Junos Snapshot Administrator in Python

Junos Snapshot Administrator in Python Guide

Release
1.0

Modified: 2017-04-05

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

Junos Snapshot Administrator in Python Junos Snapshot Administrator in Python Guide
1.0

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

YEAR 2000 NOTICE

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

END USER LICENSE AGREEMENT

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

Table of Contents

List of Figures

List of Tables

About the Documentation

Documentation and Release Notes

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

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

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

Documentation Conventions

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

Table 1: Notice Icons

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

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

Table 2: Text and Syntax Conventions

Convention	Description	Examples
Bold text like this	Represents text that you type.	To enter configuration mode, type the **configure** command: user@host> **configure**
`Fixed-width text like this`	Represents output that appears on the terminal screen.	`user@host>` **show chassis alarms** `No alarms currently active`
Italic text like this	• Introduces or emphasizes important new terms. • Identifies guide names. • Identifies RFC and Internet draft titles.	• A policy *term* is a named structure that defines match conditions and actions. • *Junos OS CLI User Guide* • RFC 1997, *BGP Communities Attribute*
Italic text like this	Represents variables (options for which you substitute a value) in commands or configuration statements.	Configure the machine's domain name: [edit] root@# **set system domain-name** *domain-name*

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

Convention	Description	Examples
Text like this	Represents names of configuration statements, commands, files, and directories; configuration hierarchy levels; or labels on routing platform components.	• To configure a stub area, include the **stub** statement at the **[edit protocols ospf area area-id]** hierarchy level. • The console port is labeled **CONSOLE**.
< > (angle brackets)	Encloses optional keywords or variables.	**stub <default-metric *metric*>;**
| (pipe symbol)	Indicates a choice between the mutually exclusive keywords or variables on either side of the symbol. The set of choices is often enclosed in parentheses for clarity.	**broadcast | multicast** **(*string1* | *string2* | *string3*)**
# (pound sign)	Indicates a comment specified on the same line as the configuration statement to which it applies.	**rsvp { # Required for dynamic MPLS only**
[ ] (square brackets)	Encloses a variable for which you can substitute one or more values.	**community name members [*community-ids*]**

Indention and braces ({ })	Identifies a level in the configuration hierarchy.	[edit]
		routing-options {
		  static {
		  route default {
		  nexthop *address*;
		  retain;
		  }
		  }
		}

; (semicolon)	Identifies a leaf statement at a configuration hierarchy level.	
GUI Conventions		
Bold text like this	Represents graphical user interface (GUI) items you click or select.	• In the Logical Interfaces box, select **All Interfaces**. • To cancel the configuration, click **Cancel**.
> (bold right angle bracket)	Separates levels in a hierarchy of menu selections.	In the configuration editor hierarchy, select **Protocols>Ospf**.

Documentation Feedback

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

- Online feedback rating system—On any page of the Juniper Networks TechLibrary site at http://www.juniper.net/techpubs/index.html, simply click the stars to rate the content, and use the pop-up form to provide us with information about your experience. Alternately, you can use the online feedback form at http://www.juniper.net/techpubs/feedback/.

- E-mail—Send your comments to techpubs-comments@juniper.net. Include the document or topic name, URL or page number, and software version (if applicable).

Requesting Technical Support

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

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

Self-Help Online Tools and Resources

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

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

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

Opening a Case with JTAC

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

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

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

PART 1

Overview

CHAPTER 1

Junos Snapshot Administrator in Python Overview

Junos Snapshot Administrator in Python Overview

Junos® Snapshot Administrator in Python (jsnapy) enables you to capture and audit runtime environment snapshots of your networked devices running the Junos OS. You can capture and validate the operational and configuration status of a device and review operational changes to a device. You create configuration files that define the scope of snapshots and customize the test criteria for the snapshot data.

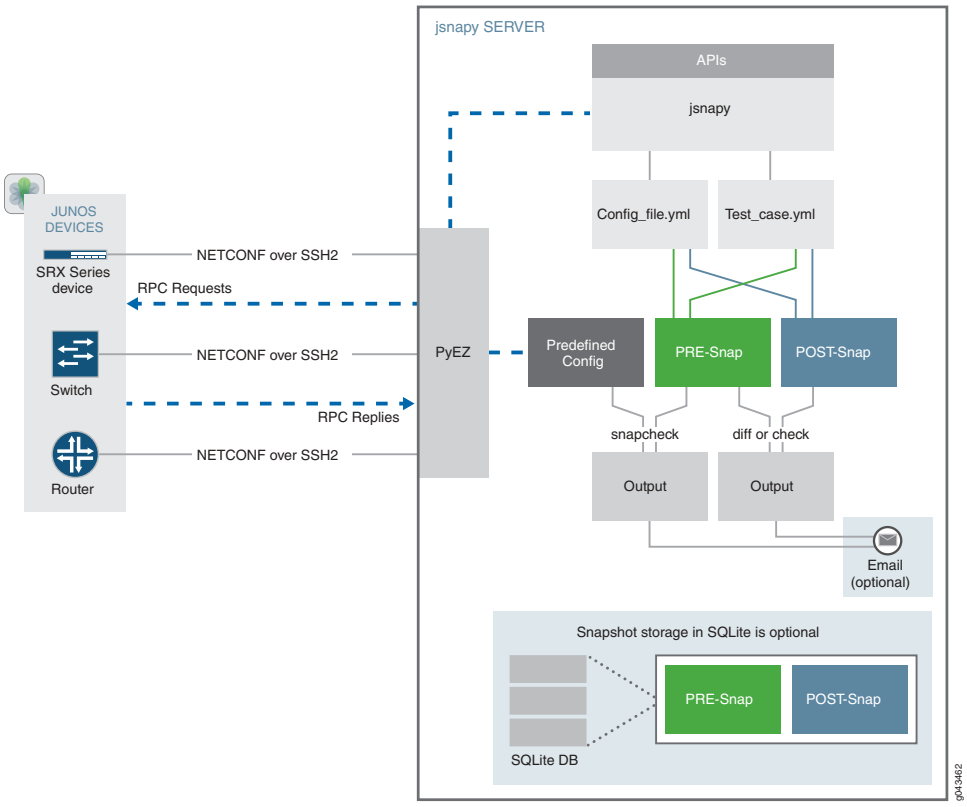
You can use Junos Snapshot Administrator in Python to perform the following functions on either a single device or list of devices running Junos OS:

- Take a snapshot of the runtime environment on a device.
- Compare two snapshots.
- Audit a device's runtime environment against pre-defined criteria.
- Use it as a module in other Python programs.
- Send e-mail alerts that include test results.
- Store and compare snapshots in an SQLite database.

For example, prior to a software or hardware upgrade on a device, you can take a pre-install and post-install snapshot of the device and then compare the two snapshots. You can then review the operational changes on the device and validate these changes from a list of expected changes.

Figure 1 on page 4 shows an operational overview of Junos Snapshot Administrator in Python.

Figure 1: Junos Snapshot Administrator in Python Operational Overview



Junos Snapshot Administrator in Python is installed on a remote server running an OS that can support Python, including Mac OS X, and many Linux distributions. The jsnapy server uses Juniper PyEZ to make NETCONF connections over SSHv2 to your networked Junos OS devices. Using YAML-formatted configuration and test files for connection and test criteria, jsnapy sends RPC requests to the devices over the NETCONF connections. RPC replies are received back at the server in the form of snapshots. You can run jsnapy from the server command line or can be included as a module in other Python applications. See "Understanding Junos Snapshot Administrator in Python when Running as a Python Module" on page 5

The snapshots are formatted as text or XML files and are stored on the server in a location designated by the **jsnapy.cfg** file located in the directory **/etc/jsnapy/**. The full list of directories and files installed with jsnapy in the **/etc/jsnapy** directory can be seen in the "Installing Junos Snapshot Administrator in Python" on page 22 Optionally, the snapshots can be stored in an SQLite database on the server. Jsnapy can then compare the snapshots either to other snapshots or to pre-defined criteria in order to audit the effects of configuration changes or to confirm proper device configuration.

The only requirement for Junos devices to work with jsnapy is that NETCONF and SSH must be enabled.

Junos Snapshot Administrator in Python logs information regarding its operations to the console and to the **/var/log/jsnapy/jsnapy.log** file by default. The level of logging

performed can be set to **DEBUG**, **INFO**, **ERROR**, or **CRITICAL** by changing parameters in the **/etc/jsnapy/logging.yml** file. Using this file, you can change the logging level, disable logging to the console, or change the locations and names of the log files that jsnapy uses. The logging level can be temporarily set to debug by using the **-v** option when calling jsnapy from the CLI, for example:

jsnapy –snap PRE -f config_file.yml -v.

Related Documentation

- Installing Junos Snapshot Administrator in Python on page 22
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7

Understanding Junos Snapshot Administrator in Python when Running as a Python Module

Junos® Snapshot Administrator in Python (jsnapy) enables you to capture and audit runtime environment snapshots of your networked devices running the Junos OS. Jsnapy can be run from the command line of a network server (jsnapy server) or called as a module from within other python scripts on the jsnapy server. See "Junos Snapshot Administrator in Python Overview" on page 3 for details about CLI operation.

Once installed on the network server, jsnapy is ready to be used as a module within other python scripts and programs. No further configuration is needed. Jsnapy retains all of its functionality when used as a module in a python script or program.

