python-nagios-helpers Documentation

Release 0.2.3

Eric Lapouyade

Feb 06, 2019

Contents

python-nagios-helpers a.k.a naghelp provides many helper classes to create python nagios plugins.

CHAPTER 1

Getting started

1.1 Install

To install:

```
pip install python-nagios-helpers
```

You may have to install some linux package:

```
sudo apt-get install libffi-dev

or

sudo yum install libffi-devel
```

1.2 Quickstart

It is higly recommended to use python-textops to manipulate collected data.

Here is an exemple of a python plugin, create a file linux_fs_full_plugin.py:

```
from naghelp import *
from textops import *

class LinuxFsFull(ActivePlugin):
    """ Basic plugin to monitor full filesystems on Linux systems"""
    cmd_params = 'user,passwd'
    tcp_ports = '22'

    def collect_data(self,data):
        data.df = Ssh(self.host.ip,self.host.user,self.host.passwd).run('df -h')
```

(continues on next page)

(continued from previous page)

```
    def parse_data(self,data):
        df = data.df.skip(1)
        data.fs_critical = df.greaterequal(98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
        data.fs_warning = df.inrange(95,98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
        data.fs_ok = df.lessthan(95,key=cuts(r'(\d+)%')).cut(col='5,4').renderitems()

    def build_response(self,data):
        self.response.add_list(CRITICAL,data.fs_critical)
        self.response.add_list(WARNING,data.fs_warning)
        self.response.add_list(OK,data.fs_ok)

if __name__ == '__main__':
   LinuxFsFull().run()
```

To excute manually:

```
python linux_fs_full_plugin.py --ip=127.0.0.1 --user=naghelp --passwd=lgpl
```

On error, it may return something liek this:

```
STATUS : CRITICAL:2, WARNING:1, OK:3
================================[  STATUS  ]================================

----( CRITICAL )--------------------------------------------------------------
/ : 98%
/home : 99%

----( WARNING )---------------------------------------------------------------
/run/shm : 95%

----( OK )--------------------------------------------------------------------
/dev : 1%
/run : 1%
/run/lock : 0%


============================[ Plugin Informations ]============================
Plugin name : __main__.LinuxFsFull
Description : Basic plugin to monitor full filesystems on Linux systems
Ports used : tcp = 22, udp = none
Execution time : 0:00:00.673851
Exit code : 2 (CRITICAL), __sublevel__=0
```

Or if no error:

```
OK

============================[ Plugin Informations ]============================
Plugin name : __main__.LinuxFsFull
Description : Basic plugin to monitor full filesystems on Linux systems
Ports used : tcp = 22, udp = none
Execution time : 0:00:00.845603
Exit code : 0 (OK), __sublevel__=0
```

Naghelp will automatically manage some options:

```
$ python linux_fs_full_plugin.py -h
Usage:
linux_fsfull.py [options]

Options:
  -h, --help         show this help message and exit
  -v                 Verbose : display informational messages
  -d                 Debug : display debug messages
  -l FILE            Redirect logs into a file
  -i                 Display plugin description
  -n                 Must be used when the plugin is started by nagios
  -s                 Save collected data in a file
                     (/tmp/naghelp/<hostname>_collected_data.json)
  -r                 Use saved collected data (option -s)
  -a                 Collect data only and print them
  -b                 Collect and parse data only and print them

  Host attributes:
    To be used to force host attributes values

    --passwd=PASSWD  Passwd
    --ip=IP          Host IP address
    --user=USER      User
    --name=NAME      Hostname
```

For more information, Read The Fabulous Manual !

1.3 Run tests

Many doctests as been developped, you can run them this way:

```
cd tests
python ./runtests.py
```

1.4 Build documentation

An already compiled documentation should be available *here<http://python-nagios-helpers.readthedocs.org>*. Nevertheless, one can build the documentation.

For HTML:

```
cd docs
make html
cd _build/html
firefox ./index.html
```

For PDF, you may have to install some linux packages:

```
sudo apt-get install texlive-latex-recommended texlive-latex-extra
sudo apt-get install texlive-latex-base preview-latex-style lacheck tipa

cd docs
make latexpdf
```

(continues on next page)

(continued from previous page)

```
cd _build/latex
evince python-nagios-helpers.pdf   (evince is a PDF reader)
```

- genindex
- modindex
- search

CHAPTER 2

How to build an active plugin

An Nagios active plugin is a script that is triggered by Nagios which is waiting 2 things :

- A message on the standard output.
- An exit code giving the error level.

A passive plugin is a script that is NOT triggered by Nagios, but by external mechanism like event handlers (syslog handlers), crontabs, mails, snmp traps etc... These plugins send message and error level through a dedicated Nagios pipe.

Naghelp actually manages only **active** plugins. We plan to extend the framework to passive plugins later.

2.1 Naghelp Vs Nagios

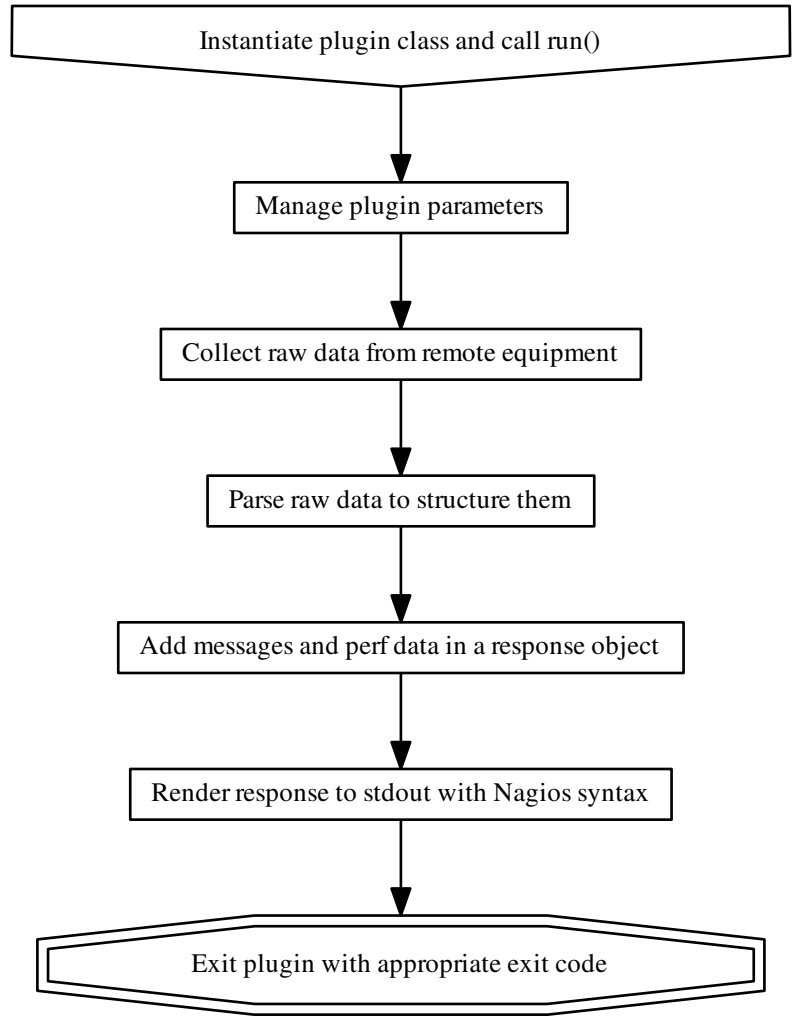
There is a little difference between a naghelp plugin and a Nagios plugin :

A naghelp plugin is a python class, a Nagios plugin is a script. To build a Nagios plugin from a naghelp plugin, you just have to instantiate a naghelp plugin class and call the `run()` method:

```
#!/usr/bin/python

from naghelp import *

class MyPlugin(ActivePlugin):
    """ My code """

if __name__ == '__main__':
    plugin = MyPlugin()
    plugin.run()
```

2.2 Plugin execution pattern

A Nagios plugin built with naghelp roughly work like this :



2.3 Plugin development

The main steps for coding a plugin with naghelp are :

- Develop a class derived from *naghelp.ActivePlugin* or derived from a project common class itself derived from *naghelp.ActivePlugin*.

  The main attributes/method to override are :

  - Attribute *cmd_params* that lists what parameters may be used on command line.

  - Attribute *required_params* tells what parameters are required
  - Attributes *tcp_ports* and *udp_ports* tell what ports to check if needed
  - Method *collect_data()* to collect raw data
  - Method *parse_data()* to parse raw data into structured data
  - Method *build_response()* to use collected and parsed data for updating response object
- Instantiate the plugin class
- run it with a `run()`

The `run()` method takes care of using attributes and calling method specified above. it also takes care of rendering the response object into Nagios string syntax, to display it onto stdout and exiting the plugin with appropriate exit code.

That's all.

2.4 A Plugin explained

In order to understand how to code a plugin, let's take the plugin from the *Getting started* and explain it line by line.

The plugin class is included into a python scripts (let's say `linux_fs_full_plugin.py`) that will be executed by Nagios directly:

```
#!/usr/bin/python
from naghelp import *
from textops import *

class LinuxFsFull(ActivePlugin):
    """ Basic plugin to monitor full filesystems on Linux systems"""
    cmd_params = 'user,passwd'
    tcp_ports = '22'

    def collect_data(self,data):
        data.df = Ssh(self.host.ip,self.host.user,self.host.passwd).run('df -h')

    def parse_data(self,data):
        df = data.df.skip(1)
        data.fs_critical = df.greaterequal(98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
        data.fs_warning = df.inrange(95,98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
        data.fs_ok = df.lessthan(95,key=cuts(r'(\d+)%')).cut(col='5,4').renderitems()

    def build_response(self,data):
        self.response.add_list(CRITICAL,data.fs_critical)
        self.response.add_list(WARNING,data.fs_warning)
        self.response.add_list(OK,data.fs_ok)

if __name__ == '__main__':
   LinuxFsFull().run()
```

Now let's explain...

2.4.1 Python interpreter

```
#!/usr/bin/python
```

The first line tells what python interpreter have to run the script. Above we supposed that naghelp has been install system-wide. But may be, you are using `virtualenv`, in such a case, you should use the correct interpreter : when activated run `which python` to see where it is, modify the first line then:

```
#!/home/myproject/myvenv/bin/python
```

If you are using buildout, replace this by a customized python interpreter, to do so, have a `/home/myproject/buildout.cfg` about like that:

```
[buildout]
...
parts = eggs tests wsgi
...
eggs =
    naghelp
    <other python packages>
    ...

[eggs]
recipe = zc.recipe.egg
eggs =
    ${buildout:eggs}
extra-paths =
    ${buildout:directory}
    ${buildout:directory}/my_project_plugins
    ...
interpreter = py2
    ...
```

generate `bin/py2` interpreter by running the command `bin/buildout`. Now, modify the plugin's first line to have

```
#!/home/myproject/bin/py2
```

2.4.2 Import modules

```
from naghelp import *
from textops import *
```

As you can see, not only we import naghelp but also python-textops : it has been developed especially for naghelp so it is highly recommended to use it. You will be able to manipulate strings and parse texts very easily.

Instead of importing these two modules, one can choose to build a `plugin_commons.py` to import and define everything you need for all your plugins, see an example in *plugin_commons*.

2.4.3 Subclass the ActivePlugin class

```
class LinuxFsFull(ActivePlugin):
```

To create your active plugin class, just subclass *naghelp.ActivePlugin*.

Nevertheless, if you have many plugin classes, it is highly recommended to subclass a class common to all your plugins which itself is derived from *naghelp.ActivePlugin* : see *plugin_commons*.

2.4.4 Specify parameters

```
cmd_params = 'user,passwd'
```

Here, by setting *cmd_params*, you are asking naghelp to accept on command line `--user` and `--passwd` options and to have a look in environment variables and in the optionnal database to see whether the informations can be found too. The values will be availabe in *collect_data()*, *parse_data()* and *build_response()* at `self.host.user` and `self.host.passwd`. By default, because of attribute *forced_params*, `ip` and `name` options are also available in the same way, you do not need to specify them.

Note that naghelp automatically sets many other options in command line, use `-h` to see the help:

```
./linux_fs_full_plugin.py -h
Usage:
linux_fs_full_plugin.py [options]

Options:
  -h, --help            show this help message and exit
  -v                    Verbose : display informational messages
  -d                    Debug : display debug messages
  -l FILE               Redirect logs into a file
  -i                    Display plugin description
  -n                    Must be used when the plugin is started by nagios
  -s                    Save collected data in a file
                        (/tmp/naghelp/<hostname>_collected_data.json)
  -r                    Use saved collected data (option -s)
  -a                    Collect data only and print them
  -b                    Collect and parse data only and print them

  Host attributes:
    To be used to force host attributes values

    --passwd=PASSWD     Password
    --ip=IP             Host IP address
    --user=USER         User
    --name=NAME         Hostname
    --subtype=SUBTYPE   Plugin subtype (usually host model)

  Specific to my project:
    -c FILE             override default path to the db.json file
```

2.4.5 Specify tcp/udp ports

```
tcp_ports = '22'
```

You can specify *tcp_ports* and/or *udp_ports* your plugin is using : by this way, the administrator will be warned what port has to be opened on his firewall. Port informations will be displayed at the end of each message in plugin informations section.

For tcp_ports, an additional check will be done if an error occurs while collecting data.

2.4.6 Redefine `collect_data()`

```
def collect_data(self,data):
    data.df = Ssh(self.host.ip,self.host.user,self.host.passwd).run('df -h')
```

collect_data() main purpose is to collect **raw data** from the remote equipment. Here are some precisions :

- You have to collect **all** data in *collect_data()*, this means it is not recommended to collect some other data in *parse_data()* nor *build_response()*.
- You have to collect **only raw** data in *collect_data()*, this means if raw data cannot be used at once and needs some kind of extraction, you have to do that after into *parse_data()* or if there is very little processing to do, into *build_response()*.
- The data collect must be optimized to make as few requests as possible to remote equipment. To do so, use `mget()`, `mrun()`, or `mwalk()` methods or `with:` blocks to keep connection opened while collecting (see module *naghelp.collect*).
- The collected data must be set onto `data` object. It is a `DictExt` dictionary that accepts dotted notation to write information (only one level at a time).
- The collected data can be saved with `-s` comand-line option and restored with `-r`.

For the example above, it is asked to run the unix command `df -h` through *naghelp.Ssh.run()* on the remote equipment at `host = self.host.ip` with `user = self.host.user` and `password = self.host.passwd`.

You can run your plugin with option `-a` to stop it just after data collect and display all harvested informations (ie `data DictExt`):

```
# ./linux_fs_full_plugin.py --ip=127.0.0.1 --user=naghelp --passwd=naghelppw -a
Collected Data =
{   'df': 'Filesystem      Size  Used Avail Use% Mounted on
/dev/sdb6            20G     19G  1,0G  95% /
udev                7,9G    4,0K  7,9G   1% /dev
tmpfs               1,6G    1,1M  1,6G   1% /run
none                5,0M       0  5,0M   0% /run/lock
none                7,9G     77M  7,8G   1% /run/shm
/dev/sda5            92G     91G  0,9G  99% /home
'}
```

2.4.7 Redefine `parse_data()`

```
def parse_data(self,data):
    df = data.df.skip(1)
    data.fs_critical = df.greaterequal(98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
    data.fs_warning = df.inrange(95,98,key=cuts(r'(\d+)%')).cut(col='5,4').
↪renderitems()
    data.fs_ok = df.lessthan(95,key=cuts(r'(\d+)%')).cut(col='5,4').renderitems()
```

parse_data() main purpose is to structure or extract informations from collected raw data. To do so, it is highly recommended to use python-textops. The `data` object is the same as the one from *collect_data()* that is a `DictExt`, so you can access collected raw data with a dotted notation. textops methods are also directly available from anywhere in the `data` object.

