nagiosplugin Documentation

Release 1.3.2

Flying Circus Internet Operations GmbH

2019-11-09

Contents

This documentation covers nagiosplugin 1.3.2.

CHAPTER 1

Contents

1.1 The nagiosplugin library

1.1.1 About

nagiosplugin is a Python class library which helps writing Nagios (or Icinga) compatible plugins easily in Python. It cares for much of the boilerplate code and default logic commonly found in Nagios checks, including:

- Nagios 3 Plugin API compliant parameters and output formatting
- Full Nagios range syntax support
- Automatic threshold checking
- Multiple independend measures
- Custom status line to communicate the main point quickly
- Long output and performance data
- Timeout handling
- Persistent "cookies" to retain state information between check runs
- Resume log file processing at the point where the last run left
- No dependencies beyond the Python standard library (except for Python 2.6).

nagiosplugin runs on POSIX and Windows systems. It is compatible with Python 2.7, and Python 3.4 through 3.7.

1.1.2 Feedback and Suggestions

nagiosplugin is currently maintained by Matt Pounsett <matt@conundrum.com>. A public issue tracker can be found at <https://github.com/mpounsett/nagiosplugin/issues> for bugs, suggestions, and patches.

1.1.3 License

The nagiosplugin package is released under the Zope Public License 2.1 (ZPL), a BSD-style Open Source license.

1.1.4 Documentation

Comprehensive documentation is available online. The examples mentioned in the tutorials can also be found in the `nagiosplugin/examples` directory of the source distribution.

1.1.5 Acknowledgements

nagiosplugin was originally written and maintained by Christian Kauhaus <kc@flyingcircus.io>. Additional contributions from the community are acknowledged in the file CONTRIBUTORS.txt

1.2 First steps with nagiosplugin

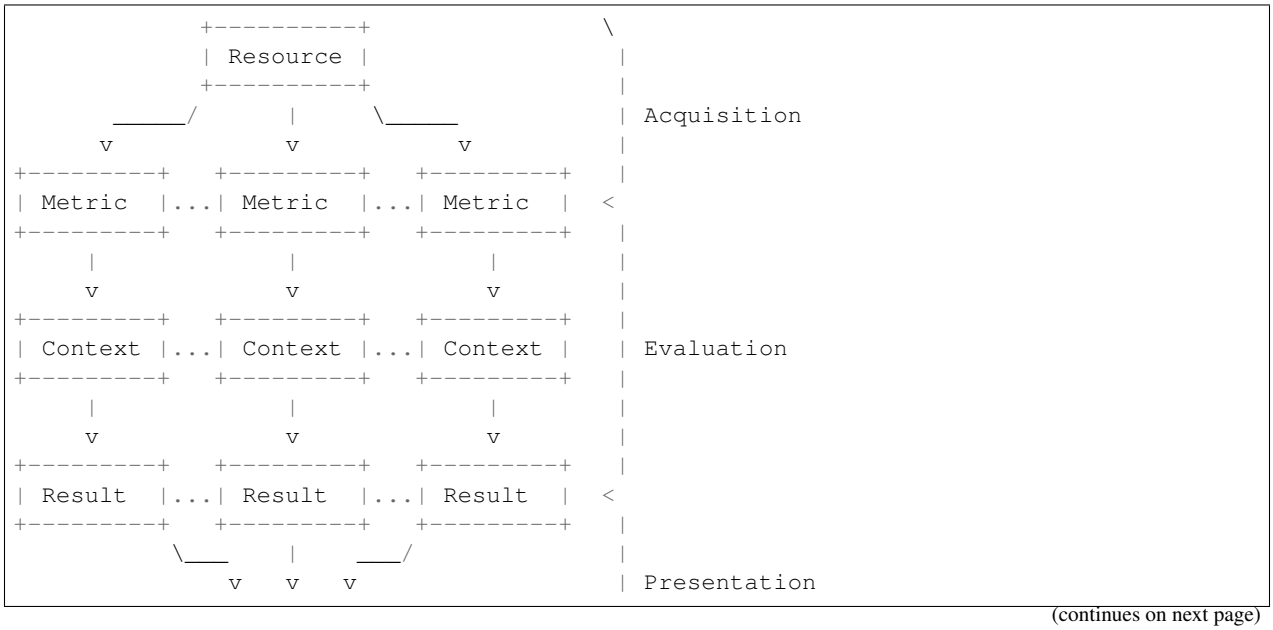
This tutorial will guide you through all important steps of writing a check with the *nagiosplugin* class library. Read this to get started.