In order to use jsnapy as a module inside of another python script, you must first use the python **import** statement so that the python script has access to jsnapy features. Because the script relies not only on jsnapy but also an understanding of Junos OS devices when working with snapshots, you must create two distinct import statements, each importing a different module from the available Juniper python libraries, jnpr.

```
from jnpr.jsnapy import SnapAdmin
from jnpr.junos import Device
```

Importing he SnapAdmin module gives the script or program access to all of the same jsnapy options available on the command line. When used within the script, the **SnapAdmin( )** function is usually assigned to a variable for use later in the script. Arguments are passed to **SnapAdmin( )** to control the available jsnapy options. The following example assigns SnapAdmin to the variable ***js*** and then calls jsnapy with each of the 3 major options, **snap**, **check**, and **snapcheck**.

Assign the SnapAdmin Function to a Variable

```
js = SnapAdmin( )
```

Call jsnapy 3 Times

```
snapshot = js.snap(config_file, "snapshot_filename")
chk = js.check(config_file, snapshot_1, snapshot_2)
snapchk = js.snapcheck(config_file, "snapshot_filename")
```

The Device module allows python scripts to be able to connect-to, login, and run commands on Junos OS devices. You supply connection IP address, username, and

password arguments when calling the Device module from within python scripts. For example, the following line from a python script creates a variable called ***device_object*** for use later in the script.

```
device_object = Device(host='192.0.2.1', user='username', password='password')
```

The variables named as argument values above can come from a variety of sources.

- You can build the configuration file information into the python script by creating a variable in which to store the data and then specifying all of the host and test information needed to connect to and test that device. For example,

```
config_data = """
hosts:
  - device: 192.0.2.1
    username : username
    passwd: password
tests:
  - test_exists.yml
  - test_does_not_contain.yml
"""
```

- You can specify an existing YAML formatted configuration file that contains connection and test information for the target device. For example:

```
config_file = /etc/jsnapy/samples/config_single_snapcheck.yml
```

- You can use a combination of a script defined device object for connection purposes and a list of existing YAML formatted test files to test the device. For example, the following line uses the previously defined **device_object** along with a list of tests to run and a name for the snapshot file.

```
js.snapcheck({'tests': ['test_exists.yml', 'test_contains.yml', 'test_is_equal.yml']},
  "snapshot_name", dev=device_object)
```

A wide variety of options are available when running jsnapy from within other Python scripts or programs. Strong familiarity with Python scripting and Junos OS device interaction is helpful in learning what is possible.

Related Documentation

- Junos Snapshot Administrator in Python Overview on page 3
- Example: Use Jsnapy as a Python module on page 61

CHAPTER 2

Junos Snapshot Administrator in Python Configuration Files Overview

Understanding Junos Snapshot Administrator in Python Configuration Files

The Junos Snapshot Administrator in Python (jsnapy) configuration files define the scope of a snapshot and specify the names of the files containing the test criteria for either a single snapshot or a comparison of two snapshots. You provide the location of the jsnapy configuration file as an argument to the **jsnapy** command or as an argument when calling jsnapy from another Python script.

Where to Store a Configuration File

The jsnapy configuration files can be stored anywhere on the jsnapy server's file system. If the configuration file does not exist in the current working directory from which **jsnapy** is called, jsnapy looks in the path specified in **/etc/jsnapy/jsnapy.cfg** under the **config_file_path** heading. If the file does not exist there, jsnapy looks in the default location (**/etc/jsnapy/**) for configuration files. If the file does not exist there either, an error is displayed. To use a specific configuration file, you can specify the full path to the configuration file on the command line when calling **jsnapy**. For example:

```
jsnapy --snap PRE -f /full/path/to/config/file/config.yml
```

Understanding the Sections of a Configuration File

Jsnapy configuration files consist of two mandatory sections: hosts and tests, and two optional sections: sqlilte and mail.

- The hosts section defines which Junos OS hosts or devices jsnapy connects to when using the configuration file. Jsnapy can connect to a single host or to multiple hosts. Configuration files that direct jsnapy to connect to a single host define the host IP address and login credentials within the configuration file.

  Configuration files that direct jsnapy to connect to multiple hosts use the include feature of YAML files to read a list of hosts and credentials from a separate YAML devices file. Devices files can be large files with many hosts listed. To use fewer hosts, the devices file can be broken up into groups that are referenced using the **group:** keyword in the hosts section of the configuration file. For more information, see"Understanding Junos Snapshot Administrator in Python Device Files" on page 13. To specify which file contains the list of devices, enter the name of the file or the full path to the file. When only the filename is supplied, the location of the file follows the rules described in "Where to Store a Configuration File" on page 7.

- The tests section tells jsnapy which test files to use when using the configuration file. The tests to be performed against the hosts are always defined in separate test files whose names are specified within the tests section of the configuration files. Multiple test filenames can be entered, each on its own line. Test filenames can be supplied as a filename or as the full path to the file. When only the filename is supplied, the location of the file follows the rules described in "Where to Store a Configuration File" on page 7. For more information, see "Understanding Junos Snapshot Administrator in Python Test Files" on page 10 and "Example: Creating the Junos Snapshot Administrator in Python Test Files" on page 32.

- The sqlite section defines the parameters for storing and comparing snapshots in an SQLite database. You can specify whether snapshots created with this configuration file should be stored in a specific SQLite database in addition to being stored in the jsnapy server's file system. You can also specify whether **--check** and **--snapcheck** optionsperform the compare operations on snapshots stored in the SQLite database. The SQLite database can store up to 51 snapshots, and these are indexed by ID numbers 0 through 50. Table 4 on page 9 describes the configuration elements needed to configure snapshot storage in an SQLite database.

- The mail section includes the name of the file containing e-mail configuration information. The mail file can be supplied as a filename or as a full path to the file. When only the filename is supplied, the location of the file follows the rules described in "Where to Store a Configuration File" on page 7. The details of what is contained in the mail file are described in "Understanding Junos Snapshot Administrator in Python Mail Files" on page 15.

Elements of a Configuration File

The jsnapy configuration files are YAML files that you build using combinations of keywords and values that are pertinent to your network. Like any YAML file, structure is derived through indentation (one or more spaces), sequence items are denoted by a

dash (-), and key-value pairs are separated by a colon (:). Table 3 on page 9 shows the supported keywords and their meaning.

Table 3: Configuration File Keywords and Descriptions

Keyword	Description
hosts	This keyword denotes the beginning of the hosts section. It must be on a line by itself and must be followed by a colon. (**hosts:**)
group	This keyword is an optional part of the hosts section. It refers to device group names within an external devices file (when contacting multiple hosts). It must be on a line by itself and must be followed by a colon. (**group:**) The group name used here must be a case-sensitive match to the group name in the external devices file. To include all hosts in the devices file, use the value **all** in place of any group name.
device	This keyword is part of the hosts section. It is preceded by a dash and followed immediately by a colon (**- devices:**). Single hosts can be contacted by specifying their IP address. Multiple hosts can be contacted by specifying a separate devices file (also in YAML format) that contains a listing of multiple device IP addresses.
include	This optional keyword is part of the hosts section. It is preceded by a dash and followed immediately by a colon. It specifies the name of a file that contains multiple device IP addresses and login credentials. It is needed only when you are connecting jsnapy to multiple hosts.
username	This keyword is part of the hosts section. It is followed immediately by a colon. It specifies a username that is used when contacting the device listed directly above it in the configuration or devices file.
passwd	This keyword is part of the hosts section. It is followed immediately by a colon. It specifies the login password used when contacting the device listed immediately above it in the configuration or devices file.
tests	This keyword denotes the beginning of the tests section. It is followed immediately by a colon. It is used to list one or more test files that are used to create or compare snapshots.
sqlite	This keyword denotes the beginning of the sqlite section. It is followed immediately by a colon. The sqlite section is used to define the parameters for storing snapshots in or comparing snapshots from an sqlite database. To complete this section, you must fill in the parameters shown in Table 4 on page 9.
mail	This keyword activates the e-mail capability. It is followed by a colon. The argument is the name of a file that contains e-mail configuration information; for example, **mail: mail.yml**. When activated, e-mail is sent regarding the results of **--check** and **--snapcheck** operations, but is not sent when using the **--snap** option.

Table 4 on page 9 shows the available parameters for filling in the sqlite section of the configuration file.

Table 4: SQLite Configuration Parameters

Parameter	Description
- store_in_sqlite:	Required when using the **--snap** option. Optional with the **--check** or **--snapcheck** options. Set to **true** to activate.
database_name	Required regardless of option. Specifies the name of the sqlite database in which to store or read the snapshot.
check_from_sqlite	Required for **--check** or **--snapcheck** options. Optional for **--snap** option. Set to **true** to activate.

Table 4: SQLite Configuration Parameters *(continued)*

Parameter	Description
compare:	Required if you are comparing two snapshots by snapshot ID number (0-50) with the **--check** or **--snapcheck** options. Ignored if you compare by snapshot name using the command line; for example, **jsnapy --check PRE POST -f config.yml**.

Inserting Comments into a Configuration File

Comments can be inserted into the file at any location by starting the comment with the pound sign (#). This is useful for users who need to manage multiple configuration files so that they can quickly understand what is contained in the file.

Related Documentation

- Example: Creating the Junos Snapshot Administrator in Python Configuration Files on page 29
- Understanding Junos Snapshot Administrator in Python Test Operators on page 16
- Junos Snapshot Administrator in Python Test Operators Summary on page 37
- Junos Snapshot Administrator in Python Overview on page 3

Understanding Junos Snapshot Administrator in Python Test Files

The Junos Snapshot Administrator in Python (jsnapy) test files define the details of what a snapshot contains. Test files, also known as test cases, are YAML-formatted files that are referenced from inside jsnapy configuration files. Configuration files are read as an argument to the **jsnapy** CLI command. Generally speaking, test files define what tests are included within the test case and the specific definitions for those tests.