`data.df` should be equal to:

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdb6            20G     19G  1,0G  95% /
udev                7,9G    4,0K  7,9G   1% /dev
tmpfs               1,6G    1,1M  1,6G   1% /run
none                5,0M       0  5,0M   0% /run/lock
none                7,9G     77M  7,8G   1% /run/shm
/dev/sda5            92G     91G  0,9G  99% /home
```

In `df = data.df.skip(1)` `skip` will skip the first line of the collect raw data and store in `df` local variable, this should be equal to:

```
/dev/sdb6            20G     19G  1,0G  95% /
udev                7,9G    4,0K  7,9G   1% /dev
tmpfs               1,6G    1,1M  1,6G   1% /run
none                5,0M       0  5,0M   0% /run/lock
none                7,9G     77M  7,8G   1% /run/shm
/dev/sda5            92G     91G  0,9G  99% /home
```

In `df.greaterequal(98,key=cuts(r'(\d+)%')),greaterequal` will select all lines where the column with a percent has a value greater or equal to 98, in our case, there is only one:

```
['/dev/sda5           92G     81G  6,9G  93% /home']
```

In `df.greaterequal(98,key=cuts(r'(\d+)%')).cut(col='5,4')` `cut` will select only column 5 and 4 in that order:

```
[['/home', '99%']]
```

In `df.greaterequal(98,key=cuts(r'(\d+)%')).cut(col='5,4').renderitems()` `renderitems` will format/render above this way:

```
['/home : 99%']
```

This will be stored in `data.fs_critical`. This is about the same for `data.fs_warning` and `data.fs_ok`.

Finally, if you want to see what has been parsed, use `-b` command line option:

```
# ./linux_fs_full_plugin.py --ip=127.0.0.1 --user=naghelp --passwd=naghelppw -b
...
Parsed Data =
{   'fs_critical': [u'/home : 99%'],
    'fs_ok': [u'/dev : 1%', u'/run : 1%', u'/run/lock : 0%', u'/run/shm : 2%'],
    'fs_warning': [u'/ : 95%']}
```

2.4.8 Redefine `build_response()`

```
def build_response(self,data):
    self.response.add_list(CRITICAL,data.fs_critical)
    self.response.add_list(WARNING,data.fs_warning)
    self.response.add_list(OK,data.fs_ok)
```

In *`build_response()`*, `data` will contain all collected data AND all parsed data in the same `DictExt`. Now, you just have to use *`PluginResponse`* methods to modify `self.response`.

In the script, we are adding lists of messages with *`add_list()`* : `self.response.add_list(OK,data.fs_ok)` is the same as writing:

```
for msg in data.fs_ok:
   if msg:
      self.response.add(OK, msg)
```

Which is equivalent to:

```
self.response.add(OK,'/dev : 1%')
self.response.add(OK,'/run : 1%')
self.response.add(OK,'/run/lock : 0%')
self.response.add(OK,'/run/shm : 2%')
```

To see what naghelp will finally send to Nagios, just execute the script

```
STATUS : CRITICAL:1, WARNING:1, OK:4
================================[  STATUS  ]================================

----( CRITICAL )------------------------------------------------------------
/home : 99%

----( WARNING )-------------------------------------------------------------
/ : 95%

----( OK )-----------------------------------------------------------------
/dev : 1%
/run : 1%
/run/lock : 0%
/run/shm : 2%


============================[ Plugin Informations ]============================
Plugin name : __main__.LinuxFsFull
Description : Basic plugin to monitor full filesystems on Linux systems
Ports used : tcp = 22, udp = none
Execution time : 0:00:00.003662
Exit code : 2 (CRITICAL), __sublevel__=0
```

As you can see :

- naghelp generated automatically a synopsis : `STATUS : CRITICAL:1, WARNING:1, OK:4`
- naghelp created the `==[ STATUS ]==` section where messages has been splitted into several sub-sections corresponding to their level.
- naghelp automatically add a `==[ Plugin Informations ]==` section with many useful informations including ports to be opened on the firewall.
- naghelp automatically used the appropriate exit code, here 2 (CRITICAL)

2.4.9 Configure Nagios

Once your plugin is developed, you have to declare a Nagios command using it:

```
define command{
    command_name    myplugin_cmd
    command_line    /path/to/linux_fs_full_plugin.py --name="$HOSTNAME$" --ip="
↪$HOSTADDRESS" --user="$ARG1$" --passwd="$ARG2"
    }
```

Then, you can define a host and a service using that Nagios command:

```
define host{
    use                     generic-host
    host_name               myequipment
    address                 1.2.3.4
}

define service{
    use                     generic-service
    host_name               myequipment
    service_description "myplugin service"
    check_command       myplugin_cmd!naghelpuser!naghelppassword
    }
```

2.5 Advanced plugin

2.5.1 Manage errors

You can abort the plugin execution when an error is encountered at monitoring level, or when it is not relevant to monitor an equipment in some conditions.

For exemple, let's say you have collected data about an equipment controller, but it is not the active one, that means data may be incomplete or not relevant : you should use the *fast_response()* or *fast_response_if()* method:

```
def build_response(self,data):
    self.fast_response_if(data.hpoa.oa.grep('^OA Role').grepc('Standby'), OK,
↪'Onboard Administration in Standby mode')
    ...
```

Above, if the monitored controller is in standby mode, we get out the plugin at once without any error (`OK` response level).

2.5.2 Create mixins

To re-use some code, you can create a plugin mixin.

Let's create a mixin that manages gauges : When a metric (for example the number of fans) is seen the first time, the metric is memorized as the reference value (`etalon`). Next times, it is compared : if the metric goes below, it means that one part has been lost and an error should be raise.

Here is the mixin:

```
class GaugeMixin(object):
    def get_gauges(self,data):
        # To be defined in child class
        # must return a tuple of tuples like this:
        # return ( ('gaugeslug','the gauge full name', <calculated gauge value as int>
↪, <responselevel>),
        #          ('gaugeslug2','the gauge2 full name', <calculated gauge2 value as␣
↪int>, <responselevel2>) )
        pass
```

(continues on next page)

(continued from previous page)

```
    def gauge_response(self,data):
        for gname,fullname,gvalue,level in self.get_gauges(data):
            self.response.add_more('%s : %s',fullname,gvalue)
            etalon_name = gname + '_etalon'
            etalon_value = self.host.get(etalon_name,None)
            if etalon_value is not None and gvalue < etalon_value:
                self.response.add(level,'%s : actual count (%s) not equal to the␣
↪reference value (%s)' % (fullname, gvalue, etalon_value))
            if gvalue:
                # save the gauge value as the new reference value in host's␣
↪persistent data
                self.host.set(etalon_name,gvalue)
            else:
                self.response.add(UNKNOWN,'%s : unknown count.' % gname.upper())

    def build_response(self,data):
        self.gauge_response(data)
        # this will call plugin build_response() or another mixin build_response() if␣
↪present.
        super(GaugeMixin,self).build_response(data)
```

Then use the mixin in your plugin by using multiple inheritage mechanism (mixin first):

```
class SunRsc(GaugeMixin,ActivePlugin):
   ...
   def get_gauges(self, data):
     return ( ('fan',       'Fans',        data.showenv.grep(r'^FAN TRAY|_FAN').grepc(
↪'OK'), WARNING ),
              ('localdisk','Local disks',data.showenv.grep(r'\bDISK\b').grepc(
↪'OK|PRESENT'), WARNING ),
              ('power',     'Powers',      data.showenv.after(r'Power Supplies:').
↪grepv(r'^---|Supply|^\s*$').grepc('OK'), WARNING ) )
```

By this way, you can re-use very easily gauge feature in many plugins. Of course, you can use several plugin mixins at a time, just remember to put the *ActivePlugin* class (or some derived from) at the end of the parent class list.

2.5.3 Use a database

It is possible to have all equipments parameters set in a common database for your project. One can configure naghelp to get parameters in that database when not found in environment variables nor as command line options. The only information that is mandatory is the equipment name : naghelp will use it as an index to retreive the information in the database. As Nagios sets `NAGIOS_HOSTNAME` environment variable, there even no need to give that parameter as command argument in nagios configuration files, naghelp will take it for you. The only place where you have to specify `--name` is while manually testing the plugin on console : the environment variable is not set in this case.

With a database, Nagios command declaration will be:

```
define command{
    command_name    myplugin_cmd
    command_line    /path/to/linux_fs_full_plugin.py
    }
```

And the service definition:

```
define service{
    use                 generic-service
    host_name           myequipment
    service_description "myplugin service"
    check_command       myplugin_cmd
    }
```

To have naghelp using a database, you have to subclass *naghelp.Host* class and redefine *_get_params_from_db()* method (see `MonitoredHost` example in that method)

Then you have to subclass *ActivePlugin* and specify that you want naghelp to use `MonitoredHost` instead of *naghelp.Host*:

```
class MyProjectActivePlugin(ActivePlugin):
   ...
   host_class = MonitoredHost
   ...
```

Then all your plugins have to derive from this class:

```
class MyPlugin(MyProjectActivePlugin):
   """ My code """
```

2.6 Create a launcher

If you have a lot of plugins, you should consider to code only naghelp classes. By this way, you will be able to define more than one plugin per python file and you will discover the joy of subclassing your own plugin classes to build some others much more faster. You will be also able to use python mixins to compact your code.

To do so, you will need a launcher that will load the right python module, instantiate the right naghelp plugin class and run it. Lets call the launcer script `pypa`, the Nagios commands.cfg will be something like this:

```
define command{
    command_name    myplugin
    command_line    /path/to/pypa my_project_plugins.myplugin.MyPlugin --name="$ARG1$
↪" --user="$ARG2$" --passwd="$ARG3"
    }
```

You just have to write a launcher once, naghelp provide a module for that, here is the `pypa` script:

```
#!/usr/bin/python
# change python interpreter if your are using virtualenv or buildout

from plugin_commons import MyProjectActivePlugin
from naghelp.launcher import launch

def main():
    launch(MyProjectActivePlugin)

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

The `launch` function will read command line first argument and instantiate the specified class with a dotted notation. It will also accept only the class name without any dot, in this case, a recursive search will be done from the directory given by `MyProjectActivePlugin.plugins_basedir` and will find the class with the right name and having

the same `plugin_type` attribute value as `MyProjectActivePlugin`. the search is case insensitive on the class name. `MyProjectActivePlugin` is the common class to all your plugins and is derived from *naghelp.ActivePlugin*.

If you start `pypa` without any parameters, it will show you all plugin classes it has discovered with their first line description:

```
$ ./pypa
Usage : bin/pypa <plugin name or path.to.module.PluginClass> [options]

Available plugins :
================================================================================================================
Name                          File                          Description
----------------------------------------------------------------------------------------
↪-------------------------
AixErrpt                      ibm_aix.py                    IBM plugin using errpt␣
↪command on all AIX systems
BrocadeSwitch                 brocade.py                    Brocade Switch Active␣
↪plugin
HpBladeC7000                  hp_blade_c7000.py             HP bladecenter C7000␣
↪plugin
HpEva                         hp_eva.py                     HP Enterprise Virtual␣
↪Array (EVA) SAN Storage Plugin
HpHpuxSyslog                  hp_hpux.py                    HPUX syslog analyzing␣
↪active plugin
HpProliant                    hp_proliant.py                HP Proliant Active␣
↪plugin
SunAlom                       sun_ctrl.py                   Sun microsystems/Oracle␣
↪plugin for hardware with ALOM controller
SunFormatFma                  sun_fma.py                    Sun microsystems/Oracle␣
↪plugin using format and fmadm commands on solaris system
SunIlom                       sun_ctrl.py                   Sun microsystems/Oracle␣
↪plugin for hardware with ILOM controller
SunRsc                        sun_ctrl.py                   Sun microsystems/Oracle␣
↪plugin for hardware with RSC controller
VIOErrlog                     ibm_aix.py                    IBM plugin using errlog␣
↪command on all VIO systems
VmwareEsxi                    vmware_esxi.py                VMWare ESXi active␣
↪plugin
----------------------------------------------------------------------------------------
↪-------------------------
```

2.7 plugin_commons

All your plugins should (**must** when using a launcher) derive from a common plugin class which itself is derived from *naghelp.ActivePlugin*. You will specify the plugins base directory, and other common attributes. All this should be placed in a file `plugin_commons.py` where you can also import and define many other things that can be common to all your plugins:

```
from naghelp import *
from textops import *

PLUGINS_DIR = '/path/to/my_project_plugins'

class MyProjectActivePlugin(ActivePlugin):
```

(continues on next page)

(continued from previous page)

```
   abstract = True                              # to avoid launcher to find this class
   plugins_basemodule = 'my_project_plugins.' # prefix to add to have module␣
↪accessible from python path
   plugins_basedir = PLUGINS_DIR
   plugin_type = 'myproject_plugin'           # you choose whatever you want but not
↪'plugin'
   host_class = MonitoredHost                 # only if you have a database managed␣
↪by a derived Host class
```

Then, a typical code for your plugins would be like this, here `/path/to/my_project_plugins/myplugin.py`:

```
from plugin_commons import *

class MyPlugin(MyProjectActivePlugin):
   """ My code """
```

2.8 Debug

To debug a plugin, the best way is to activate the debug mode with `-d` flag:

```
$ python ./linux_fs_full_plugin.py --ip=127.0.0.1 --user=naghelp --passwd=naghelppw -d
2016-01-12 10:12:29,912 - naghelp - DEBUG - Loading data from /home/elapouya/projects/
↪sebox/src/naghelp/tests/hosts/127.0.0.1/plugin_persistent_data.json :
2016-01-12 10:12:29,913 - naghelp - DEBUG - {   u'ip': u'127.0.0.1',
    u'name': u'127.0.0.1',
    u'passwd': u'naghelppw2',
    u'user': u'naghelp'}
2016-01-12 10:12:29,913 - naghelp - INFO - Start plugin __main__.LinuxFsFull for 127.
↪0.0.1
2016-01-12 10:12:29,913 - naghelp - DEBUG - Host informations :
2016-01-12 10:12:29,914 - naghelp - DEBUG - _params_from_db = {   u'ip': u'127.0.0.1',
    u'name': u'127.0.0.1',
    u'passwd': u'naghelppw2',
    u'user': u'naghelp'}
2016-01-12 10:12:29,914 - naghelp - DEBUG - _params_from_env = {   }
2016-01-12 10:12:29,914 - naghelp - DEBUG - _params_from_cmd_options = {   'ip': '127.
↪0.0.1',
    'name': None,
    'passwd': 'naghelppw',
    'subtype': None,
    'user': 'naghelp'}
2016-01-12 10:12:29,914 - naghelp - DEBUG -
------------------------------------------------------------
ip           : 127.0.0.1
name         : 127.0.0.1
passwd       : naghelppw
user         : naghelp
------------------------------------------------------------
2016-01-12 10:12:30,101 - naghelp - DEBUG - #### Ssh( naghelp@127.0.0.1 ) ############
↪###
2016-01-12 10:12:30,255 - naghelp - DEBUG - is_connected = True
2016-01-12 10:12:30,255 - naghelp - DEBUG -   ==> df -h
2016-01-12 10:12:30,775 - naghelp - DEBUG - #### Ssh : Connection closed #############
↪##
```

(continues on next page)

(continued from previous page)

```
2016-01-12 10:12:30,775 - naghelp - INFO - Data are collected
2016-01-12 10:12:30,776 - naghelp - DEBUG - Collected Data =
{   'df': 'Sys. de fichiers Taille Utilis\xc3\xa9 Dispo Uti% Mont\xc3\xa9 sur
/dev/sdb6           20G     12G  6,5G  65% /
udev               7,9G    4,0K  7,9G   1% /dev
tmpfs              1,6G    1,1M  1,6G   1% /run
none               5,0M       0  5,0M   0% /run/lock
none               7,9G    119M  7,8G   2% /run/shm
/dev/sda5           92G     81G  6,9G  93% /home
'}
2016-01-12 10:12:30,777 - naghelp - INFO - Data are parsed
2016-01-12 10:12:30,777 - naghelp - DEBUG - Parsed Data =
{   'fs_critical': [],
    'fs_ok': [   '/ : 65%',
                 '/dev : 1%',
                 '/run : 1%',
                 '/run/lock : 0%',
                 '/run/shm : 2%',
                 '/home : 93%'],
    'fs_warning': []}
2016-01-12 10:12:30,778 - naghelp - DEBUG - Saving data to /home/elapouya/projects/
↪sebox/src/naghelp/tests/hosts/127.0.0.1/plugin_persistent_data.json :
ip           : 127.0.0.1
name         : 127.0.0.1
passwd       : naghelppw
user         : naghelp
2016-01-12 10:12:30,778 - naghelp - INFO - Plugin output summary : None
2016-01-12 10:12:30,778 - naghelp - DEBUG - Plugin output :
################################################################################
OK
=================================[  STATUS  ]=================================

----( OK )-------------------------------------------------------------------
/ : 65%
/dev : 1%
/run : 1%
/run/lock : 0%
/run/shm : 2%
/home : 93%


============================[ Plugin Informations ]============================
Plugin name : __main__.LinuxFsFull
Description : Basic plugin to monitor full filesystems on Linux systems
Ports used : tcp = 22, udp = none
Execution time : 0:00:00.866135
Exit code : 0 (OK), __sublevel__=0
```

2.8.1 Debug params

Before looking your code, check the plugin is receiving the right parameters : They are taken from command line options in priority THEN from envrionment variables THEN from database and persistent data.