1.2.1 Key concepts

nagiosplugin has a fine-grained class model with clear separation of concerns. This allows plugin writers to focus on one particular tasks at a time while writing plugins. Nagios/Icinga plugins need to perform three step: data *acquisition*, *evaluation*, and *presentation*. Each step has an associated class (Resource, Context, Summary) and information between tasks is passed with structured value objects (Metric, Result).

1.2.2 Classes overview

Here is a diagram with the most important classes and their relationships:



(continues on next page)

(continued from previous page)

```
                +---------+                  |
                | Summary |                  |
                +---------+                 /
```

Resource A model of the thing being monitored. It should usually have the same name as the whole plugin. Generates one or more metrics.

Example: system load

Metric A single measured data point. A metric consists of a name, a value, a unit, and optional minimum and maximum bounds. Most metrics are scalar (the value can be represented as single number).

Example: load1=0.75

Context Additional information to evaluate a metric. A context has usually a warning and critical range which allows to determine if a given metric is OK or not. Contexts also include information on how to present a metric in a human-readable way.

Example: warning=0.5, critical=1.0

Result Product of a metric and a context. A result consists of a state ("ok", "warning", "critical", "unknown"), some explanatory text, and references to the objects that it was generated from.

Example: WARNING - load1 is 0.75

Summary Condenses all results in a single status line. The status line is the plugin's most important output: it appears in mails, text messages, pager alerts etc.

Example: LOAD WARNING - load1 is 0.75 (greater than 0.5)

The following tutorials which will guide you through the most important features of *nagiosplugin*.

Hint: Study the source code in the `nagiosplugin/examples` directory for complete examples.

1.2.3 Tutorials

Tutorial #1: 'Hello world' check

In the first tutorial, we will develop `check_world`. This check will determine if the world exists. The algorithm is simple: if the world would not exist, the check would not execute.

This minimalistic check consists of a `Resource` World which models the part of the world that is interesting for the purposes of our check. Resource classes must define a `Resource.probe()` method which returns a list of metrics. We just return a single `Metric` object that states that the world exists.

```
#!python

"""Hello world Nagios check."""

import nagiosplugin


class World(nagiosplugin.Resource):

    def probe(self):
        return [nagiosplugin.Metric('world', True, context='null')]
```

(continues on next page)

(continued from previous page)

```
def main():
    check = nagiosplugin.Check(World())
    check.main()


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

We don't have a context to evaluate the returned metric yet, so we resort to the built-in "null" context. The "null" context does nothing with its associated metrics.

We now create a `Check` object that is fed only with the resource object. We could put context and summary objects into the `Check()` constructor as well. This will be demonstrated in the next tutorial. There is also no command line processing nor timeout handling nor output control. We call the `Check.main()` method to evaluate resources, construct text output and exit with the appropriate status code.

Running the plugin creates very simple output:

```
$ check_world.py
WORLD OK
```

The plugin's exit status is 0, signalling success to the calling process.

Tutorial #2: check_load

In this tutorial, we will discuss important basic features that are present in nearly every check. These include command line processing, metric evaluation with scalar contexts, status line formatting and logging.

The **`check_load`** plugin resembles the one found in the standard Nagios plugins collection. It allows to check the system load average against thresholds.

Data acquisition

First, we will subclass `Resource` to generate metrics for the 1, 5, and 15 minute load averages.

```
class Load(nagiosplugin.Resource):
    """Domain model: system load.

    Determines the system load parameters and (optionally) cpu count.
    The `probe` method returns the three standard load average numbers.
    If `percpu` is true, the load average will be normalized.

