pynag Documentation

Release 0.9.0

Pall Sigurdsson and Tomas Edwardsson

July 23, 2014

Contents

Release 0.9.0

Date July 23, 2014

CHAPTER 1

Introduction

1.1 About pynag

Pynag is a all around python interface to Nagios and bretheren (Icinga, Naemon and Shinken) as well as providing a command line interface to them for managing them.

CHAPTER 2

The pynag module

2.1 pynag Package

2.2 Subpackages

2.2.1 Control Package

Control Package

The Control module includes classes to control the Nagios service and the Command submodule wraps Nagios commands.

class pynag.Control.**daemon**(*nagios_bin='/usr/bin/nagios'*, *nagios_cfg='/etc/nagios/nagios.cfg'*, *nagios_init=None*, *sudo=True*, *shell=None*, *service_name='nagios'*, *nagios_config=None*)

Bases: object

Control the nagios daemon through python

```
>>> from pynag.Control import daemon
>>>
>>> d = daemon()
>>> d.restart()
```

SYSTEMD = 3

SYSV_INIT_SCRIPT = 1

SYSV_INIT_SERVICE = 2

reload()

Reloads Nagios.

Returns Return code of the reload command ran by pynag.Utils.runCommand()

Return type int

restart()

Restarts Nagios via it's init script.

Returns Return code of the restart command ran by pynag.Utils.runCommand()

Return type int

`running``()`
Checks if the daemon is running

Returns Whether or not the daemon is running

Return type bool

`start``()`
Start the Nagios service.

Returns Return code of the start command ran by pynag.Utils.runCommand()

Return type int

`status``()`
Obtain the status of the Nagios service.

Returns Return code of the status command ran by pynag.Utils.runCommand()

Return type int

`stop``()`
Stop the Nagios service.

Returns Return code of the stop command ran by pynag.Utils.runCommand()

Return type int

`systemd_service_path` **= '/usr/lib/systemd/system'**

`verify_config``()`
Run nagios -v config_file to verify that the conf is working

Returns True – if pynag.Utils.runCommand() returns 0, else None

Subpackages

Command Package

`Command` Package The Command module is capable of sending commands to Nagios via the configured communication path.

`pynag.Control.Command.`**`acknowledge_host_problem`**`(`*host_name*, *sticky*, *notify*, *persistent*, *author*, *comment*, *command_file=None*, *timestamp=0*`)`
Allows you to acknowledge the current problem for the specified host. By acknowledging the current problem, future notifications (for the same host state) are disabled. If the "sticky" option is set to two (2), the acknowledgement will remain until the host returns to an UP state. Otherwise the acknowledgement will automatically be removed when the host changes state. If the "notify" option is set to one (1), a notification will be sent out to contacts indicating that the current host problem has been acknowledged. If the "persistent" option is set to one (1), the comment associated with the acknowledgement will survive across restarts of the Nagios process. If not, the comment will be deleted the next time Nagios restarts.

`pynag.Control.Command.`**`acknowledge_svc_problem`**`(`*host_name*, *service_description*, *sticky*, *notify*, *persistent*, *author*, *comment*, *command_file=None*, *timestamp=0*`)`
Allows you to acknowledge the current problem for the specified service. By acknowledging the current problem, future notifications (for the same servicestate) are disabled. If the "sticky" option is set to two (2), the acknowledgement will remain until the service returns to an OK state. Otherwise the acknowledgement will automatically be removed when the service changes state. If the "notify" option is set to one (1), a notification

will be sent out to contacts indicating that the current service problem has been acknowledged. If the "persistent" option is set to one (1), the comment associated with the acknowledgement will survive across restarts of the Nagios process. If not, the comment will be deleted the next time Nagios restarts.

`pynag.Control.Command.`**`add_host_comment`**(*host_name*, *persistent*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Adds a comment to a particular host. If the "persistent" field is set to zero (0), the comment will be deleted the next time Nagios is restarted. Otherwise, the comment will persist across program restarts until it is deleted manually.

`pynag.Control.Command.`**`add_svc_comment`**(*host_name*, *service_description*, *persistent*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Adds a comment to a particular service. If the "persistent" field is set to zero (0), the comment will be deleted the next time Nagios is restarted. Otherwise, the comment will persist across program restarts until it is deleted manually.

`pynag.Control.Command.`**`change_contact_host_notification_timeperiod`**(*contact_name*, *notification_timeperiod*, *command_file=None*, *timestamp=0*)

Changes the host notification timeperiod for a particular contact to what is specified by the "notification_timeperiod" option. The "notification_timeperiod" option should be the short name of the timeperiod that is to be used as the contact's host notification timeperiod. The timeperiod must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_contact_modattr`**(*contact_name*, *value*, *command_file=None*, *timestamp=0*)

This command changes the modified attributes value for the specified contact. Modified attributes values are used by Nagios to determine which object properties should be retained across program restarts. Thus, modifying the value of the attributes can affect data retention. This is an advanced option and should only be used by people who are intimately familiar with the data retention logic in Nagios.