In debug traces, what is set to `self.host` is written here:

```
2016-01-12 10:12:29,914 - naghelp - DEBUG -
------------------------------------------------------------
ip            : 127.0.0.1
name          : 127.0.0.1
passwd        : naghelppw
user          : naghelp
------------------------------------------------------------
```

2.8.2 Debug `collect_data`

The main pitfall is to succeed to connect to the remote host.

Naghelp provides possibilities to customize patterns for login steps or to find the prompt. This could be done on *Telnet*, *Expect* or even *Ssh*.

The main symptom when a pattern is wrong, this usually hangs the collect, and the plugin should then raise an *TimeoutError* exception.

To debug patterns, please do some testing by collecting data manually yourself and test your patterns on output. Please check patterns syntax in `re` module.

Use `-a` plugin option to check what has been collected

Note: The prompt pattern is used to recognize when a command output has finished and when naghelp can send another one.

2.8.3 Debug `parse_data`

The main pitfall is to correctly parse or extract data from collected raw data. Use `-b` plugin option to check what has been collected and what has be parsed. Use options `-s` and `-r` to avoid wasting time to collect data.

Check textops manual to see how to do parse texts. There is no particular tip but to have experience with regular expressions.

2.8.4 Debug `build_response`

The goal is to check that all equipment errors/infos are correctly detected. There is a little trick to check everything : The first time, use your plugin with `-s` option to save collected data : To test all cases, you can simulate errors by modifying the file for collected data. Then, use `-r` to restore modified data and see how your plugin is working.

The file for collected data is located at the path shown in the plugin usage (`-h` option), by default it is `/tmp/naghelp/<hostname>_collected_data.json`

- genindex
- modindex
- search

CHAPTER 3

Collect

This module provides many funcions and classes to collect data remotely and locally

3.1 Expect

class naghelp.**Expect**(*spawn*, *login_steps=None*, *prompt=None*, *logout_cmd=None*, *logout_steps=None*, *context={}*, *timeout=30*, *expected_pattern='\S'*, *unexpected_pattern='<timeout>'*, *filter=None*, **args*, ***kwargs*)

Interact with a spawn command

Expect is a class that "talks" to other interactive programs. It is based on pexpect and is focused on running one or many commands. *Expect* is to be used when *Telnet* and *Ssh* are not applicable.

Parameters

- **spawn** (*str*) – The command to start and to communicate with
- **login_steps** (*list or tuple*) – steps to execute to reach a prompt
- **prompt** (*str*) – A pattern that matches the prompt
- **logout_cmd** (*str*) – command to execute before closing communication (Default : None)
- **logout_steps** (*list or tuple*) – steps to execute before closing communication (Default : None)
- **context** (*dict*) – Dictionary that will be used in steps to .format() strings (Default : None)
- **timeout** (*int*) – Maximum execution time (Default : 30)
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found in methods that collect data (like run,mrun,get,mget,walk,mwalk...) if None, there is no test. By default, tests the result is not empty.

- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, it tests <timeout>.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.

On object creation, *Expect* will :

- spawn the specified command (`spawn`)
- follow `login_steps`
- and wait for the specified `prompt`.

on object *run()* or *mrun()*, it will :

- execute the specified command(s)
- wait the specified prompt between each command
- return command(s) output wihout prompt string
- then close the interaction (see just below)

on close, *Expect* will :

- execute the `logout_cmd`
- follow `logout_steps`
- finally terminate the spawned command

What are Steps ?

During login and logout, one can specify steps. A step is one or more pattern/answer tuples. The main tuple syntax is:

```
(
    (
        ( step1_pattern1, step1_answer1),
        ( step1_pattern2, step1_answer2),
        ...
    ),
    (
        ( step2_pattern1, step2_answer1),
        ( step2_pattern2, step2_answer2),
        ...
    ),
    ...
)
```

For a same step, *Expect* will search for any of the specified patterns, then will respond the corresponding answer. It will go to the next step only when one of the next step's patterns is found. If not, will stay at the same step looking again for any of the patterns. In order to simplify tuple expression, if there is only one tuple in a level, the parenthesis can be removed. For answer, you have to specify a string : do not forget the newline otherwise you will get stuck. One can also use `Expect.BREAK` to stop following the steps, `Expect.RESPONSE` to raise an error with the found pattern as message. `Expect.KILL` does the same but also kills the spawned command.

Here are some `login_steps` examples :

The spawned command is just waiting a password:

```
(r'(?i)Password[^:]*: ','wwwpw\n')
```

The spawned command is waiting a login then a password:

```
(
    (r'(?i)Login[^:]*: ','www\n'),
    (r'(?i)Password[^:]*: ','wwwpassword\n')
)
```

The spawned command is waiting a login then a password, but may ask a question at login prompt:

```
(
    (
        (r'(?i)Are you sure to connect \?','yes'),
        (r'(?i)Login[^:]*: ','www\n')
    ),
    ('(?i)Password[^:]*: ','wwwpassword\n')
)
```

You can specify a context dictionary to *Expect* to format answers strings. With `context = { 'user':'www','passwd':'wwwpassword' }` login_steps becomes:

```
(
    (
        (r'(?i)Are you sure to connect \?','yes'),
        (r'(?i)Login[^:]*: ','{user}\n')
    ),
    ('(?i)Password[^:]*: ','{passwd}\n')
)
```

mrun(*cmds*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)

Execute many commands at the same time

Runs a dictionary of commands at the specified prompt and then close the interaction. Timeout will not raise any error but will return None for the running command. It returns a dictionary where keys are the same as the `cmds` dict and the values are the commmands output.

Parameters

- **cmds** (*dict or list of items*) – The commands to be executed by the spawned command
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the interaction.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The commands output

Return type textops.DictExt

Example

SSH with multiple commands:

```
e = Expect('ssh www@localhost',
            login_steps=('(?i)Password[^:]*: ','wwwpassword\n'),
            prompt=r'www@[^\$]*\$ ',
            logout_cmd='exit')
print e.mrun({'cur_dir':'pwd','big_files':'find . -type f -size +10000'})
```

Will return something like:

```
{
    'cur_dir' : '/home/www',
    'big_files' : 'bigfile1\nbigfile2\nbigfile3\n...'
}
```

Note: This example emulate *Ssh* class. *Expect* is better for non-standard commands that requires human interations.

run(*cmd=None*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)
Execute one command

Runs a single command at the specified prompt and then close the interaction. Timeout will not raise any error but will return None. If you want to execute many commands without closing the interation, use `with` syntax.

Parameters

- **cmd** (*str*) – The command to be executed by the spawned command
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the interaction.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The command output or None on timeout

Return type textops.StrExt

Examples

Doing a ssh through Expect:

```
e = Expect('ssh www@localhost',
           login_steps=('(?i)Password[^:]*: ','wwwpassword\n'),
           prompt=r'www@[^\$]*\$ ',
           logout_cmd='exit')
print e.run('ls -la')
```

Expect/ssh with multiple commands:

```
with Expect('ssh www@localhost',
           login_steps=('(?i)Password[^:]*: ','wwwpassword\n'),
           prompt=r'www@[^\$]*\$ ',
           logout_cmd='exit') as e:
    cur_dir = e.run('pwd').strip()
    big_files_full_path = e.run('find %s -type f -size +10000' % cur_dir)
print big_files_full_path
```

Note: These examples emulate *Ssh* class. *Expect* is better for non-standard commands that requires human interations.

3.2 Http

class naghelp.**Http**(*expected_pattern='\S'*, *unexpected_pattern='<timeout>'*, *filter=None*, **args*, ***kwargs*)

Http class helper

This class helps to collect web pages.

Parameters

- **host** (*str*) – IP address or hostname to connect to
- **port** (*int*) – port number to use (Default : 80 TCP)
- **timeout** (*int*) – Time in seconds before raising an error or a None value

get(*url*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, **args*, ***kwargs*)

get one URL

Parameters

- **url** (*str*) – The url to get
- **timeout** (*int*) – Time in seconds before raising an error or a None value

Returns The page or NoAttr if URL is reachable but returned a Http Error

Return type str

mget(*urls*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, **args*, ***kwargs*)

Get multiple URLs at the same time

Parameters

- **urls** (*dict*) – The urls to get

- **timeout** (*int*) – Time in seconds before raising an error or a None value

Returns List of pages or NoAttr if not availables

Return type textops.DictExt

mpost(*urls*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, **args*, ***kwargs*)
Post multiple URLs at the same time

Parameters

- **urls** (*dict*) – The urls to get
- **timeout** (*int*) – Time in seconds before raising an error or a None value

Returns List of pages or NoAttr if not availables

Return type textops.DictExt

post(*url*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, **args*, ***kwargs*)
post one URL

Parameters

- **url** (*str*) – The url to get
- **timeout** (*int*) – Time in seconds before raising an error or a None value

Returns The page or NoAttr if URL is reachable but returned a Http Error

Return type str

3.3 Snmp

class naghelp.**Snmp**(*host*, *community='public'*, *version=None*, *timeout=30*, *port=161*, *user=None*, *auth_passwd=None*, *auth_protocol=''*, *priv_passwd=None*, *priv_protocol=''*, *object_identity_to_string=True*, **args*, ***kwargs*)
Snmp class helper

This class helps to collect OIDs from a remote snmpd server. One can issue some snmpget and/or snmpwalk. Protocols 1, 2c and 3 are managed. It uses pysnmp library.

Parameters

- **host** (*str*) – IP address or hostname to connect to
- **community** (*str*) – community to use (For protocol 1 and 2c)
- **version** (*int*) – protocol to use : None, 1,2,'2c' or 3 (Default: None). If None, it will use protocol 3 if a user is specified, 2c otherwise.
- **timeout** (*int*) – Time in seconds before raising an error or a None value
- **port** (*int*) – port number to use (Default : 161 UDP)
- **user** (*str*) – protocol V3 authentication user
- **auth_passwd** (*str*) – snmp v3 authentication password
- **auth_protocol** (*str*) – snmp v3 auth protocol ('md5' or 'sha')
- **priv_passwd** (*str*) – snmp v3 privacy password
- **priv_protocol** (*str*) – snmp v3 privacy protocol ('des' or 'aes')

exists(*oid_or_mibvar*)
Return True if the OID exists

It return False if the OID does not exists or raise an exception if snmp server is unreachable

Parameters **oid_or_mibvar** (*str or ObjectIdentity*) – an OID path or a pysnmp ObjectIdentity

Returns True if OID exists

Return type bool

Examples

To collect a numerical OID:

```
>>> snmp = Snmp('demo.snmplabs.com')
>>> snmp.exists('1.3.6.1.2.1.1.1.0')
True
>>> snmp.exists('1.3.6.1.2.1.1.1.999')
False
```

get(*oid_or_mibvar*)
get one OID

Parameters **oid_or_mibvar** (*str or ObjectIdentity*) – an OID path or a pysnmp ObjectIdentity

Returns OID value. The python type depends on OID MIB type.

Return type textops.StrExt or int

Examples

To collect a numerical OID:

```
>>> snmp = Snmp('demo.snmplabs.com')
>>> snmp.get('1.3.6.1.2.1.1.1.0')
'SunOS zeus.snmplabs.com 4.1.3_U1 1 sun4m'
```

To collect an OID with label form:

```
>>> snmp = Snmp('demo.snmplabs.com')
>>> snmp.get('iso.org.dod.internet.mgmt.mib-2.system.sysDescr.0')
'SunOS zeus.snmplabs.com 4.1.3_U1 1 sun4m'
```

To collect an OID with MIB symbol form:

```
>>> snmp = Snmp('demo.snmplabs.com')
>>> snmp.get(('SNMPv2-MIB', 'sysDescr', 0))
'SunOS zeus.snmplabs.com 4.1.3_U1 1 sun4m'
```

mget(*vars_oids*)
Get multiple OIDs at the same time

This method is much more faster than doing multiple *get()* because it uses the same network request. In addition, one can request a range of OID. To build a range, just use a dash between to integers :

this OID will be expanded with all integers in between : For instance, '1.3.6.1.2.1.1.2-4.1' means : [1.3.6.1.2.1.1.2.1, 1.3.6.1.2.1.1.3.1, 1.3.6.1.2.1.1.4.1]

Parameters **vars_oids** (*dict*) – keyname/OID dictionary

Returns

List of tuples (OID,value). Values type are int or textops.StrExt

Return type textops.DictExt

Example

```
>>> snmp = Snmp('demo.snmplabs.com')
>>> print snmp.mget({'uname':'1.3.6.1.2.1.1.0','other':'1.3.6.1.2.1.1.2-9.0'})
{'uname' : 'SunOS zeus.snmplabs.com 4.1.3_U1 1 sun4m',
 'other' : ['value for 1.3.6.1.2.1.1.2.0', 'value for 1.3.6.1.2.1.1.3.0', etc.
↪.. ] }
```

mwalk(*vars_oids*, *ignore_errors=False*)

Walk from multiple OID root pathes

Parameters **vars_oids** (*dict*) – keyname/OID root path dictionary

Returns

A dictionary of list of tuples (OID,value). Values type are int or textops.StrExt

Return type textops.DictExt

Example

```
>>> snmp = Snmp('localhost')
>>> print snmp.mwalk({'node1' : '1.3.6.1.2.1.1.9.1.2', 'node2' : '1.3.6.1.2.1.
↪1.9.1.3'})
{'node1': [('1.3.6.1.2.1.1.9.1.2.1', ObjectIdentity(ObjectIdentifier('1.3.6.1.
↪6.3.10.3.1.1'))),
           ('1.3.6.1.2.1.1.9.1.2.2', ObjectIdentity(ObjectIdentifier('1.3.6.1.
↪6.3.11.3.1.1')))
           ... ],
 'node2': [('1.3.6.1.2.1.1.9.1.3.1', 'The SNMP Management Architecture MIB.'),
           ('1.3.6.1.2.1.1.9.1.3.2', 'The MIB for Message Processing and␣
↪Dispatching.'),
           ... ]}
```

normalize_oid(*oid*)

Normalize OID object in order to be used with pysnmp methods

Basically, it converts OID with a tuple form into a ObjectIdentity form, keeping other forms unchanged.

Parameters **oid** (*str,tuple or ObjectIdentity*) – The OID to normalize

Returns OID form that is ready to be used with pysnmp

Return type str or ObjectIdentity

Examples

```
>>> s=Snmp('demo.snmplabs.com')
>>> s.normalize_oid(('SNMPv2-MIB', 'sysDescr2', 0))
ObjectIdentity('SNMPv2-MIB', 'sysDescr2', 0)
>>> s.normalize_oid('1.3.6.1.2.1.1.1.0')
'1.3.6.1.2.1.1.1.0'
```

walk(*oid_or_mibvar*, *ignore_errors=False*)

Walk from a OID root path

Parameters **oid_or_mibvar** (*str or ObjectIdentity*) – an OID path or a pysnmp ObjectIdentity

Returns

List of tuples (OID,value). Values type are int or `textops.StrExt`

Return type `textops.ListExt`

Example

```
>>> snmp = Snmp('localhost')
>>> for oid,val in snmp.walk('1.3.6.1.2.1.1'):
...     print oid,'-->',val
...
1.3.6.1.2.1.1.1.0 --> SunOS zeus.snmplabs.com 4.1.3_U1 1 sun4m
1.3.6.1.2.1.1.2.0 --> 1.3.6.1.4.1.20408
     ...
```

3.4 Ssh

class naghelp.**Ssh**(*host*, *user*, *password=None*, *timeout=30*, *auto_accept_new_host=True*, *prompt_pattern=None*, *get_pty=False*, *expected_pattern='\S'*, *unexpected_pattern='<timeout>'*, *filter=None*, *add_stderr=True*, **args*, ***kwargs*)

Ssh class helper

This class create a ssh connection in order to run one or many commands.

