Snimpy

Release 1.0.0

May 29, 2021

Contents

Snimpy is a Python-based tool providing a simple interface to build SNMP query. Here is a very simplistic example that allows us to display the routing table of a given host:

```
load("IP-FORWARD-MIB")
m=M("localhost", "public", 2)
routes = m.ipCidrRouteNextHop
for x in routes:
    net, netmask, tos, src = x
    print("%15s/%-15s via %-15s src %-15s" % (net, netmask, routes[x], src))
```

You can either use *Snimpy* interactively throught its console (derived from Python own console or from IPython if available) or write *Snimpy* scripts which are just Python scripts with some global variables available.

CHAPTER 1

Why another tool?

There are a lot of SNMP tools available but most of them have important drawback when you need to reliably automatize operations.

snmpget, *snmpset* and *snmpwalk* are difficult to use in scripts. Errors are printed on standard output and there is no easy way to tell if the command was successful or not. Moreover, results can be multiline (a long HexString for example). At least, automatisation is done through the shell and OID or bit manipulation are quite difficult.

Net-SNMP provides officiel bindings for Perl and Python. Unfortunately, the integration is quite poor. You don't have an easy way to load and browse MIBs and error handling is inexistant. For example, the Python bindings will return None for a non-existant OID. Having to check for this on each request is quite cumbersome.

For Python, there are other bindings. For example, pysnmp provides a pure Python implementation. However, MIBs need to be compiled. Moreover, the exposed interface is still low-level. Sending a simple SNMP GET can either take 10 lines or one line wrapped into 10 lines.

The two main points of *Snimpy* are:

- very high-level interface relying on MIBs
- raise exceptions when something goes wrong

Meantime, another Pythonic SNMP library based on Net-SNMP has been released: Easy SNMP. Its interface is a less Pythonic than *Snimpy* but it doesn't need MIBs to work.

CHAPTER 2

Contents

2.1 Installation

At the command line:

```
$ easy_install snimpy
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv snimpy
$ pip install snimpy
```

Snimpy requires libsmi, a library to access SMI MIB information. You need to install both the library and the development headers. If *Snimpy* complains to not find `smi.h`, you can help by specifying where this file is located by exporting the appropriate environment variable:

```
$ export C_INCLUDE_PATH=/opt/local/include
```

On Debian/Ubuntu, you can install libsmi with:

```
$ sudo apt-get install libffi-dev libsmi2-dev snmp-mibs-downloader
```

On RedHat and similar, you can use:

```
$ sudo yum install libffi-devel libsmi-devel
```

On OS X, if you are using homebrew, you can use:

```
$ brew install libffi
$ brew install libsmi
```

On Debian and Ubuntu, *Snimpy* is also available as a package you can install with:

```
$ sudo apt-get install snimpy
```

2.2 Usage

2.2.1 Invocation

There are three ways to use *Snimpy*:

1. Interactively through a console.
2. As a script interpreter.
3. As a regular Python module.

Interactive use

Snimpy can be invoked with either *snimpy* or *python -m snimpy*. Without any other arhument, the interactive console is spawned. Otherwise, the given script is executed and the remaining arguments are served as arguments for the script.

When running interactively, you get a classic Python environment. There are two additional objects available:

- The *load()* method that takes a MIB name or a path to a filename. The MIB will be loaded into memory and made available in all SNMP managers:

```
load("SNMPv2-MIB")
load("/usr/share/mibs/ietf/IF-MIB")
```

- The *M* class which is used to instantiate a manager (a SNMP client):

```
m = M()
m = M(host="localhost", community="private", version=2)
m = M("localhost", "private", 2)
m = M(community="private")
m = M(version=3,
      secname="readonly",
      authprotocol="MD5", authpassword="authpass",
      privprotocol="AES", privpassword="privpass")
```

A manager instance contains all the scalars and the columns in MIB loaded with the *load()* method. There is no table, node or other entities. For a scalar, getting and setting a value is a simple as:

```
print(m.sysDescr)
m.sysName = "newhostname"
```

For a column, you get a dictionary-like interface:

```
for index in m.ifTable:
    print(repr(m.ifDescr[index]))
m.ifAdminStatus[3] = "down"
```

If you care about efficiency, note that the above snippet will walk the table twice: once to retrieve the index to loop over and once to retrieve the values. This could be avoided with:

```
for index, value in m.ifDescr.iteritems():
    print(repr(value))
```