- Understanding the Structure of a Test File on page 10
- Elements of a Test File on page 11

Understanding the Structure of a Test File

There are only two sections to a test file: the tests_include list, and the details section. There is a details section for each test named in the include list. Test files begin with an include statement: **tests_include:**. The include statement denotes the beginning of a list of tests that are contained within the test case. Each test name in the list is preceded by a dash (-); for example, **- test_name**. You can name the tests whatever you want, except test names cannot include spaces or special characters. Valid test names include: **check_interfaces_up**, **show_software_version**, or **test_bgp_neighbors**.

NOTE: If the tests_include list does not exist in the test file, but there are tests defined in the details section, all tests detailed within the test file are run. You can skip the running of specific tests within a test file by leaving the test name off of the tests_include list.

The details section begins with the name of the test from the include list, followed by a colon; for example, **check_interfaces_up:**. The purpose of the details section is to provide all of the test criteria for the test case, including:

- Junos OS CLI or RPC commands (one per test) that are executed on the device. The output of these commands forms the basis of the snapshot.
- The Extensible Markup Path Language (XPath) that identifies the starting node in the Junos OS hierarchy from which the tests are run.
- Test operators that help to analyze the output from the commands.
- Info messages that are returned as the result of successful execution of the test operators.
- Error messages that are returned as the result of failure of the test operators.

When using the **get-config** RPC command, you can filter for specific configuration information using the **filter_xml:** statement. As an example, the **filter_xml: configuration/system/login**statement causes the device to filter out all of the configuration elements except those in the Junos OS /system/login hierarchy. This statement only works on the specific RPC command **get-config**. For more information, see "Example: Creating the Junos Snapshot Administrator in Python Test Files" on page 32.

Elements of a Test File

Like the other YAML files used with jsnapy, test files are structured using key-value pairs. The keywords and values are separated by a colon (:).

For example **rpc: get-config**.

Table 5 on page 11 shows the keys available for use in a jsnapy test file and describes each key.

Table 5: Test File Keywords and Descriptions

Keyword	Description

tests_include	This keyword denotes the beginning of a jsnapy test file and the beginning of the include statement. It is followed by a colon (:) and a carriage-return. Each element of the list that follows is the name of a test case, preceded by a space, a dash, and another space and must be on a line by itself. For example:
	
	`tests_include:`
	`- show_version`
	`- check_interface_state`
	`- return_bgp_neighbors`

command	This keyword is used to specify what Junos OS command is run on the device by jsnapy. It is preceded by a space, a dash (-), and another space and followed immediately by a colon (:). For example: **- command: show version**
rpc	This keyword is used to specify what Junos OS RPC command should be run on the device by jsnapy. It is preceded by a space, a dash (-), and another space and followed immediately by a colon (:). For example: **- rpc: get-software-version**

Table 5: Test File Keywords and Descriptions *(continued)*

Keyword	Description
format	This keyword is used to format the device output. Available options are: **text** and **xml**. When comparing output that is formatted as text, only the **--diff** operation is supported. The **--check** and **--snapcheck** operations do not work on text formatted snapshots.
kwargs	This keyword only works with RPC commands. It is ignored if the **command** keyword is used in the test case definition. The keyword is used to provide arguments to the **rpc** command being sent to the device; for example: when using the **get-interface-information rpc** command, arguments like **interface-name** and **media-type** could be used to provide specific information about the interface.
item	This keyword denotes the beginning of a test definition. It is preceded by a space, a dash (-), and another space and followed immediately by a colon (:). For example: **- item:** This keyword also instructs jsnapy to match only the first node in the **xpath**.
iterate	This keyword denotes the beginning of a test definition. It is preceded by a space, a dash (-), and another space and followed immediately by a colon (:). For example: **- iterate:** This keyword also instructs jsnapy to continue to match all nodes in the **xpath**. NOTE: A test definition can use either **iterate** or **item**. They cannot be used together.
xpath	This keyword denotes the path in the Junos OS hierarchy from which you want to start tests. It is preceded by two spaces and followed by a colon (:). For example: **xpath:** The values used with this keyword are path syntax values. For example, the **/configuration/system/services**xpath selects the [**system services**] portion of the Junos OS hierarchy.
id	This keyword denotes an XPATH expression relative to the node-set being iterated that specifies a unique data element that maps the first snapshot data item with the second snapshot data item. The keyword is followed immediately by a colon (:). It is possible to use single or multiple **id** keywords under the same **iterate** or **item** XPATH blocks.
tests	This keyword denotes the beginning of the test definition. It is followed by a colon. Tests are activated only when the **--check** and **--snapcheck** options are used. Tests are not performed when taking snapshots using the **--snap** option. See "Junos Snapshot Administrator in Python Test Operators Summary" on page 37 for details about the available tests.
info	This keyword specifies what information is logged upon successful completion of the tests if the jsnapy logging level is set to debug. The information can use a jinja2 formatted template that pulls node values from the snapshot files. For example: **info: "Test successful! The admin status of the interface is <{{pre['admin-status']}}>."**

Table 5: Test File Keywords and Descriptions *(continued)*

Keyword	Description
err	This keyword specifies what information will be logged upon failure of the test if the jsnapy logging level is set to debug. The information can use a jinja2 formatted template that pulls node values from the snapshot files. For example: **err: "Test Failed! The admin-status of the interface is <{{post['admin-status']}}>."**
pre	When used with jinja2 formatted variables in the info or err message sections, this keyword causes jsnapy to search for the named variable inside of the pre-change snapshot.
post	When used with jinja2 formatted templates in the info or err message sections, this keyword causes jsnapy to search for the named variable inside of the post-change snapshot.

Related Documentation

- Example: Creating the Junos Snapshot Administrator in Python Test Files on page 32
- Understanding Junos Snapshot Administrator in Python Test Operators on page 16
- Junos Snapshot Administrator in Python Test Operators Summary on page 37

Understanding Junos Snapshot Administrator in Python Device Files

The Junos Snapshot Administrator in Python (jsnapy) devices file is a YAML file that contains IP addresses, group names, and login credentials for multiple Junos OS devices. IP addresses and login credentials within the file can be grouped by user-defined criteria such as model, device type, or location. There are no naming restrictions for the device file, so multiple device files can be created and any of them can be referenced by the jsnapy configuration files.

- Junos Snapshot Administrator in Python Device File Elements on page 13
- Example Devices File on page 14

Junos Snapshot Administrator in Python Device File Elements

Jsnapy device files are built using combinations of keywords and values that are pertinent to your network. Like any YAML file, structure is derived through indentation (one or more spaces), sequence items are denoted by a dash (-), and key value pairs are separated by a colon (:). The IP addresses used to identify the Junos OS hosts to contact appear on lines by themselves, preceded by a dash (-) and followed by a colon (:); for example:

- 192.0.2.1:

If you want to create groups of devices in the devices file, you can enter a group name on a line by itself, followed by a colon (:); for example:

MX_Devices:

The host IP addresses listed after the group name are considered a part of this group until the next group name is encountered in the devices file. This allows smaller groups to be contacted by referencing the group name (or names) from within the hosts section

of the configuration file. If there are no group names configured in the devices file, the entire list of devices is contacted whenever that particular devices file is referenced in the configuration file.

Login credentials follow each IP address unless an SSH trust relationship has been established between the jsnapy server and the Junos OS host. The login credentials specify the username and password needed for that particular host IP using the keywords **username:** and **passwd:**, respectively.

Comments can be inserted into the devices file at any location by simply starting the comment with a hash (#); for example, **#This is a comment**.

Jsnapy device files should be stored in the same directory as the configuration file that is being used with the **jsnapy** command.

The following commented example shows the structure of a jsnapy devices file.

Example Devices File

```
user@host:~> cat devices_example.yml
#Junos Snapshot Administrator in Python - Example Devices File
MX_Devices: #This is a group name.
 - 192.0.2.33: #IP address of a Junos OS host
   username: username
   passwd: password
 - 192.0.2.65:
   username: yourname
   passwd: yourpassword
QFX: #<-- Group name
 - 192.0.2.12:
   username: switchuser
   password: password1
 - 198.51.100.10:
   username: switchuser
   passwd: password1
Finance:
 - 192.0.2.36:
   username: financeuser
   passwd: 1password
 - 192.0.2.11:
   username: financeuser
   passwd: 1password
RandD:
 - 198.51.100.1 #No login credentials needed for these hosts because of existing
 SSH trust relationships.
 - 198.51.100.254
```

Related Documentation

- Junos Snapshot Administrator in Python Overview on page 3
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7
- Understanding Junos Snapshot Administrator in Python Test Files on page 10

Understanding Junos Snapshot Administrator in Python Mail Files

The Junos Snapshot Administrator in Python (jsnapy) mail files are YAML-formatted files that define the details needed for jsnapy to send e-mail regarding the results of completed snapshots and comparison tests.

General Information

You can name mail files whatever you want except that the name cannot contain spaces or special characters. The mail file should be stored in the same directory as the jsnapy configuration file to which it is linked. When a mail file is referenced from a configuration file, e-mail will be sent every time jsnapy is run with that configuration using either the **--check** or **--snapcheck** options, regardless of success or failure of the tests that are run on the Junos OS host. If jsnapy is run with the **--check** option, then no mail is sent.

After installation and initialization, there is a sample mail file named **send_mail.yml** in the **/etc/jsnapy/samples** directory. This file shows the available fields and some sample values.

```
user@jsnapy-server:~> cat /etc/jsnapy/samples/send_mail.yml
to: abc@juniper.net
from: foo@gmail.com
sub: "Sample Jsnapy Results, please verify"
recipient_name: ABC
passwd: pass123
#server: smtp.gmail.com optional
sender_name: "Juniper Networks"
```

This sample mail file will not work for sending mail without modification.