`pynag.Control.Command.`**`change_contact_modhattr`**(*contact_name*, *value*, *command_file=None*, *timestamp=0*)

This command changes the modified host attributes value for the specified contact. Modified attributes values are used by Nagios to determine which object properties should be retained across program restarts. Thus, modifying the value of the attributes can affect data retention. This is an advanced option and should only be used by people who are intimately familiar with the data retention logic in Nagios.

`pynag.Control.Command.`**`change_contact_modsattr`**(*contact_name*, *value*, *command_file=None*, *timestamp=0*)

This command changes the modified service attributes value for the specified contact. Modified attributes values are used by Nagios to determine which object properties should be retained across program restarts. Thus, modifying the value of the attributes can affect data retention. This is an advanced option and should only be used by people who are intimately familiar with the data retention logic in Nagios.

`pynag.Control.Command.`**`change_contact_svc_notification_timeperiod`**(*contact_name*, *notification_timeperiod*, *command_file=None*, *timestamp=0*)

Changes the service notification timeperiod for a particular contact to what is specified by the "notification_timeperiod" option. The "notification_timeperiod" option should be the short name of the timeperiod that is to be used as the contact's service notification timeperiod. The timeperiod must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_custom_contact_var`**(*contact_name*, *varname*, *varvalue*, *command_file=None*, *timestamp=0*)
Changes the value of a custom contact variable.

`pynag.Control.Command.`**`change_custom_host_var`**(*host_name*, *varname*, *varvalue*, *command_file=None*, *timestamp=0*)
Changes the value of a custom host variable.

`pynag.Control.Command.`**`change_custom_svc_var`**(*host_name*, *service_description*, *varname*, *varvalue*, *command_file=None*, *timestamp=0*)
Changes the value of a custom service variable.

`pynag.Control.Command.`**`change_global_host_event_handler`**(*event_handler_command*, *command_file=None*, *timestamp=0*)
Changes the global host event handler command to be that specified by the "event_handler_command" option. The "event_handler_command" option specifies the short name of the command that should be used as the new host event handler. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_global_svc_event_handler`**(*event_handler_command*, *command_file=None*, *timestamp=0*)
Changes the global service event handler command to be that specified by the "event_handler_command" option. The "event_handler_command" option specifies the short name of the command that should be used as the new service event handler. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_host_check_command`**(*host_name*, *check_command*, *command_file=None*, *timestamp=0*)
Changes the check command for a particular host to be that specified by the "check_command" option. The "check_command" option specifies the short name of the command that should be used as the new host check command. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_host_check_timeperiod`**(*host_name*, *timeperiod*, *command_file=None*, *timestamp=0*)
Changes the valid check period for the specified host.

`pynag.Control.Command.`**`change_host_event_handler`**(*host_name*, *event_handler_command*, *command_file=None*, *timestamp=0*)
Changes the event handler command for a particular host to be that specified by the "event_handler_command" option. The "event_handler_command" option specifies the short name of the command that should be used as the new host event handler. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_host_modattr`**(*host_name*, *value*, *command_file=None*, *timestamp=0*)
This command changes the modified attributes value for the specified host. Modified attributes values are used by Nagios to determine which object properties should be retained across program restarts. Thus, modifying the value of the attributes can affect data retention. This is an advanced option and should only be used by people who are intimately familiar with the data retention logic in Nagios.

`pynag.Control.Command.`**`change_max_host_check_attempts`**(*host_name*, *check_attempts*, *command_file=None*, *timestamp=0*)
Changes the maximum number of check attempts (retries) for a particular host.

`pynag.Control.Command.`**`change_max_svc_check_attempts`**(*host_name*, *service_description*, *check_attempts*, *command_file=None*, *timestamp=0*)

Changes the maximum number of check attempts (retries) for a particular service.

`pynag.Control.Command.`**`change_normal_host_check_interval`**(*host_name*, *check_interval*, *command_file=None*, *timestamp=0*)

Changes the normal (regularly scheduled) check interval for a particular host.

`pynag.Control.Command.`**`change_normal_svc_check_interval`**(*host_name*, *service_description*, *check_interval*, *command_file=None*, *timestamp=0*)

Changes the normal (regularly scheduled) check interval for a particular service

`pynag.Control.Command.`**`change_retry_host_check_interval`**(*host_name*, *service_description*, *check_interval*, *command_file=None*, *timestamp=0*)

Changes the retry check interval for a particular host.

`pynag.Control.Command.`**`change_retry_svc_check_interval`**(*host_name*, *service_description*, *check_interval*, *command_file=None*, *timestamp=0*)

Changes the retry check interval for a particular service.

`pynag.Control.Command.`**`change_svc_check_command`**(*host_name*, *service_description*, *check_command*, *command_file=None*, *timestamp=0*)

Changes the check command for a particular service to be that specified by the “check_command” option. The “check_command” option specifies the short name of the command that should be used as the new service check command. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_svc_check_timeperiod`**(*host_name*, *service_description*, *check_timeperiod*, *command_file=None*, *timestamp=0*)

Changes the check timeperiod for a particular service to what is specified by the “check_timeperiod” option. The “check_timeperiod” option should be the short name of the timeperod that is to be used as the service check timeperiod. The timeperiod must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_svc_event_handler`**(*host_name*, *service_description*, *event_handler_command*, *command_file=None*, *timestamp=0*)

Changes the event handler command for a particular service to be that specified by the “event_handler_command” option. The “event_handler_command” option specifies the short name of the command that should be used as the new service event handler. The command must have been configured in Nagios before it was last (re)started.