Furthermore, you can pass partial index values to *iteritems()* to limit walked table rows to a specific subset:

```
for index, value in m.ipNetToMediaPhysAddress.iteritems(10):
    print(repr(value))
```

If you don't need values you can use subscript syntax for this as well:

```
for index in m.ipNetToMediaPhysAddress[10]:
    print(repr(index))
```

Another way to avoid those extra SNMP requests is to enable the caching mechanism which is disabled by default:

```
import time
m = M("localhost", cache=True)
print(m.sysUpTime)
time.sleep(1)
print(m.sysUpTime)
time.sleep(1)
print(m.sysUpTime)
time.sleep(10)
print(m.sysUpTime)
```

You can also specify the number of seconds data should be cached:

```
m = M("localhost", cache=20)
```

Also note that iterating over a table require an accessible index. Old MIB usually have accessible indexes. If this is not the case, you'll have to iterate on a column instead. For example, the first example could be written as:

```
for index in m.ifDescr:
    print(repr(m.ifDescr[index]))
```

If you want to group several write into a single request, you can do it with *with* keyword:

```
with M("localhost", "private") as m:
    m.sysName = "toto"
    m.ifAdminStatus[20] = "down"
```

It's also possible to set a custom timeout and a custom value for the number of retries. For example, to wait 2.5 seconds before timeout occurs and retry 10 times, you can use:

```
m = M("localhost", timeout=2.5, retries=10)
```

Snimpy will stop on any error with an exception. This allows you to not check the result at each step. Your script can't go awry. If this behaviour does not suit you, it is possible to suppress exceptions when querying inexistant objects. Instead of an exception, you'll get *None*:

```
m = M("localhost", none=True)
```

If for some reason, you need to specify the module you want to use to lookup a node, you can do that using the following syntax:

```
print(m['SNMPv2-MIB'].sysDescr)
print(m['IF-MIB'].ifNumber)
```

Script interpreter

Snimpy can be run as a script interpreter. There are two ways to do this. The first one is to invoke *Snimpy* and provide a script name as well as any argument you want to pass to the script:

```
$ snimpy example-script.py arg1 arg2
$ python -m snimpy example-script.py arg1 arg2
```

The second one is to use *Snimpy* as a shebang interpreter. For example, here is a simple script:

```
#!/usr/bin/env snimpy

load("IF-MIB")
m = M("localhost")
print(m.ifDescr[0])
```

The script can be invoked as any shell script.

Inside the script, you can use any valid Python code. You also get the *load()* method and the *M* class available, like for the interactive use.

Regular Python module

Snimpy can also be imported as a regular Python module:

```
from snimpy.manager import Manager as M
from snimpy.manager import load

load("IF-MIB")
m = M("localhost")
print(m.ifDescr[0])
```

2.2.2 About "major SMI errors"

If you get an exception like *RAPID-CITY contains major SMI errors (check with smilint -s -l1)*, this means that there are some grave errors in this MIB which may lead to segfaults if the MIB is used as is. Usually, this means that some identifier are unknown. Use *smilint -s -l1 YOUR-MIB* to see what the problem is and try to solve all problems reported by lines beginning by *[1]*.

For example:

```
$ smilint -s -l1 rapid_city.mib
rapid_city.mib:30: [1] failed to locate MIB module `IGMP-MIB'
rapid_city.mib:32: [1] failed to locate MIB module `DVMRP-MIB'
rapid_city.mib:34: [1] failed to locate MIB module `IGMP-MIB'
rapid_city.mib:27842: [1] unknown object identifier label `igmpInterfaceIfIndex'
rapid_city.mib:27843: [1] unknown object identifier label `igmpInterfaceQuerier'
rapid_city.mib:27876: [1] unknown object identifier label `dvmrpInterfaceIfIndex'
rapid_city.mib:27877: [1] unknown object identifier label `dvmrpInterfaceOperState'
rapid_city.mib:27894: [1] unknown object identifier label `dvmrpNeighborIfIndex'
rapid_city.mib:27895: [1] unknown object identifier label `dvmrpNeighborAddress'
rapid_city.mib:32858: [1] unknown object identifier label `igmpCacheAddress'
rapid_city.mib:32858: [1] unknown object identifier label `igmpCacheIfIndex'
```

To solve the problem here, load *IGMP-MIB* and *DVMRP-MIB* before loading *rapid_city.mib*. *IGMP-MIB* should be pretty easy to find. For *DVMRP-MIB*, try Google.

