rx_babr:0, rx_trunc: 0
rx_length_errors: 0, rx_frame_errors: 0 rx_crc_errors: 0
rx_errors: 0, rx_ints: 0, collisions: 0
eberr:0, tx_babt: 0, tx_underrun: 0
tx_timeout: 0, tx_timeout: 0,tx_window_errors: 0
tx_aborted_errors: 0, tx_ints: 106531, resets: 1

```

The following Table extracts each section of the output that starts with **TSEC Ethernet Device Driver:**. In this case, the value for **key: name** is derived from the capturing group in the regular expression defined in **item**.

```

---
DevicesLocalTable:
  command: show devices local
  target: fpcl
  item: 'TSEC Ethernet Device Driver: (\.?\w+),'
  key: name
  view: DevicesLocalView

DevicesLocalView:
  fields:
    TSEC_status_counters: _TSECStatusCountersTable
    receive_counters: _ReceiveTable
    transmit_per_queue: _TransmitQueueTable

```


You can also define **item** as an asterisk (*) if you want to match against the entire section of output instead of matching each line. When you include **item: '*'**, in most cases, you must also include the **title** parameter to specify the heading for the section of output to extract. Extracting the output using **item: '*'** is useful when you include the **regex** parameter in a View and want each expression to match against the entire text string. Otherwise, the **regex** expressions are combined and matched against each line.

The following Table extracts the text block under the heading **Receive:** and matches each regular expression against the entire text string:

```
_ReceiveTable:
  item: '*'
  title: 'Receive:'
  view: _ReceiveView

_ReceiveView:
  regex:
    bytes: '(\d+) bytes'
    packets: '(\d+) packets'
    FCS_errors: '(\d+) FCS errors'
    broadcast_packets: '(\d+) broadcast packets'
```

Table Item Key (key)

The optional **key** property defines the output fields that are used to uniquely identify a Table item. You can identify a Table item using a single key or a list of keys. If the Table and View return iterative data, the **key** value must reference a variable or field name defined in the View.

Consider the following **show chassis fan** output:

Item	Status	RPM	Measurement
Fan 1	OK	5280	Spinning at normal speed
Fan 2	OK	5280	Spinning at normal speed
Fan 3	OK	5280	Spinning at normal speed
Fan 4	OK	5280	Spinning at normal speed
Fan 5	OK	5280	Spinning at normal speed

The following Table defines the Table item key as **fan-name**, which maps to the value under the **Item** column in the output.

```
ChassisFanTable:
    command: show chassis fan
    key: fan-name
    view: ChassisFanView
ChassisFanView:
    columns:
        fan-name: Item
        fan-status: Status
        fan-rpm: RPM
        fan-measurement: Measurement
```

When you retrieve and print the data in the Junos PyEZ application, each item in the resulting dictionary uses this field's value as its key.

```
dict_keys(['Fan 1', 'Fan 2', 'Fan 3', 'Fan 4', 'Fan 5'])
```

You can also define **key** as a list to identify a Table item using a composite key. For example:

```
---
FPCIPv4AddressTable:
    command: show ipv4 address
    target: fpc1
    key:
        - name
        - addr
    view: FPCIPv4AddressView
FPCIPv4AddressView:
    columns:
        index: Index
        addr: Address
        name: Name
```


Selected Keys (key_items)

The **key** parameter defines the output fields that uniquely identify a Table item. When you include the **key** parameter in a Table, you can use the optional **key_items** parameter to only return data for certain key values. **key_items** defines the key or list of keys of the Table items for which to retrieve data. You can define the **key_items** parameter in the Table definition or in the Junos PyEZ application.

Consider the following **show chassis fan** output:

Item	Status	RPM	Measurement
Fan 1	OK	5280	Spinning at normal speed
Fan 2	OK	5280	Spinning at normal speed
Fan 3	OK	5280	Spinning at normal speed
Fan 4	OK	5280	Spinning at normal speed
Fan 5	OK	5280	Spinning at normal speed

The following Table defines the Table item key as **fan-name** and only retrieves the data for the Table item with a key value equal to **Fan 1**.

```
ChassisFanTable:
  command: show chassis fan
  key: fan-name
  key_items:- Fan 1
  view: ChassisFanView
ChassisFanView:
  columns:
    fan-name: Item
    fan-status: Status
    fan-rpm: RPM
    fan-measurement: Measurement
```

In the Junos PyEZ application, to define the **key_items** to return or to override the **key_items** defined in the Table, set the **key_items** argument in the Table's **get()** method to a list or tuple of the desired items, for example:

```
from jnpr.junos import Device
from jnpr.junos.command.chassis_fan import ChassisFanTable
from pprint import pprint
import json
```



```

with Device(host='router.example.com') as dev:
    fans = ChassisFanTable(dev)
    fans.get(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">key_items=['Fan 1', 'Fan 2']</fo:inline>)
    pprint(json.loads(fans.to_json()))

```

```

user@host:~$ python3 junos-pyez-get-fan-data.py

```

```

{'Fan 1': {'fan-measurement': 'Spinning at normal speed',
           'fan-name': 'Fan 1',
           'fan-rpm': 5280,
           'fan-status': 'OK'},
 'Fan 2': {'fan-measurement': 'Spinning at normal speed',
           'fan-name': 'Fan 2',
           'fan-rpm': 5280,
           'fan-status': 'OK'}}

```

Section Title (title)

Tables can include the optional **title** parameter to define the starting point for a section in the command output from which to extract and parse the data. When the Table defines **item: ***, you must include **title** to specify the heading for the section of output to extract

For example, consider the following command output, which is enclosed within a larger set of output:

```

...
TSEC status counters:
kernel_dropped:0, rx_large:0 rx_short: 0
rx_nonoctet: 0, rx_crcerr: 0, rx_overrun: 0
rx_bsy: 0,rx_babr:0, rx_trunc: 0
rx_length_errors: 0, rx_frame_errors: 0 rx_crc_errors: 0
rx_errors: 0, rx_ints: 2250110, collisions: 0
eberr:0, tx_babt: 0, tx_underrun: 0
tx_timeout: 0, tx_timeout: 0,tx_window_errors: 0
tx_aborted_errors: 0, tx_ints: 2038385, resets: 1
...

```


The following Table uses the **title** parameter to extract and parse data from the **TSEC status counters** section of the output. In this case, the Table defines **item** as '*', which considers the data as a single text string.

```
_TSECStatusCountersTable:
  item: '*'
  title: 'TSEC status counters:'
  view: _TSECStatusCountersView

_TSECStatusCountersView:
  regex:
    kernel_dropped: 'kernel_dropped:(\d+)'
    rx_large: 'rx_large:(\d+)'
```

Field Delimiter (delimiter)

Some operational commands return output comprised of only key-value pairs. If you want to simply retrieve the entire set of data and extract each key-value pair, you can use the **delimiter** parameter to define how to split each data pair instead of defining a separate View. Junos PyEZ uses the delimiter to split the data at the specified boundary and stores each key-value pair in a dictionary.

Consider the following output for the **show link stats vty** command.