Jsnapy uses sendmail on the jsnapy server to send SMTP e-mail to the Gmail server **smtp.gmail.com** by default.

Understanding the Structure of a Mail File

The jsnapy mail file has several required parameter-value pairs and two optional ones. All parameter-value pairs are separated by a colon (:). Table 6 on page 15 describes both the required and optional parameters for jsnapy mail file configuration.

Table 6: Key-Value Pairs in jsnapy Mail Files

Parameter	Description
to	This value is the e-mail address to which you want the e-mail messages sent.
from	This value is an e-mail address that is used as the from field of the outgoing e-mail. It is also used as the username for login to the SMTP server specified by the **server** parameter. NOTE: If no value is given for the **server** parameter, then this value should be a valid e-mail address @gmail.com.

Table 6: Key-Value Pairs in jsnapy Mail Files *(continued)*

Parameter	Description
sub	This value is used as the subject line in the outgoing e-mail.
recipient_name	This value is used as the salutation in the body of the e-mail.
passwd	This value is the password associated with the e-mail account value in the **from** parameter.
server	This optional value is the IP address or DNS name of an SMTP server that jsnapy will use for sending e-mail. If no value is given, jsnapy will use smtp.gmail.com for sending e-mail.
port	This optional value tells jsnapy to send e-mail on a TCP port other than port 587 (SMTP).
sender_name	This value is the name used as the signature line in the body of the e-mail.

Related Documentation

- Junos Snapshot Administrator in Python Overview on page 3
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7

Understanding Junos Snapshot Administrator in Python Test Operators

Junos Snapshot Administrator in Python (jsnapy) enables you to capture and audit runtime environment snapshots of your networked devices running Junos OS. The **jsnapy** configuration file defines the scope of a snapshot and specifies the test files to use for either a single snapshot or a comparison of two snapshots. The **--snapcheck** option takes a single snapshot and evaluates the results, and the **--check** option compares the results of two separate snapshots. Within the test file, you specify the commands to be run on the Junos device, the XPath from which to start, an optional id value, and the tests to run against the command output. The test cases use test operators to either evaluate data elements in a single snapshot or compare data elements in two separate snapshots.

Jsnapy provides numerous relational operators that test for existence, equality, inequality, size, and inclusion in or exclusion from a specific range or list. Specific operators work with different operand types including strings, numbers, and XML elements. You should construct test cases using test operators that pertain to the type of check being performed. For a list of available operators, see "Junos Snapshot Administrator in Python Test Operators Summary" on page 37.

Junos Snapshot Administrator in Python uses a few test operators, **delta**, **list-no-less**, **list-no-more**, and **no-diff**, to compare elements or element values in two separate snapshots. Test cases using these test operators are executed when you use the **--check** option. When you use the **--snapcheck** option, which is specific to a single collection, test cases using these test operators are effectively ignored. Junos Snapshot Administrator in Python outputs a message when a test case is ignored, as shown in the following sample output:

```
--------------------------------------------------------------------------
CHECKING SECTION: ospf-checks
```

```
------------------------------------------------------------------------------
INFO: snapcheck mode: skipping test: list-not-less
INFO: snapcheck mode: skipping test: no-diff
+ TEST PASSED: All OSPF neighbors are up
+ TEST PASSED: OSPF neighbors must have the same priority value
INFO: snapcheck mode: skipping test: no-diff
```

Related Documentation

- Junos Snapshot Administrator in Python Test Operators Summary on page 37
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7

PART 2

Installation

CHAPTER 3

Installing Junos Snapshot Administrator in Python

Installing Junos Snapshot Administrator in Python

You install Junos Snapshot Administrator in Python (jsnapy) on a remote server in the network. Prior to installing jsnapy, ensure that the remote server is running an OS that is capable of running Python 2.6 or later. This includes, but is not limited to:

- Linux (Debian, Ubuntu, Fedora, CentOS, and FreeBSD)
- Mac OS X

NOTE: **Although Microsoft Windows can run Python 2.6, jsnapy is not supported on Windows.**

Due to the wide range of possible supported OSs, there are dependencies within each OS that must also be fulfilled. Table 7 on page 23 shows the dependencies for each OS.

Table 7: jsnapy Dependencies

OS	Dependencies
CentOS	• pip • python-devel • libxml2-devel • libxslt-devel • gcc • openssl • libffi-devel
Debian	• python-pip • python-dev • libxml2-dev • libxslt-dev • libssl-dev • libffi-dev
Fedora	• python-pip • python-devel • libxml2-devel • libxslt-devel • gcc • openssl • libffi-devel
FreeBSD	• py27-pip • libxml2 • libxslt
OSX	• xcode • xquartz • pip

Table 7: jsnapy Dependencies *(continued)*

OS	Dependencies
Ubuntu	• python-pip • python-dev • libxml2-dev • libxslt-dev • libssl-dev • libffi-dev

There are two methods that can be used to install jsnapy: using the Python package manager, pip, or using a Python setup script, setup.py, from cloned github source code.

To install jsnapy using pip, enter the following command

user@jsnapy-server:~> sudo pip install git+https://github.com/Juniper/jsnapy.git

To download the source code and install with the setup script:

1. Clone the source code from github using one of the following options:
   a. Use the git command at the server CLI:

      user@jsnapy-server:~> git clone https://github.com/Juniper/jsnapy.

   b. Download the source code as a zip file from https://github.com/Juniper/jsnapy by clicking the Clone or Download button.

      NOTE: One of the options after clicking the **Clone** or **Download** button is to Open in Desktop. This option refers to the GitHub Desktop. This installation guide does not cover the installation or use of GitHub Desktop or installing jsnapy using GitHub Desktop.

2. Unzip the downloaded file:

   user@server:~> unzip jsnapy-master.zip

3. Go to the jsnapy-master folder:

   user@server:~> cd jsnapy-master

4. Use pip to install jsnapy:

   user@server:~> sudo pip install dist/jsnapy-0.1.tar.gz

Jsnapy is under ongoing development by Juniper and is posted on GitHub. As such, there are often updates available. To update jsnapy to the latest development code after it is installed on your server, use the pip command with the update flag, as shown:

```
user@server:~> sudo pip install -U git+https://github.com/Juniper/jsnapy.git
```

To update to the latest release code, use the pip command with the update flag, as shown:

```
user@server:~> sudo pip install -U jsnapy
```

The jsnapy installer creates the files and directories listed in Table 7 on the jsnapy server under **/etc/jsnapy**:

Table 8: Directories and Files Contained in /etc/jsnapy

Directory or File Name	Purpose
jsnapy.cfg	This file specifies the default paths for configuration files, snapshot files, and test files.
logging.yml	This file specifies the settings for logging of jsnapy events and messages.
samples	This directory contains an assortment of sample configuration and test files.
snapshots	This directory is the default location for the storage of snapshot files created by jsnapy. The snapshots are stored here by device and test.
testfiles	This directory is the default location for storing jsnapy test files.

Related Documentation

- Junos Snapshot Administrator in Python Overview on page 3
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7

PART 3

Configuration

CHAPTER 4

Creating Junos Snapshot Administrator in Python Configuration Files

Example: Creating the Junos Snapshot Administrator in Python Configuration Files

This example demonstrates how to create some basic Junos Snapshot Administrator in Python (jsnapy) configuration files. The examples in this topic us specific numbers of spaces sometimes before any other text is entered on a line. These spaces are required for proper formatting of the YAML file.When usernames and passwords need to be typed in these examples, replace the word **user** with a valid username and the **<password>** text with a valid password.

Requirements

- Junos Snapshot Administrator in Python Release 1.0 is installed on the server.

Overview

This topic describes, how to create multiple jsnapy configuration files:

- A configuration file that connects to one specific host and runs one specific test
- A configuration file that connects to one specific host, runs two tests, and sends the results by e-mail
- A configuration file that connects to a group of hosts, runs one test, and stores the snapshot in an SQLite database

Junos Snapshot Administrator in Python configuration files consist of two mandatory sections: hosts and tests, and two optional sections: sqlilte and mail. The configuration files that direct jsnapy to connect (using PyEZ and NETCONF) to a single host define the host IP address and credentials within the configuration file, while the configuration files that direct jsnapy to connect to multiple hosts use the include feature of YAML files to read in a list of hosts and credentials from a separate YAML file. The tests to be performed against the hosts are always defined in separate YAML test files whose names are specified within the tests section of the configuration files. Multiple test files can be

specified. For more information regarding test files, see "Understanding Junos Snapshot Administrator in Python Test Files" on page 10 and "Creating the Junos Snapshot Administrator in Python Test Files" on page 32 .

SQLite configuration is done within the configuration file, while e-mail configuration is done in a separate mail configuration file, the name of which is specified in the configuration file. See <understanding mail config> and <creating mail config> for information about setting up jsnapy to use e-mail.

Comments can be inserted into the file at any location by starting the comment with the pound sign (#).

One Host, One Test

Step-by-Step Procedure

Two sections are required for this configuration file: hosts and tests. For this example, use a text editor to create a new file named **configuration_one_host_one_test.yml**.

1. Create the hosts section.

```
hosts:
```

2. Specify the IP address and login credentials.

```
  - device: 192.0.2.12
    username: user
    passwd: <password>
```

3. Create the tests section.

```
tests:
```

4. Specify the test file to use.

```
  - test_exists.yml
```

Results

The resulting configuration file can be shown in the terminal using the **cat** command.

```
user@jsnapy-server:~> cat configuration_one_host_one_test.yml
hosts:
  - device: 192.0.2.12
    username: user
    password: <password>
tests:
  - test_exists.yml
```


One Host, Two Tests with E-mail

Step-by-Step Procedure

Three sections are required for this configuration file: hosts, tests and mail. For this example, use a text editor to create a new file named **configuration_one_host_two_tests_e-mail.yml**.