`pynag.Control.Command.`**`change_svc_modattr`**(*host_name*, *service_description*, *value*, *command_file=None*, *timestamp=0*)

This command changes the modified attributes value for the specified service. Modified attributes values are used by Nagios to determine which object properties should be retained across program restarts. Thus, modifying the value of the attributes can affect data retention. This is an advanced option and should only be used by people who are intimately familiar with the data retention logic in Nagios.

pynag.Control.Command.**change_svc_notification_timeperiod**(*host_name*, *service_description*, *notification_timeperiod*, *command_file=None*, *timestamp=0*)

Changes the notification timeperiod for a particular service to what is specified by the "notification_timeperiod" option. The "notification_timeperiod" option should be the short name of the timeperiod that is to be used as the service notification timeperiod. The timeperiod must have been configured in Nagios before it was last (re)started.

pynag.Control.Command.**del_all_host_comments**(*host_name*, *command_file=None*, *timestamp=0*)

Deletes all comments assocated with a particular host.

pynag.Control.Command.**del_all_svc_comments**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Deletes all comments associated with a particular service.

pynag.Control.Command.**del_host_comment**(*comment_id*, *command_file=None*, *timestamp=0*)

Deletes a host comment. The id number of the comment that is to be deleted must be specified.

pynag.Control.Command.**del_host_downtime**(*downtime_id*, *command_file=None*, *timestamp=0*)

Deletes the host downtime entry that has an ID number matching the "downtime_id" argument. If the downtime is currently in effect, the host will come out of scheduled downtime (as long as there are no other overlapping active downtime entries).

pynag.Control.Command.**del_svc_comment**(*comment_id*, *command_file=None*, *timestamp=0*)

Deletes a service comment. The id number of the comment that is to be deleted must be specified.

pynag.Control.Command.**del_svc_downtime**(*downtime_id*, *command_file=None*, *timestamp=0*)

Deletes the service downtime entry that has an ID number matching the "downtime_id" argument. If the downtime is currently in effect, the service will come out of scheduled downtime (as long as there are no other overlapping active downtime entries).

pynag.Control.Command.**delay_host_notification**(*host_name*, *notification_time*, *command_file=None*, *timestamp=0*)

Delays the next notification for a parciular service until "notification_time". The "notification_time" argument is specified in time_t format (seconds since the UNIX epoch). Note that this will only have an affect if the service stays in the same problem state that it is currently in. If the service changes to another state, a new notification may go out before the time you specify in the "notification_time" argument.

pynag.Control.Command.**delay_svc_notification**(*host_name*, *service_description*, *notification_time*, *command_file=None*, *timestamp=0*)

Delays the next notification for a parciular service until "notification_time". The "notification_time" argument is specified in time_t format (seconds since the UNIX epoch). Note that this will only have an affect if the service stays in the same problem state that it is currently in. If the service changes to another state, a new notification may go out before the time you specify in the "notification_time" argument.

pynag.Control.Command.**disable_all_notifications_beyond_host**(*host_name*, *command_file=None*, *timestamp=0*)

Disables notifications for all hosts and services "beyond" (e.g. on all child hosts of) the specified host. The current notification setting for the specified host is not affected.

pynag.Control.Command.**disable_contact_host_notifications**(*contact_name*, *command_file=None*, *timestamp=0*)

Disables host notifications for a particular contact.

`pynag.Control.Command.`**`disable_contact_svc_notifications`**(*contact_name*, *command_file=None*, *timestamp=0*)
    Disables service notifications for a particular contact.

`pynag.Control.Command.`**`disable_contactgroup_host_notifications`**(*contactgroup_name*, *command_file=None*, *timestamp=0*)
    Disables host notifications for all contacts in a particular contactgroup.

`pynag.Control.Command.`**`disable_contactgroup_svc_notifications`**(*contactgroup_name*, *command_file=None*, *timestamp=0*)
    Disables service notifications for all contacts in a particular contactgroup.

`pynag.Control.Command.`**`disable_event_handlers`**(*command_file=None*, *timestamp=0*)
    Disables host and service event handlers on a program-wide basis.

`pynag.Control.Command.`**`disable_failure_prediction`**(*command_file=None*, *timestamp=0*)
    Disables failure prediction on a program-wide basis. This feature is not currently implemented in Nagios.

`pynag.Control.Command.`**`disable_flap_detection`**(*command_file=None*, *timestamp=0*)
    Disables host and service flap detection on a program-wide basis.

`pynag.Control.Command.`**`disable_host_and_child_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables notifications for the specified host, as well as all hosts "beyond" (e.g. on all child hosts of) the specified host.

`pynag.Control.Command.`**`disable_host_check`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables (regularly scheduled and on-demand) active checks of the specified host.

`pynag.Control.Command.`**`disable_host_event_handler`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables the event handler for the specified host.

`pynag.Control.Command.`**`disable_host_flap_detection`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables flap detection for the specified host.