    This check requires Linux-style /proc files to be present.
    """

    def __init__(self, percpu=False):
        self.percpu = percpu

    def cpus(self):
        _log.info('counting cpus with "nproc"')
        cpus = int(subprocess.check_output(['nproc']))
```

(continues on next page)

(continued from previous page)

```
        _log.debug('found %i cpus in total', cpus)
        return cpus

    def probe(self):
        _log.info('reading load from /proc/loadavg')
        with open('/proc/loadavg') as loadavg:
            load = loadavg.readline().split()[0:3]
        _log.debug('raw load is %s', load)
        cpus = self.cpus() if self.percpu else 1
        load = [float(l) / cpus for l in load]
        for i, period in enumerate([1, 5, 15]):
            yield nagiosplugin.Metric('load%d' % period, load[i], min=0,
                                      context='load')
```

check_load has two modes of operation: the load averages may either be takes as read from the kernel or normalized by cpu. Accordingly, the `Load()` constructor has a parameter two switch normalization on.

In the `Load.probe()` method the check reads the load averages from the `/proc` filesystem and extracts the interesting values. For each value, a `Metric` object is returned. Each metric has a generated name ("load1", "load5", "load15") and a value. We don't declare a unit of measure since load averages come without unit. All metrics will share the same context "load" which means that the thresholds for all three values will be the same.

Note: Deriving the number of CPUs from `/proc` is a little bit messy and deserves an extra method. Resource classes may encapsulate arbitrary complex measurement logic as long they define a `Resource.probe()` method that returns a list of metrics. In the code example shown above, we sprinkle some logging statements which show effects when the check is called with an increased logging level (discussed below).

Evaluation

The **check_load** plugin should accept warning and critical ranges and determine if any load value is outside these ranges. Since this kind of logic is pretty standard for most of all Nagios/Icinga plugins, *nagiosplugin* provides a generalized context class for it. It is the `ScalarContext` class which accepts a warning and a critical range as well as a template to present metric values in a human-readable way.

When `ScalarContext` is sufficient, it may be configured during instantiation right in the `main` function. A first version of the `main` function looks like this:

```
def main():
    argp = argparse.ArgumentParser(description=__doc__)
    argp.add_argument('-w', '--warning', metavar='RANGE', default='',
                      help='return warning if load is outside RANGE')
    argp.add_argument('-c', '--critical', metavar='RANGE', default='',
                      help='return critical if load is outside RANGE')
    argp.add_argument('-r', '--percpu', action='store_true', default=False)
    args = argp.parse_args()
    check = nagiosplugin.Check(
        Load(args.percpu),
        nagiosplugin.ScalarContext('load', args.warning, args.critical))
    check.main()
```

Note that the context name "load" is referenced by all three metrics returned by the `Load.probe` method.

This version of **check_load** is already functional:

```
$ ./check_load.py
LOAD OK - load1 is 0.11
| load15=0.21;;;0 load1=0.11;;;0 load5=0.18;;;0

$ ./check_load.py -c 0.1:0.2
LOAD CRITICAL - load15 is 0.22 (outside 0.1:0.2)
| load15=0.22;;0.1:0.2;0 load1=0.11;;0.1:0.2;0 load5=0.2;;0.1:0.2;0
# exit status 2

$ ./check_load.py -c 0.1:0.2 -r
LOAD OK - load1 is 0.105
| load15=0.11;;0.1:0.2;0 load1=0.105;;0.1:0.2;0 load5=0.1;;0.1:0.2;0
```

In the first invocation (lines 1–3), **check_load** reports only the first load value which looks bit arbitrary. In the second invocation (lines 5–8), we set a critical threshold. The range specification is parsed automatically according to the *Nagios plugin API* and the first metric that lies outside is reported. In the third invocation (lines 10–12), we request normalization and all values fit in the range this time.

Result presentation

Although we now have a running check, the output is not as informative as it could be. The first line of output (status line) is very important since the information presented therein should give the admin a clue what is going on. We want the first line to display:

- a load overview when there is nothing wrong
- which load value violates a threshold, if applicable
- which threshold is being violated, if applicable.

The last two points are already covered by the `Result` default implementation, but we need to tweak the summary to display a load overview as stated in the first point:

```
class LoadSummary(nagiosplugin.Summary):
    """Status line conveying load information.