1. Create the hosts section.

```
hosts:
```

2. Specify the IP address and login credentials.

```
  - device: 192.0.2.11
    username: user
    passwd: <password>
```

3. Create the tests section.

```
tests:
```

4. Specify the test file to use.

```
   - test_not_less.yml
  - test_not_more.yml
```

5. Specify the e-mail configuration file.

```
mail: send_mail.yml
```

Results

The resulting configuration file can be shown in the terminal using the **cat** command.

```
user@jsnapy-server:~> cat configuration_one_host_two_tests_e-mail.yml
hosts:
  - device: 192.0.2.11
    username: user
    password: <password>
tests:
  - test_not_less.yml
  - test_not_more.yml
mail: send_mail.yml
```

Multiple Hosts, One Test with SQLite Database Storage

Step-by-Step Procedure

Three sections are required for this configuration file: hosts, tests, and sqlite. Use a text editor to create a new file named **configuration_multiple_hosts_one_test_sqlite**.

1. Create the hosts section.

```
hosts:
```

2. Specify the file in which the host IP addresses and login credentials are contained.

```
- include:  devices.yml
  group: MX
```

3. Create the tests section.

```
tests:
```

4. Specify the test file to use.

```
- test_diff.yml
```

5. Create the sqlite section.

```
sqlite:
```

6. Configure the slqlite parameters.

```
- store_in_sqlite: yes
  check_from_sqlite: yes
  database_name: jsnapy.db
```

Results The resulting configuration file can be shown in the terminal using the **cat** command.

```
user@jsnapy-server:~> cat configuration_multiple_hosts_one_tests_sqlite.yml
hosts:
  - include: devices.yml
    group: MX
tests:
  - test_diff.yml
sqlite:
  - store_in_sqlite: yes
    check_from_sqlite: yes
    database_name: jsnapy.db
```

Related Documentation

- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7
- Junos Snapshot Administrator in Python Overview on page 3

Example: Creating the Junos Snapshot Administrator in Python Test Files

This example demonstrates how to create some basic Junos Snapshot Administrator in Python (jsnapy) test files. The examples in this topic use specific numbers of spaces sometimes before any other text is entered on a line. These spaces are required for proper formatting of the YAML file.

- Requirements on page 33
- Overview on page 33

Requirements

- Junos Snapshot Administrator in Python Release 1.0 is installed on the server.

Overview

This topic describes how to create two test files using various elements available for test file creation:

- A test file named **test_sw_version.yml** that just captures the Junos OS software release version
- A test file named **filter_config_info.yml** that requests filtered information from the **get_config** RPC command.

Jsnapy test files contain the detailed commands, tests, hierarchy locations, and test elements necessary to create or compare snapshots of the runtime environments of Junos OS devices. You can create as many test files as you want, store them wherever you want on the jsnapy server, and name them as you see fit (with the exception of spaces and special characters). There are two primary parts to a jsnapy test file: the named list of tests under a tests_include heading, and the detail sections, one detail section for each named test under the tests_include section.

Comments can be inserted into the test file at any location by starting the comment with the pound sign (#).

Capture Junos OS Software Release Version

Step-by-Step Procedure

For this example, use a text editor to create a new file named **test_sw_version.yml**. Create the file in the same directory as the configuration file that you intend to use or in the **/etc/jsnapy/testfiles/** directory.

1. Create the tests_include section.

```
tests_include:
```

2. Set the test name.

```
  - test_sw_version
```

3. Start the test details section.

```
test_sw_version:
```

4. Set the command to be sent. In this case it is the Junos OS command **show version**.

```
  - command: show version
```

5. Specify whether your tests look for the first instance of a value or all instances of a value. In this case we are looking for only the first instance.

```
    - item:
```

6. Specify the XPATH location from which to start the snapshot or search.

```
        xpath: '//software-information'
```

7. Begin the tests section, if there is one. In this case there will be a tests section.

```
        tests:
```

8. Specify what tests will be run. In this case we will enable this test file to check whether the Junos OS version is the same between snapshots.

```
          - all-same: junos-version
```

9. Set the error message to display. In the even that the test fails, this message will be displayed.

```
            err: "Test Failed!!! The versions are not the same. From the
   PRE snapshot, the version is: <{{pre['junos-version']}}>. From the POST
   snapshot, the version is <{{post['junos-version']}}>!! "
```

10. Set the info message to display. In the event that the test succeeds, this message will be displayed.

```
            info: "Test Succeeded!! The Junos OS version is:
   <{{post['junos-version']}}>!!!"
```

Results

The resulting test file can be shown in the terminal using the **cat** command.

```
user@jsnapy-server:~> cat test_sw_version.yml
tests_include:
  - test_sw_version

test_sw_version:
  - command: show version
  - item:
      xpath: '//software-information'
      tests:
        - all-same: junos-version
          err: "Test Failed!!! The versions are not the same. From the PRE
snapshot, the version is: <{{pre['junos-version']}}>. From the POST snapshot, the
 version is <{{post['junos-version']}}>!! "
          info: "Test Succeeded!! The Junos OS version is:
<{{post['junos-version']}}>!!!"
```

Request Filtered Configuration Information

Step-by-Step Procedure

This example creates a test file that uses an RPC command, **get-config**, to retrieve the device configuration. An argument is provided to the RPC command so that jsnapy filters that device configuration information so that the snapshot contains only the filtered information rather than the entire configuration. In this example we will configure the test file to return only the **host-name** information.

1. Create the tests_include section.

```
tests_include:
```

2. Set the test name.

```
  - test_rpc_filtering
```

3. Begin the test details section.

```
test_rpc_filterin:
```

4. Set the RPC command to be used.

```
  - rpc: get-config
```

5. Provide the arguments keyword to tell jsnapy that arguments follow.

```
  - kwargs:
```

6. Enter the argument name and value. In this case we are filtering for the host name.

```
      filter_xml: configuration/system/host-name
```

Results

The resulting configuration can be shown in the terminal using the **cat** command.

```
user@jsnapy-server:~> cat filter_config_info.yml
tests_include:
  - test_rpc_filtering

test_rpc_filtering:
  - rpc: get-config
  - kwargs:
      filter_xml: configuration/system/host-name
```

Related Documentation

- Understanding Junos Snapshot Administrator in Python Test Files on page 10
- Understanding Junos Snapshot Administrator in Python Mail Files on page 15

- Understanding Junos Snapshot Administrator in Python Device Files on page 13
- Junos Snapshot Administrator in Python Overview on page 3

CHAPTER 5

Junos Snapshot Administrator in Python Test Operators

Junos Snapshot Administrator in Python Test Operators Summary

Junos Snapshot Administrator in Python (jsnapy) enables you to capture and audit runtime environment snapshots of your networked devices running Junos OS. The Junos Snapshot Administrator configuration file defines the scope of a snapshot and the evaluation criteria for either a single snapshot or a comparison of two snapshots. The configuration files reference the names of test files that define the details of the tests to be run. Within the test file, you can create test cases that evaluate or compare content from specific Junos OS commands. The test cases use test operators to either evaluate data elements in a single snapshot or compare data elements in two separate snapshots.

Table 9 on page 38 lists the Junos Snapshot Administrator test operators along with a brief description of each operator. Operators are grouped by operand type.

Table 9: Junos Snapshot Administrator Test Operators

Operator	Description
Compare Elements or Element Values in Two Snapshots	
delta	Compare the change in value of a specified data element, which must be present in both snapshots, to a specified delta. You can specify the delta as an absolute, positive, or negative percentage, or an absolute, positive, or negative fixed value.
list-not-less	Determine if the specified items are present in the first snapshot but are not present in the second snapshot.
list-not-more	Determine if the specified items are present in the second snapshot but are not present in the first snapshot.
no-diff	Compare specified data elements that are present in both snapshots, and verify that the value is the same.
Operate on Elements with Numeric or String Values	
all-same	Check if all content values for the specified elements are the same. Optionally, you can check if all content values for the specified elements are the same as the content value of a reference item.
is-equal	Test if an XML element string or integer value matches a given value.
not-equal	Test if an XML element string or integer value does not match a given value.
Operate on Elements with Numeric Values	
in-range	Test if the XML element value is within a given numeric range.
is-gt	Test if the XML element value is greater than a given numeric value.
is-lt	Test if the XML element value is less than a given numeric value.
not-range	Test if the XML element value is outside a given numeric range.
Operate on Elements with String Values	
contains	Determine if an XML element string-value contains the provided string value.
is-in	Determine if an XML element string-value is included in a specified list of string values.
not-in	Determine if an XML element string value is excluded from a specified list of string values.
Operate on XML Elements	
exists	Verify the existence of an XML element in the snapshot.

Table 9: Junos Snapshot Administrator Test Operators *(continued)*

Operator	Description
not-exists	Verify the lack of existence of an XML element in the snapshot.

Related Documentation

- Understanding Junos Snapshot Administrator in Python Test Operators on page 16
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7
- Junos Snapshot Administrator in Python Overview on page 3

all-same operator

Syntax

```
all-same xpath-expression, [ [reference-item] ] {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that checks if all content values for the specified elements are the same. If you include the optional reference item, the operator checks if all content values for the specified elements are the same as the content value of the reference item.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

reference-item—(Optional) XPath expression specifying the reference element against which value all other element values are compared.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following example code checks that all OSPF neighbors have the same priority value. If any priority values are different, the code generates the error message.