```
PPP LCP/NCP: 0
HDLC keepalives: 0
RSVP: 0
ISIS: 0
OSPF Hello: 539156
OAM: 0
BFD: 15
UBFD: 0
LMI: 0
LACP: 0
ETHOAM: 0
SYNCE: 0
PTP: 0
L2TP: 0
LNS-PPP: 0
ARP: 4292
```



```
ELMI: 0
VXLAN MRESOLVE: 0
Unknown protocol: 42
```

The following Table defines the delimiter as the colon (:) character:

```
---
FPCLinkStatTable:
  command: show link stats
  target: fpcl
  delimiter: ":"
```

When you retrieve the data in the Junos PyEZ application, the Table splits each line of output at the delimiter and stores the key-value pairs in a dictionary.

```
{'ARP': 4292, 'ELMI': 0, 'SYNCE': 0, 'ISIS': 0, 'BFD': 15, 'PPP LCP/NCP': 0,
'OAM': 0, 'ETHOAM': 0, 'LACP': 0, 'LMI': 0, 'Unknown protocol': 42,
'UBFD': 0, 'L2TP': 0, 'HDLC keepalives': 0, 'LNS-PPP': 0,
'OSPF Hello': 539156, 'RSVP': 0, 'VXLAN MRESOLVE': 0, 'PTP': 0}
```

Eval Expression (eval)

You can use the optional **eval** parameter to add or modify key-value pairs in the final data returned by the Table and View. The **eval** parameter maps a key name to a string containing a mathematical expression that gets evaluated by the Python **eval** function. You can include the **eval** parameter in both Tables and Views, and **eval** can define and calculate multiple values.

When you use **eval** in a Table, it references the full data dictionary for the calculation, and the key and calculated value are added as a single additional item to the dictionary. When you use **eval** in a View, the expression is calculated on each iteration of the data, and the calculated value is added to the data for that iteration. If the **eval** key name matches a key defined in the View, **eval** replaces the value of that key with the calculated value. If the **eval** key name does not match a key defined in the View, **eval** adds the new key and calculated value to the data.

The **eval** expression can reference the **data** dictionary returned by the View. The expression must reference **data** using a Jinja template variable, so that Junos PyEZ can replace the variable with the dictionary when the expression is evaluated.

The following example uses **eval** in the Table definition to include a single additional entry in the data dictionary. The added item's key is **cchip_errors_from_lkup_chip**, and its value is the sum of the number of interrupts.

```
---
CChipLiInterruptStatsTable:
  command: show xmchip {{ chip_instance }} li interrupt-stats
  target: NULL
  args:
    chip_instance: 0
  key:
    - li_block
    - name
    eval:cchip_errors_from_lkup_chip: "reduce(lambda x,y: x+y, [v['interrupts']
for k,v in {{ data }}.items()])"
  view: CChipLiInterruptStatsView

CChipLiInterruptStatsView:
  columns:
    li_block: LI Block
    name: Interrupt Name
    interrupts: Number of Interrupts
    last_occurance: Last Occurrence
```

You can also define **eval** in the Table to calculate and add multiple key-values pairs, for example:

```
---
CChipDRDErrTable:
  command: show xmchip {{ instance }} drd error-stats
  args:
    instance: 0
  target: NULL
  key: Interrupt Name
  item: '*'
  eval:
    cchip_drd_wan_errors: sum([v['interrupt_count'] for k, v in
    {{ data }}.items() if k.endswith('_0')])
    cchip_drd_fab_errors: sum([v['interrupt_count'] for k, v in
    {{ data }}.items() if k.endswith('_1')])
  view: CChipDRDErrView
```



```

CChipDRDErrView:
    regex:
        cchip_drd_wan_timeouts: 'Total WAN reorder ID timeout errors:\s+(\d+)'
        cchip_drd_fab_timeouts: 'Total fabric reorder ID timeout errors:\s+(\d+)'
    columns:
        interrupt_name: Interrupt Name
        interrupt_count: Number of Interrupts
    filters:
        - interrupt_count

```

For information about using **eval** in a View, see ["Eval Expression \(eval\)" on page 243](#).

Table View (view)

The **view** property associates the Table definition with a particular View. A View defines how the Table output should be parsed and maps your user-defined Python variable names to output fields in the selected Table items. You can customize the View to only select certain fields from the Table items.

If the output consists of only key-value pairs, you can use the Table's **delimiter** parameter to extract the data and store the key-value pairs in a dictionary. In this case, you do not need to define a separate View.

RELATED DOCUMENTATION

[Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output | 238](#)

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output | 259](#)

Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output

IN THIS SECTION

- [Summary of Parameters in Views for Parsing Unstructured Output | 238](#)
- [View Name | 239](#)
- [columns | 240](#)
- [Eval Expression \(eval\) | 243](#)
- [exists | 245](#)
- [fields | 246](#)
- [filters | 251](#)
- [regex | 253](#)

Junos PyEZ operational (op) Tables for unstructured output extract data from the text output of a CLI command executed on a device running Junos OS or a vty command executed on a given Flexible PIC Concentrator (FPC). A Table is associated with a *View*, which is used to access fields in the Table items and map them to user-defined Python variables. 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 variables to data 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. The keys defined in the View must be valid Python variable names.

This topic discusses the different components of the View.

Summary of Parameters in Views for Parsing Unstructured Output

Junos PyEZ Views, like Tables, are formatted using YAML. Views that parse unstructured output can include a number of parameters, which are summarized in *Table 1*.

Table 21: Parameters in Views for Junos PyEZ Op Tables for Unstructured Output

View Parameter	Description
View Name	User-defined identifier for the View.
columns	(Optional) List of column titles in the command output.
eval	(Optional) Associative array, or dictionary, of one or more key-value pairs that map a user-defined key to a string containing a mathematical expression. For each iteration of the data, the expression is evaluated using the Python eval function. The key and the calculated value are added to the data corresponding to that iteration.
exists	(Optional) Associative array, or dictionary, of key-value pairs that map a user-defined key to a string. If the string is present in the output, the variable is set to True, otherwise, the variable is set to False.
fields	(Optional) Associative array, or dictionary, of key-value pairs that map a user-defined key to the name of a nested Table that parses a specific section of the command output.
filters	(Optional) List of one or more keys defined under columns . The final set of data only includes data from the selected columns.
regex	(Optional) List of regular expressions to match desired content.

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.

The following example defines a View named **ChassisFanView**, which is referenced by the Table's **view** parameter:

```
---
ChassisFanTable:
  command: show chassis fan
  key: fan-name
  view: ChassisFanViewChassisFanView:
  columns:
    fan-name: Item
    fan-status: Status
    fan-rpm: RPM
    fan-measurement: Measurement
```

columns

You can use the **columns** parameter in a View to extract and parse command output that is formatted in rows and columns.

Consider the following **show ospf neighbor** command output:

Address	Interface	State	ID	Pri	Dead
198.51.100.2	ge-0/0/0.0	Full	192.168.0.2	128	37
198.51.100.6	ge-0/0/1.0	Full	192.168.0.3	128	34

To extract the data, include the **columns** parameter in the View, and map your Python variable name to the column name. The application stores the key and the value extracted from the command output for that column as a key-value pair in the dictionary for the given item.

The following View extracts the data from each column in the **show ospf neighbor** command output:

```
---
OspfNeighborTable:
  command: show ospf neighbor
  key: Address
  view: OspfNeighborView
OspfNeighborView:
  columns:
    neighbor_address: Address
```



```

interface: Interface
neighbor_state: State
neighbor_id: ID
neighbor_priority: Pri
activity_timer: Dead

```

When you retrieve and print the data in the Junos PyEZ application, the data for each neighbor includes the column keys and corresponding data.