Parameters

- **host** (*str*) – IP address or hostname to connect to
- **user** (*str*) – The username to use for login
- **password** (*str*) – The password
- **timeout** (*int*) – Time in seconds before raising an error or a None value
- **prompt_pattern** (*str*) – None by Default. If defined, the way to run commands is to capture the command output up to the prompt pattern. If not defined, it uses paramiko exec_command() method (preferred way).
- **get_pty** (*bool*) – Create a pty, this is useful for some ssh connection (Default: False)
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found in methods that collect data (like run,mrun,get,mget,walk,mwalk...) if None, there is no test. By default, tests the result is not empty.

- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, it tests <timeout>.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.
- **add_stderr** (*bool*) – If True, the stderr will be added at the end of results (Default: True)
- **port** (*int*) – port number to use (Default : 0 = 22)
- **pkey** (*PKey*) – an optional private key to use for authentication
- **key_filename** (*str*) – the filename, or list of filenames, of optional private key(s) to try for authentication
- **allow_agent** (*bool*) – set to False to disable connecting to the SSH agent
- **look_for_keys** (*bool*) – set to False to disable searching for discoverable private key files in `~/.ssh/`
- **compress** (*bool*) – set to True to turn on compression
- **sock** (*socket*) – an open socket or socket-like object (such as a *.Channel*) to use for communication to the target host
- **gss_auth** (*bool*) – `True` if you want to use GSS-API authentication
- **gss_kex** (*bool*) – Perform GSS-API Key Exchange and user authentication
- **gss_deleg_creds** (*bool*) – Delegate GSS-API client credentials or not
- **gss_host** (*str*) – The targets name in the kerberos database. default: hostname
- **banner_timeout** (*float*) – an optional timeout (in seconds) to wait for the SSH banner to be presented.

mrun(*cmds*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)

Execute many commands at the same time

Runs a dictionary of commands at the specified prompt and then close the connection. Timeout will not raise any error but will return None for the running command. It returns a dictionary where keys are the same as the `cmds` dict and the values are the commmands output.

Parameters

- **cmds** (*dict or list of items*) – The commands to be executed by remote host
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the connection.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the

command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The commands output

Return type `textops.DictExt`

Example

SSH with multiple commands:

```
ssh = Ssh('localhost','www','wwwpassword')
print ssh.mrun({'cur_dir':'pwd','big_files':'find . -type f -size +10000'})
```

Will return something like:

```
{
    'cur_dir' : '/home/www',
    'big_files' : 'bigfile1\nbigfile2\nbigfile3\n...'
}
```

To be sure to have the commands order respected, use list of items instead of a dict:

```
ssh = Ssh('localhost','www','wwwpassword')
print ssh.mrun( (('cmd','./mycommand'),('cmd_err','echo $?')) )
```

run(*cmd*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)

Execute one command

Runs a single command at the usual prompt and then close the connection. Timeout will not raise any error but will return None. If you want to execute many commands without closing the connection, use `with` syntax.

Parameters

- **cmd** (*str*) – The command to be executed
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the connection.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The command output or None on timeout

Return type `textops.StrExt`

Examples

SSH with default login/password/prompt:

```
ssh = Ssh('localhost','www','wwwpassword')
print ssh.run('ls -la')
```

SSH with multiple commands (use `with` to keep connection opened). This is usefull when one command depend on another one:

```
with Ssh('localhost','www','wwwpassword') as ssh:
    cur_dir = ssh.run('pwd').strip()
    big_files_full_path = ssh.run('find %s -type f -size +10000' % cur_dir)
print big_files_full_path
```

run_channels(*cmd*, *timeout=30*, *auto_close=True*, ***kwargs*)

Execute one command

Runs a single command at the usual prompt and then close the connection. Timeout will not raise any error but will return None. If you want to execute many commands without closing the connection, use `with` syntax. This function returns 3 values : stdout dump, stderr dump and exit status. Pattern testing and filtering is not available in this method (see `run()` method if you want them)

Parameters

- **cmd** (*str*) – The command to be executed
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the connection.

Returns output on stdout, errors on stderr and exit status

Return type out (str), err (str), status (int)

Examples

SSH with default login/password/prompt:

```
ssh = Ssh('localhost','www','wwwpassword')
out, err, status =  ssh.run_channels('ls -la')
print 'out = %s\nerr = %s\nstatus=%s' % (out, err, status)
```

SSH with multiple commands (use `with` to keep connection opened). This is usefull when one command depend on another one:

```
with Ssh('localhost','www','wwwpassword') as ssh:
    cur_dir, err, status = ssh.run_channels('pwd').strip()
    if not err:
        big_files_full_path = ssh.run('find %s -type f -size +10000' % cur_
↪dir)
print big_files_full_path
```

run_script(*script*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, *auto_strip=True*, *format_dict={}*, ***kwargs*)

Execute a script

Returns The script output or None on timeout

Return type textops.StrExt

3.5 Telnet

class naghelp.**Telnet**(*host*, *user*, *password=None*, *timeout=30*, *port=0*, *login_pattern=None*, *passwd_pattern=None*, *prompt_pattern=None*, *autherr_pattern=None*, *sleep=0*, *sleep_login=0*, *expected_pattern='\S'*, *unexpected_pattern='<timeout>'*, *filter=None*, **args*, ***kwargs*)

Telnet class helper

This class create a telnet connection in order to run one or many commands.

Parameters

- **host** (*str*) – IP address or hostname to connect to
- **user** (*str*) – The username to use for login
- **password** (*str*) – The password
- **timeout** (*int*) – Time in seconds before raising an error or a None value
- **port** (*int*) – port number to use (Default : 0 = 23)
- **login_pattern** (*str or list*) – The pattern to recognize the login prompt (Default : `login\s*:`). One can specify a string, a re.RegexObject, a list of string or a list of re.RegexObject
- **passwd_pattern** (*str or list*) – The pattern to recognize the password prompt (Default : `Password\s*:`). One can specify a string, a re.RegexObject, a list of string or a list of re.RegexObject
- **prompt_pattern** (*str*) – The pattern to recognize the usual prompt (Default : `[\r\n][^\s]*\s?[\$#>:]+\s`). One can specify a string or a re.RegexObject.
- **autherr_pattern** (*str*) – The pattern to recognize authentication error (Default : `bad password|login incorrect|login failed|authentication error`). One can specify a string or a re.RegexObject.
- **sleep** (*int*) – Add delay in seconds before each write/expect
- **sleep_login** (*int*) – Add delay in seconds before login
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found in methods that collect data (like run,mrun,get,mget,walk,mwalk...) if None, there is no test. By default, tests the result is not empty.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, it tests <timeout>.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.

mrun(*cmds*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)

Execute many commands at the same time

Runs a dictionary of commands at the specified prompt and then close the connection. Timeout will not raise any error but will return None for the running command. It returns a dictionary where keys are the same as the `cmds` dict and the values are the commmands output.

Parameters

- **cmds** (*dict or list of items*) – The commands to be executed by remote host
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the connection.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The commands output

Return type `textops.DictExt`

Example

Telnet with multiple commands:

```
tn = Telnet('localhost','www','wwwpassword')
print tn.mrun({'cur_dir':'pwd','big_files':'find . -type f -size +10000'})
```

Will return something like:

```
{
    'cur_dir' : '/home/www',
    'big_files' : 'bigfile1\nbigfile2\nbigfile3\n...'
}
```

run(*cmd*, *timeout=30*, *auto_close=True*, *expected_pattern=0*, *unexpected_pattern=0*, *filter=0*, ***kwargs*)
Execute one command

Runs a single command at the usual prompt and then close the connection. Timeout will not raise any error but will return None. If you want to execute many commands without closing the connection, use `with` syntax.

Parameters

- **cmd** (*str*) – The command to be executed
- **timeout** (*int*) – A timeout in seconds after which the result will be None
- **auto_close** (*bool*) – Automatically close the connection.
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found if None, there is no test. By default, use the value defined at object level.

- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, use the value defined at object level.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By default, use the filter defined at object level.

Returns The command output or None on timeout

Return type `textops.StrExt`

Examples

Telnet with default login/password/prompt:

```
tn = Telnet('localhost','www','wwwpassword')
print tn.run('ls -la')
```

Telnet with custom password prompt (password in french), note the `(?i)` for the case insensitive:

```
tn = Telnet('localhost','www','wwwpassword',password_pattern=r'(?i)Mot de␣
↪passe\s*:')
print tn.run('ls -la')
```

Telnet with multiple commands (use `with` to keep connection opened). This is usefull when one command depend on another one:

```
with Telnet('localhost','www','wwwpassword') as tn:
    cur_dir = tn.run('pwd').strip()
    big_files_full_path = tn.run('find %s -type f -size +10000' % cur_dir)
print big_files_full_path
```

3.6 Local commands

naghelp.**runsh**(*cmd*, *context={}*, *timeout=30*, *expected_pattern='\\S'*, *unexpected_pattern=None*, *filter=None*, *key=''*)

Run a local command with a timeout

If the command is a string, it will be executed within a shell.
If the command is a list (the command and its arguments), the command is executed without a shell.
If a context dict is specified, the command is formatted with that context (`str.format()`)

Parameters

- **cmd** (*str or a list*) – The command to run
- **context** (*dict*) – The context to format the command to run (Optional)
- **timeout** (*int*) – The timeout in seconds after with the forked process is killed and TimeoutException is raised (Default : 30s).

- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found. if None, there is no test. By default, tests the result is not empty.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, there is no test.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.
- **key** (*str*) – a key string to appear in UnexpectedResultError if any.

Returns Command execution stdout as a list of lines.

Return type `textops.ListExt`

Note: It returns **ONLY** stdout. If you want to get stderr, you need to redirect it to stdout.

Examples

```
>>> for line in runsh('ls -lad /etc/e*'):
...     print line
...
-rw-r--r-- 1 root root  392 oct.   8  2013 /etc/eclipse.ini
-rw-r--r-- 1 root root  350 mai   21  2012 /etc/eclipse.ini_old
drwxr-xr-x 2 root root 4096 avril 13  2014 /etc/elasticsearch
drwxr-xr-x 3 root root 4096 avril 25  2012 /etc/emacs
-rw-r--r-- 1 root root   79 avril 25  2012 /etc/environment
drwxr-xr-x 2 root root 4096 mai    1  2012 /etc/esound
```

```
>>> print runsh('ls -lad /etc/e*').grep('eclipse').tostr()
-rw-r--r-- 1 root root  392 oct.   8  2013 /etc/eclipse.ini
-rw-r--r-- 1 root root  350 mai   21  2012 /etc/eclipse.ini_old
```

```
>>> l=runsh('LANG=C ls -lad /etc/does_not_exist')
>>> print l
[]
>>> l=runsh('LANG=C ls -lad /etc/does_not_exist 2>&1')
>>> print l
['ls: cannot access /etc/does_not_exist: No such file or directory']
```

naghelp.**runshex**(*cmd*, *context={}*, *timeout=30*, *expected_pattern='\\S'*, *unexpected_pattern=None*, *filter=None*, *key=''*, *unexpected_stderr=True*)

Run a local command with a timeout

If the command is a string, it will be executed within a shell.

If the command is a list (the command and its arguments), the command is executed without a shell.

If a context dict is specified, the command is formatted with that context (`str.format()`)

Parameters

- **cmd** (*str or a list*) – The command to run
- **context** (*dict*) – The context to format the command to run (Optional)
- **timeout** (*int*) – The timeout in seconds after with the forked process is killed and TimeoutException is raised (Default : 30s).
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found. if None, there is no test. By default, tests the result is not empty.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, there is no test.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.
- **key** (*str*) – a key string to appear in UnexpectedResultError if any.
- **unexpected_stderr** (*bool*) – When True (Default), it raises an error if stderr is not empty

Returns stdout, stderr, return code tuple

Return type tuple

Note: It returns **ONLY** stdout. If you want to get stderr, you need to redirect it to stdout.

---

naghelp.**mrunsh**(*cmds*, *context={}*, *cmd_timeout=30*, *total_timeout=60*, *expected_pattern='\\S'*, *unexpected_pattern=None*, *filter=None*)

Run multiple local commands with timeouts

It works like *runsh()* except that one must provide a dictionary of commands. It will generate the same dictionary where values will be replaced by command execution output. It is possible to specify a by-command timeout and a global timeout for the whole dictionary.

Parameters

- **cmds** (*dict*) – dictionary where values are the commands to execute.

  If the command is a string, it will be executed within a shell.
  If the command is a list (the command and its arguments), the command is executed without a shell.
  If a context dict is specified, the command is formatted with that context (`str.format()`)

- **context** (*dict*) – The context to format the command to run
- **cmd_timeout** (*int*) – The timeout in seconds for a single command
- **total_timeout** (*int*) – The timeout in seconds for the all commands
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found. if None, there is no test. By default, tests the result is not empty.

- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, there is no test.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.

Returns Dictionary where each value is the Command execution stdout as a list of lines.

Return type `textops.DictExt`

Note: Command execution returns **ONLY** stdout. If you want to get stderr, you need to redirect it to stdout.

Examples

```
>>> mrunsh({'now':'LANG=C date','quisuisje':'whoami'})
{'now': ['Wed Dec 16 11:50:08 CET 2015'], 'quisuisje': ['elapouya']}
```

naghelp.**mrunshex**(*cmds*, *context={}*, *cmd_timeout=30*, *total_timeout=60*, *expected_pattern='\\S'*, *unexpected_pattern=None*, *filter=None*, *unexpected_stderr=True*)

Run multiple local commands with timeouts

It works like *runshex()* except that one must provide a dictionary of commands. It will generate the same dictionary where values will be replaced by command execution output. It is possible to specify a by-command timeout and a global timeout for the whole dictionary. stderr are store in `<key>_stderr` and return codes in `<key>_rcode`

Parameters

- **cmds** (*dict*) – dictionary where values are the commands to execute.

  If the command is a string, it will be executed within a shell.
  If the command is a list (the command and its arguments), the command is executed without a shell.
  If a context dict is specified, the command is formatted with that context (`str.format()`)

- **context** (*dict*) – The context to format the command to run
- **cmd_timeout** (*int*) – The timeout in seconds for a single command
- **total_timeout** (*int*) – The timeout in seconds for the all commands
- **expected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is not found. if None, there is no test. By default, tests the result is not empty.
- **unexpected_pattern** (*str or regex*) – raise UnexpectedResultError if the pattern is found if None, there is no test. By default, there is no test.
- **filter** (*callable*) – call a filter function with `result, key, cmd` parameters. The function should return the modified result (if there is no return statement, the original result is used). The filter function is also the place to do some other checks : `cmd` is the command

that generated the `result` and `key` the key in the dictionary for `mrun`, `mget` and `mwalk`. By Default, there is no filter.

- **unexpected_stderr** (*bool*) – When True (Default), it raises an error if stderr is not empty

Returns Dictionary where each value is the Command execution stdout as a list of lines.

Return type `textops.DictExt`

Note: Command execution returns **ONLY** stdout. If you want to get stderr, you need to redirect it to stdout.

Examples

```
>>> mrunsh({'now':'LANG=C date','quisuisje':'whoami'})
{'now': ['Wed Dec 16 11:50:08 CET 2015'], 'quisuisje': ['elapouya']}
```

3.7 Others

naghelp.**search_invalid_port**(*ip*, *ports*)

Returns the first invalid port encountered or None if all are reachable

Parameters

- **ip** (*str*) – ip address to test
- **ports** (*str or list of int*) – list of ports to test

Returns first invalid port encountered or None if all are reachable

Examples

```
>>> search_invalid_port('8.8.8.8','53')
(None)
>>> search_invalid_port('8.8.8.8','53,22,80')
22
```

3.8 Exceptions

exception naghelp.**NotConnected**

Exception raised when trying to collect data on an already close connection

After a run()/mrun() without a `with:` clause, the connection is automatically closed. Do not do another run()/mrun() in the row otherwise you will have the exception. Solution : use `with:`

exception naghelp.**ConnectionError**

Exception raised when trying to initialize the connection

It may come from bad login/password, bad port, inappropriate parameters and so on

exception naghelp.**CollectError**
Exception raised when a collect is unsuccessful

It may come from internal error from libraries pexpect/telnetlib/pysnmp. This includes some internal timeout exception.

exception naghelp.**TimeoutError**
Exception raised when a connection or a collect it too long to process

It may come from unreachable remote host, too long lasting commands, bad pattern matching on Expect/Telnet/Ssh for connection or prompt search steps.

exception naghelp.**UnexpectedResultError**
Exception raised when a command gave an unexpected result

This works with `expected_pattern` and `unexpected_pattern` available in some collecting classes.

- genindex
- modindex
- search

CHAPTER 4

Host

This module defined the naghelp Host object.