```
ospf-check {
   command show ospf neighbor;
   iterate ospf-neighbor {
      all-same neighbor-priority {
         info OSPF neighbors must have the same priority value;
         err "OSPF interface %s has mismatch priority %s", interface-name,
neighbor-priority;
      }
   }
}
```

The following example code checks that all OSPF neighbors have the same priority value as that of interface ae19.0. If any priority values are different from that of the reference interface, the code generates the error message.

```
ospf-check {
   command show ospf neighbor;
   iterate ospf-neighbor {
      all-same neighbor-priority, [interface-name = 'ae19.0']  {
         info OSPF neighbors must have the same priority value as ae19;
         err "OSPF interface %s has mismatch priority %s", interface-name,
neighbor-priority;
      }
   }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- is-equal operator on page 44
- not-equal operator on page 51

contains operator

Syntax

```
contains xpath-expression, test-string {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that determines if an XML element string value contains the provided *test-string* value.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

test-string—String searched for in the XML element value. Enclose the string in quotation marks.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples Given the following partial XML output from the **show version** operational mode command:

```
<multi-routing-engine-results>
     <multi-routing-engine-item>
          <re-name>re0</re-name>
          <software-information>
               <host-name>R1</host-name>
               <product-model>mx960</product-model>
               <product-name>mx960</product-name>
```



```
                    <package-information>
                         <name>junos</name>
                         <comment>JUNOS Base OS boot [10.4R7.5]</comment>
                    </package-information>
                    <package-information>
                         <name>jbase</name>
                        <comment>JUNOS Base OS Software Suite [10.4R7.5]</comment>

                    </package-information>
...
```

The following test case selects the first **package-information** node set, and checks the child **comment** element for a Junos OS release number of 10.4. If the 10.4 release string is not found, the code generates the error message. The error string includes the comment value containing the release number found on that Routing Engine.

```
version-check {
   command show version invoke-on all-routing-engines;
   iterate //software-information {
      contains package-information[1]/comment, "10.4R" {
         info Checking Junos version for 10.4 release;
         err "Found %s on RE %s", package-information[1]/comment, ../re-name;

      }
   }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- is-in operator on page 46
- not-in operator on page 53

delta operator

Syntax

```
id id;
delta xpath-expression, delta-value {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that compares the change in value of a specified data element, which must be present in both snapshots, to a specified delta. You can specify the delta as an absolute, positive, or negative percentage, or as an absolute, positive, or negative fixed value.

Parameters *delta-value*—Delta value expressed as a percentage or fixed value and against which the change in the element value is compared.

Expressed as a percentage:

- Absolute percentage—Indicates that either an increase or decrease in the value greater than the absolute percentage is an error, for example, 10%.
- Positive percentage—Indicates that only an increase in the value greater than the absolute percentage is an error, for example, +10%.
- Negative percentage—Indicates that only a decrease in the value greater than the absolute percentage is an error, for example, -10%.

Expressed as a fixed value:

- Absolute fixed value—Indicates that either an increase or decrease in the value greater than the absolute value is an error, for example, 500.
- Positive fixed value—Indicates that only an increase in the value greater than the absolute value is an error, for example, +500.
- Negative fixed value—Indicates that only a decrease in the value greater than the absolute value is an error, for example, -500.

err *string*—Statement generated when the test case returns false.

id *id*—XPath expression relative to the data content that specifies a unique data element that maps the first snapshot data item to the second snapshot data item. To create a unique ID based on multiple element values, define multiple **id** statements.

info *string*—Description of the test case.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following test cases check the BGP route prefix count after a maintenance window. To ensure that the BGP peers were restored, the code checks the prefix counts for a delta change of -10%. That is, if the new prefix count is less than 90% of the original prefix count, the test reports an error.

```
bgp-checks {
   command show bgp summary;
   iterate bgp-rib {
      id name;
      delta total-prefix-count, -10% {
         info BGP total prefix count should not change by more than -10%;
         err "  BGP rib: %s total prefix count has exceeded threshold", name;

         err "     pre-check: %s, post-check: %s", $PRE/total-prefix-count,
$POST/total-prefix-count;
      }
      delta active-prefix-count, -10% {
         info BGP active prefix count should not change by more than -10%;
         err "  BGP rib: %s total prefix count has exceeded threshold", name;

         err "     pre-check: %s, post-check: %s", $PRE/active-prefix-count,
 $POST/active-prefix-count;
      }
```



```
    }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- list-not-less operator on page 48
- list-not more operator on page 49
- no-diff operator on page 50

exists operator

Syntax

```
exists xpath-expression {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that verifies the existence of an XML element in the snapshot.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

xpath-expression—XPath expression selecting the elements to test.

Usage Examples The following test case checks for active chassis alarms. If the **no-active-alarm** element exists, there are no active chassis alarms. Otherwise, the code reports an error indicating that there are active alarms.

```
alarm-checks {
   command show chassis alarms;
   item alarm-summary {
      exists no-active-alarm {
         info No chassis alarms;
         err "There are %s chassis alarms", active-alarm-count;
      }
   }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- not-exists operator on page 52

in-range operator

Syntax

```
in-range xpath-expression, integer-start, integer-end {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that tests if an XML element value is within a given numeric range.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

integer-start—Numeric value defining the start of the range.

integer-end—Numeric value defining the end of the range.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following test case checks the OSPF database and tests if each router has between 5 and 10 links. If the **link-count** value falls outside the specified range, the code reports an error.

```
ospf-db-checks {
     command show ospf database detail;
     iterate //ospf-router-lsa {
          in-range link-count, 5, 10 {
               info OSPF router links [5 - 10];
             err "Router %s has %s links", ../advertising-router, link-count;

          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- is-gt operator on page 45
- is-lt operator on page 47
- not-range operator on page 54

is-equal operator

Syntax

```
is-equal xpath-expression, value {
     info string;
     err "string";
```

```
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that tests if the XML element string or integer value matches a given value.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

value—String or integer value against which the element value is compared. Enclose string values in quotation marks.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following test case prints an error if the Routing Engine in slot 0 (re0) is not the master Routing Engine.

```
re0-master {
     command show chassis routing-engine;
     item route-engine[slot = '0'] {
          is-equal mastership-state, "master" {
               info re0 is always master;
               err "  re0 is not master, rather %s", mastership-state;
               err "     Correct so that re0 is the master Routing Engine!";
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- all-same operator on page 39
- not-equal operator on page 51

is-gt operator

Syntax

```
is-gt xpath-expression, integer-value {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that tests if an XML element value is greater than a given numeric value.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

integer-value—Numeric value against which to compare the XML element value.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples

The following test case checks that each OSPF interface has at least 1 neighbor by verifying that the **neighbor-count** is greater than 0. If the neighbor count is 0, the code reports an error.

```
ospf-int-checks {
     command show ospf interface;
     iterate ospf-interface {
          is-gt neighbor-count, 0 {
               info OSPF interfaces must have at least 1 neighbor;
               err "OSPF interface %s does not have any neighbors",
interface-name;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- in-range operator on page 44
- is-lt operator on page 47
- not-range operator on page 54

is-in operator

Syntax

```
is-in xpath-expression, string-list {
     info string;
     err "string";
    [err "string"];
}
```

Release Information

Operator introduced in Junos Snapshot Administrator Release 1.0.

Description

Junos Snapshot Administrator test operator that determines if an XML element string value is included in the specified list of string values.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

string-list—Comma-separated list of strings against which to compare the XML element value for inclusion. Enclose each string in quotation marks.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples Given the following XML output from the **show rsvp session** operational mode command:

```
<rsvp-session-data>
     <session-type>Ingress</session-type>
     <count>3</count>
     <rsvp-session junos:style="brief">
          <destination-address>10.255.20.137</destination-address>
          <source-address>10.255.20.167</source-address>
          <lsp-state>Dn</lsp-state>
          <route-count>0</route-count>
          <rsb-count>0</rsb-count>
          <resv-style>-</resv-style>
          <label-in>-</label-in>
          <label-out>-</label-out>
          <name>test</name>
     </rsvp-session>
...
```

The following test case checks if the RSVP session **lsp-state** has a value of either **Up** or **NotInService**. If the **lsp-state** value is not in the specified string list, the code reports an error.

```
rsvp-checks {
     command show rsvp session;
     iterate rsvp-session-data/rsvp-session {
          is-in lsp-state, "Up", "NotInService" {
               info RSVP LSP state is [Up | NotInService];
             err " RSVP session to %s has LSP state %s.", destination-address,
 lsp-state;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- contains operator on page 40
- not-in operator on page 53

is-lt operator

Syntax

```
is-lt xpath-expression, integer-value {
     info string;
     err "string";
    [err "string"];
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that tests if an XML element value is less than a given numeric value.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

integer-value—Numeric value against which to compare the XML element value.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples

The following test case checks whether a BGP peer flaps by testing if the **flap-count** is less than 5. If the **flap-count** is greater than 5, the code reports an error.

```
bgp-peer-checks {
     command show bgp neighbor;
     iterate bgp-peer {
          is-lt flap-count, 5 {
               info BGP peer flap-count < 5;
               err "BGP peer %s has %s flaps", peer-address, flap-count;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- in-range operator on page 44
- is-gt operator on page 45
- not-range operator on page 54

list-not-less operator

Syntax

```
id id;
list-not-less {
     info string;
     err "string";
    [err "string";]
}
```

Release Information

Operator introduced in Junos Snapshot Administrator Release 1.0.

Description

Junos Snapshot Administrator test operator that determines if the specified XML elements are present in the first snapshot but are not present in the second snapshot. The **list-not-less** test-operator validates the existence in the second snapshot of the elements defined by the **id** statement.

Parameters

err *string*—Statement generated when the test case returns false.

id *id*—XPath expression relative to the data content that specifies a unique data element that maps the first snapshot data item to the second snapshot data item. To create a unique ID based on multiple element values, define multiple **id** statements.

info *string*—Description of the test case.

Usage Examples

The following test case checks if the OSPF neighbors that existed before the device maintenance are present after the device maintenance. If any OSPF neighbors present in the first snapshot are missing from the second snapshot, the code reports an error.