```

from jnpr.junos import Device
from jnpr.junos.command.ospf_neighbor import OspfNeighborTable
from pprint import pprint
import json

with Device(host='router1.example.com') as dev:
    stats = OspfNeighborTable(dev)
    stats.get()
    pprint(json.loads(stats.to_json()))

```

```

user@host:~$ python3 junos-pyez-ospf-neighbors.py

```

```

{'198.51.100.2': {'activity_timer': 39,
                  'interface': 'ge-0/0/0.0',
                  'neighbor_address': '198.51.100.2',
                  'neighbor_id': '192.168.0.2',
                  'neighbor_priority': 128,
                  'neighbor_state': 'Full'},
 '198.51.100.6': {'activity_timer': 36,
                  'interface': 'ge-0/0/1.0',
                  'neighbor_address': '198.51.100.6',
                  'neighbor_id': '192.168.0.3',
                  'neighbor_priority': 128,
                  'neighbor_state': 'Full'}}

```

To filter the data to only include data from selected columns, include the **filters** parameter in the View. For more information, see *No Link Title*.

Some command output includes column titles that span multiple lines, for example:

```
FI interrupt statistics
-----

-----
Stream Total RLIM   Total      Cell timeout Total Reorder Total cell   Total number
      request   PT/MALLOC Ignored      cell timeout drops in    of times
      counter   Usemeter      errors      secure mode entered into
      saturation Drops                                secure mode
-----
36      0          0          1          1          0          0
128     0          0          1          49         0          0
142     0          0          1          53         0          0
-----
...
```

To define a multiline column title in a View, set the column key element equal to a list of the words in each row for that title. The following View defines the columns for the previous command output:

```
CChipFiStatsTable:
  command: show mqss {{ chip_instance }} fi interrupt-stats
  target: fpc8
  args:
    chip_instance: 0
  key: Stream
  view: CChipFiStatsView

CChipFiStatsView:
  columns:
    stream: Stream
    req_sat:
      - Total RLIM
      - request
      - counter
      - saturation
    cchip_fi_malloc_drops:
      - Total
      - PT/MALLOC
      - Usemeter
      - Drops
```



```

cell_timeout_ignored:
  - Cell timeout
  - Ignored
cchip_fi_cell_timeout:
  - Total Reorder
  - cell timeout
  - errors
drops_in_secure:
  - Total cell
  - drops in
  - secure mode
times_in_secure:
  - Total number
  - of times
  - entered into
  - secure mode

```

Eval Expression (eval)

You can use the optional **eval** parameter to add or modify key-value pairs in the final data returned by the Table and View. The **eval** parameter maps a key name to a string containing a mathematical expression that gets evaluated by the Python **eval** function. You can include the **eval** parameter in both Tables and Views, and **eval** can define and calculate multiple values.

When you use **eval** in a Table, it references the full data dictionary for the calculation, and the key and calculated value are added as a single additional item to the dictionary. When you use **eval** in a View, the expression is calculated on each iteration of the data, and the calculated value is added to the data for that iteration. If the **eval** key name matches a key defined in the View, **eval** replaces the value of that key with the calculated value. If the **eval** key name does not match a key defined in the View, **eval** adds the new key and calculated value to the data.

The **eval** expression can reference the **data** dictionary returned by the View. The expression must reference **data** using a Jinja template variable, so that Junos PyEZ can replace the variable with the dictionary when the expression is evaluated.

The following example uses **eval** in the View definition. The **cpu** entry modifies the existing value of the **cpu** field for each item in the data dictionary, and the **max** entry creates a new key-value pair for each item in the data dictionary.

```
---
FPCThread:
  command: show threads
  target: Null
  key: Name
  view: FPCThreadView

FPCThreadView:
  columns:
    pid-pr: PID PR
    state: State
    name: Name
    stack: Stack Use
    time: Time (Last/Max/Total)
    cpu: cpueval:
    cpu: "{{ { cpu } }}"[:-1]"
    max: "({ { time } }).split('/')[1]"
```

Consider the following sample output for the **show threads** vty command:

PID	PR	State	Name	Stack Use	Time (Last/Max/Total)	cpu
1	H	asleep	Maintenance	680/32768	0/5/5 ms	0%
2	L	running	Idle	1452/32768	0/22/565623960 ms	80%
3	H	asleep	Timer Services	1452/32768	0/7/1966 ms	0%
...						

The View's **eval** parameter modifies each **cpu** entry to omit the percent sign (%). As a result, the data includes '0' instead of '0%'. In addition, it adds a new key, **max**, and its calculated value for each item.

```
'Maintenance': { 'cpu': '0',
                  'max': '5',
                  'name': 'Maintenance',
                  'pid-pr': '1 H',
                  'stack': '680/32768',
                  'state': 'asleep',
                  'time': '0/5/5 ms'},
```



```
'Timer Services': {'cpu': '0',
                   'max': '7',
                   'name': 'Timer Services',
                   'pid-pr': '3 H',
                   'stack': '1452/32768',
                   'state': 'asleep',
                   'time': '0/7/1966 ms'},
...

```

For examples that use **eval** in the Table definition, see ["Eval Expression \(eval\)" on page 235](#).

exists

You can use the optional **exists** parameter in a View to indicate if a string is present in the command output. **exists** is a dictionary of key-value pairs that map a user-defined Python variable name to the string to match in the command output. If the string is present in the output, the variable is set to **True**. Otherwise, the variable is set to **False**.

Consider the **show host_loopback status-summary** vty command output.

```
SENT: Ukern command: show host_loopback status-summary

Host Loopback Toolkit Status Summary:

No detected wedges

No toolkit errors

```

The following Table defines **exists** to test if the command output includes a **No detected wedges** string or a **No toolkit errors** string:

```
---
HostlbStatusSummaryTable:
  command: show host_loopback status-summary
  target: fpc1
  view: HostlbStatusSummaryView

HostlbStatusSummaryView:
  exists:

```



```
no_detected_wedges: No detected wedges
no_toolkit_errors: No toolkit errors
```

When you use the Table and View to test for the strings and print the resulting values in your Junos PyEZ application, both variables are set to **True** in this case.

```
{'no_detected_wedges': True, 'no_toolkit_errors': True}
```

fields

Command output can be lengthy and complex, and you might need different logic to parse different sections of the output. In some cases, you cannot adequately parse the command output using a single Table and View. To parse this type of output, you can include the optional **fields** parameter in the View. **fields** is a dictionary of key-value pairs that map a user-defined key to the name of a nested Table that selects a specific section of the command output. Each nested Table can reference its own View, which is used to parse the data selected by that Table.

Consider the **show xmchip 0 pt stats** vty command output, which has two different sections of data:

```
SENT: Ukern command: show xmchip 0 pt stats

WAN PT statistics (Index 0)
-----

PCT entries used by all WI-1 streams      : 0
PCT entries used by all WI-0 streams      : 0
PCT entries used by all LI streams        : 0
CPT entries used by all multicast packets : 0
CPT entries used by all WI-1 streams      : 0
CPT entries used by all WI-0 streams      : 0
CPT entries used by all LI streams        : 0

Fabric PT statistics (Index 1)
-----

PCT entries used by all FI streams        : 0
PCT entries used by all WI (Unused) streams : 0
PCT entries used by all LI streams        : 0
```



```

CPT entries used by all multicast packets      : 0
CPT entries used by all FI streams             : 0
CPT entries used by all WI (Unused) streams    : 0
CPT entries used by all LI streams             : 0

```

The following **XMChipStatsView** View uses the **fields** parameter to define two additional Tables, which are used to parse the two different sections of the command output. The **_WANPTStatTable** and **_FabricPTStatTable** Tables extract the data from the **WAN PT statistics** and the **Fabric PT statistics** sections, respectively. In this case, the Tables use the **delimiter** parameter to extract and split the data, so they do not need to reference a separate View.