The Host object will store all informations about the equipment to monitor. It could be for exemple :

- The hostname
- Host IP address
- The user login
- The user password
- The snmp community
- ...

One can add some additional data during the plugin execution, they will be persistent accross all plugin executions (plugin objects call *save_data()* and *load_data()* methods). This is useful for :

- Counters
- Gauges
- Timestamps
- Flags
- ...

Informations come from 3 sources in this order :

- From a database (text file, sqlite, mysql ...)
- From environment variables
- From command line

Informations from command line have priority over environment vars which have priority over those from database.

class naghelp.**Host**(*plugin*)
Contains equipment informations

Host object is a dict with some additional methods.

Parameters **plugin** (Plugin) – The plugin object that is used to monitor the equipment

Informations can be accessed and modified by 2 ways :

- By attribute
- By Index

To save and load custom data, one could do a *save_data()* and *load_data()* but this is automatically done by the plugin itself (see *naghelp.ActivePlugin.run()*)

Examples :

```
>>> os.environ['NAGIOS_HOSTNAME']='host_to_be_monitored'
>>> plugin = ActivePlugin()
>>> host = Host(plugin)
>>> print host.name
host_to_be_monitored
>>> host.my_custom_data = 'last check time'
>>> print host.my_custom_data
last check time
>>> print host['my_custom_data']
last check time
>>> host.save_data()
>>> host._get_persistent_filename()
'/tmp/naghelp/host_to_be_monitored_persistent_data.json'
>>> print open(host._get_persistent_filename()).read()
{
    "name": "host_to_be_monitored",
    "my_custom_data": "last check time"
}
```

```
>>> os.environ['NAGIOS_HOSTNAME']='host_to_be_monitored'
>>> plugin = ActivePlugin()
>>> host = Host(plugin)
>>> print host.my_custom_data

>>> host.load_data()
>>> print host.my_custom_data
last check time
```

_get_env_to_param()

Returns a dict for the environment variable to extract

The keys are the environment variables to extract, the values are the attribute name to use for the Host object.

Only a few environment variables are automatically extracted, they are renamed when saved into Host object :

Environment Variables	Stored as
NAGIOS_HOSTNAME	name
NAGIOS_HOSTALIAS	alias
NAGIOS_HOSTADDRESS	ip
NAGIOS_HOSTGROUPNAMES	groups
NAGIOS_HOSTGROUPNAME	group

_get_params_from_db(*hostname*)
Get host informations from database

Getting informations from a database is optionnal. If needed, it is the developer responsibility to subclass `Host` class and redefine the method `_get_params_from_db(hostname)` that returns a dict with informations about `hostname`. In this method, the developer must also load persistent data that may come from a same cache file as the database. The default behaviour for this method is only to load persistent data from a database flat file (.json).

Parameters **hostname** (*str*) – The name of the equipment as declared in Nagios configuration files.

Returns A dictionary that contains equipment informations AND persistent data merged.

Return type dict

Example

Here is an example of *_get_params_from_db()* override where informations about a monitored host are stored in a json file located at DB_JSON_FILE

```
class MonitoredHost(Host):
    def _get_params_from_db(self,hostname):
        # The first step MUST be to read the persistent data file ( = cache␣
↪file )
        params = self._plugin.load_data(self._get_persistent_filename()) or␣
↪DictExt()

        # Check whether the database file has changed
        db_file_modif_time = int(os.path.getmtime(DB_JSON_FILE))
        if db_file_modif_time == params['db_file_modif_time']:
            # if not, return the cached data
            return params

        # If database file has changed :
        db = json.load(open(DB_JSON_FILE))
        # find hostname in db :
        for h in db.monitored_hosts:
            if h['datas']['hostname'] == hostname:
                # merge the new data into the persistent data dict (will be␣
↪cached)
                params.update(h.datas)
                params['db_file_modif_time'] = db_file_modif_time
                params['name'] = params.get('hostname','noname')
                return params
        return params
```

Note: You can do about the same for SQlite or MySQL, do not forget to load persistent data file as a first step and merge data from database after. Like the above example, you can use the persistent data json file as a cache for your database. By this way, persistent data AND database data are saved in the same file within a single operation. The dictionary returned by this method will be saved automatically by the *naghelp.ActivePlugin.run()* method as persistent data.

_get_persistent_filename()
Get the full path for the persisten .json file

It uses *persistent_filename_pattern* to get the file pattern, and apply the hostname to build the full path.

debug()
Log Host informations for debug

Note: To see debug on python console, call `naghelp.activate_debug()`

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

load_data()
load data to the *Host* object

That is from database and/or persistent file then from environment variables and then from command line.

persistent_filename_pattern = '/tmp/naghelp/%s_persistent_data.json'
Default persistent .json file path pattern (note the %s that will be replaced by the hostname)

save_data()
Save data to a persistent place

It actually saves the whole dict into a .json file. This is automatically called by the :meth:naghelp.ActivePlugin.run method.

to_list(*lst*, *defvalue='-'*)
Formats a list of strings with Host informations

It works like *to_str()* except that the input is a list of strings : This useful when a text has been splitted into lines.

Parameters

- **str** (*str*) – A format string
- **defvalue** (*str*) – String to display when a data is not available

Returns The list with formatted string

Return type list

Examples

```
>>> os.environ['NAGIOS_HOSTNAME']='host_to_be_monitored'
>>> os.environ['NAGIOS_HOSTADDRESS']='192.168.0.33'
>>> plugin = ActivePlugin()
>>> host = Host(plugin)
>>> host.load_data()
>>> lst = ['{name} as got IP={ip} and custom data "{my_custom_data}"',
... 'Not available data are replaced by a dash: {other_data}']
>>> print host.to_list(lst)
[u'host_to_be_monitored as got IP=192.168.0.33 and custom data "last check␣
↪time"',
'Not available data are replaced by a dash: -']
```

Note: As you noticed, `NAGIOS_HOSTNAME` environment variable is stored as `name` in Host object and `NAGIOS_HOSTADDRESS` as `ip`. `my_custom_data` is a persistent data that has been automatically loaded because set into a previous example.

to_str(*str*, *defvalue='-'*)

Formats a string with Host informations

Not available data are replaced by a dash

Parameters

- **str** (*str*) – A format string
- **defvalue** (*str*) – String to display when a data is not available

Returns the formatted string

Return type str

Examples

```
>>> os.environ['NAGIOS_HOSTNAME']='host_to_be_monitored'
>>> os.environ['NAGIOS_HOSTADDRESS']='192.168.0.33'
>>> plugin = ActivePlugin()
>>> host = Host(plugin)
>>> host.load_data()
>>> print host.to_str('{name} as got IP={ip} and custom data "{my_custom_data}
↪"')
host_to_be_monitored as got IP=192.168.0.33 and custom data "last check time"
>>> print host.to_str('Not available data are replaced by a dash: {other_data}
↪')
Not available data are replaced by a dash: -
>>> print host.to_str('Or by whatever you want: {other_data}','N/A')
Or by whatever you want: N/A
```

Note: As you noticed, `NAGIOS_HOSTNAME` environment variable is stored as `name` in Host object and `NAGIOS_HOSTADDRESS` as `ip`. `my_custom_data` is a persistent data that has been automatically loaded because set into a previous example.

- genindex
- modindex
- search

CHAPTER 5

PerfData

class naghelp.**PerfData**(*label*, *value*, *uom=None*, *warn=None*, *crit=None*, *minval=None*, *maxval=None*)

PerfData class is a convenient way to build a performance data object without taking care of the syntax needed by Nagios. The PerfData object can then be added into to *PluginResponse* object, with method *PluginResponse.add_perf_data()*.

Parameters

- **label** (*str*) – The metric name
- **value** (*str*) – The metric value
- **uom** (*str*) – unit of measurement, is one of:
  - no unit specified - assume a number (int or float) of things (eg, users, processes, load averages)
  - s - seconds (also us, ms)
  - % - percentage
  - B - bytes (also KB, MB, TB)
  - c - a continous counter (such as bytes transmitted on an interface)
- **warn** (*str*) – WARNING threshold
- **crit** (*str*) – CRITICAL threshold
- **min** (*str*) – Minimum value
- **max** (*str*) – Maximum value

Examples

```
>>> perf = PerfData('filesystem_/','55','%','95','98','0','100')
>>> print perf
filesystem_/=55%;95;98;0;100
>>> perf.value = 99
>>> print perf
filesystem_/=99%;95;98;0;100
```

- genindex
- modindex
- search

CHAPTER 6

Response

6.1 ResponseLevel

class naghelp.**ResponseLevel**(*name*, *exit_code*)

Object to use when exiting a naghelp plugin

Instead of using numeric code that may be hard to memorize, predefined objects has be created :

Response level object	exit code
OK	0
WARNING	1
CRITICAL	2
UNKNOWN	3

To exit a plugin with the correct exit code number, one have just to call the *exit()* method of the wanted ResonseLevel object

exit()

This is the official way to exit a naghelp plugin

Example

```
>>> level = CRITICAL
>>> level.exit()
    SystemExit: 2
```

info()

Get name and exit code for a Response Level

Examples

```
>>> level = CRITICAL
>>> level.info()
'CRITICAL (exit_code=2)'
>>> level.name
'CRITICAL'
>>> level.exit_code
2
```

6.2 PluginResponse

class naghelp.**PluginResponse**(*default_level=OK*)

Response to return to Nagios for a naghelp plugin

Parameters **default_level** (*ResponseLevel*) – The level to return when no level messages has been added to the response (for exemple when no error has be found). usually it is set to `OK` or `UNKNOWN`

A naghelp response has got many sections :

- A synopsis (The first line that is directl visible onto Nagios interface)
- A body (informations after the first line, only visible in detailed views)
- Some performance Data

The body itself has got some sub-sections :

- Begin messages (Usually for a title, an introduction . . .)
- Levels messages, that are automatically splitted into Nagios levels in this order:
  - Critical messages
  - Warning messages
  - Unkown messages
  - OK messages
- More messages (Usually to give more information about the monitored host)
- **End messages (Custom conclusion messages. naghelp `Plugin` use this section** to add automatically some informations about the plugin.

Each section can be updated by adding a message through dedicated methods.

PluginResponse object takes care to calculate the right ResponseLevel to return to Nagios : it will depend on the Levels messages you will add to the plugin response. For example, if you add one `OK` message and one `WARNING` message, the response level will be `WARNING`. if you add again one `CRITICAL` message then an `OK` message , the response level will be `CRITICAL`.

About the synopsis section : if not manualy set, the PluginResponse class will build one for you : It will be the unique level message if you add only one in the response or a summary giving the number of messages in each level.

Examples

```
>>> r = PluginResponse(OK)
>>> print r
OK
```

add(*level*, *msg*, **args*, ***kwargs*)

Add a message in levels messages section and sets the response level at the same time

Use this method each time your plugin detects a WARNING or a CRITICAL error. You can also use this method to add a message saying there is an UNKNOWN or OK state somewhere. You can use this method several times and at any time until the *send()* is used. This method updates the calculated ResponseLevel. When the response is rendered, the added messages are splitted into sub-section

Parameters

- **level** (ResponseLevel) – the message level (Will affect the final response level)
- **msg** (*str*) – the message to add in levels messages section.
- **args** (*list*) – if additional arguments are given, msg will be formatted with % (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, msg will be formatted with str.format()

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> r.add(CRITICAL,'The system crashed')
>>> r.add(WARNING,'Found some almost full file system')
>>> r.add(UNKNOWN,'Cannot find FAN %s status',0)
>>> r.add(OK,'Power {power_id} is ON',power_id=1)
>>> print r.get_current_level()
CRITICAL
>>> print r
STATUS : CRITICAL:1, WARNING:1, UNKNOWN:1, OK:1
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
The system crashed

----( WARNING )-------------------------------------------------------------
↪--
Found some almost full file system

----( UNKNOWN )-------------------------------------------------------------
↪--
Cannot find FAN 0 status

----( OK )------------------------------------------------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
Power 1 is ON
```

add_begin(*msg*, **args*, ***kwargs*)

Add a message in begin section

You can use this method several times and at any time until the *send()* is used. The messages will be displayed in begin section in the same order as they have been added. This method does not change the calculated ResponseLevel.

Parameters

- **msg** (*str*) – the message to add in begin section.
- **args** (*list*) – if additionnal arguments are given, `msg` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, `msg` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> r.add_begin('='*40)
>>> r.add_begin('{hostname:^40}', hostname='MyHost')
>>> r.add_begin('='*40)
>>> r.add_begin('Date : %s, Time : %s','2105-12-18','14:55:11')
>>> r.add_begin('\n')
>>> r.add(CRITICAL,'This is critical !')
>>> print r
This is critical !
========================================
                 MyHost
========================================
Date : 2105-12-18, Time : 14:55:11

==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !
```

add_comment(*level*, *msg*, **args*, ***kwargs*)

Add a comment in levels messages section and sets the response level at the same time

it works like *add()* except that the message is not counted into the synopsis

Parameters

- **level** (`ResponseLevel`) – the message level (Will affect the final response level)
- **msg** (*str*) – the message to add in levels messages section.
- **args** (*list*) – if additionnal arguments are given, `msg` will be formatted with `%` (old-style python string formatting)

- **kwargs** (*dict*) – if named arguments are given, `msg` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> r.add_comment(CRITICAL,'Here are some errors')
>>> r.add(CRITICAL,'error 1')
>>> r.add(CRITICAL,'error 2')
>>> print r
STATUS : CRITICAL:2
==================================[  STATUS  ]==================================

----( CRITICAL )-------------------------------------------------------------
Here are some errors
error 1
error 2
```

add_elif(**add_ifs*, ***kwargs*)

Multi-conditionnal message add

This works like *add_if()* except that it accepts multiple tests. Like python `elif`, the method stops on first True test and send corresponding message. If you want to build the equivalent of a *default* message, just use `True` as the last test.

Parameters

- **add_ifs** (*list*) – list of tuple (test,level,message).
- **kwargs** (*dict*) – if named arguments are given, messages will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> logs = '''
... Power 0 is critical
... Power 1 is OK
... Power 2 is degraded
... Power 3 is failed
... Power 4 is OK
... Power 5 is degraded
... Power 6 is smoking
... '''
>>> from textops import *
>>> for log in logs | rmblank():
...     r.add_elif( (log|haspattern('critical|failed'), CRITICAL, log),
...                 (log|haspattern('degraded|warning'), WARNING, log),
```

(continues on next page)

(continued from previous page)

```
...                 (log|haspattern('OK'), OK, log),
...                 (True, UNKNOWN, log) )
>>> print r.get_current_level()
CRITICAL
>>> print r
STATUS : CRITICAL:2, WARNING:2, UNKNOWN:1, OK:2
==================================[  STATUS  ]==================================

----( CRITICAL )-------------------------------------------------------------
Power 0 is critical
Power 3 is failed

----( WARNING )-------------------------------------------------------------
Power 2 is degraded
Power 5 is degraded

----( UNKNOWN )-------------------------------------------------------------
Power 6 is smoking

----( OK )------------------------------------------------------------------
Power 1 is OK
Power 4 is OK
```

add_end(*msg*, **args*, ***kwargs*)

Add a message in end section

You can use this method several times and at any time until the *send()* is used. The messages will be displayed in end section in the same order as they have been added. This method does not change the calculated ResponseLevel.

Parameters

- **msg** (*str*) – the message to add in end section.
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are give, `msg` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> r.add_end('='*40)
>>> r.add_end('{hostname:^40}', hostname='MyHost')
>>> r.add_end('='*40)
>>> r.add_end('Date : %s, Time : %s','2105-12-18','14:55:11')
>>> r.add_end('\n')
>>> r.add(CRITICAL,'This is critical !')
>>> print r
```

(continues on next page)

(continued from previous page)

```
This is critical !

==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !


=======================================
                MyHost
=======================================
Date : 2105-12-18, Time : 14:55:11
```

add_if (*test*, *level*, *msg=None*, *header=None*, *footer=None*, **args*, ***kwargs*)

Test than add a message in levels messages section and sets the response level at the same time

This works like *add()* except that it is conditionnal : `test` must be True. If no message is given, the value of `test` is used.

Parameters

- **test** (*any*) – the message is added to the response only if bool(test) is True.
- **level** (`ResponseLevel`) – the message level (Will affect the final response level)
- **msg** (*str*) – the message to add in levels messages section. If no message is given, the value of test is used.
- **header** (*str*) – Displayed before the message as a level comment if not None (Default : None)
- **footer** (*str*) – Displayed after the message as a level comment if not None (Default : None)
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, `msg` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> logs = '''
... Power 0 is critical
... Power 1 is OK
... Power 2 is degraded
... Power 3 is failed
... Power 4 is OK
... Power 5 is degraded
... '''
>>> from textops import *
>>> nb_criticals = logs | grepc('critical|failed')
```

(continues on next page)

(continued from previous page)

```
>>> print nb_criticals
2
>>> warnings = logs | grep('degraded|warning').tostr()
>>> print warnings
Power 2 is degraded
Power 5 is degraded
>>> unknowns = logs | grep('unknown').tostr()
>>> print unknowns

>>> r.add_if(nb_criticals,CRITICAL,'{n} power(s) are critical',n=nb_criticals)
>>> r.add_if(warnings,WARNING)
>>> r.add_if(unknowns,UNKNOWN)
>>> print r.get_current_level()
CRITICAL
>>> print r
STATUS : CRITICAL:1, WARNING:1
==================================[  STATUS
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
2 power(s) are critical

----( WARNING )-------------------------------------------------------------
↪--
Power 2 is degraded
Power 5 is degraded
```

add_list (*level*, *msg_list*, *header=None*, *footer=None*, **args*, ***kwargs*)

Add several level messages having a same level

Sometimes, you may have to specify a list of faulty parts in the response : this can be done by this method in a single line. If a message is empty in the list, it is not added.

Parameters

- **level** (ResponseLevel) – the message level (Will affect the final response level)
- **msg_list** (*list*) – the messages list to add in levels messages section.
- **header** (*str*) – Displayed before the message as a level comment if not None (Default : None) one can use `{_len}` in the comment to get list count.
- **footer** (*str*) – Displayed after the message as a level comment if not None (Default : None) one can use `{_len}` in the comment to get list count.
- **args** (*list*) – if additional arguments are given, messages in `msg_list` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, messages in `msg_list` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
```

(continues on next page)

(continued from previous page)

```
OK
>>> logs = '''
... Power 0 is critical
... Power 1 is OK
... Power 2 is degraded
... Power 3 is failed
... Power 4 is OK
... Power 5 is degraded
... '''
>>> from textops import grep
>>> criticals = logs >> grep('critical|failed')
>>> warnings = logs >> grep('degraded|warning')
>>> print criticals
['Power 0 is critical', 'Power 3 is failed']
>>> print warnings
['Power 2 is degraded', 'Power 5 is degraded']
>>> r.add_list(CRITICAL,criticals)
>>> r.add_list(WARNING,warnings)
>>> print r.get_current_level()
CRITICAL
>>> print r
STATUS : CRITICAL:2, WARNING:2
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
Power 0 is critical
Power 3 is failed

----( WARNING )-------------------------------------------------------------
↪--
Power 2 is degraded
Power 5 is degraded
```

```
>>> r = PluginResponse()
>>> r.add_list(WARNING,['Power warning1','Power warning2'],'{_len} Power␣
↪warnings:','Power warnings : {_len}')
>>> r.add_list(WARNING,['CPU warning1','CPU warning2'],'{_len} CPU warnings:',
↪'CPU warnings : {_len}')
>>> print r
STATUS : WARNING:4
==================================[  STATUS ␣
↪]==================================

----( WARNING )-------------------------------------------------------------
↪--
2 Power warnings:
Power warning1
Power warning2
Power warnings : 2
2 CPU warnings:
CPU warning1
CPU warning2
CPU warnings : 2
```

add_many (*lst*, **args*, ***kwargs*)

Add several level messages NOT having a same level

This works like *add_list()* except that instead of giving a list of messages one have to specify a list of tuples (level,message). By this way, one can give a level to each message into the list. If a message is empty in the list, it is not added.

Parameters

- **lst** (*list*) – A list of (level,message) tuples to add in levels messages section.
- **args** (*list*) – if additional arguments are given, messages in `lst` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, messages in `lst` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> logs = '''
... Power 0 is critical
... Power 1 is OK
... Power 2 is degraded
... Power 3 is failed
... Power 4 is OK
... Power 5 is degraded
... '''
>>> from textops import *
>>> errors = [ (CRITICAL if error|haspatterni('critical|failed') else WARNING,
↪error)
...             for error in logs | grepv('OK') ]
>>> print errors
[(WARNING, ''), (CRITICAL, 'Power 0 is critical'), (WARNING, 'Power 2 is␣
↪degraded'),
(CRITICAL, 'Power 3 is failed'), (WARNING, 'Power 5 is degraded')]
>>> r.add_many(errors)
>>> print r.get_current_level()
CRITICAL
>>> print r
STATUS : CRITICAL:2, WARNING:2
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )--------------------------------------------------------------
↪--
Power 0 is critical
Power 3 is failed

----( WARNING )---------------------------------------------------------------
↪--
Power 2 is degraded
Power 5 is degraded
```

add_mlist (*levels*, *lists*, *headers=[]*, *footers=[]*, **args*, ***kwargs*)
Add a list of lists of messages

This is useful with `textops.sgrep` utility that returns lists of lists of messages. first level will be affected to first list, second level to the second list and so on.

Parameters

- **levels** (`ResponseLevel`) – a list of message levels. use a `None` level to skip a list.
- **lists** (*list*) – list of message lists.
- **header** (*str*) – Displayed before the message as a level comment if not None (Default : None) one can use `{_len}` in the comment to get list count.
- **footer** (*str*) – Displayed after the message as a level comment if not None (Default : None) one can use `{_len}` in the comment to get list count.
- **args** (*list*) – if additional arguments are given, messages in `msg_list` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are given, messages in `msg_list` will be formatted with `str.format()`

Examples

```
>>> from textops import StrExt
>>> r = PluginResponse(OK)
>>> logs = StrExt('''
... Power 0 is critical
... Power 1 is OK
... Power 2 is degraded
... Power 3 is failed
... Power 4 is OK
... Power 5 is degraded
... ''')
>>>
>>> print logs.sgrep(('critical|failed','degraded|warning'))
[['Power 0 is critical', 'Power 3 is failed'],
['Power 2 is degraded', 'Power 5 is degraded'],
['', 'Power 1 is OK', 'Power 4 is OK']]
>>> r.add_mlist((CRITICAL,WARNING,OK),logs.sgrep(('critical|failed',
↪'degraded|warning')))
>>> print r
STATUS : CRITICAL:2, WARNING:2, OK:2
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
Power 0 is critical
Power 3 is failed

----( WARNING )-------------------------------------------------------------
↪--
Power 2 is degraded
Power 5 is degraded

----( OK )-----------------------------------------------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
Power 1 is OK
Power 4 is OK

>>> r = PluginResponse(OK)
>>> r.add_mlist((CRITICAL,WARNING,None),logs.sgrep(('critical|failed',
↪'degraded|warning')))
>>> print r
STATUS : CRITICAL:2, WARNING:2
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
Power 0 is critical
Power 3 is failed

----( WARNING )-------------------------------------------------------------
↪--
Power 2 is degraded
Power 5 is degraded
```

add_more(*msg*, **args*, ***kwargs*)

Add a message in "more messages" section (aka "Additionnal informations")

You can use this method several times and at any time until the *send()* is used. The messages will be displayed in the section in the same order as they have been added. This method does not change the calculated ResponseLevel.

Parameters

- **msg** (*str*) – the message to add in end section.
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are give, `msg` will be formatted with `str.format()`

Note: The "Additionnal informations" section title will be added automatically if the section is not empty.

Examples

```
>>> r = PluginResponse(OK)
>>> r.add(CRITICAL,'This is critical !')
>>> r.add_more('Date : %s, Time : %s','2105-12-18','14:55:11')
>>> print r
This is critical !
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !
```

(continues on next page)

(continued from previous page)

```
==========================[ Additionnal informations
↪]==========================
Date : 2105-12-18, Time : 14:55:11
```

add_perf_data(*data*)

Add performance object into the response

Parameters **data** (str or *PerfData*) – the perf data string or PerfData object to add to the response. Have a look to Performance data string syntax.

Examples

```
>>> r = PluginResponse(OK)
>>> r.add_begin('Begin\n')
>>> r.add_end('End')
>>> r.add_perf_data(PerfData('filesystem_/','55','%','95','98','0','100'))
>>> r.add_perf_data(PerfData('filesystem_/usr','34','%','95','98','0','100'))
>>> r.add_perf_data('cpu_wait=88%;40;60;0;100')
>>> r.add_perf_data('cpu_user=12%;80;95;0;100')
>>> print r
OK|filesystem_/=55%;95;98;0;100
Begin
End| filesystem_/usr=34%;95;98;0;100 cpu_wait=88%;40;60;0;100 cpu_user=12%;80;
↪95;0;100
```

escape_msg(*msg*)

Escapes forbidden chars in messages

Nagios does not accept the pipe symbol in messages because it is a separator for performance data. This method escapes or replace such forbidden chars. Default behaviour is to replace the pipe `|` by an exclamation mark `!`.

Parameters **msg** (*str*) – The message to escape

Returns

str : The escaped message

get_current_level()

get current level

If no level has not been set yet, it will return the default_level. Use this method if you want to know what ResponseLevel will be sent.

Returns the response level to be sent

Return type *ResponseLevel*

Examples

```
>>> r = PluginResponse(OK)
>>> print r.get_current_level()
OK
>>> r.set_level(WARNING)
```

(continues on next page)

(continued from previous page)

```
>>> print r.get_current_level()
WARNING
```

get_default_synopsis()
Returns the default synopsis

This method is called if no synopsis has been set manually, the response synopsis (first line of the text returned by the plugin) will be :

- The error message if there is only one level message
- Otherwise, some statistics like : `STATUS : CRITICAL:2, WARNING:2, UNKNOWN:1, OK:2`

If you want to have a different default synopsis, you can subclass the *PluginResponse* class and redefine this method.

Examples

```
>>> r = PluginResponse(OK)
>>> r.add(CRITICAL,'This is critical !')
>>> print r.get_default_synopsis()
This is critical !
>>> r.add(WARNING,'This is just a warning.')
>>> print r.get_default_synopsis()
STATUS : CRITICAL:1, WARNING:1
```

get_output(*body_max_length=None*)
Renders the whole response following the Nagios syntax

This method is automatically called when the response is sent by *send()*. If you want to have a different output, you can subclass the *PluginResponse* class and redefine this method.

Returns The response text output following the Nagios syntax

Return type str

Note: As __str__ directly calls *get_output()*, printing a *PluginResponse* object is equivalent to call *get_output()*.

Example

```
>>> r = PluginResponse(OK)
>>> r.add(CRITICAL,'This is critical !')
>>> r.add(WARNING,'This is just a warning.')
```

```
>>> print r.get_output()
STATUS : CRITICAL:1, WARNING:1
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
```

(continues on next page)

(continued from previous page)

```
This is critical !

----( WARNING )--------------------------------------------------------------
↪--
This is just a warning.
```

```
>>> print r
STATUS : CRITICAL:1, WARNING:1
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !

----( WARNING )--------------------------------------------------------------
↪--
This is just a warning.
```

get_sublevel()

get sublevel

Returns sublevel (0,1,2 or 3)

Return type int

Exemples:

```
>>> r = PluginResponse(OK)
>>> print r.get_sublevel()
0
>>> r.set_sublevel(2)
>>> print r.get_sublevel()
2
```

level_msgs_render()

Renders level messages

This method is automatically called when the response is rendered by `get_outpout()`. If you want to have a different output, you can subclass the *PluginResponse* class and redefine this method.

Returns The foramtted level messages

Return type str

Example

```
>>> r = PluginResponse(OK)
>>> r.add(CRITICAL,'This is critical !')
>>> r.add(WARNING,'This is just a warning.')
>>> print r.level_msgs_render()
==================================[  STATUS ␣
↪]==================================
```

(continues on next page)

(continued from previous page)

```
----( CRITICAL )--------------------------------------------------------------
 ↪--
This is critical !

----( WARNING )---------------------------------------------------------------
 ↪--
This is just a warning.
```

section_format(*title*)
Returns the section title string

This method is automatically called when the response is rendered by `get_outpout()`. If you want to have a different output, you can subclass the *PluginResponse* class and redefine this method.

Parameters **title** (*str*) – the section name to be formatted

Returns The foramtted section title

Return type str

Example

```
>>> r = PluginResponse(OK)
>>> print r.section_format('My section')
================================[ My section␣
 ↪]================================
```

send(*level=None*, *synopsis=''*, *msg=''*, *sublevel=None*, *nagios_host=None*, *nagios_svc=None*, *nagios_cmd=None*)
Send the response to Nagios

This method is automatically called by *naghelp.ActivePlugin.run()* method and follow these steps :

- if defined, force a level, a sublevel, a synopsis or add a last message
- render the response string following the Nagios syntax
- display the string on stdout
- exit the plugin with the exit code corresponding to the response level.

Parameters

- **level** (*ResponseLevel*) – force a level (optional),
- **synopsis** (*str*) – force a synopsis (optional),
- **msg** (*str*) – add a last level message (optional),
- **sublevel** (*int*) – force a sublevel [0-3] (optional),
- **nagios_host** (*str*) – nagios hostname (only for passive response)
- **nagios_svc** (*str*) – nagios service (only for passive response)
- **nagios_cmd** (*str or pynag obj*) – if None, the response goes to stdout : this is for active plugin. if string the response is sent to the corresponding file for debug. if it is a pynag cmd object, the response is sent to the nagios pipe for host `nagios_host` and for service `nagios_svc` : this is for a passive plugin.

set_level (*level*)
Manually set the response level

Parameters **level** (*ResponseLevel*) – OK, WARNING, CRITICAL or UNKNOWN

Examples

```
>>> r = PluginResponse(OK)
>>> print r.level
None
>>> r.set_level(WARNING)
>>> print r.level
WARNING
```

set_sublevel (*sublevel*)
sets sublevel attribute

Parameters **sublevel** (*int*) – 0,1,2 or 3 (Default : 0)

From time to time, the CRITICAL status meaning is not detailed enough : It may be useful to color it by a sub-level. The `sublevel` value is not used directly by *PluginResponse*, but by *ActivePlugin* class which adds a `__sublevel__=<sublevel>` string in the plugin informations section. This string can be used for external filtering.

Actually, the sublevel meanings are :

Sub-level	Description
0	The plugin is 100% sure there is a critical error
1	The plugin was able to contact remote host (ping) but got no answer from agent (time-out,credentials)
2	The plugin was unable to contact the remote host, it may be a network issue (firewall, ...)
3	The plugin raised an unexpected exception : it should be a bug.

set_synopsis (*msg*, **args*, ***kwargs*)
Sets the response synopsis.

By default, if no synopsis has been set manually, the response synopsis (first line of the text returned by the plugin) will be :

- The error message if there is only one level message
- Otherwise, some statistics like : `STATUS : CRITICAL:2, WARNING:2, UNKNOWN:1, OK:2`

If something else is wanted, one can define a custom synopsis with this method.

Parameters

- **msg** (*str*) – the synopsis.
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)
- **kwargs** (*dict*) – if named arguments are give, `msg` will be formatted with `str.format()`

Examples

```
>>> r = PluginResponse(OK)
>>> r.add(CRITICAL,'This is critical !')
>>> print r
This is critical !
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !

>>> r.set_synopsis('Mayday, Mayday, Mayday')
>>> print r
Mayday, Mayday, Mayday
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
This is critical !
```

subsection_format(*title*)

Returns the subsection title string

This method is automatically called when the response is rendered by `get_outpout()`. If you want to have a different output, you can subclass the *PluginResponse* class and redefine this method.

Parameters **title** (*str*) – the subsection name to be formatted

Returns The foramtted subsection title

Return type str

Example

```
>>> r = PluginResponse(OK)
>>> print r.subsection_format('My subsection')
----( My subsection )--------------------------------------------------------
↪--
```

- genindex
- modindex
- search

CHAPTER 7

Plugin module

7.1 Plugin

class naghelp.plugin.**Plugin**
Plugin base class

This is an abstract class used with *ActivePlugin*, it brings :

- plugin search in a directory of python files
- plugin instance generation
- plugin logging management
- plugin command line options management
- plugin persistent data management

This abstract class should be used later with `naghelp.PassivePlugin` when developed.

add_cmd_options()
This method can be customized to add some OptionParser options for the current plugin

Example:

```
self._cmd_parser.add_option('-z', action='store_true', dest='super_debug',
                            default=False, help='Activate the super debug mode
↪')
```

add_logger_console_handler()
Activate logging to the console

add_logger_file_handler()
Activate logging to the log file

classmethod debug(*msg*, **args*, ***kwargs*)
log a debug message

Parameters

- **msg** (*str*) – the message to log
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)

Examples

This logs a debug message in log file and/or console:

```
p = Plugin()
p.debug('my_variable = %s',my_variable)
```

classmethod error(*msg*, **args*, ***kwargs*)
log an error message

Parameters

- **msg** (*str*) – the message to log
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)

Examples

This logs an error message in log file and/or console:

```
p = Plugin()
p.error('Syntax error in line %s',36)
```

classmethod find_plugins()
Recursively find all plugin classes for all python files present in a directory.