```
ospf-check {
     command show ospf neighbor;
     iterate ospf-neighbor {
          id interface-name;
          list-not-less {
               info OSPF interface list check;
               err "OSPF interface gone missing: %s going to %s",
interface-name, neighbor-address;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- delta operator on page 41
- list-not more operator on page 49
- no-diff operator on page 50

list-not more operator

Syntax

```
id id;
list-not-more {
     info string;
     err "string";
    [err "string";]

}
```

Release Information

Operator introduced in Junos Snapshot Administrator Release 1.0.

Description

Junos Snapshot Administrator test operator that determines if the specified XML elements are present in the second snapshot but are not present in the first snapshot. The **list-not-more** test-operator validates the existence in the first snapshot of the elements defined by the **id** statement.

Parameters

err *string*—Statement generated when the test case returns false.

id *id*—XPath expression relative to the data content that specifies a unique data element that maps the first snapshot data item to the second snapshot data item. To create a unique ID based on multiple element values, define multiple **id** statements.

info *string*—Description of the test case.

Usage Examples The following example check chassis alarms. This configuration uses both **list-not-less** and **list-not-more** to check for any changes in the alarms. The data elements referenced in the **list-not-more err** section are elements from the second snapshot that were not present in the first snapshot.

```
alarm-checks {
     command show chassis alarms;
     item alarm-summary {
          not-exists active-alarm-count {
               info No chassis alarms;
               err "There are %s chassis alarms", active-alarm-count;
          }
     }
     iterate alarm-detail {
          id alarm-description;
          list-not-less {
               info Alarm Gone Missing;
               err "-Alarm: %s", alarm-description;
     }
          list-not-more {
               info Alarm Got More;
               err "+Alarm: %s", alarm-description;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- delta operator on page 41
- list-not-less operator on page 48
- no-diff operator on page 50

no-diff operator

Syntax

```
id id;
no-diff xpath-expression {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that compares specified data elements that are present in both the first and second snapshot collections and verifies that the value is the same.

Parameters err *string*—Statement generated when the test case returns false.

id *id*—XPath expression relative to the data content that specifies a unique data element that maps the first snapshot data item to the second snapshot data item. To create a unique ID based on multiple element values, define multiple **id** statements.

info *string*—Description of the test case.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples

The following test case reports an error if the OSPF neighbor address has changed. The **$PRE** and **$POST** variables in the second **err** statement reference the first and second data collections, respectively.

```
ospf-check {
     command show ospf neighbor;
     iterate ospf-neighbor {
          id interface-name;
          no-diff neighbor-address {
               info OSPF neighbor change check;
               err "OSPF interface %s neighbor changed!", interface-name;
             err "   was going to %s, now going to %s", $PRE/neighbor-address,
 $POST/neighbor-address;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- delta operator on page 41
- list-not-less operator on page 48
- list-not more operator on page 49

not-equal operator

Syntax

```
not-equal xpath-expression, value {
     info string;
     err "string";
    [err "string";]
}
```

Release Information

Operator introduced in Junos Snapshot Administrator Release 1.0.

Description

Junos Snapshot Administrator test operator that tests if the XML element string or integer value does not match a given value.

Parameters

err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

value—String or integer value against which the element value is compared. Enclose string values in quotation marks.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following test case prints an error if the Routing Engine in slot 0 (re0) is the master Routing Engine.

```
re0-master {
     command show chassis routing-engine;
     item route-engine[slot = '0'] {
          not-equal mastership-state, "master" {
               info re0 must not be master;
               err "     Correct so that re0 is not  the master Routing
Engine!";
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- all-same operator on page 39
- is-equal operator on page 44

not-exists operator

Syntax

```
not-exists xpath-expression {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that verifies the lack of existence of an XML element in the snapshot.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

xpath-expression—XPath expression specifying the elements to test.

Usage Examples The following test case checks for active chassis alarms. If the **active-alarm-count** element does not exist, there are no active chassis alarms. Otherwise, the code reports an error indicating that there are active alarms.

```
alarm-checks {
   command show chassis alarms;
   item alarm-summary {
```

```
        not-exists active-alarm-count {
           info No chassis alarms;
           err "There are %s chassis alarms", active-alarm-count;
        }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- exists operator on page 43

not-in operator

Syntax

```
not-in xpath-expression, string-list {
     info string;
     err "string";
    [err "string";]

}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that determines if an XML element string value is excluded from the specified list of string values.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

string-list—Comma-separated list of strings against which to compare the XML element value for exclusion. Enclose each string in quotation marks.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following test case checks that the RSVP session **lsp-state** element does not have a value of **Dn** or **Failed**. If the **lsp-state** value is in the specified string list, the code reports an error.

```
rsvp-checks {
     command show rsvp session;
     iterate rsvp-session-data/rsvp-session {
          not-in lsp-state, "Dn", "Failed" {
               info RSVP LSP state is not [Dn | Failed];
             err " RSVP session to %s has LSP state %s.", destination-address,
 lsp-state;
          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- contains operator on page 40
- is-in operator on page 46

not-range operator

Syntax

```
not-range xpath-expression, integer-start, integer-end {
     info string;
     err "string";
    [err "string";]
}
```

Release Information Operator introduced in Junos Snapshot Administrator Release 1.0.

Description Junos Snapshot Administrator test operator that tests if an XML element value is outside of a given numeric range.

Parameters err *string*—Statement generated when the test case returns false.

info *string*—Description of the test case.

integer-start—Numeric value defining the start of the range.

integer-end—Numeric value defining the end of the range.

xpath-expression—XPath expression selecting the elements to evaluate.

Usage Examples The following example checks the OSPF database and tests if each router has either less than 5 links or greater than 10 links. If the **link-count** value falls inside the specified range, the code reports an error.

```
ospf-db-checks {
     command show ospf database detail;
     iterate //ospf-router-lsa {
          not-range link-count, 5, 10 {
               info OSPF router links < 5 or > 10;
             err "Router %s has %s links", ../advertising-router, link-count;

          }
     }
}
```

Related Documentation

- *Understanding Junos Snapshot Administrator Test Operators*
- *Junos Snapshot Administrator Test Operators Summary*
- in-range operator on page 44
- is-gt operator on page 45

- is-lt operator on page 47

PART 4

Administration

CHAPTER 6

Using Junos Snapshot Administrator in Python

Using Junos Snapshot Administrator in Python

You can use Junos Snapshot Administrator in Python (jsnapy) on a device running Junos OS to capture and save a runtime environment snapshot, compare two snapshots, or capture a snapshot and immediately evaluate it.

When you take a snapshot, you provide a snapshot name. Jsnapy uses the snapshot name, target device name from the configuration.yml or device.yml file, and test file test section strings to generate snapshot filenames that uniquely identify that snapshot. For example, say you are collecting data from device junos-device.example.com, you define the snapshot name **SNAP1**, and your test file has two test sections named **ospf-checks** and **bgp-checks**. When you take a snapshot, Junos Snapshot Administrator creates the following output files:

- **junos-device.example.com_SNAP1_ospf_checks.xml**
- **junos-device.example.com_SNAP1_bgp_checks.xml**

The snapshot files are stored on the jsnapy server in the location specified by the **snapshot_path** parameter in the YAML file **/etc/jsnapy/jsnapy.cfg**. By default, the location is **/etc/jsnapy/snapshots**.

The following sections outline the Junos Snapshot Administrator commands:

Taking a Snapshot

To take a snapshot of a device, enter the following on the jsnapy server's command line:

```
user@jsnapy-server$ jsnapy --snap snapshot-name -f configuration-filename
```

The command parameters are:

- ***snapshot-name***—String used in the output filenames to uniquely identify that snapshot.
- ***configuration-filename***—Snapshot configuration filename.

For example, prior to a maintenance upgrade, user bsmith takes a snapshot of the device. The snapshot name is **preupgrade** and the configuration filename is **config-snapshot.conf**. Since bsmith did not specify his password in the configuration file config-snapshot.yml, upon connecting, the device prompts for the user's password.

```
bsmith@server$ jsnap --snap preupgrade -f config-snapshot.yml

Connecting to device junos-device.example.com ...............

Enter Password for username <bsmith> :
Connecting to device junos-device.example.com ...............
Tests Included: test_version_check
Taking snapshot of COMMAND: show version
Tests Included: check_chassis_fpc
Taking snapshot of COMMAND: show chassis fpc
```

Comparing Two Snapshots

To compare two existing snapshots using the test criteria, enter the following on the jsnapy server's command line:

```
user@jsnapy-server$ jsnapy --check snapshot1 snapshot2 -f configuration-filename
```

The command parameters are:

- ***snapshot1***—String used in the output filename to uniquely identify the first snapshot. Often this is entered as PRE, preupgrage, pre-change, or some name to denote that this snapshot came first.
- ***snapshot2***—String used in the output filename to uniquely identify the second snapshot. Often this is entered as POST, postupgrage, post-change, or some name to denote that this snapshot came second.
- ***configuration-filename***—Snapshot configuration file name.