`pynag.Control.Command.`**`disable_host_freshness_checks`**(*command_file=None*, *timestamp=0*)
    Disables freshness checks of all hosts on a program-wide basis.

`pynag.Control.Command.`**`disable_host_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables notifications for a particular host.

`pynag.Control.Command.`**`disable_host_svc_checks`**(*host_name*, *command_file=None*, *timestamp=0*)
    Enables active checks of all services on the specified host.

`pynag.Control.Command.`**`disable_host_svc_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
    Disables notifications for all services on the specified host.

`pynag.Control.Command.`**`disable_hostgroup_host_checks`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
    Disables active checks for all hosts in a particular hostgroup.

`pynag.Control.Command.`**`disable_hostgroup_host_notifications`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)

Disables notifications for all hosts in a particular hostgroup. This does not disable notifications for the services associated with the hosts in the hostgroup - see the DISABLE_HOSTGROUP_SVC_NOTIFICATIONS command for that.

`pynag.Control.Command.`**`disable_hostgroup_passive_host_checks`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)

Disables passive checks for all hosts in a particular hostgroup.

`pynag.Control.Command.`**`disable_hostgroup_passive_svc_checks`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)

Disables passive checks for all services associated with hosts in a particular hostgroup.

`pynag.Control.Command.`**`disable_hostgroup_svc_checks`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)

Disables active checks for all services associated with hosts in a particular hostgroup.

`pynag.Control.Command.`**`disable_hostgroup_svc_notifications`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)

Disables notifications for all services associated with hosts in a particular hostgroup. This does not disable notifications for the hosts in the hostgroup - see the DISABLE_HOSTGROUP_HOST_NOTIFICATIONS command for that.

`pynag.Control.Command.`**`disable_notifications`**(*command_file=None*, *timestamp=0*)

Disables host and service notifications on a program-wide basis.

`pynag.Control.Command.`**`disable_passive_host_checks`**(*host_name*, *command_file=None*, *timestamp=0*)

Disables acceptance and processing of passive host checks for the specified host.

`pynag.Control.Command.`**`disable_passive_svc_checks`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Disables passive checks for the specified service.

`pynag.Control.Command.`**`disable_performance_data`**(*command_file=None*, *timestamp=0*)

Disables the processing of host and service performance data on a program-wide basis.

`pynag.Control.Command.`**`disable_service_flap_detection`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Disables flap detection for the specified service.

`pynag.Control.Command.`**`disable_service_freshness_checks`**(*command_file=None*, *timestamp=0*)

Disables freshness checks of all services on a program-wide basis.

`pynag.Control.Command.`**`disable_servicegroup_host_checks`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Disables active checks for all hosts that have services that are members of a particular hostgroup.

`pynag.Control.Command.`**`disable_servicegroup_host_notifications`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Disables notifications for all hosts that have services that are members of a particular servicegroup.

`pynag.Control.Command.`**`disable_servicegroup_passive_host_checks`**`(`*servicegroup_name, command_file=None, timestamp=0*`)`

Disables the acceptance and processing of passive checks for all hosts that have services that are members of a particular service group.

`pynag.Control.Command.`**`disable_servicegroup_passive_svc_checks`**`(`*servicegroup_name, command_file=None, timestamp=0*`)`

Disables the acceptance and processing of passive checks for all services in a particular servicegroup.

`pynag.Control.Command.`**`disable_servicegroup_svc_checks`**`(`*servicegroup_name, command_file=None, timestamp=0*`)`

Disables active checks for all services in a particular servicegroup.

`pynag.Control.Command.`**`disable_servicegroup_svc_notifications`**`(`*servicegroup_name, command_file=None, timestamp=0*`)`

Disables notifications for all services that are members of a particular servicegroup.

`pynag.Control.Command.`**`disable_svc_check`**`(`*host_name, service_description, command_file=None, timestamp=0*`)`

Disables active checks for a particular service.

`pynag.Control.Command.`**`disable_svc_event_handler`**`(`*host_name, service_description, command_file=None, timestamp=0*`)`

Disables the event handler for the specified service.

`pynag.Control.Command.`**`disable_svc_flap_detection`**`(`*host_name, service_description, command_file=None, timestamp=0*`)`

Disables flap detection for the specified service.

`pynag.Control.Command.`**`disable_svc_notifications`**`(`*host_name, service_description, command_file=None, timestamp=0*`)`

Disables notifications for a particular service.

`pynag.Control.Command.`**`enable_all_notifications_beyond_host`**`(`*host_name, command_file=None, timestamp=0*`)`

Enables notifications for all hosts and services "beyond" (e.g. on all child hosts of) the specified host. The current notification setting for the specified host is not affected. Notifications will only be sent out for these hosts and services if notifications are also enabled on a program-wide basis.

`pynag.Control.Command.`**`enable_contact_host_notifications`**`(`*contact_name, command_file=None, timestamp=0*`)`

Enables host notifications for a particular contact.

`pynag.Control.Command.`**`enable_contact_svc_notifications`**`(`*contact_name, command_file=None, timestamp=0*`)`

Disables service notifications for a particular contact.

`pynag.Control.Command.`**`enable_contactgroup_host_notifications`**(*contactgroup_name*, *command_file=None*, *timestamp=0*)
Enables host notifications for all contacts in a particular contactgroup.