It finds all python files inside `YourPluginsBaseClass.plugins_basedir` then look for all classes having the attribute `plugin_type` with the value `YourPluginsBaseClass.plugin_type`

It returns a dictionary where keys are plugin class name in lower case and the values are a dictionary containing :

Keys	Values
name	the class name (case sensitive)
module	the plugin module name with a full dotted path
path	the module file path
desc	the plugin description (first docstring line)

classmethod find_plugins_import_errors()
Find all import errors all python files present in a directory.

It finds all python files inside `YourPluginsBaseClass.plugins_basedir` and try to import them. If an error occurs, the file path and linked exception is memorized.

It returns a list of tuples containing the file path and the exception.

found_plugins = {}
Plugins discovered during find_plugins() method

get_cmd_usage()
Returns the command line usage

classmethod get_instance(*plugin_name*, ***extra_options*)
Generate a plugin instance from its name string

This method is useful when you only know at execution time the name of the plugin to instantiate.

If you have an active plugin class called `HpProliant` in a file located at `/home/me/myplugin_dir/hp/hp_proliant.py` then you can get the plugin instance this way :

For all your plugins, you should first subclass the ActivePlugin to override plugins_basedir:

```
class MyProjectActivePlugin(ActivePlugin):
    plugins_basedir = '/home/me/myplugin_dir'
    plugin_type = 'myproject_plugin'  # you choose whatever you want but not
↪'plugin'

class HpProliant(MyProjectActivePlugin):
    ''' My code '''
```

Check that `/home/me/myplugin_dir` is in your python path.

Then you can get an instance by giving only the class name (case insensitive):

```
plugin = MyProjectActivePlugin.get_instance('hpproliant')
```

Or with the full doted path (case sensitive this time):

```
plugin = MyProjectActivePlugin.get_instance('hp.hp_proliant.HpProliant')
```

In first case, it is shorter but a recursive search will occur to find the class in all python files located in `/home/me/myplugin_dir/`. It is mainly useful in python interactive console.

In second case, as you specified the full path, the class is found at once : it is faster and should be used in production.

Once the class is found, an instance is created and returned.

Of course, if you do not need to get an instance from a string : it is more pythonic to do like this:

```
from hp.hp_proliant import HpProliant
plugin = HpProliant()
```

get_logger_console_level()
gets logger level specific for the console output.

Note : This is possible to set different logger level between log file and console

get_logger_file_level()
gets logger level specific for log file output.

Note : This is possible to set different logger level between log file and console

get_logger_file_logfile()
get log file path

get_logger_format()
gets logger format, by default the one defined in `logger_format` attribute

get_logger_level()
gets logger level. By default sets to `logging.ERROR` to get only errors

classmethod get_plugin(*plugin_name*)
find a plugin and return its module and its class name

To get the class itself, one have to get the corresponding module's attribute:

```
module,class_name = MyProjectActivePlugin.get_plugin('hpproliant')
plugin_class = getattr(module,class_name,None)
```

Parameters **plugin_name** (*str*) – the plugin name to find (case insensitive)

Returns A tuple containing the plugin's module and plugin's class name

Return type module, str

classmethod get_plugin_class(*plugin_name*)
get the plugin class from its name

If the dotted notation is used, the string is case sensitive and the corresponding module is loaded at once, otherwise the plugin name is case insensitive and a recursive file search is done from the directory `plugin_class.plugins_basedir`

Parameters **plugin_name** (*str*) – the plugin name to find.

Returns plugin's class object or None if not found.

Return type class object or None

You can get plugin class object by giving only the class name (case insensitive):

```
plugin_class = MyProjectActivePlugin.get_plugin_class('hpproliant')
```

Or with the full dotted path (case sensitive this time):

```
plugin_class = MyProjectActivePlugin.get_plugin_class('hp.hp_proliant.
↪HpProliant')
```

If you need a plugin instance, prefer using *get_instance()*

classmethod get_plugin_desc()
Returns the plugin description. By default return the class docstring.

handle_cmd_options()
Parse command line options

The parsed options are stored in `self.options` and arguments in `self.args`

classmethod info(*msg*, **args*, ***kwargs*)
log an informational message

Parameters

- **msg** (*str*) – the message to log
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)

Examples

This logs an informational message in log file and/or console:

```
p = Plugin()
p.info('Date : %s',datetime.now())
```

init_cmd_options()
Create OptionParser instance and add some basic options

This is automatically called when the plugin is run. Avoid to override this method, prefer to customize *add_cmd_options()*

init_logger()
Initialize logging

classmethod load_data(*filename*)
Load and de-serialize data from a file

The input file must be a json file.

Parameters **filename** (*str*) – The file path to load.

Returns The restored data or `NoAttr` on error.

Return type `textops.DictExt`

Examples

```
>>> open('/tmp/mydata','w').write('''{
...   "powers": {
...     "1": "OK",
...     "2": "Degraded",
...     "3": "OK",
...     "4": "Failed"
...   },
...   "nb_disks": 36
... }''')
>>> data = ActivePlugin.load_data('/tmp/mydata')
>>> print data
{u'powers': {u'1': u'OK', u'3': u'OK', u'2': u'Degraded', u'4': u'Failed'}, u
↪'nb_disks': 36}
>>> print type(data)
<class 'textops.base.DictExt'>
```

See *save_data()* to know how `/tmp/mydata` has been generated.

logger_format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
The logging format to use

logger_logbackup = 5
Log file backup file number

logger_logsize = 1000000
Log file max size

manage_cmd_options()
Manage commande line options

OptionParser instance is created, options are added, then command line is parsed.

plugin_type = 'plugin'
For plugin search, it will search for classes having this attribute in all python files

plugins_basedir = '/path/to/your/plugins/python/modules'
For plugin search, it will search recursively from this directory

plugins_basemodule = 'plugins.python.'
For plugin search, the module prefix to add to have the module accessible from python path. Do not forget the ending dot. You can set empty if your plugin modules are in python path.

Note: Do not forget to put empty `__init__.py` file in each directory leading to your plugins.

---

classmethod save_data(*filename*, *data*, *ignore_error=True*)
Serialize and save data into a file

The data must be a dictionary where values must be simple types : str, int, float, date, list and/or dict. The data are serialized into json format.

Parameters

- **filename** (*str*) – The file path to store the data. The directories are created if not present.
- **data** (*dict*) – The dictionary to save.
- **ignore_error** (*bool*) – ignore errors if True (Default: True)

Notes

The data dictionary keys must be strings. If you specify integers, they will be replaced by a string.

Examples

```
>>> data={'powers': {1:'OK', 2:'Degraded',3:'OK', 4:'Failed'}, 'nb_disks': 36␣
↪}
>>> ActivePlugin.save_data('/tmp/mydata',data)
>>> print open('/tmp/mydata').read()
{
    "powers": {
        "1": "OK",
        "2": "Degraded",
        "3": "OK",
        "4": "Failed"
    },
    "nb_disks": 36
}
```

classmethod warning(*msg*, **args*, ***kwargs*)
log a warning message

Parameters

- **msg** (*str*) – the message to log
- **args** (*list*) – if additional arguments are given, `msg` will be formatted with `%` (old-style python string formatting)

Examples

This logs an warning message in log file and/or console:

```
p = Plugin()
p.warning('This variable is not used in line %s',36)
```

7.2 ActivePlugin

class naghelp.**ActivePlugin**(***extra_options*)

Python base class for active nagios plugins

This is the base class for developping Active Nagios plugin with the naghelp module

build_response(*data*)

Build a response

This method should be overridden when developing a new plugin. You must use data dictionary to decide what alerts and/or informations to send to Nagios. To do so, a *PluginResponse* object has already been initialized by the framework and is available at `self.response` : you just have to use *add** methods. It is highly recommended to call the super/parent `build_response()` at the end to easily take advantage of optional mixins like *naghelp.GaugeMixin*.

Parameters **data** (`textops.DictExt`) – the data dictionary where are collected and parsed data

Example

After getting a raw syslog output, we extract warnings and critical errors:

```
def build_response(self,data):
    self.response.add_list(WARNING,data.warnings)
    self.response.add_list(CRITICAL,data.criticals)
    ...
    super(MyPluginClass,self).build_response(data)
```

See *parse_data()* and *collect_data()* examples to see how data has been updated.

check_host_required_fields()

Checks host required fields

This checks the presence of values for host parameters specified in attribute *required_params*. If this attribute is empty, all parameters specified in attribute *cmd_params* will be considerated as required. The method will also check that either *name* or *ip* parameter value is present.

check_ports()

Checks port

This method is called when an error occurs while collecting data from host : It will check whether the tcp ports are reachable or not. If not, the plugin exits with a fast response.

cmd_params = ''

Attribute that must contain a list of all possible *Host* parameters for the current plugin

This will automatically add options to the `optparse.OptionParser` object. This means that the given parameters can be set at command line (use '-h' for plugin help to see them appear). This also ask naghelp

to get parameters from environment variable or an optional database if available. Once the parameters value found, naghelp will store them into the host object at the same index. For example, if `plugin.cmd_params = 'user,passwd'` then parameters values will be available at `self.host.user` and `self.host.passwd` inside the definition of *collect_data()*.

It is highly recommended to use the following parameters name as their description has been already defined by the method *get_plugin_host_params_tab()* :

NAME	DESCRIPTION
name	Hostname
ip	Host IP address
subtype	Plugin subtype (usually host model)
user	User
passwd	Password
console_ip	Console or controller IP address
snmpversion	SNMP protocal version (1,2 or 3)
community	SNMP community
community_alt	SNMP community for other device
authpp	SNMP authentification passphrase
authproto	SNMP authentification protocal (md5 or sha)
privpp	SNMP privacy passphrase
privproto	SNMP privacy protocal (des or aes)
protocol	ssh or telnet
port	Port number
collect_cmd_timeout	Maximum time allowed for one collect command
collect_all_timeout	Maximum time allowed for the whole collect process
maxwarn	Gauge max value for a warning status
maxcrit	Gauge max value for a critical status
minwarn	Gauge min value for a warning status
mincrit	Gauge min value for a critical status
options	Additionnal options

Note that `name` and `ip` are hard coded : you must use them for Nagios hostname and hostaddress. The same for `subtype`, `protocol` and `port` that are hard coded for port testing.

The parameter list can be a python list or a coma separated string.

You can force all your plugin to have some default parameters (like 'name' and 'ip') : to do so, use the plugin attribute *forced_params*.

Note: **Do not** include the parameters that are not *Host* related (like plugin debug mode flag, verbose mode flag, plugin description flag etc...). These parameters are already checked by `optparse.OptionParser` and do not need to get their value from environment variables or a database.

---

collect_data(*data*)

Collect data from monitored host

This method should be overridden when developing a new plugin. One should use *naghelp.collect* module to retrieve raw data from monitored equipment. Do not parse raw data in this method : see *parse_data()*. Note that no data is returned : one just have to modify `data` with a dotted notation.

Parameters **data** (`textops.DictExt`) – the data dictionary to write collected raw data to.

Example

Here we execute the command `dmesg` on a remote host via SSH to get last logs:

```
def collect_data(self,data):
    data.syslog = Ssh(self.host.ip,self.host.user,self.host.passwd).run('dmesg
↪')
```

See *parse_data()* example to see how data has been parsed. See *build_response()* example to see how data will be used.

collected_data_filename_pattern = '/tmp/naghelp/%s_collected_data.json'
Attribute giving the pattern for the persistent data file path. `%s` will be replaced by the monitored host name (or IP if host name not specified)

data
The place to put collected and parsed data

As data is a `textops.DictExt` object, one can use the dotted notation for reading and for writing.

default_level = OK
Attribute giving the response level to return if no level has been set.

By default, naghelp consider that if no level message has been added to the response, there is no errors and return the `OK` level to Nagios.

In some situation, one may prefer to send an `UKNOWN` state by default.

do_feature()
Run the appropriate feature depending on the options given

do_monitoring()
Run the monitoring feature of the plugin

It will take care of everything in that order :

1. Collect monitoring informations with *collect_data()*
2. Check ports if an error occured while collecting data
3. Parse collected data with *parse_data()*
4. Build a response with *build_response()*
5. Save persistent data (save the *host* object to a json file)
6. Add plugin information at the response ending
7. Send the response (render the response to stdout and exit the plugin with appropriate exit code)

doctest_begin()
For doctest usage only

doctest_end()
For doctest usage only

error(*msg*, *sublevel=3*, *exception=None*, **args*, ***kwargs*)
Log an error and exit the plugin

Not only it logs an error to console and/or log file, it also send a fast response that will exit the plugin. If the exception that has generated the error is not derived from CollectError, A stack and available data are also dumped.

Parameters

- **msg** (*str*) – Message body. Note that it adds a begin message (not a message level).
- **sublevel** (*int*) – The message sublevel (displayed into plugin information section)
- **exception** (*Exception*) – The exception that is the error's origin (Optional).

fast_response(*level*, *synopsis*, *msg=''*, *sublevel=1*)
Exit the plugin at once by sending a basic message level to Nagios

This is used mainly on errors : the goal is to avoid the plugin to go any further.

Parameters

- **level** (*ResponseLevel*) – Response level to give to Nagios
- **synopsis** (*str*) – Response title
- **msg** (*str*) – Message body. Note that it adds a begin message (not a message level).
- **sublevel** (*int*) – The message sublevel (displayed into plugin information section)

fast_response_if(*test*, *level*, *synopsis*, *msg=''*, *sublevel=1*)
If test is True, exit the plugin at once by sending a basic message level to Nagios

This works like *fast_response()* except that it exits only if test is True.

Parameters

- **test** (*bool*) – Must be True to send response and exit plugin.
- **level** (*ResponseLevel*) – Response level to give to Nagios
- **synopsis** (*str*) – Response title
- **msg** (*str*) – Message body. Note that it adds a begin message (not a message level).
- **sublevel** (*int*) – The message sublevel (displayed into plugin information section)

forced_params = 'name,ip,collect_cmd_timeout,collect_all_timeout'
Attribute you can set to force all your plugins to have some default *Host* parameters. These parameters are automatically added to the plugin attribute *cmd_params*.

Default is `'name,ip'` because all the monitored equipments must have a Nagios name and an IP.

get_plugin_host_params_desc()
Builds a dictionary giving description of plugin host parameters

This merges *cmd_params* and *forced_params* paramters and returns their description

get_plugin_host_params_tab()
Returns a dictionary of Host parameters description

This dictionary helps naghelp to build the plugin help (`-h` option in command line).

It is highly recommended to use, in the attribute *cmd_params*, only keys from this dictionary. If it is not the case, a pseudo-description will be calculated when needed.

If you want to create specific parameters, add them in the dictionary with their description by overriding this method in a subclass.

get_plugin_informations()
Get plugin informations

This method builds a text giving some informations about the plugin, it will be placed at the end of the plugin response.

Example

Here an output:

```
>>> p = ActivePlugin()
>>> print p.get_plugin_informations()

============================[ Plugin Informations
↪]=============================
Plugin name : naghelp.plugin.ActivePlugin
Description : Python base class for active nagios plugins
Execution date : ...
Execution time : ...
Ports used : tcp = none, udp = none
Exit code : 0 (OK), __sublevel__=0
```

get_tcp_ports()
Returns tcp ports

Manages `port` and `protocol` host parameters if defined

get_udp_ports()
Returns udp ports

Manages `port` host parameter if defined

handle_cmd_options()
Parse command line options

The parsed options are stored in `plugin.options` and arguments in `plugin.args`

If the user requests plugin description, it is displayed and the plugin exited with UNKOWN response level.