    We specialize the `ok` method to present all three figures in one
    handy tagline. In case of problems, the single-load texts from the
    contexts work well.
    """

    def __init__(self, percpu):
        self.percpu = percpu

    def ok(self, results):
        qualifier = 'per cpu ' if self.percpu else ''
        return 'loadavg %sis %s' % (qualifier, ', '.join(
            str(results[r].metric) for r in ['load1', 'load5', 'load15']))
```

The `Summary` class has three methods which can be specialized: `ok()` to return a status line when there are no problems, `problem()` to return a status line when the overall check status indicates problems, and `verbose()` to generate additional output. All three methods get a set of `Result` objects passed in. In our code, the `ok` method

queries uses the original metrics referenced by the result objects to build an overview like "loadavg is 0.19, 0.16, 0.14".

Check setup

The last step in this tutorial is to put the pieces together:

```
@nagiosplugin.guarded
def main():
    argp = argparse.ArgumentParser(description=__doc__)
    argp.add_argument('-w', '--warning', metavar='RANGE', default='',
                      help='return warning if load is outside RANGE')
    argp.add_argument('-c', '--critical', metavar='RANGE', default='',
                      help='return critical if load is outside RANGE')
    argp.add_argument('-r', '--percpu', action='store_true', default=False)
    argp.add_argument('-v', '--verbose', action='count', default=0,
                      help='increase output verbosity (use up to 3 times)')
    args = argp.parse_args()
    check = nagiosplugin.Check(
        Load(args.percpu),
        nagiosplugin.ScalarContext('load', args.warning, args.critical),
        LoadSummary(args.percpu))
    check.main(verbose=args.verbose)


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

In the `main()` function we parse the command line parameters using the standard `argparse.ArgumentParser` class. Watch the `Check` object creation: its constructor can be fed with a variable number of `Resource`, `Context`, and `Summary` objects. In this tutorial, instances of our specialized `Load` and `LoadSummary` classes go in.

We did not specialize a `Context` class to evaluate the load metrics. Instead, we use the supplied `ScalarContext` which compares a scalar value against two ranges according to the range syntax defined by the Nagios plugin API. The default `ScalarContext` implementation covers the majority of evaluation needs. Checks using non-scalar metrics or requiring special logic should subclass `Context` to fit their needs.

The check's `main()` method runs the check, prints the check's output including summary, log messages and *performance data* to *stdout* and exits the plugin with the appropriate exit code.

Note the `guarded()` decorator in front of the main function. It helps the code part outside `Check` to behave: in case of uncaught exceptions, it ensures that the exit code is **3** (unknown) and that the exception string is properly formatted. Additionally, logging is set up at an early stage so that even messages logged from constructors are captured and printed at the right place in the output (between status line and performance data).

Tutorial #3: check_users

In the third tutorial, we will learn how to process multiple metrics. Additionally, we will see how to use logging and verbosity levels.

Multiple metrics

A plugin can perform several measurements at once. This is often necessary to perform more complex state evaluations or improve latency. Consider a check that determines both the number of total logged in users and the number of unique

logged in users.

A Resource implementation could look like this:

```
class Users(nagiosplugin.Resource):

    def __init__(self):
        self.users = []
        self.unique_users = set()

    def list_users(self):
        """Return logged in users as list of user names."""
        [...]
        return users

    def probe(self):
        """Return both total and unique user count."""
        self.users = self.list_users()
        self.unique_users = set(self.users)
        return [nagiosplugin.Metric('total', len(self.users), min=0,
                                    context='users'),
                nagiosplugin.Metric('unique', len(self.unique_users), min=0,
                                    context='users')]
```

The `probe()` method returns a list containing two metric objects. Alternatively, the `probe()` method can act as generator and yield metrics:

```
def probe(self):
    """Return both total and unique user count."""
    self.users = self.list_users()
    self.unique_users = set(self.users)
    yield nagiosplugin.Metric('total', len(self.users), min=0,
                              context='users')
    yield nagiosplugin.Metric('unique', len(self.unique_users), min=0,
                              context='users')]
```

This may be more comfortable than constructing a list of metrics first and returning them all at once.