`pynag.Control.Command.`**`enable_contactgroup_svc_notifications`**(*contactgroup_name*, *command_file=None*, *timestamp=0*)
Enables service notifications for all contacts in a particular contactgroup.

`pynag.Control.Command.`**`enable_event_handlers`**(*command_file=None*, *timestamp=0*)
Enables host and service event handlers on a program-wide basis.

`pynag.Control.Command.`**`enable_failure_prediction`**(*command_file=None*, *timestamp=0*)
Enables failure prediction on a program-wide basis. This feature is not currently implemented in Nagios.

`pynag.Control.Command.`**`enable_flap_detection`**(*command_file=None*, *timestamp=0*)
Enables host and service flap detection on a program-wide basis.

`pynag.Control.Command.`**`enable_host_and_child_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables notifications for the specified host, as well as all hosts "beyond" (e.g. on all child hosts of) the specified host. Notifications will only be sent out for these hosts if notifications are also enabled on a program-wide basis.

`pynag.Control.Command.`**`enable_host_check`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables (regularly scheduled and on-demand) active checks of the specified host.

`pynag.Control.Command.`**`enable_host_event_handler`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables the event handler for the specified host.

`pynag.Control.Command.`**`enable_host_flap_detection`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables flap detection for the specified host. In order for the flap detection algorithms to be run for the host, flap detection must be enabled on a program-wide basis as well.

`pynag.Control.Command.`**`enable_host_freshness_checks`**(*command_file=None*, *timestamp=0*)
Enables freshness checks of all hosts on a program-wide basis. Individual hosts that have freshness checks disabled will not be checked for freshness.

`pynag.Control.Command.`**`enable_host_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables notifications for a particular host. Notifications will be sent out for the host only if notifications are enabled on a program-wide basis as well.

`pynag.Control.Command.`**`enable_host_svc_checks`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables active checks of all services on the specified host.

`pynag.Control.Command.`**`enable_host_svc_notifications`**(*host_name*, *command_file=None*, *timestamp=0*)
Enables notifications for all services on the specified host. Note that notifications will not be sent out if notifications are disabled on a program-wide basis.

`pynag.Control.Command.`**`enable_hostgroup_host_checks`**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables active checks for all hosts in a particular hostgroup.

pynag.Control.Command.**enable_hostgroup_host_notifications**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables notifications for all hosts in a particular hostgroup. This does not enable notifications for the services associated with the hosts in the hostgroup - see the ENABLE_HOSTGROUP_SVC_NOTIFICATIONS command for that. In order for notifications to be sent out for these hosts, notifications must be enabled on a program-wide basis as well.

pynag.Control.Command.**enable_hostgroup_passive_host_checks**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables passive checks for all hosts in a particular hostgroup.

pynag.Control.Command.**enable_hostgroup_passive_svc_checks**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables passive checks for all services associated with hosts in a particular hostgroup.

pynag.Control.Command.**enable_hostgroup_svc_checks**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables active checks for all services associated with hosts in a particular hostgroup.

pynag.Control.Command.**enable_hostgroup_svc_notifications**(*hostgroup_name*, *command_file=None*, *timestamp=0*)
Enables notifications for all services that are associated with hosts in a particular hostgroup. This does not enable notifications for the hosts in the hostgroup - see the ENABLE_HOSTGROUP_HOST_NOTIFICATIONS command for that. In order for notifications to be sent out for these services, notifications must be enabled on a program-wide basis as well.

pynag.Control.Command.**enable_notifications**(*command_file=None*, *timestamp=0*)
Enables host and service notifications on a program-wide basis.

pynag.Control.Command.**enable_passive_host_checks**(*host_name*, *command_file=None*, *timestamp=0*)
Enables acceptance and processing of passive host checks for the specified host.

pynag.Control.Command.**enable_passive_svc_checks**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)
Enables passive checks for the specified service.

pynag.Control.Command.**enable_performance_data**(*command_file=None*, *timestamp=0*)
Enables the processing of host and service performance data on a program-wide basis.

pynag.Control.Command.**enable_service_freshness_checks**(*command_file=None*, *timestamp=0*)
Enables freshness checks of all services on a program-wide basis. Individual services that have freshness checks disabled will not be checked for freshness.

pynag.Control.Command.**enable_servicegroup_host_checks**(*servicegroup_name*, *command_file=None*, *timestamp=0*)
Enables active checks for all hosts that have services that are members of a particular hostgroup.

pynag.Control.Command.**enable_servicegroup_host_notifications**(*servicegroup_name*, *command_file=None*, *timestamp=0*)
Enables notifications for all hosts that have services that are members of a particular servicegroup. In order for notifications to be sent out for these hosts, notifications must also be enabled on a program-wide basis.

`pynag.Control.Command.`**`enable_servicegroup_passive_host_checks`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Enables the acceptance and processing of passive checks for all hosts that have services that are members of a particular service group.

`pynag.Control.Command.`**`enable_servicegroup_passive_svc_checks`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Enables the acceptance and processing of passive checks for all services in a particular servicegroup.