Download it and use *smistrip* to get the MIB. You can check that the problem is solved with this command:

```
$ smilint -p ../cisco/IGMP-MIB.my -p ./DVMRP-MIB -s -l1 rapid_city.mib
```

You will get a lot of errors in *IGMP-MIB* and *DVMRP-MIB* but no line with *[1]*: everything should be fine. To load *rapid_city.mib*, you need to do this:

```
load("../cisco/IGMP-MIB.my")
load("./DVMRP-MIB")
load("rapid_city.mib")
```

2.3 API reference

While *Snimpy* is targeted at being used interactively or through simple scripts, you can also use it from your Python program.

It provides a high-level interface as well as lower-level ones. However, the effort is only put in th `manager` module and other modules are considered as internal details.

2.3.1 `manager` module

2.3.2 Internal modules

Those modules shouldn't be used directly.

`mib` module

`snmp` module

This module is a low-level interface to build SNMP requests, send them and receive answers. It is built on top of pysnmp but the exposed interface is far simpler. It is also far less complete and there is an important dependency to the `basictypes` module for type coercing.

exception snimpy.snmp.**SNMPAuthorization**

exception snimpy.snmp.**SNMPBadValue**

exception snimpy.snmp.**SNMPCommitFailed**

exception snimpy.snmp.**SNMPEndOfMibView**

exception snimpy.snmp.**SNMPException**
SNMP related base exception. All SNMP exceptions are inherited from this one. The inherited exceptions are named after the name of the corresponding SNMP error.

exception snimpy.snmp.**SNMPGen**

exception snimpy.snmp.**SNMPInconsistentName**

exception snimpy.snmp.**SNMPInconsistentValue**

exception snimpy.snmp.**SNMPNoAccess**

exception snimpy.snmp.**SNMPNoCreation**

exception snimpy.snmp.**SNMPNoSuchInstance**

exception snimpy.snmp.**SNMPNoSuchName**

exception snimpy.snmp.**SNMPNoSuchObject**

exception snimpy.snmp.**SNMPNotWritable**

exception snimpy.snmp.**SNMPReadOnly**

exception snimpy.snmp.**SNMPResourceUnavailable**

exception snimpy.snmp.**SNMPTooBig**

exception snimpy.snmp.**SNMPUndoFailed**

exception snimpy.snmp.**SNMPWrongEncoding**

exception snimpy.snmp.**SNMPWrongLength**

exception snimpy.snmp.**SNMPWrongType**

exception snimpy.snmp.**SNMPWrongValue**

class snimpy.snmp.**Session**(*host*, *community='public'*, *version=2*, *secname=None*, *authprotocol=None*, *authpassword=None*, *privprotocol=None*, *privpassword=None*, *contextname=None*, *bulk=40*, *none=False*)

SNMP session. An instance of this object will represent an SNMP session. From such an instance, one can get information from the associated agent.

bulk

Get bulk settings.

Returns *False* if bulk is disabled or a non-negative integer for the number of repetitions.

get(**oids*)

Retrieve an OID value using GET.

Parameters **oids** – a list of OID to retrieve. An OID is a tuple.

Returns a list of tuples with the retrieved OID and the raw value.

retries

Get number of times a request is retried.

Returns Number of retries for each request.

set(**args*)

Set an OID value using SET. This function takes an odd number of arguments. They are working by pair. The first member is an OID and the second one is `basictypes.Type` instace whose *pack()* method will be used to transform into the appropriate form.

Returns a list of tuples with the retrieved OID and the raw value.

timeout

Get timeout value for the current session.

Returns Timeout value in microseconds.

walk(**oids*)

Walk from given OIDs but don't return any "extra" results. Only results in the subtree will be returned.

Parameters **oid** – OIDs used as a start point

Returns a list of tuples with the retrieved OID and the raw value.

walkmore(**oids*)

Retrieve OIDs values using GETBULK or GETNEXT. The method is called "walk" but this is either

a GETBULK or a GETNEXT. The later is only used for SNMPv1 or if bulk has been disabled using *bulk()* property.

Parameters **oids** – a list of OID to retrieve. An OID is a tuple.

Returns a list of tuples with the retrieved OID and the raw value.

basictypes module

2.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.4.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/vincentbernat/snimpy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Snimpy could always use more documentation, whether as part of the official Snimpy docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/vincentbernat/snimpy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.4.2 Get Started!

Ready to contribute? Here's how to set up *snimpy* for local development.