To assign a `Context` to a `Metric`, pass the context's name in the metric's **context** parameter. Both metrics use the same context "users". This way, the main function must define only one context that applies the same thresholds to both metrics:

```
@nagiosplugin.guarded
def main():
    argp = argparse.ArgumentParser()
    [...]
    args = argp.parse_args()
    check = nagiosplugin.Check(
        Users(),
        nagiosplugin.ScalarContext('users', args.warning, args.critical,
                                   fmt_metric='{value} users logged in'))
    check.main()
```

Multiple contexts

The above example defines only one context for all metrics. This may not be practical. Each metric should get its own context now. By default, a metric is matched by a context of the same name. So we just leave out the **context**

parameters:

```
def probe(self):
    [...]
    return [nagiosplugin.Metric('total', len(self.users), min=0),
            nagiosplugin.Metric('unique', len(self.unique_users), min=0)]
```

We then define two contexts (one for each metric) in the `main()` function:

```
@nagiosplugin.guarded
def main():
    [...]
    args = argp.parse_args()
    check = nagiosplugin.Check(
        Users(),
        nagiosplugin.ScalarContext('total', args.warning, args.critical,
                                   fmt_metric='{value} users logged in'),
        nagiosplugin.ScalarContext(
            'unique', args.warning_unique, args.critical_unique,
            fmt_metric='{value} unique users logged in'))
    check.main(args.verbose, args.timeout)
```

Alternatively, we can require every context that fits in metric definitions.

Logging and verbosity levels

nagiosplugin integrates with the logging module from Python's standard library. If the main function is decorated with `guarded` (which is heavily recommended), the logging module gets automatically configured before the execution of the `main()` function starts. Messages logged to the *nagiosplugin* logger (or any sublogger) are processed with nagiosplugin's integrated logging.

Consider the following example check:

```
import argparse
import nagiosplugin
import logging

_log = logging.getLogger('nagiosplugin')


class Logging(nagiosplugin.Resource):

    def probe(self):
        _log.warning('warning message')
        _log.info('info message')
        _log.debug('debug message')
        return [nagiosplugin.Metric('zero', 0, context='default')]


@nagiosplugin.guarded
def main():
    argp = argparse.ArgumentParser()
    argp.add_argument('-v', '--verbose', action='count', default=0)
    args = argp.parse_args()
    check = nagiosplugin.Check(Logging())
    check.main(args.verbose)
```

(continues on next page)

(continued from previous page)

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

The verbosity level is set in the `check.main()` invocation depending on the number of "-v" flags. Let's test this check:

```
$ check_verbose.py
LOGGING OK - zero is 0 | zero=0
warning message (check_verbose.py:11)
$ check_verbose.py -v
LOGGING OK - zero is 0
warning message (check_verbose.py:11)
| zero=0
$ check_verbose.py -vv
LOGGING OK - zero is 0
warning message (check_verbose.py:11)
info message (check_verbose.py:12)
| zero=0
$ check_verbose.py -vvv
LOGGING OK - zero is 0
warning message (check_verbose.py:11)
info message (check_verbose.py:12)
debug message (check_verbose.py:13)
| zero=0
```

When called with *verbose=0,* both the summary and the performance data are printed on one line and the warning message is displayed. Messages logged with *warning* or *error* level are always printed. Setting *verbose* to 1 does not change the logging level but enable multi-line output. Additionally, full tracebacks would be printed in the case of an uncaught exception. Verbosity levels of 2 and 3 enable logging with *info* or *debug* levels.

This behaviour conforms to the "Verbose output" suggestions found in the Nagios plug-in development guidelines.

The initial verbosity level is 1 (multi-line output). This means that tracebacks are printed for uncaught exceptions raised in the initialization phase (before `Check.main()` is called). This is generally a good thing. To suppress tracebacks during initialization, call `guarded()` with an optional `verbose` parameter. Example:

```
@nagiosplugin.guarded(verbose=0)
def main():
    [...]
```

Note: The initial verbosity level takes effect only until `Check.main()` is called with a different verbosity level.