`pynag.Control.Command.`**`enable_servicegroup_svc_checks`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Enables active checks for all services in a particular servicegroup.

`pynag.Control.Command.`**`enable_servicegroup_svc_notifications`**(*servicegroup_name*, *command_file=None*, *timestamp=0*)

Enables notifications for all services that are members of a particular servicegroup. In order for notifications to be sent out for these services, notifications must also be enabled on a program-wide basis.

`pynag.Control.Command.`**`enable_svc_check`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Enables active checks for a particular service.

`pynag.Control.Command.`**`enable_svc_event_handler`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Enables the event handler for the specified service.

`pynag.Control.Command.`**`enable_svc_flap_detection`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Enables flap detection for the specified service. In order for the flap detection algorithms to be run for the service, flap detection must be enabled on a program-wide basis as well.

`pynag.Control.Command.`**`enable_svc_notifications`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

Enables notifications for a particular service. Notifications will be sent out for the service only if notifications are enabled on a program-wide basis as well.

`pynag.Control.Command.`**`find_command_file`**(*cfg_file=None*)

Returns path to nagios command_file by looking at what is defined in nagios.cfg

Args: cfg_file (str): Path to nagios.cfg configuration file

Returns: str. Path to the nagios command file

Raises: PynagError

`pynag.Control.Command.`**`process_file`**(*file_name*, *delete*, *command_file=None*, *timestamp=0*)

Directs Nagios to process all external commands that are found in the file specified by the <file_name> argument. If the <delete> option is non-zero, the file will be deleted once it has been processes. If the <delete> option is set to zero, the file is left untouched.

`pynag.Control.Command.`**`process_host_check_result`**(*host_name*, *status_code*, *plugin_output*, *command_file=None*, *timestamp=0*)

This is used to submit a passive check result for a particular host. The “status_code” indicates the state of the host check and should be one of the following: 0=UP, 1=DOWN, 2=UNREACHABLE. The "plugin_output" argument contains the text returned from the host check, along with optional performance data.

`pynag.Control.Command.`**`process_service_check_result`**(*host_name*, *service_description*, *return_code*, *plugin_output*, *command_file=None*, *timestamp=0*)

This is used to submit a passive check result for a particular service. The "return_code" field should be one of the following: 0=OK, 1=WARNING, 2=CRITICAL, 3=UNKNOWN. The "plugin_output" field contains text output from the service check, along with optional performance data.

`pynag.Control.Command.`**`read_state_information`**(*command_file=None*, *timestamp=0*)

Causes Nagios to load all current monitoring status information from the state retention file. Normally, state retention information is loaded when the Nagios process starts up and before it starts monitoring. WARNING: This command will cause Nagios to discard all current monitoring status information and use the information stored in state retention file! Use with care.

`pynag.Control.Command.`**`remove_host_acknowledgement`**(*host_name*, *command_file=None*, *timestamp=0*)

This removes the problem acknowledgement for a particular host. Once the acknowledgement has been removed, notifications can once again be sent out for the given host.

`pynag.Control.Command.`**`remove_svc_acknowledgement`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)

This removes the problem acknowledgement for a particular service. Once the acknowledgement has been removed, notifications can once again be sent out for the given service.

`pynag.Control.Command.`**`restart_program`**(*command_file=None*, *timestamp=0*)

Restarts the Nagios process.

`pynag.Control.Command.`**`save_state_information`**(*command_file=None*, *timestamp=0*)

Causes Nagios to save all current monitoring status information to the state retention file. Normally, state retention information is saved before the Nagios process shuts down and (potentially) at regularly scheduled intervals. This command allows you to force Nagios to save this information to the state retention file immediately. This does not affect the current status information in the Nagios process.

`pynag.Control.Command.`**`schedule_and_propagate_host_downtime`**(*host_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for a specified host and all of its children (hosts). If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The specified (parent) host downtime can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the specified (parent) host should not be triggered by another downtime entry.

pynag.Control.Command.**schedule_and_propagate_triggered_host_downtime**(*host_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for a specified host and all of its children (hosts). If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). Downtime for child hosts are all set to be triggered by the downtime for the specified (parent) host. The specified (parent) host downtime can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the specified (parent) host should not be triggered by another downtime entry.

pynag.Control.Command.**schedule_forced_host_check**(*host_name*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules a forced active check of a particular host at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Forced checks are performed regardless of what time it is (e.g. timeperiod restrictions are ignored) and whether or not active checks are enabled on a host-specific or program-wide basis.

pynag.Control.Command.**schedule_forced_host_svc_checks**(*host_name*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules a forced active check of all services associated with a particular host at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Forced checks are performed regardless of what time it is (e.g. timeperiod restrictions are ignored) and whether or not active checks are enabled on a service-specific or program-wide basis.

pynag.Control.Command.**schedule_forced_svc_check**(*host_name*, *service_description*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules a forced active check of a particular service at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Forced checks are performed regardless of what time it is (e.g. timeperiod restrictions are ignored) and whether or not active checks are enabled on a service-specific or program-wide basis.

pynag.Control.Command.**schedule_host_check**(*host_name*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules the next active check of a particular host at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Note that the host may not actually be checked at the time you specify. This could occur for a number of reasons: active checks are disabled on a program-wide or service-specific basis, the host is already scheduled to be checked at an earlier time, etc. If you want to force the host check to occur at the time you specify, look at the SCHEDULE_FORCED_HOST_CHECK command.

pynag.Control.Command.**schedule_host_downtime**(*host_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for a specified host. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The specified host downtime can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the specified host should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_host_svc_checks`**(*host_name*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules the next active check of all services on a particular host at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Note that the services may not actually be checked at the time you specify. This could occur for a number of reasons: active checks are disabled on a program-wide or service-specific basis, the services are already scheduled to be checked at an earlier time, etc. If you want to force the service checks to occur at the time you specify, look at the SCHEDULE_FORCED_HOST_SVC_CHECKS command.