```

XMChipStatsTable:
  command: show xmchip 0 pt stats
  target: fpc1
  view: XMChipStatsView

XMChipStatsView:
  fields:
    wan_pt_stats: _WANPTStatTable
    fabric_pt_stats: _FabricPTStatTable

_WANPTStatTable:
  title: WAN PT statistics (Index 0)
  delimiter: ":"

_FabricPTStatTable:
  title: Fabric PT statistics (Index 1)
  delimiter: ":"

```

When you retrieve and print the data in the Junos PyEZ application, each key defined under **fields** contains the data selected and parsed by the corresponding Table.

```

{'fabric_pt_stats': {'CPT entries used by all FI streams': 0,
                    'CPT entries used by all LI streams': 0,
                    'CPT entries used by all WI (Unused) streams': 0,
                    'CPT entries used by all multicast packets': 0,
                    'PCT entries used by all FI streams': 0,
                    'PCT entries used by all LI streams': 0,
                    'PCT entries used by all WI (Unused) streams': 0},
 'wan_pt_stats': {'CPT entries used by all LI streams': 0,

```



```
'CPT entries used by all WI-0 streams': 0,
'CPT entries used by all WI-1 streams': 0,
'CPT entries used by all multicast packets': 0,
'PCT entries used by all LI streams': 0,
'PCT entries used by all WI-0 streams': 0,
'PCT entries used by all WI-1 streams': 0}}
```

As another example, consider the **show ttp statistics vty** command output:

TTP Statistics:

	Receive	Transmit
	-----	-----
L2 Packets	4292	1093544
L3 Packets	542638	0
Drops	0	0
Netwk Fail	0	0
Queue Drops	0	0
Unknown	0	0
Coalesce	0	0
Coalesce Fail	0	0

TTP Transmit Statistics:

	Queue 0	Queue 1	Queue 2	Queue 3
	-----	-----	-----	-----
L2 Packets	1093544	0	0	0
L3 Packets	0	0	0	0

TTP Receive Statistics:

	Control	High	Medium	Low	Discard
	-----	-----	-----	-----	-----
L2 Packets	0	0	4292	0	0
L3 Packets	0	539172	3466	0	0
Drops	0	0	0	0	0
Queue Drops	0	0	0	0	0
Unknown	0	0	0	0	0
Coalesce	0	0	0	0	0
Coalesce Fail	0	0	0	0	0

TTP Receive Queue Sizes:

```
Control Plane : 0 (max is 4473)
High          : 0 (max is 4473)
Medium        : 0 (max is 4473)
```



```

Low                : 0 (max is 2236)

TTP Transmit Queue Size: 0 (max is 6710)

```

The **FPCTTPStatsView** View uses the **fields** parameter to reference multiple nested Tables, which extract the data in the different sections of the output. Each Table references its own View or uses the **delimiter** parameter to parse the data in that section.

```

---
FPCTTPStatsTable:
  command: show ttp statistics
  target: fpc2
  view: FPCTTPStatsView

FPCTTPStatsView:
  fields:
    TTPStatistics: _FPCTTPStatisticsTable
    TTPTransmitStatistics: _FPCTTPTransmitStatisticsTable
    TTPReceiveStatistics: _FPCTTPReceiveStatisticsTable
    TTPQueueSizes: _FPCTTPQueueSizesTable

_FPCTTPStatisticsTable:
  title: TTP Statistics
  view: _FPCTTPStatisticsView
_FPCTTPStatisticsView:
  columns:
    rcvd: Receive
    tras: Transmit

_FPCTTPTransmitStatisticsTable:
  title: TTP Transmit Statistics
  view: _FPCTTPTransmitStatisticsView
_FPCTTPTransmitStatisticsView:
  columns:
    queue0: Queue 0
    queue1: Queue 1
    queue2: Queue 2
    queue3: Queue 3
  filters:
    - queue2

_FPCTTPReceiveStatisticsTable:

```



```

        'rcvd': 0,
        'tras': 173},
    'Queue Drops': {'name': 'Queue Drops',
        'rcvd': 0,
        'tras': 0},
    'Unknown': {'name': 'Unknown', 'rcvd': 0, 'tras': 0}},
    'TTPTransmitStatistics': {'L2 Packets': {'queue2': 0},
        'L3 Packets': {'queue2': 0}}}
```

filters

The **columns** parameter extracts data from command output that is formatted in rows and columns. When you include the **columns** parameter in a View, you can optionally include the **filters** parameter to filter which column data is included in the final output. The **filters** parameter defines a list of one or more keys defined under **columns**. The final set of data only includes data from the selected columns. You can provide default filters in the View definition, and you can also define or override filter values in the Junos PyEZ application.

Consider the **show ospf neighbor** command output:

Address	Interface	State	ID	Pri	Dead
198.51.100.2	ge-0/0/0.0	Full	192.168.0.2	128	37
198.51.100.6	ge-0/0/1.0	Full	192.168.0.3	128	34

In the following View, the **columns** parameter defines keys for all of the columns in the corresponding command output, but the **filters** parameter only includes the data from the **Address** and **State** columns in the data dictionary.

```

---
OspfNeighborFiltersTable:
    command: show ospf neighbor
    key: Address
    view: OspfNeighborFiltersView
OspfNeighborFiltersView:
    columns:
        neighbor_address: Address
        interface: Interface
        neighbor_state: State
        neighbor_id: ID
```



```

neighbor_priority: Pri
activity_timer: Dead
filters:
  - neighbor_address
  - neighbor_state

```

The following Junos PyEZ code first calls `get()` without any arguments, which retrieves the data using the default list of filters defined in the View. The second call to `get()` includes the `filters` argument, which overrides the filter list defined in the View.

```

from jnpr.junos import Device
from Tables.show_ospf_neighbor_filter import OspfNeighborFiltersTable
from pprint import pprint
import json

with Device(host='router1.example.com') as dev:
    stats = OspfNeighborFiltersTable(dev)

    stats.get()
    pprint(json.loads(stats.to_json()))
    print('\n')

    stats.get(<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">filters=['neighbor_address', 'neighbor_id', 'neighbor_state']</
fo:inline>)
    pprint(json.loads(stats.to_json()))

```

When you execute the application, the first call to `get()` uses the filters defined in the View, and the second call uses the filters defined in the call, which override those defined in the View.