It is advisable to sprinkle logging statements in the plugin code, especially into the resource model classes. A logging example for a users check could look like this:

```
class Users(nagiosplugin.Resource):

    [...]

    def list_users(self):
        """Return list of logged in users."""
        _log.info('querying users with "%s" command', self.who_cmd)
        users = []
        try:
            for line in subprocess.check_output([self.who_cmd]).splitlines():
```

(continues on next page)

(continued from previous page)

```
                _log.debug('who output: %s', line.strip())
                users.append(line.split()[0].decode())
        except OSError:
            raise nagiosplugin.CheckError(
                'cannot determine number of users ({} failed)'.format(
                    self.who_cmd))
        _log.debug('found users: %r', users)
        return users
```

Interesting items to log are: the command which is invoked to query the information from the system, or the raw result to verify that parsing works correctly.

1.3 Topic Guides

Topic guides are meant as explanatory tests which expand on one specific area of the library. Expect more to come here.

1.3.1 Plugin Debugging

Debugging plugins can sometimes be complicated since there are so many classes, which are tied together in an implicit way. I have collected some frequent questions about debugging.

An uncaught exception makes the plugin return UNKNOWN. Where is the cause?

When your plugin raises an exception, you may get very little output. Example:

```
$ check_users.py
USERS UNKNOWN: RuntimeError: error
```

Set the **verbose** parameter of `main()` to some value greater than zero and you will get the full traceback:

```
$ check_users.py -v
USERS UNKNOWN: RuntimeError: error
Traceback (most recent call last):
  File "nagiosplugin/runtime.py", line 38, in wrapper
    return func(*args, **kwds)
  File "nagiosplugin/examples/check_users.py", line 104, in main
    check.main(args.verbose, args.timeout)
  File "nagiosplugin/check.py", line 110, in main
    runtime.execute(self, verbose, timeout)
  File "nagiosplugin/runtime.py", line 118, in execute
    with_timeout(self.timeout, self.run, check)
  File "nagiosplugin/platform/posix.py", line 19, in with_timeout
    func(*args, **kwargs)
  File "nagiosplugin/runtime.py", line 107, in run
    check()
  File "nagiosplugin/check.py", line 95, in __call__
    self._evaluate_resource(resource)
  File "nagiosplugin/check.py", line 73, in _evaluate_resource
    metrics = resource.probe()
  File "nagiosplugin/examples/check_users.py", line 57, in probe
    self.users = self.list_users()
```

(continues on next page)

(continued from previous page)

```
  File "nagiosplugin/examples/check_users.py", line 34, in list_users
    raise RuntimeError('error')
RuntimeError: error
```

A Check constructor dies with "cannot add type <...>"

When you see the following exception raised from `Check()` (or `Check.add()`):

```
UNKNOWN: TypeError: ("cannot add type <class '__main__.Users'> to check", <__main__.
↪Users object at 0x7f0c64f73f90>)
```

chances are high that you are trying to add an object that is not an instance from Resource, Context, Summary, or Results or its subclasses. A common error is to base a resource class on `object` instead of `Resource`.

I'm trying to use pdb but I get a timeout after 10s

When using an interactive debugger like pdb on plugins, you may experience that your debugging session is aborted with a timeout after 10 seconds. Just set the **timeout** parameter in `main()` to 0 to avoid this.

1.4 API docs

The *nagiosplugin* module consists of several submodules which are discussed in detail as follows. Refer to the "*First steps with nagiosplugin*" section for an introduction on how to use them for typical plugins.

1.4.1 Core API

The core API consists of all functions and classes which are called in a plugin's `main` function. A typical main function is decorated with `guarded()` and creates a `Check` object. The check instance is fed with instances of `Resource`, `Context`, or `Summary` (respective custom subclasses). Finally, control is passed to the check's `main()` method.

Note: All classes that plugin authors typically need are imported into the *nagiosplugin* name space. For example, use

```
import nagiosplugin
# ...
check = nagiosplugin.Check()
```

to get a `Check` instance.