1. Fork the *snimpy* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/snimpy.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv snimpy
$ cd snimpy/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
  $ flake8 snimpy tests
       $ python setup.py test
  $ tox

To get flake8 and tox, just pip install them into your virtualenv.
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.4+. Check https://travis-ci.org/vincentbernat/snimpy/pull_requests and make sure that the tests pass for all supported Python versions.

2.4.4 Tips

To run a subset of tests:

```
$ python -m nose tests/test_snmp.py
```

2.5 License

Snimpy is licensed under the ISC license. It basically means: do whatever you want with it as long as the copyright sticks around, the conditions are not modified and the disclaimer is present.

2.5.1 Development Lead

- Vincent Bernat <bernat@luffy.cx>

2.5.2 Contributors

- Jakub Wroniecki
- Julian Taylor

2.5.3 ISC License

2.6 History

2.6.1 1.0.0 (2021-05-29)

- Drop compatibility with Python 2.

2.6.2 0.8.14 (2020-04-26)

- Add `items()` in addition to `iteritems()` This is an iterator on Python 3 and return a list of items in Python 2.

2.6.3 0.8.13 (2018-10-12)

- Compatibility with Python 3.7.
- Fix an issue with implied index when reusing indexes between tables.

2.6.4 0.8.12 (2017-10-02)

- Support for more recent versions of IPython.
- Support for SNMPv3 context name.
- Support for notification nodes (MIB only).

2.6.5 0.8.11 (2016-08-13)

- Fix IPython interactive shell.
- Fix IPv6 handling for sessions.
- Ability for a session to return None instead of raising an exception.

2.6.6 0.8.10 (2016-02-16)

- Ability to walk a table (if the first index is accessible).
- Ability to do a partial walk (courtesy of Alex Unigovsky).

2.6.7 0.8.8 (2015-11-15)

- Fix thread-safety problem introduced in 0.8.6. This also undo any improvement advertised in 0.8.6 when using multiple threads. However, performance should be kept when using a single thread.

2.6.8 0.8.7 (2015-11-14)

- Ability to specify a module name when querying a manager.
- Compatibility with PySNMP 4.3
- Array-like interface for OIDs.
- Ability to restrict lookups to a specific MIB: m['IF-MIB'].ifDescr.
- Fix multithread support with SNMPv3 (with a performance impact).

2.6.9 0.8.6 (2015-06-24)

- Major speed improvement.
- Major memory usage improvement.

2.6.10 0.8.5 (2015-04-04)

- Ability to set SMI search path (with `mib.path()`)
- Fix documentation build on *Read the Doc*.
- Add a loose mode to manager to loosen type coercion.

2.6.11 0.8.4 (2015-02-10)

- More CFFI workarounds, including cross-compilation support.
- Ability to override a node type.
- Automatic workaround for "SNMP too big" error message.

2.6.12 0.8.3 (2014-08-18)

- IPv6 support.

2.6.13 0.8.2 (2014-06-08)

- Minor bugfixes.

2.6.14 0.8.1 (2013-10-25)

- Workaround a problem with CFFI extension installation.

2.6.15 0.8.0 (2013-10-19)

- Python 3.3 support. Pypy support.
- PEP8 compliant.
- Sphinx documentation.
- Octet strings with a display hint are now treated differently than plain octet strings (unicode). Notably, they can now be set using the displayed format (for example, for MAC addresses).

2.6.16 0.7.0 (2013-09-23)

- Major rewrite.
- SNMP support is now provided through PySNMP.
- MIB parsing is still done with `libsmi` but through CFFI instead of a C module.
- More unittests. Many bugfixes.

2.6.17 0.6.4 (2013-03-21)

- GETBULK support.
- MacAddress SMI type support.

2.6.18 0.6.3 (2012-04-13)

- Support for IPython 0.12.
- Minor bugfixes.

2.6.19 0.6.2 (2012-01-19)

- Ability to return None instead of getting an exception.

2.6.20 0.6.1 (2012-01-14)

- Thread safety and efficiency.

2.6.21 0.6 (2012-01-10)

- SNMPv3 support

2.6.22 0.5.1 (2011-08-07)

- Compatibility with IPython 0.11.
- Custom timeouts and retries.

2.6.23 0.5 (2010-02-03)

- Check conformity of loaded modules.
- Many bugfixes.

2.6.24 0.4 (2009-06-06)

- Allow to cache requests.

2.6.25 0.3 (2008-11-23)

- Provide a manual page.
- Use a context manager to group SET requests.

2.6.26 0.2.1 (2008-09-28)

- First release on PyPI.

Python Module Index

S

Index