```

user@host:~$ python3 junos-pyez-ospf-filters.py

```

```

{'198.51.100.2': {'neighbor_address': '198.51.100.2', 'neighbor_state': 'Full'},
 '198.51.100.6': {'neighbor_address': '198.51.100.6', 'neighbor_state': 'Full'}}

{'198.51.100.2': {'neighbor_address': '198.51.100.2',
                  'neighbor_id': '192.168.0.2',
                  'neighbor_state': 'Full'},
 '198.51.100.6': {'neighbor_address': '198.51.100.6',

```



```
'neighbor_id': '192.168.0.3',
'neighbor_state': 'Full'}}
```

regex

You can use the optional **regex** parameter in a View to match and extract specific fields in the command output. **regex** is a dictionary that maps keys to regular expressions. If the corresponding Table does not define **item: '*'**, Junos PyEZ combines the regular expressions and matches the result against each line of output. However, if the Table defines **item: '*'** to extract the data as a single text string, Junos PyEZ instead matches each individual regular expression against the entire text string.

The capturing group defined in the regular expression determines the data that is extracted from the field and stored in the data dictionary. If you define a capturing group, only the value for that group is stored in the data. Otherwise, Junos PyEZ stores the value that matches the full expression. For example, **(d+.d+)** retrieves and stores a float value from the string search expression, whereas **(d+).d+** only stores the integer portion of the data. If you define multiple groups, only the value for the first group is stored.

Junos PyEZ leverages the **pyparsing** module to define a number of built-in keywords that you can use in place of a regular expression. *Table 2* lists the keyword, a brief description, and the corresponding expression, where **pp** is derived from **import pyparsing as pp**.

Table 22: regex Parameter Keywords

Keyword	Description	Expression
hex_numbers	Word containing only hexadecimal characters	<pre>hex_numbers = pp.OneOrMore(pp.Word(pp.nums, min=1)) & pp.OneOrMore(pp.Word('abcdefABCDEF', min=1))</pre>
numbers	Word consisting of an integer or float value	<pre>numbers = (pp.Word(pp.nums) + pp.Optional(pp.Literal('.') + pp.Word(pp.nums))).setParseAction(lambda i: ''.join(i))</pre>

percentage	Word composed of digits and a trailing percentage sign (%)	<code>percentage = pp.Word(pp.nums) + pp.Literal('%')</code>
printables	One or more words composed of printable characters (any non-whitespace characters)	<code>printables = pp.OneOrMore(pp.Word(pp.printables))</code>
word	Word composed of alpha or alphanumeric characters	<code>word = pp.Word(pp.alphanums) pp.Word(pp.alphas)</code>
words	One or more word strings	<code>words = (pp.OneOrMore(word)).setParseAction(lambda i: ' '.join(i))</code>

Consider the following **show icmp statistics** command output. Each section of output is under a specific section title, and the data consists of a number and one or more words.

```
ICMP Statistics:
    0 requests
    0 throttled
    0 network unreachablees
    0 ttl expired
    0 redirects
    0 mtu exceeded
    0 source route denials
    0 filter prohibited
    0 other unreachablees
```



```

    0 parameter problems
    0 ttl captured
    0 icmp/option handoffs
    0 igmp v1 handoffs
    0 tag te requests
    0 tag te to RE

ICMP Errors:
    0 unknown unreachable
    0 unsupported ICMP type
    0 unprocessed redirects
    0 invalid ICMP type
    0 invalid protocol
    0 bad input interface
    0 bad route lookup
    0 bad nh lookup
    0 bad cf mtu
    0 runts

ICMP Discards:
    0 multicasts
    0 bad source addresses
    0 bad dest addresses
    0 IP fragments
    0 ICMP errors
    0 unknown originators

ICMP Debug Messages:
    0 throttled

ICMP Rate Limit Settings:
    500 pps per iff
    1000 pps total

```

To parse the previous output, the main View defines **fields**, which references nested Tables and Views that parse each section of output. The nested Views define the **regex** parameter to match against the data extracted by the corresponding Table.

```

---
ICMPStatsTable:
  command: show icmp statistics
  target: fpcl

```



```

view: ICMPStatsView

ICMPStatsView:
  fields:
    discards: _ICMPDiscardsTable
    errors: _ICMPErrorsTable
    rate: _ICMPRateTable

  _ICMPDiscardsTable:
    title: ICMP Discards
    key: name
    view: _ICMPDiscardsView

  _ICMPDiscardsView:
    regex:
      value: \d+
      name: '(\w+(\s\w+)*)'

  _ICMPErrorsTable:
    title: ICMP Errors
    key: name
    view: _ICMPErrorsView

  _ICMPErrorsView:
    regex:
      error: numbers
      name: words

  _ICMPRateTable:
    title: ICMP Rate Limit Settings
    key: name
    view: _ICMPRateView

  _ICMPRateView:
    regex:
      rate: numbers
      name: words

```

For example, the **_ICMPDiscardsTable** Table selects the data under the **ICMP Discards** section in the command output. The **_ICMPDiscardsView** View defines two keys, **value** and **name**, which map to regular expressions. **value** matches one or more digits, and **name** matches one or more words. Because

the Table does not define **item: '*'**, the regular expressions are combined and matched against each line of data in that section.

```
_ICMPDiscardsTable:
  title: ICMP Discards
  key: name
  view: _ICMPDiscardsView

_ICMPDiscardsView:
  regex:
    value: \d+
    name: '(\w+(\s\w+)*)'
```

The **_ICMPErrorsTable** Table selects the data under the **ICMP Errors** section in the command output. The **_ICMPErrorsView** View defines the **error** and **name** keys and uses the built-in keywords **numbers** and **words** in place of explicitly defining regular expressions.

```
_ICMPErrorsTable:
  title: ICMP Errors
  key: name
  view: _ICMPErrorsView

_ICMPErrorsView:
  regex:
    error: numbers
    name: words
```

If the Table defines **item: '*'**, the extracted data is considered to be one text string. In this case, each regular expression in the corresponding view matches against the entire string.

Consider the **show ithrottle id 0** command output.

```
SENT: Ukern command: show ithrottle id 0

ID  Usage %  Cfg State  Oper State  Name
--  -
0   50.0     1          1          TOE ithrottle

Throttle Times:                In hptime ticks  In ms
                                -
Timer Interval                  333333          5.000
```


Allowed time	166666	2.500
Allowed excess	8333	0.125
Start time	488655082	n/a
Run time this interval	0	0.000
Deficit	0	0.000
Run time max	17712	0.266
Run time total	144154525761	2162317

Min Usage Perc: 25.0

Max Usage Perc: 50.0

AdjustUsageEnable: 1

Throttle Stats:

Starts	: 65708652
Stops	: 65708652
Checks	: 124149442
Enables	: 0
Disables	: 0
AdjUp	: 6
AdjDown	: 4

The following Table uses **item: '*'** to extract the data as a single string. The View's **regex** parameter defines three regular expressions. Each regex pattern is matched against the entire string. Because the regular expressions define capturing groups, Junos PyEZ only stores the data that matches the group.

```
IthrottleIDTable:
  command: show ithrottle id {{ id }}
  args:
    id: 0
  item: '*'
  target: fpc1
  view: IthrottleIDView

IthrottleIDView:
  regex:
    min_usage: 'Min Usage Perc: (\d+\.\d+)'
    max_usage: 'Max Usage Perc: (\d+\.\d+)'
    usg_enable: 'AdjustUsageEnable: (\d)'
  fields:
    throttle_stats: _ThrottleStatsTable

_ThrottleStatsTable:
```



```
title: Throttle Stats
delimiter: ":"
```

When you retrieve and print the data in the Junos PyEZ application, the data includes the three **regex** items, which contain the value matched by the capturing group for that expression.