You should customize this method if you want to check some options before running the plugin main part.

host
This will contain the *Host* object. not that it is devrived from a dict.

host_class
alias of `naghelp.host.Host`

init_cmd_options()
Initialize command line options

This create `optparse.OptionParser` instance and add some basic options

It also add options corresponding to Host parameters. The host parameters will be stored first into OptionParse's options object (`plugin.options`), later it is set to host object.

This method is automatically called when the plugin is run. Avoid to override this method, prefer to customize `add_cmd_options()`

nagios_status_on_error = CRITICAL
Attribute giving the *ResponseLevel* to return to Nagios on error.

options
Attribute that contains the command line options as parsed by `optparse.OptionParser`

parse_data(*data*)
Parse data

This method should be overridden when developing a new plugin. When raw data are not usable at once, one should parse them to structure the informations. *parse_data()* will get the data dictionary updated

by *collect_data()*. One should then use python-textops to parse the data. There is no data to return : one just have to modify `data` with a dotted notation.

Parameters **data** (`textops.DictExt`) – the data dictionary to read collected raw data and write parsed data.

Example

After getting a raw syslog output, we extract warnings and critical errors:

```
def parse_data(self,data):
    data.warnings  = data.syslog.grepi('EMS Event Notification').grep(
↪'MAJORWARNING|SERIOUS')
    data.criticals = data.syslog.grepi('EMS Event Notification').grep(
↪'CRITICAL')
```

See *collect_data()* example to see how data has been updated. See *build_response()* example to see how data will be used.

Note: The data dictionary is the same for collected data and parsed data, so do not use already existing keys for collected data to store new parsed data.

---

plugin_type = 'active'
Attribute for the plugin type

This is used during plugin recursive search : should be the same string accross all your plugins

required_params = None
Attribute that contains the list of parameters required for the *Host* object

For example, if your plugin need to connect to a host that requires a password, you must add 'passwd' in the list. The list of possible Host parameters you can add should be keys of the dictionary returned by the method *get_plugin_host_params_tab()*.

At execution time, `naghelp` will automatically check the required parameters presence : they could come from command line option, environment variable or a database (see *naghelp.Host*).

The parameter list can be a python list or a coma separated string. If the list is `None` (by default), this means that all parameters from attribute *cmd_params* are required.

response_class
alias of `naghelp.response.PluginResponse`

restore_collected_data()
Restore collected data

During development and testing, it may boring to wait the data to be collected : The idea is to save them once, and then use them many times : This is useful when developing *parse_data()* and *build_response()*

This method is called when using `-r` option on command line.

run()
Run the plugin with options given in command line, database or plugin initial extra_options

It will take care of everything in that order :

1. Manage command line options (uses *cmd_params*)
2. Create the *Host* object (store it in attribute *host*)

3. Activate logging (if asked in command line options with `-v` or `-d`)
4. Load persistent data into *host*
5. call do_feature() method
6. save host persistent data

save_collected_data()
Save collected data

During development and testing, it may boring to wait the data to be collected : The idea is to save them once, and then use them many times : This is useful when developing *parse_data()* and *build_response()*

This method is called when using `-s` option on command line.

save_host_data()
Save data before sending the built response

tcp_ports = ''
Attribute that lists the tcp_ports used by the plugin

naghelp will check specified ports if a problem is detected while collecting data from host. naghelp also use this attribute in plugin summary to help administrator to configure their firewall.

The attribute is ignored if the plugin uses the `protocol` or `port` parameters.

The ports list can be a python list or a coma separated string.

udp_ports = ''
Attribute that lists the udp_ports used by the plugin

naghelp uses this attribute in plugin summary to help administrator to configure their firewall.

The attribute is ignored if the plugin uses the `protocol` or `port` parameters.

The ports list can be a python list or a coma separated string.

usage = 'usage: \n%prog [options]'
Attribute for the command line usage

warning(*msg*, **args*, ***kwargs*)
Log a warning and add a warning message level

Not only it logs a warning to console and/or log file, it also add a warning in response's message level section

Parameters **msg** (*str*) – The message to log and add.

- genindex
- modindex
- search

CHAPTER 8

Launcher

naghelp.launcher.**launch**(*plugin_base_class*)

Load the class specified in command line then instantiate and run it.

It will read command line first argument and instantiate the specified class with a dotted notation. It will also accept only the class name without any dot, in this case, a recursive search will be done from the directory given by `plugin_base_class.plugins_basedir` and will find the class with the right name and having the same `plugin_type` attribute value as `plugin_base_class`. the search is case insensitive on the class name. Once the plugin instance has been create, the `run()` method is executed. If you start your launcher without any parameters, it will show you all plugin classes it has discovered in `plugin_base_class.plugins_basedir` with their first line description.

Parameters **plugin_base_class** (*naghelp.ActivePlugin*) – the base class from which all your active plugins are inherited. This class must redefine attributes *plugins_basedir* and *plugin_type*.

This function has to be used in a launcher script you may want to write to start a class as a Nagios plugin, here is a an example:

```
#!/usr/bin/python
# change python interpreter if your are using virtualenv or buildout

from plugin_commons import MyProjectActivePlugin
from naghelp.launcher import launch

def main():
    launch(MyProjectActivePlugin)

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

Then you can run your plugin class like that (faster):

```
/path/to/your/launcher my_project_plugins.myplugin.MyPlugin --name=myhost --
↪user=nagiosuser --passwd=nagiospwd
```

or (slower):

```
/path/to/your/launcher myplugin --name=myhost --user=nagiosuser --passwd=nagiospwd
```

naghelp.launcher.**usage**(*plugin_base_class*, *error=''*)
Prints launcher usage and display all available plugin classes

- genindex
- modindex
- search

CHAPTER 9

Mixins

This module contains mixins to extended naghelp with some additional features

9.1 GaugeMixin

class naghelp.**GaugeMixin**

Gauge response helper Mixin

This mixin helps to build a response when one expect a value (gauge) to not move up,down or to be in range. it has be to be declared in the parent classes of a plugin class, before ActivePlugin class. Methods have to be used into *naghelp.ActivePlugin.build_response()* method. One must call the `super` *naghelp.ActivePlugin.build_response()* at the end

Example:

```
MyPluginWithGauges(GaugeMixin, ActivePlugin):
    ...
    def build_response(self,data):
        ...
        self.gauge_response_etalon_down('fan',   'Fans',   data.boards.grep('Fan␣
↪Tray').grepc('OK'), WARNING )
        self.gauge_response_etalon_down('board', 'Boards', data.boards.grep('SB|IB
↪').grepc('Available|Assigned'), WARNING )
        self.gauge_response_etalon_down('power', 'Powers', data.boards.grep('^PS
↪').grepc('OK'), WARNING )
        ...
        super(MyPluginWithGauges,self).build_response(data)
```

gauge_etalon_clear(*id*)

Clear the reference value for an "etalon" gauge

By this way, you ask the plugin to learn a new reference value

Parameters **id** (*str*) – The id of the gauge

gauge_etalon_set(*id*, *value*)
Force a reference value for an "etalon" gauge

Parameters

- **id** (*str*) – The id of the gauge
- **value** (*any*) – The value that will be the new reference value

gauge_response_etalon_change(*id*, *label*, *value*, *level*)
Remember a value, detect if it has changed

At the first call the value is stored in host persistent data. The next times, it adds a CRITICAL or WARNING response if the value has changed. The new value is stored and become the new reference value.

Parameters

- **id** (*str*) – The id of the gauge : an arbitrary string without space (aka slug). This is used for storing the value in persistent data and for debug purposes.
- **label** (*str*) – The gauge meaning. This will be used to build the response message
- **value** (*any*) – The value to test
- **level** (*naghelp.ResponseLevel*) – WARNING or CRITICAL

Example

```
>>> class MyPluginWithGauges(GaugeMixin, ActivePlugin):
...     pass
...
>>> p=MyPluginWithGauges()
>>> p.doctest_begin()                    # only for doctest
>>> p.gauge_etalon_clear('tempcursor')   # only for doctest
>>> p.gauge_response_etalon_change('tempcursor','Temperature cursor',20,
↪CRITICAL)
>>> print p.response
OK
=========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 20
>>> p.doctest_end()
>>> p=MyPluginWithGauges()
>>> p.doctest_begin()                    # only for doctest
>>> p.gauge_response_etalon_change('tempcursor','Temperature cursor',21,
↪CRITICAL)
>>> print p.response
Temperature cursor : actual value (21) has changed (was 20)
==================================[  STATUS ␣
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
Temperature cursor : actual value (21) has changed (was 20)

=========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 21
>>> p.doctest_end()                      # only for doctest
```

(continues on next page)

(continued from previous page)

```
>>> p=MyPluginWithGauges()
>>> p.doctest_begin()
>>> p.gauge_response_etalon_change('tempcursor','Temperature cursor',21,
↪CRITICAL)
>>> print p.response
OK
==========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 21
>>> p.doctest_end()
```

gauge_response_etalon_change_list(*id*, *label_values*, *level*)

Call *gauge_response_etalon_change()* for all `(label,values)` tuple in the list `label_values`

gauge_response_etalon_down(*id*, *label*, *value*, *level*)

Remember a value, detect if it has changed by going down

At the first call the value is stored in host persistent data. The next times, it adds a CRITICAL or WARNING response if the value has changed by going down. The new value is stored and become the new reference value.

Parameters

- **id** (*str*) – The id of the gauge : an arbitrary string without space (aka slug). This is used for storing the value in persistent data and for debug purposes.
- **label** (*str*) – The gauge meaning. This will be used to build the response message
- **value** (*any*) – The value to test
- **level** (*naghelp.ResponseLevel*) – WARNING or CRITICAL

Example

```
>>> class MyPluginWithGauges(GaugeMixin, ActivePlugin):
...     pass
...
>>> p=MyPluginWithGauges()                    # 1st plugin execution
>>> p.doctest_begin()                         # only for doctest
>>> p.gauge_etalon_clear('tempcursor')        # only for doctest
>>> p.gauge_response_etalon_down('tempcursor','Temperature cursor',20,
↪CRITICAL)
>>> print p.response
OK
==========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 20
>>> p.doctest_end()
>>> p=MyPluginWithGauges()                    # 2nd plugin execution
>>> p.doctest_begin()                         # only for doctest
>>> p.gauge_response_etalon_down('tempcursor','Temperature cursor',19,
↪CRITICAL)
>>> print p.response
Temperature cursor : actual value (19) is less than the reference value (20...
... )
==================================[  STATUS ␣
↪]==================================
```

(continues on next page)

(continued from previous page)

```
----( CRITICAL )--------------------------------------------------------------
↪--
Temperature cursor : actual value (19) is less than the reference value (20)

=========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 19
>>> p.doctest_end()                          # only for doctest
>>> p=MyPluginWithGauges()
>>> p.doctest_begin()                        # only for doctest
>>> p.gauge_response_etalon_down('tempcursor','Temperature cursor',19,
↪CRITICAL)
>>> print p.response
OK
=========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 19
>>> p.doctest_end()                          # only for doctest
```

gauge_response_etalon_down_list(*id*, *label_values*, *level*)

Call *gauge_response_etalon_down()* for all `(label,values)` tuple in the list `label_values`

gauge_response_etalon_up(*id*, *label*, *value*, *level*)

Remember a value, detect if it has changed by going up

At the first call the value is stored in host persistent data. The next times, it adds a CRITICAL or WARNING response if the value has changed by going up. The new value is stored and become the new reference value.

Parameters

- **id** (*str*) – The id of the gauge : an arbitrary string without space (aka slug). This is used for storing the value in persistent data and for debug purposes.
- **label** (*str*) – The gauge meaning. This will be used to build the response message
- **value** (*any*) – The value to test
- **level** (*naghelp.ResponseLevel*) – WARNING or CRITICAL

Example

```
>>> class MyPluginWithGauges(GaugeMixin, ActivePlugin):
...     pass
...
>>> p=MyPluginWithGauges()                   # 1st plugin execution
>>> p.doctest_begin()                        # only for doctest
>>> p.gauge_etalon_clear('tempcursor')       # only for doctest
>>> p.gauge_response_etalon_up('tempcursor','Temperature cursor',20,CRITICAL)
>>> print p.response
OK
=========================[ Additionnal informations␣
↪]==========================
Temperature cursor : 20
>>> p.doctest_end()
```

(continues on next page)

(continued from previous page)

```
>>> p=MyPluginWithGauges()                    # 2nd plugin execution
>>> p.doctest_begin()                         # only for doctest
>>> p.gauge_response_etalon_up('tempcursor','Temperature cursor',21,CRITICAL)
>>> print p.response
Temperature cursor : actual value (21) is more than the reference value (20...
... )
==================================[  STATUS
↪]==================================

----( CRITICAL )-------------------------------------------------------------
↪--
Temperature cursor : actual value (21) is more than the reference value (20)

=========================[ Additionnal informations
↪]=========================
Temperature cursor : 21
>>> p.doctest_end()                           # only for doctest
>>> p=MyPluginWithGauges()
>>> p.doctest_begin()                         # only for doctest
>>> p.gauge_response_etalon_up('tempcursor','Temperature cursor',21,CRITICAL)
>>> print p.response
OK
=========================[ Additionnal informations
↪]=========================
Temperature cursor : 21
>>> p.doctest_end()                           # only for doctest
```

gauge_response_etalon_up_list(*id*, *label_values*, *level*)

Call *gauge_response_etalon_up()* for all `(label,values)` tuple in the list `label_values`

gauge_response_threshold(*id*, *label*, *value*, *warn_min=None*, *crit_min=None*, *warn_max=None*, *crit_max=None*)

Test a value and add a WARNING or CRITICAL response if the value is out of range

It also add gauges value in the response's additional informations section

Parameters

- **id** (*str*) – The id of the gauge : an arbitrary string without space (aka slug). This is used for debug purposes.
- **label** (*str*) – The gauge meaning. This will be used to build the response message
- **value** (*str, int or float*) – The value to test. If the value is a string, it will detect the first numeric value.
- **warn_min** (*int or float*) – The lower threshold for WARNING response
- **crit_min** (*int or float*) – The lower threshold for CRITICAL response
- **warn_max** (*int or float*) – The upper threshold for WARNING response
- **crit_max** (*int or float*) – The upper threshold for CRITICAL response

Example

```
>>> class MyPluginWithGauges(GaugeMixin, ActivePlugin):
...     pass
```

(continues on next page)

(continued from previous page)

```
...
>>> p=MyPluginWithGauges()
>>> p.gauge_response_threshold('test','Test gauge','90',0,10,70,100)
>>> print p.response
Test gauge : 90 >= MAX WARNING (70)
==================================[  STATUS
↪]==================================

----( WARNING )-------------------------------------------------------------
↪--
Test gauge : 90 >= MAX WARNING (70)

=========================[ Additionnal informations
↪]=========================
Test gauge : 90
>>> p=MyPluginWithGauges()
>>> p.gauge_response_threshold('test','Test gauge','-10',0,10,70,100)
>>> print p.response
Test gauge : -10 <= MIN CRITICAL (10)
==================================[  STATUS
↪]==================================

----( CRITICAL )------------------------------------------------------------
↪--
Test gauge : -10 <= MIN CRITICAL (10)

=========================[ Additionnal informations
↪]=========================
Test gauge : -10
>>> p=MyPluginWithGauges()
>>> p.gauge_response_threshold('test','Test gauge','Temperature=110C',0,10,70,
↪100)
>>> print p.response
Test gauge : 110 >= MAX CRITICAL (100)
==================================[  STATUS
↪]==================================

----( CRITICAL )------------------------------------------------------------
↪--
Test gauge : 110 >= MAX CRITICAL (100)

=========================[ Additionnal informations
↪]=========================
Test gauge : Temperature=110C
```

gauge_response_threshold_list(*id*, *label_values*, *warn_min=None*, *crit_min=None*, *warn_max=None*, *crit_max=None*)

Test a list of values and add a WARNING or CRITICAL response if the value is out of range

It calls *gauge_response_threshold()* for each `(label,value)` specified in the `label_values` list.

Parameters

- **id** (*str*) – The id of the gauge : an arbitrary string without space (aka slug). This is used for debug purposes.
- **label_values** (*list*) – A list of tuple `(label,value)` where `label` is the gauge meaning and `value` is the value to test.

- **warn_min** (*int or float*) – The lower threshold for WARNING response
- **crit_min** (*int or float*) – The lower threshold for CRITICAL response
- **warn_max** (*int or float*) – The upper threshold for WARNING response
- **crit_max** (*int or float*) – The upper threshold for CRITICAL response

- genindex
- modindex
- search

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W