`pynag.Control.Command.`**`schedule_host_svc_downtime`**(*host_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for all services associated with a particular host. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The service downtime entries can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the services should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_hostgroup_host_downtime`**(*hostgroup_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for all hosts in a specified hostgroup. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The host downtime entries can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the hosts should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_hostgroup_svc_downtime`**(*hostgroup_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for all services associated with hosts in a specified servicegroup. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The service downtime entries can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the services should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_servicegroup_host_downtime`**(*servicegroup_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for all hosts that have services in a specified servicegroup. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The host downtime entries can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the hosts should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_servicegroup_svc_downtime`**(*servicegroup_name*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for all services in a specified servicegroup. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The service downtime entries can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the services should not be triggered by another downtime entry.

`pynag.Control.Command.`**`schedule_svc_check`**(*host_name*, *service_description*, *check_time*, *command_file=None*, *timestamp=0*)

Schedules the next active check of a specified service at "check_time". The "check_time" argument is specified in time_t format (seconds since the UNIX epoch). Note that the service may not actually be checked at the time you specify. This could occur for a number of reasons: active checks are disabled on a program-wide or service-specific basis, the service is already scheduled to be checked at an earlier time, etc. If you want to force the service check to occur at the time you specify, look at the SCHEDULE_FORCED_SVC_CHECK command.

`pynag.Control.Command.`**`schedule_svc_downtime`**(*host_name*, *service_description*, *start_time*, *end_time*, *fixed*, *trigger_id*, *duration*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Schedules downtime for a specified service. If the "fixed" argument is set to one (1), downtime will start and end at the times specified by the "start" and "end" arguments. Otherwise, downtime will begin between the "start" and "end" times and last for "duration" seconds. The "start" and "end" arguments are specified in time_t format (seconds since the UNIX epoch). The specified service downtime can be triggered by another downtime entry if the "trigger_id" is set to the ID of another scheduled downtime entry. Set the "trigger_id" argument to zero (0) if the downtime for the specified service should not be triggered by another downtime entry.

`pynag.Control.Command.`**`send_command`**(*command_id*, *command_file=None*, *timestamp=0*, **args*)

Send one specific command to the command pipe

Args: command_id (str): Identifier string of the nagios command Eg: `ADD_SVC_COMMENT`

command_file (str): Path to nagios command file.

timestamp (int): Timestamp in time_t format of the time when the external command was sent to the command file. If 0 of None, it will be set to time.time(). Default 0.

args: Command arguments.

`pynag.Control.Command.`**`send_custom_host_notification`**(*host_name*, *options*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Allows you to send a custom host notification. Very useful in dire situations, emergencies or to communicate with all admins that are responsible for a particular host. When the host notification is sent out, the $NOTIFICATIONTYPE$ macro will be set to "CUSTOM". The <options> field is a logical OR of the following integer values that affect aspects of the notification that are sent out: 0 = No option (default), 1 = Broadcast (send notification to all normal and all escalated contacts for the host), 2 = Forced (notification is sent out regardless of current time, whether or not notifications are enabled, etc.), 4 = Increment current notification # for the host (this is not done by default for custom notifications). The comment field can be used with the $NOTIFICATIONCOMMENT$ macro in notification commands.

`pynag.Control.Command.`**`send_custom_svc_notification`**(*host_name*, *service_description*, *options*, *author*, *comment*, *command_file=None*, *timestamp=0*)

Allows you to send a custom service notification. Very useful in dire situations, emergencies or to communicate with all admins that are responsible for a particular service. When the service notification is sent out, the $NOTIFICATIONTYPE$ macro will be set to "CUSTOM". The <options> field is a logical OR of the following integer values that affect aspects of the notification that are sent out: 0 = No option (default), 1 = Broadcast (send notification to all normal and all escalated contacts for the service), 2 = Forced (notification is sent out regardless of current time, whether or not notifications are enabled, etc.), 4 = Increment current notification # for the service(this is not done by default for custom notifications). The comment field can be used with the $NOTIFICATIONCOMMENT$ macro in notification commands.

`pynag.Control.Command.`**`set_host_notification_number`**(*host_name*, *notification_number*, *command_file=None*, *timestamp=0*)

Sets the current notification number for a particular host. A value of 0 indicates that no notification has yet been sent for the current host problem. Useful for forcing an escalation (based on notification number) or replicating notification information in redundant monitoring environments. Notification numbers greater than zero have no noticeable affect on the notification process if the host is currently in an UP state.