```
{'max_usage': 50.0,
 'min_usage': 25.0,
 'throttle_stats': {'AdjDown': 4,
                    'AdjUp': 6,
                    'Checks': 124149442,
                    'Disables': 0,
                    'Enables': 0,
                    'Starts': 65708652,
                    'Stops': 65708652},
 'usg_enable': 1}
```

RELATED DOCUMENTATION

[Defining Junos PyEZ Operational Tables for Parsing Unstructured Output | 221](#)

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output | 259](#)

Using Junos PyEZ Operational Tables and Views that Parse Unstructured Output

Junos PyEZ operational (op) Tables for unstructured output extract data from the text output of a CLI command executed on a device running Junos OS or a vty command executed on a given Flexible PIC Concentrator (FPC). 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.command.ospf_neighbor import OspfNeighborTable
```



```
with Device(host='router.example.com') as dev:
    stats = OspfNeighborTable(dev)
```

To use the Table in your Junos PyEZ application to execute the command and retrieve the data, call the Table's `get()` method and supply any required or optional parameters. If the Table defines default arguments, for example, for the `args`, `filters`, `key_items`, or `target` parameters, the `get()` method uses these defaults unless you override them in the argument list.

```
from jnpr.junos import Device
from jnpr.junos.command.ospf_neighbor import OspfNeighborTable
from pprint import pprint
import json

with Device(host='router.example.com') as dev:
    stats = OspfNeighborTable(dev)
    stats.get()
    pprint(json.loads(stats.to_json()))
```

RELATED DOCUMENTATION

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Defining Junos PyEZ Operational Tables for Parsing Unstructured Output | 221](#)

[Defining Views for Junos PyEZ Operational Tables that Parse Unstructured Output | 238](#)

Defining Junos PyEZ Configuration Tables

IN THIS SECTION

- [Table Name | 263](#)
- [Configuration Scope \(get or set\) | 263](#)
- [Key Field \(key-field\) | 265](#)
- [Required Keys \(required_keys\) | 267](#)

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. [Table 23 on page 261](#) summarizes the parameters and 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 23: Junos PyEZ Configuration Table Parameters

Table Parameter Name	Table Parameter	Table Type	Description
Table name	–	get or set	User-defined Table identifier.
Configuration scope	get or set	–	<p>XPath expression relative to the top-level <configuration> element that identifies the configuration hierarchy level at which to select or configure objects, depending on the Table type.</p> <p>Specify get to retrieve configuration objects or specify set to both configure and retrieve objects.</p> <p>These objects become the reference for the associated View.</p>

Table 23: Junos PyEZ Configuration Table Parameters (*Continued*)

Table Parameter Name	Table Parameter	Table Type	Description
Configuration resource key field	key-field	set	<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 <name> 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>
Required keys	required_keys	get or set	<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 <name> element.</p> <p>Users must include all required keys as arguments to the get() method when retrieving the configuration data in their application.</p>
Table View	view	get or set	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

```



```

UIView:
  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

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 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:

```
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-  
style="italic">table_object</fo:inline>.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

with Device(host='router1.example.com') as dev:
    users = UserConfigTable(dev)
    users.username = 'admin'
    users.userclass = 'super-user'
    ...
```

If the configuration Table defines fields for 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

with Device(host='router1.example.com') as dev:
    users = UserTable(dev)
    users.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 Views for Junos PyEZ Configuration Tables" on page 268](#).

RELATED DOCUMENTATION

[Defining Views for Junos PyEZ Configuration Tables | 268](#)

[Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279](#)

[Loading Inline or External Tables and Views in Junos PyEZ Applications | 194](#)

[Defining Junos PyEZ Operational Tables for Parsing Structured Output | 200](#)

Defining Views for Junos PyEZ Configuration Tables

IN THIS SECTION

- [View Name | 271](#)
- [Fields \(fields\) | 272](#)
- [Field Options \('get' Tables\) | 274](#)
- [Field Options \('set' Tables\) | 275](#)
- [Groups \(groups\) and Field Groups \(fields_\) | 277](#)

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 24 on page 269](#).

Table 24: Junos PyEZ Configuration View Parameters

View Parameter Name	View Parameter	Table Type	Description
View name	–	get or set	User-defined View identifier.
Field items	fields	get or set	<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 field names must be valid Python variable names. The XPath expressions are relative to the context defined by the get or set property for that Table.</p> <p>When the Table scope uses get, the fields identify the data to extract from the configuration. When the Table scope uses set, the fields identify the elements that you can configure or retrieve, depending on the operation.</p>

Table 24: Junos PyEZ Configuration View Parameters *(Continued)*

View Parameter Name	View Parameter	Table Type	Description
Field groups	fields_group	get or set	<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 field names must be valid Python variable names. The XPath expressions are relative to the context set by the corresponding groups parameter.</p> <p>When the Table scope uses get, the fields identify the data to extract from the configuration. When the Table scope uses set, the fields identify the elements that you can configure or retrieve, depending on the operation.</p>
Groups	groups	get or set	<p>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 get or set property for that Table.</p>

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

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 field names must be valid Python variable names. 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
  ...

UserView:
  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

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.get()
    for account in users:
        print("Username is {}\nUser class is {}".format(account.name,
account.userclass))

```

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. *No Link Title* and *No Link Title* 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:

```
UserView:
  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 2 and *Table 3* 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 25: Type Checks for 'set' Configuration Tables

type Value	Description	Example
bool	Field only accepts Boolean values of True or False	enable: { 'enable': { 'type': 'bool' } }

Table 25: Type Checks for 'set' Configuration Tables *(Continued)*

type Value	Description	Example
enum	Field only accepts one of the values defined in the enum list	<code>enc : { 'encapsulation' : { 'type' : { 'enum' : ['vlan-ccc','vlan-vpls'] } } }</code>
float	Field only accepts floating point values	<code>drift : { 'clock-drift' : { 'type' : 'float' } }</code>
int	Field only accepts integer values	<code>uid: { 'uid' : { 'type' : 'int' } }</code>
str	Field only accepts string values	<code>name: { 'name': { 'type': 'str' } }</code>

Table 26: Constraint Checks for 'set' Configuration Tables

Constraint Check Name	Description	Example
default	<p>Default value for a field.</p> <p>A field uses its default value when the user does not explicitly configure the field. If the user calls the reset() method to reset field values in the application, fields that have a defined default are set to that value.</p>	<code>native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'default' : 501 } }</code>
maxValue	Maximum value for a field, which is interpreted based on the field type .	<code>native_vlan : { 'native-vlan-id' : { 'type' : 'int', 'minValue' : 0, 'maxValue' : 4094 } }</code>
minValue	Minimum value for a field, which is interpreted based on the field type .	<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 method:

```
<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>

```

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 field names must be valid Python variable names.

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 | 260](#)

[Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279](#)

Using Junos PyEZ Configuration Tables to Retrieve Configuration Data

IN THIS SECTION

- [Retrieving Configuration Items | 280](#)
- [Specifying Inheritance and Group Options | 282](#)
- [Accessing Table Items | 284](#)
- [Iterating Through a Table | 285](#)

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

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
```

The following sections discuss how to then retrieve and manipulate the data:

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

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">users.get()</fo:inline>
```


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

with Device(host='router.example.com') as dev:
    interfaces = InterfaceTable(dev)
    interfaces.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'}

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.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.

```
<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-family="Lato
Bold">{'inherit':''}</fo:inline>
```

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.

```
if '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

with Device(host='router.example.com') as dev:
    users = UserTable(dev)
    users.get()

    for user in users:
        print("Username is {} \nUser class is {}".format(user.username,
user.userclass))
```

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 | 260](#)

[Defining Views for Junos PyEZ Configuration Tables | 268](#)

Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources

IN THIS SECTION

- [Creating the Structured Resource | 287](#)
- [Using the Resource in a Junos PyEZ Application | 289](#)

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 this topic.