---

nagiosplugin.check

Example: Skeleton main function

The following pseudo code outlines how `Check` is typically used in the main function of a plugin:

```
def main():
    check = nagiosplugin.Check(MyResource1(...), MyResource2(...),
                               MyContext1(...), MyContext2(...),
                               MySummary(...))
    check.main()
```

nagiosplugin.resource

nagiosplugin.context

Example ScalarContext usage

Configure a ScalarContext with warning and critical ranges found in ArgumentParser's result object `args` and add it to a check:

```
c = Check(..., ScalarContext('metric', args.warning, args.critical), ...)
```

nagiosplugin.summary

nagiosplugin.runtime

1.4.2 Intermediate data API

The following classes allow to handle intermediate data that are used during the plugin's execution in a structured way. Most of them are used by the *nagiosplugin* library itself to create objects which are passed into code written by plugin authors. Other classes (like `Metric`) are used by plugin authors to generate intermediate data during *acquisition* or *evaluation* steps.

Note: All classes that plugin authors typically need are imported directly into the *nagiosplugin* name space. For example, use

```
import nagiosplugin
# ...
result = nagiosplugin.Result(nagiosplugin.Ok)
```

to get a `Result` instance.

---

nagiosplugin.metric

nagiosplugin.state

Note: `ServiceState` is not imported into the *nagiosplugin* top-level name space since there is usually no need to access it directly.

State subclasses

The state subclasses are singletons. Plugin authors should use the class name (without parentheses) to access the instance. For example:

```
state = nagiosplugin.Critical
```

nagiosplugin.performance

nagiosplugin.range

nagiosplugin.result

1.4.3 Auxiliary Classes

nagiosplugin's auxiliary classes are not strictly required to write checks, but simplify common tasks and provide convenient access to functionality that is regularly needed by plugin authors.

Note: All classes that plugin authors typically need are imported directly into the *nagiosplugin* name space. For example, use

```
import nagiosplugin
# ...
with nagiosplugin.Cookie(path) as cookie:
    # ...
```

to get a cookie.

---

nagiosplugin.cookie

Cookie example

Increment a connection count saved in the cookie by `self.new_conns`:

```
with nagiosplugin.Cookie(self.statefile) as cookie:
    cookie['connections'] = cookie.get('connections', 0) + self.new_conns
```

Note that the new content is committed automatically when exiting the `with` block.

nagiosplugin.logtail

LogTail example

Calls `process()` for each new line in a log file:

```
cookie = nagiosplugin.Cookie(self.statefile)
with nagiosplugin.LogTail(self.logfile, cookie) as newlines:
    for line in newlines:
        process(line.decode())
```

1.5 Glossary

acquisition First step of check execution in the context of the nagiosplugin library. Data is retrieved from the system under surveillance using custom code. This is where the meat of a plugin is. Data acquisition is performed by one or more *domain model* objects which are usually `Resource` subclasses.

domain model One or more classes that abstract the properties of the system under surveillance that are relevant for the check. The domain model code should not be interspersed with secondary aspects like data representation or interfacing with outside monitoring infrastructure.

evaluation Second step of check execution in the context of the nagiosplugin library. Data generated in the *acquisition* step is evaluated according to criteria specified in `Context` objects.

Nagios plugin API Documents that define how a Nagios/Icinga compatible plugin must be called and how it should respond. There is a main document and an appendix for Nagios 3 extensions.

perfdata See *performance data*.

performance data Part of the plugin output which is passed to external programs by Nagios.

presentation Third step of check execution in the context of the nagiosplugin library. Outcomes from the *evaluation* step are condensed into a compact summary which is suited to inform the admin about relevant system state. Data presentation is the responsibility of `Summary` objects which also generate the *performance data* output section.

range String notation defined in the *Nagios plugin API* to express a set of acceptable values. Values outside a range trigger a warning or critical condition.

unit of measure Property of a metric which is returned in *Performance Data* and is used for example as axis label in performance graphs. Nagios plugins should only use base units like *s*, *B*, etc. instead of scaled units like *days*, *MiB* etc.

uom See *Unit of Measure*.