For example, prior to and immediately following a maintenance upgrade, user bsmith takes a snapshot of the device. The snapshot names are **preupgrade** and **postupgrade**. To compare these two snapshots using the criteria defined in the configuration file **config-snapshot.yml**, bsmith issues the following command:

```
bsmith@server$ jsnap --check preupgrade postupgrade -f config-snapshot.yml
```

The Junos Snapshot Administrator output displays the target router and the test results for each of the active test sections in the configuration file. Sample output is shown here:

```
************************** Device: junos-device.example.com
**************************
Tests Included: test_version_check
******************** Command: show version ********************
PASS | All "junos-version" was changed between the pre and post snapshot. It is
now 15.1R3.3 [ 1 matched ]
```

```
------------------------------ Final Result!! ------------------------------
Total No of tests passed: 1
Total No of tests failed: 0
Overall Tests passed!!!
```

Taking and Evaluating a Snapshot

To take a snapshot and immediately evaluate it based on a pre-defined set of criteria, enter the following on the jsnapy server's command line:

user@jsnapy-server$ **jsnap --snapcheck** ***snapshot-name*** **-f** ***configuration-filename***

The command parameters are:

- ***snapshot-name***—String used in the output filenames to uniquely identify that snapshot.
- ***configuration-filename***—Snapshot configuration filename.

For example, a user wants to confirm the number of software packages installed on the router junos-device.example.com. To do this, he runs the following command on the jsnapy server's command line:

user@jsnapy-server$ **jsnapy --snapcheck postupgrade -f config-software-check.yml**

The Junos Snapshot Administrator output displays the target router and the test results for each of the active test sections in the configuration file. Sample output is shown here:

```
Connecting to device junos-device.example.com ...............

Taking snapshot of COMMAND: show version
*************************** Device: junos-device.example.com
***************************
Tests Included: test_version_check
*************************** Command: show version ***************************
PASS | All "//package-information/name" exists at xpath "//software-information"
 [ 59 matched ]
------------------------------ Final Result!! ------------------------------
Total No of tests passed: 1
Total No of tests failed: 0
Overall Tests passed!!!
```

Related Documentation

- Junos Snapshot Administrator in Python Overview on page 3
- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7
- Understanding Junos Snapshot Administrator in Python Test Operators on page 16

Example: Use Jsnapy as a Python module

This example shows how to use the features of Junos Snapshot Administrator in Python (jsnapy) in a python script or program. When username and password entries are required in this script, replace <username> and <password> with appropriate values.

A default installation of jsnapy includes many sample configuration and test files in the **/etc/jsnapy/samples/** directory, including several Python script examples named

module_check.py, **module_data.py**, **module_device.py**, and **module_snapcheck.py**. These files each demonstrate different features of jsnapy running as a module within a python script.

This example will use a slightly modified version of the Python script file, **module_data.py**, to demonstrate running jsnapy with the **snapcheck** option. The script demonstrates how to pass YAML configuration data from within the Python script.

Requirements

This example uses the following hardware and software components:

- A device running Junos OS
- An instance of jsnapy installed on a server (jsnapy server)
- A text editor with which to view and change the script

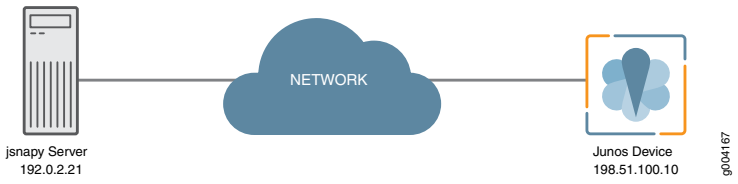
Before you write your own Python scripts for jsnapy, be sure you have a thorough understanding of both Python programming concepts and jsnapy operation.

Overview

In this example, we examine the provided Python script **/etc/jsnapy/samples/module_data.py**. We will see how the script imports jsnapy as a module, how it defines jsnapy configuration parameters, and how it displays the results of the **snapcheck** operation on a remote Junos OS device.

Topology

This example uses a simple topology where the jsnapy server connects to a single remote Junos OS device. Connections to more than one device can be achieved but will not be discussed in this example.



Examining the Python Script

The python script performs these tasks:

- Imports the required Python modules into the script.
- Defines a variable that is used to call the SnapAdmin() function from the jsnapy module.
- Defines the jsnapy configuration parameters and assigns them to a variable.
- Calls jsnapy with the **snapcheck** option using the previously defined configuration variable as the configuration argument and the snapshot filename **pre**.
- Prints the results to the terminal of the jsnapy server.

NOTE: In its original form, the Python script, **module_data.py**, imports and calls the Data pretty printer module, **pprint**. When called from the script, this module causes python to print all of the snapshot data that jsnapy receives from the Junos OS device to the terminal of the jsnapy server. Due to the length of this data, the call to **pprint** is commented out for this example by prepending the call with the pound sign (#).

Note the Comments

Step-by-Step Procedure

- Because we are examining an existing script in this example, we will point out the comments that appear as the first line of the file. Comments can be placed anywhere within the script by beginning the comment with the pound sign (#).

```
### Example showing how to pass yaml data in same file ###

from jnpr.junos import Device
```

Import Python Modules

Step-by-Step Procedure

- This script uses three imported modules, one for jsnapy, one for printing complex data to the terminal, and one for working with Junos OS devices.

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

Define a Variable for the Call to SnapAdmin

Step-by-Step Procedure

- Here the script assigns the SnapAdmin() function from the jsnapy module to a variable named **js** for ease of use and to be able to pass arguments when calling jsnapy.

```
js = SnapAdmin()
```

Define jsnapy Configuration Parameters

Step-by-Step Procedure

- Here the script defines the Junos OS host and which tests will be performed on that host. It assigns these parameters to the variable **config_data**. The triple quotes allow **config_data** to contain new-line characters.

NOTE: Another way to specify the Junos Os hosts and tests to be performed is to reference a jsnapy configuration file. The configuration file can define one or more hosts and one or more test files. Because the configuration data is contained within the python script in this example, the **config_data** variable does not reference a configuration file outside of the script but is filled with the key-value pairs that appear between the triple-quotes (“ “ “).

```
config_data = """
hosts:
  - device: 198.51.100.10
    username : <username>
    passwd: <password>
tests:
  - test_exists.yml
  - test_contains.yml
  - test_is_equal.yml
"""
```

Call jsnapy With the Defined Configuration Data

Step-by-Step Procedure

- Here the script calls jsnapy with the **snapcheck** option and passes the configuration information and the name of the snapshot file. The information returned as a result of this call is stored as values in the **snapchk** variable.

NOTE: You can access all of the available jsnapy options, check, snap, and snapcheck by appending the option after the call to SnapAdmin(). For example, js.check(*config_file*, *“snapshot1”, “snapshot2”*), js.snap(*config_file*, *“snapshot_name”*), and js.snapcheck(*config_file*, *“snapshot_name”*).

```
snapchk = js.snapcheck(config_data, "pre")
```

Print the Results to the Terminal

Step-by-Step Procedure

- Here the script loops through the returned values and prints them to the jsnapy server terminal in a readable format. This is also where we alter the script to prevent it from printing the entire snapshot to the terminal.

```
for val in snapchk:
    print "Tested on", val.device
    print "Final result: ", val.result
    print "Total passed: ", val.no_passed
    print "Total failed:", val.no_failed
    #pprint(dict(val.test_details))
```

Results

For ease of viewing, the contents of the modified script are shown below by running the **cat** command on the file **module_data.py**.

```
user@jsnapy-server$ cat module_data.py

### Example showing how to pass yaml data in same file ###
from jnpr.jsnapy import SnapAdmin
from pprint import pprint
from jnpr.junos import Device

js = SnapAdmin()

config_data = """
hosts:
  - device: 198.51.100.10
    username : <username>
    passwd: <password>
tests:
  - test_exists.yml
  - test_contains.yml
  - test_is_equal.yml
"""

snapchk = js.snapcheck(config_data, "pre")
for val in snapchk:
    print "Tested on", val.device
    print "Final result: ", val.result
    print "Total passed: ", val.no_passed
    print "Total failed:", val.no_failed
    #pprint(dict(val.test_details))
```

Verification

Verifying the Operation of the Script

Purpose Once your Python script and the needed jsnapy configuration and test files are complete, you can verify the operation of the script by running it from the jsnapy server terminal.

Action

```
user@jsnapy-server$ python module_data.py
Connecting to device 198.51.100.10 ................
Taking snapshot of COMMAND: show version
Taking snapshot of COMMAND: show version invoke-on all-routing-engines
Taking snapshot of COMMAND: show interfaces terse lo*
************************** Device: 198.51.100.10 **************************
Tests Included: test_version_check
************************** Command: show version **************************
PASS | All "//package-information/name" exists at xpath "//software-information"
 [ 59 matched ]
************************** Device: 198.51.100.10 **************************
Tests Included: test_version_check
************ Command: show version invoke-on all-routing-engines ************
Test Failed!!! Junos version does not contains package name as jbase
Test Failed!!! Junos version does not contains package name as jbase
FAIL | All "//package-information/name[1]" do not contains j" [ 57 matched / 2
failed ]
************************** Device: 198.51.100.10 **************************
Tests Included: test_interfaces_terse
********************* Command: show interfaces terse lo* *********************
PASS | All "admin-status" is equal to "up" [ 1 matched ]
------------------------------ Final Result!! ------------------------------
Total No of tests passed: 2
Total No of tests failed: 1
Overall Tests failed!!!
Tested on 198.51.100.10
Final result:  Failed
Total passed:  2
Total failed: 1
```

Meaning

Troubleshooting

To troubleshoot [item], perform these tasks:

- Troubleshooting [item] on page 67

Troubleshooting [item]

Problem

Solution

Related Documentation

- Understanding Junos Snapshot Administrator in Python Configuration Files on page 7
- Understanding Junos Snapshot Administrator in Python Test Files on page 10
- Example: Creating the Junos Snapshot Administrator in Python Configuration Files on page 29