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>
            <full-name>J Smith</full-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 260](#) and ["Defining Views for Junos PyEZ Configuration Tables" on page 268](#).

```

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 ["Loading Inline or External Tables and Views in Junos PyEZ Applications" on page 194](#).

Using the Resource in a Junos PyEZ Application

To configure the resource in your Junos PyEZ application:

1. Create a **Device** instance 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 290](#).

```
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()
```

7. Close the device connection.

```
dev.close()
```

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 120](#) and ["Using Junos PyEZ to Commit the Configuration" on page 148](#).

RELATED DOCUMENTATION

[Defining Junos PyEZ Configuration Tables | 260](#)

[Defining Views for Junos PyEZ Configuration Tables | 268](#)

[Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS | 290](#)

Using Junos PyEZ Configuration Tables to Configure Structured Resources on Devices Running Junos OS

IN THIS SECTION

- [General Configuration Process | 291](#)

- [Configuring Statements Consisting of a Fixed-Form Keyword | 295](#)
- [Configuring Multiple Values for the Same Statement | 296](#)
- [Configuring Multiple Instances of the Same Statement | 298](#)
- [Configuring Multiple Instances of the Same Resource | 300](#)
- [Deleting Containers or Leaf Statements | 301](#)
- [Configuring Properties Corresponding to Junos XML Attributes | 303](#)
- [Using append\(\) to Generate the Junos XML Configuration Data | 307](#)
- [Viewing Your Configuration Changes | 308](#)
- [Controlling the RPC Timeout Interval | 310](#)

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. This topic discusses 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

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
import crypt

with Device(host='router.example.com') as dev:
    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.

```

<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
style="italic">table_object</fo:inline>.<fo:inline xmlns:fo="http://www.w3.org/
1999/XSL/Format" font-style="italic">fieldname</fo:inline> = <fo:inline
xmlns:fo="http://www.w3.org/1999/XSL/Format" font-style="italic">value</
fo:inline>

```

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, which in this example is **username**.

The following code imports **UserConfigTable** and configures values for the **username**, **userclass**, and **password** fields. The **crypt** module calculates the hash of the user's password for the configuration. This example requires Python3 to use the **crypt()** function with a single argument.

```
# Python 3
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree
import crypt

with Device(host='router.example.com') as dev:
    uc = UserConfigTable(dev)

    uc.username = 'user1'
    uc.userclass = 'operator'
    uc.password = crypt.crypt('lab123')
    ...
```

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, deleting a leaf or container statement, or configuring an object property that corresponds to a Junos XML attribute, see the following sections:

- [No Link Title](#)
- [No Link Title](#)
- [No Link Title](#)
- [No Link Title](#)
- [No Link Title](#)
- [No Link Title](#)

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. 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.

```
uc.append()
```

After building the XML, the **append()** method also resets all fields to 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 *No Link Title*.

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 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 *No Link Title*.

```
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

```
uc.set()
```

NOTE: When you create the Table instance with a context manager (**with ... as** syntax) that includes the **mode** parameter to use a specific configuration mode, the context manager handles opening and locking and closing and unlocking the database. In this case, you only need to call the **load()** and **commit()** methods to configure the device. Calling the **lock()** or **set()** method results in a **LockError** exception.

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 the using the different methods to load and commit the configuration data, see ["Using Junos PyEZ to Configure Devices Running Junos OS" on page 120](#) and ["Using Junos PyEZ to Commit the Configuration" on page 148](#).

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 *No Link Title*.

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 279](#).

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>
```



```

        ...
    </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

with Device(host='router.example.com') as dev:
    sc = ServicesConfigTable(dev)

    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">sc.ftp = True</fo:inline>
    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 [ value1value2value3 ...];
```


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

with Device(host='router.example.com') as dev:
    intf = InterfacesConfigTable(dev)

    intf.name = 'ge-0/0/1'
    intf.mode = 'trunk'
    intf.native_vlan = 510
    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">intf.vlan_list = [510, 520, 530]</fo:inline>

    intf.append()
    intf.set()
```

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 in a Junos PyEZ application. 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 **lxml** 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 example configures two user objects and commits the changes:

```

from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree
import crypt

with Device(host='router.example.com') as dev:
    uc = UserConfigTable(dev)

    uc.username = 'user1'
    uc.userclass = 'operator'
    uc.uid = 1005
    uc.password = crypt.crypt('lab123')
    uc.append()

    uc.username = 'user2'
    uc.userclass = 'operator'

```



```
uc.uid = 1006
uc.password = crypt.crypt('lab456')
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:

```
from jnpr.junos import Device
from myTables.ConfigTables import UserConfigTable
from lxml import etree
import crypt

users = ['user1', 'user2']
uids = ['1005', '1006']
passwds = ['lab123', 'lab456']

with Device(host='router.example.com') as dev:
    uc = UserConfigTable(dev)
    for user, uid, passwd in zip(users, uids, passwds):
        uc.username = user
        uc.userclass = 'operator'
        uc.uid = uid
        uc.password = crypt.crypt(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: user/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: 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

with Device(host='router.example.com') as dev:
    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

with Device(host='router.example.com') as dev:
    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>
```

Configuring Properties Corresponding to Junos XML Attributes

Some configuration mode commands, for example `deactivate` or `protect`, apply or remove a specific property, such as the `inactive` or `protect` property, to a configuration statement. In the CLI, this property

is indicated by a tag preceding the configuration statement. The Junos XML configuration indicates this property using an XML attribute for the object.

For example, the following command deactivates the given interface.

```
[edit]
```

```
user@host# deactivate interfaces ge-1/0/2
```

When you view the configuration in the CLI, the **inactive** tag precedes the interface name.

```
[edit]
```

```
user@host# show interfaces
```

```
inactive: ge-1/0/2 {
  description "to CustomerA";
  unit 0 {
    family inet {
      address 198.51.100.1/24;
    }
  }
}
```

Similarly, in the Junos XML output, the **<interface>** element for the same interface includes the **inactive="inactive"** attribute.