1.6 Release History

1.6.1 1.3.2 (2019-11-09)

- Include `doc` and `tests` directories in source distribution to support Gentoo package building tests (#22)
- Update official python support to 2.7, 3.4+ in README

1.6.2 1.3.1 (2019-11-08)

- Fixed a packaging bug

1.6.3 1.3.0 (2019-11-08)

- New maintainer/contributor information and project home
- Updated tests and package metadata for recent Python 3 versions

- Newer tooling for tests/documentation

1.6.4 1.2.4 (2016-03-12)

- Add optional keyword parameter `verbose` to Runtime.guarded(). This parameter allows to set verbose level in the early execution phase (#13).
- Allow Context.evaluate() return either a Result or ServiceState object. In case the latter is returned, it gets automatically wrapped in a Result object (#6).

1.6.5 1.2.3 (2015-10-30)

- Fix bug that caused a UnicodeDecodeError when using non-ASCII characters in fmt_metric (#12).
- Print perfdata always on a single line (even in multi-line mode) to improve compatibility with various monitoring systems (#11).

1.6.6 1.2.2 (2014-05-27)

- Mention that nagiosplugin also runs with Python 3.4 (no code changes necessary).
- Make name prefix in status output optional by allowing to assign None to Check.name.
- Accept bare metric as return value from Resource.probe().
- Fix bug where Context.describe() was not used to obtain metric description (#13162).

1.6.7 1.2.1 (2014-03-19)

- Fix build failures with LANG=C (#13140).
- Remove length limitation of perfdata labels (#13214).
- Fix formatting of large integers as Metric values (#13287).
- Range: allow simple numerals as argument to Range() (#12658).
- Cookie: allow for empty state file specification (#12788).

1.6.8 1.2 (2013-11-08)

- New `Summary.empty` method is called if there are no results present (#11593).
- Improve range violation wording (#11597).
- Ensure that nagiosplugin install correctly with current setuptools (#12660).
- Behave and do not attach anything to the root logger.
- Add debugging topic guide. Explain how to disable the timeout when using pdb (#11592).

1.6.9 1.1 (2013-06-19)

- Identical to 1.1b1.

1.6.10 1.1b1 (2013-05-28)

- Made compatible with Python 2.6 (#12297).
- Tutorial #3: check_users (#11539).
- Minor documentation improvements.

1.6.11 1.0.0 (2013-02-05)

- LogTail returns lines as byte strings in Python 3 to avoid codec issues (#11564).
- LogTail gives a line-based iterator instead of a file object (#11564).
- Basic API docs for the most important classes (#11612).
- Made compatible with Python 2.7 (#11533).
- Made compatible with Python 3.3.

1.6.12 1.0.0b1 (2012-10-29)

- Improve error reporting for missing contexts.
- Exit with code 3 if no metrics have been generated.
- Improve default Summary.verbose() to list all threshold violations.
- Move main source repository to https://bitbucket.org/gocept/nagiosplugin/ (#11561).

1.6.13 1.0.0a2 (2012-10-26)

- API docs for the most important classes (#7939).
- Added two tutorials (#9425).
- Fix packaging issues.

1.6.14 1.0.0a1 (2012-10-25)

- Completely reworked API. The new API is not compatible with the old 0.4 API so you must update your plugins.
- Python 3 support.
- The `Cookie` class is now basically a persistent dict and accepts key/value pairs. Cookie are stored as JSON files by default so they can be inspected by the system administrator (#9400).
- New `LogTail` class which provides convenient access to constantly growing log files which are eventually rotated.

1.6.15 0.4.5 (2012-06-18)

- Windows port. *nagiosplugin* code now runs under pywin32 (#10899).
- Include examples in egg release (#9901).

1.6.16 0.4.4 (2011-07-18)

Bugfix release to fix issues reported by users.

- Improve Mac OS X compatibility (#8755).
- Include examples in distribution (#8555).

1.6.17 0.4.3 (2010-12-17)

- Change __str__ representation of large numbers to avoid scientific notation.

1.6.18 0.4.2 (2010-10-11)

- Packaging issues.

1.6.19 0.4.1 (2010-09-21)

- Fix distribution to install correctly.
- Documentation: tutorial and topic guides.

1.6.20 0.4 (2010-08-17)

- Initial public release.

CHAPTER 2

Indices and tables

- genindex
- search

To download the package, see the PyPI page.

Python Module Index

n

Index