```
user@host# show interfaces | display xml
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
  <configuration junos:changed-seconds="1544581124" junos:changed-
    localtime="2018-12-11 18:18:44 PST">
    <interfaces>
      <interface inactive="inactive">
        <name>ge-1/0/2</name>
```



```

        <description>to CustomerA</description>
        <unit>
            <name>0</name>
            <family>
                <inet>
                    <address>
                        <name>198.51.100.1/24</name>
                    </address>
                </inet>
            </family>
        </unit>
    </interface>
</interfaces>
</configuration>
</rpc-reply>

```

Junos PyEZ configuration Tables enable you to define supported XML attributes for an object when configuring structured resources. Consider the following Junos PyEZ configuration Table and View:

```

InterfaceTable:
    set: interfaces
    key-field:
        - name
    view: InterfaceView

InterfaceView:
    fields:
        interface: interface
        name: interface/name
        desc: interface/description
        unit_name: interface/unit/name
        ip_address: interface/unit/family/inet/address

```

To define the XML attribute for a given configuration object, set its field as defined by the View to a dictionary containing the attribute and its value. For example, to define an interface but immediately deactivate it, set the field corresponding to the **<interface>** element to **{'inactive':'inactive'}**. The following example configures and deactivates the given interface:

```

from jnpr.junos import Device
from myTables.ifConfigTable import InterfaceTable
from lxml import etree

```



```

with Device(host='router.example.com') as dev:
    intf = InterfaceTable(dev)
    intf.name = 'ge-1/0/2'
    intf.unit_name = 0
    intf.ip_address = '198.51.100.1/24'
    intf.desc = 'to CustomerA'
    intf.interface = {'inactive':'inactive'}

    intf.append()

    configXML = intf.get_table_xml()
    if (configXML is not None):
        print (etree.tostring(configXML, encoding='unicode', pretty_print=True))
    else:
        print (configXML)

    intf.set()

```

The application prints the Junos XML configuration data returned by the `get_table_xml()` method. The interface element with identifier 'ge-1/0/2' includes the `inactive="inactive"` attribute.

```

<configuration>
  <interfaces>
    <interface inactive="inactive">
      <name>ge-1/0/2</name>
      <unit>
        <name>0</name>
        <family>
          <inet>
            <address>198.51.100.1/24</address>
          </inet>
        </family>
      </unit>
      <description>to CustomerA</description>
    </interface>
  </interfaces>
</configuration>

```


To activate an inactive object, set the View field corresponding to the inactive object to {'active':'active'}.

```
from jnpr.junos import Device
from myTables.ifConfigTable import InterfaceTable
from lxml import etree

with Device(host='router.example.com') as dev:
    intf = InterfaceTable(dev)
    intf.name = 'ge-1/0/2'
    intf.interface = {'active':'active'}

    intf.append()
    intf.set()
```

Similarly, to protect the configuration element or remove the **protect** attribute from a protected element, set the appropriate field value to {'protect':'protect'} or {'unprotect':'unprotect'}. For more information about XML attributes in the Junos OS configuration, see the *Junos XML Management Protocol Developer Guide*.

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
import crypt

with Device(host='router.example.com') as dev:
    uc = UserConfigTable(dev)
    uc.username = 'user1'
    uc.userclass = 'operator'
    uc.password = crypt.crypt('lab123')

    <fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
```



```
weight="bold">uc.append()</fo:inline>
uc.set()
```

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
import crypt

with Device(host='router.example.com') as dev:
    uc = UserConfigTable(dev)
    uc.username = 'user1'
    uc.userclass = 'operator'
    uc.password = crypt.crypt('lab123')
    uc.append()

    configXML = uc.<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">get_table_xml()</fo:inline>
    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()

<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">uc.pdiff()</fo:inline>

...
```



```
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 **Devicetimeout** 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 = crypt.crypt('lab123')
uc.append()

<fo:inline xmlns:fo="http://www.w3.org/1999/XSL/Format" font-
weight="bold">uc.set(timeout=300)</fo:inline>
```

RELATED DOCUMENTATION

[Defining Junos PyEZ Configuration Tables | 260](#)

[Defining Views for Junos PyEZ Configuration Tables | 268](#)

[Overview of Using Junos PyEZ Configuration Tables to Define and Configure Structured Resources | 287](#)

[Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279](#)

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 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:

```
from jnpr.junos.op.xcvr import XcvrTable

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 that Parse Structured Output | 216](#)

[Using Junos PyEZ Configuration Tables to Retrieve Configuration Data | 279](#)

8

CHAPTER

Troubleshooting Junos PyEZ

[Troubleshooting jnpr.junos Import Errors | 315](#)

[Troubleshooting Junos PyEZ Connection Errors | 316](#)

[Troubleshooting Junos PyEZ Authentication Errors When Managing Devices
Running Junos OS | 318](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos
OS | 320](#)

Troubleshooting jnpr.junos Import Errors

IN THIS SECTION

- Problem | 315
- Cause | 315
- Solution | 315

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 12](#).

To verify that Junos PyEZ is successfully installed, start Python and import the **jnpr.junos** package.

```
[user@server ~]$ python
```

```
>>> import junpr.junos
>>> junpr.junos.__version__
'2.2.0'
```

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 | 12](#)

[Junos PyEZ Modules Overview | 8](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

Troubleshooting Junos PyEZ Connection Errors

IN THIS SECTION

- [Problem | 317](#)
- [Cause | 317](#)
- [Solution | 317](#)

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 Retrieve Facts from Devices Running Junos OS | 50](#)

[Troubleshooting jnpr.junos Import Errors | 315](#)

[Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS | 318](#)

[Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS | 320](#)

Troubleshooting Junos PyEZ Authentication Errors When Managing Devices Running Junos OS

IN THIS SECTION

- [Problem | 318](#)
- [Cause | 319](#)
- [Solution | 319](#)

Problem

Description

Junos PyEZ generates an error regarding failed authentication. For example:

```
unable to connect to dcl1a.example.com: ConnectAuthError(dcl1a.example.com)
```

or

```
jnpr.junos.tty_ssh:SSH Auth Error
```


Cause

The device running Junos OS or the console server through which the application connects might fail to authenticate the user for the following reasons:

- The user does not have an account on the device running Junos OS or on the console server through which it connects.
- The user has an account with a text-based password configured on the device running Junos OS and the console server, if one is used, but the wrong password or no password is supplied for the user when creating the **Device** instance.
- The user has an account and authenticates using SSH keys with the device running Junos OS or a console server connected to the device, 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 or console servers where appropriate and that the SSH public key 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 | 41](#)

[Using Junos PyEZ to Retrieve Facts from Devices Running Junos OS | 50](#)

Troubleshooting Junos PyEZ Errors When Configuring Devices Running Junos OS

IN THIS SECTION

- [Troubleshooting Timeout Errors | 320](#)
- [Troubleshooting Configuration Lock Errors | 321](#)
- [Troubleshooting Configuration Change Errors | 322](#)

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

IN THIS SECTION

- [Problem | 320](#)
- [Cause | 321](#)
- [Solution | 321](#)

Problem

Description

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

```
RpcTimeoutError(host: dcl1a.example.com, cmd: commit-configuration, timeout: 30)
```


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 **Devicetimeout** property to an appropriate value, or include the **timeout=seconds** argument in the **commit()** or **set()** method when you load and commit configuration data on a device. For example:

```
dev = Device(host="<fo:inline xmlns:fo='http://www.w3.org/1999/XSL/Format' font-  
style='italic'>host</fo:inline>")  
dev.open()  
dev.timeout = 300  
  
with Config(dev, mode='exclusive') as cu:  
    cu.load(path="junos-config.conf", merge=True)  
    cu.commit(timeout=360)  
  
dev.close()
```

Troubleshooting Configuration Lock Errors

IN THIS SECTION

- [Problem | 322](#)
- [Cause | 322](#)
- [Solution | 322](#)

Problem

Description

The Junos PyEZ code generates a `LockError` message indicating that the configuration cannot be locked.

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

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

IN THIS SECTION

- [Problem | 323](#)
- [Cause | 323](#)
- [Solution | 323](#)

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 | 120](#)

[Using the Junos PyEZ Config Utility to Configure Devices Running Junos OS | 129](#)

[Example: Using Junos PyEZ to Load Configuration Data from a File | 154](#)