`pynag.Control.Command.`**`set_svc_notification_number`**(*host_name*, *service_description*, *notification_number*, *command_file=None*, *timestamp=0*)

Sets the current notification number for a particular service. A value of 0 indicates that no notification has yet been sent for the current service problem. Useful for forcing an escalation (based on notification number) or replicating notification information in redundant monitoring environments. Notification numbers greater than zero have no noticeable affect on the notification process if the service is currently in an OK state.

`pynag.Control.Command.`**`shutdown_program`**(*command_file=None*, *timestamp=0*)

Shuts down the Nagios process.

`pynag.Control.Command.`**`start_accepting_passive_host_checks`**(*command_file=None*, *timestamp=0*)

Enables acceptance and processing of passive host checks on a program-wide basis.

`pynag.Control.Command.`**`start_accepting_passive_svc_checks`**(*command_file=None*, *timestamp=0*)

Enables passive service checks on a program-wide basis.

`pynag.Control.Command.`**`start_executing_host_checks`**(*command_file=None*, *timestamp=0*)

Enables active host checks on a program-wide basis.

`pynag.Control.Command.`**`start_executing_svc_checks`**(*command_file=None*, *timestamp=0*)

Enables active checks of services on a program-wide basis.

`pynag.Control.Command.`**`start_obsessing_over_host`**(*host_name*, *command_file=None*, *timestamp=0*)

Enables processing of host checks via the OCHP command for the specified host.

`pynag.Control.Command.`**`start_obsessing_over_host_checks`**(*command_file=None*, *timestamp=0*)
Enables processing of host checks via the OCHP command on a program-wide basis.

`pynag.Control.Command.`**`start_obsessing_over_svc`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)
Enables processing of service checks via the OCSP command for the specified service.

`pynag.Control.Command.`**`start_obsessing_over_svc_checks`**(*command_file=None*, *timestamp=0*)
Enables processing of service checks via the OCSP command on a program-wide basis.

`pynag.Control.Command.`**`stop_accepting_passive_host_checks`**(*command_file=None*, *timestamp=0*)
Disables acceptance and processing of passive host checks on a program-wide basis.

`pynag.Control.Command.`**`stop_accepting_passive_svc_checks`**(*command_file=None*, *timestamp=0*)
Disables passive service checks on a program-wide basis.

`pynag.Control.Command.`**`stop_executing_host_checks`**(*command_file=None*, *timestamp=0*)
Disables active host checks on a program-wide basis.

`pynag.Control.Command.`**`stop_executing_svc_checks`**(*command_file=None*, *timestamp=0*)
Disables active checks of services on a program-wide basis.

`pynag.Control.Command.`**`stop_obsessing_over_host`**(*host_name*, *command_file=None*, *timestamp=0*)
Disables processing of host checks via the OCHP command for the specified host.

`pynag.Control.Command.`**`stop_obsessing_over_host_checks`**(*command_file=None*, *timestamp=0*)
Disables processing of host checks via the OCHP command on a program-wide basis.

`pynag.Control.Command.`**`stop_obsessing_over_svc`**(*host_name*, *service_description*, *command_file=None*, *timestamp=0*)
Disables processing of service checks via the OCSP command for the specified service.

`pynag.Control.Command.`**`stop_obsessing_over_svc_checks`**(*command_file=None*, *timestamp=0*)
Disables processing of service checks via the OCSP command on a program-wide basis.

2.2.2 Model Package

`Model` Package

This module provides a high level Object-Oriented wrapper around pynag.Parsers.config.

Example:

```
>>> from pynag.Model import Service, Host
>>>
>>> all_services = Service.objects.all
>>> my_service = all_services[0]
>>> print my_service.host_name
localhost
>>>
>>> example_host = Host.objects.filter(host_name="host.example.com")
>>> canadian_hosts = Host.objects.filter(host_name__endswith=".ca")
```

```
>>>
>>> for i in canadian_hosts:
...     i.alias = "this host is located in Canada"
...     i.save()
```

class `pynag.Model.`**`Command`**(*item=None*, *filename=None*, ***kwargs*)

Bases: `pynag.Model.ObjectDefinition`

`command_line`
This is the %s attribute for object definition

`command_name`
This is the %s attribute for object definition

`object_type` **= 'command'**

`objects` **= <pynag.Model.ObjectFetcher object at 0x7fb0e557b990>**

`rename`(*shortname*)
Rename this command, and reconfigure all related objects

class `pynag.Model.`**`Contact`**(*item=None*, *filename=None*, ***kwargs*)

Bases: `pynag.Model.ObjectDefinition`

`add_to_contactgroup`(*contactgroup*)

`address`
This is the %s attribute for object definition

`alias`
This is the %s attribute for object definition

`can_submit_commands`
This is the %s attribute for object definition

`contact_name`
This is the %s attribute for object definition

`contactgroups`
This is the %s attribute for object definition

`delete`(*recursive=False*, *cleanup_related_items=True*)
Delete this contact and optionally remove references in groups and escalations

Works like ObjectDefinition.delete() except:

Arguments: cleanup_related_items – If True, remove all references to this contact in contactgroups and escalations recursive – If True, remove escalations/dependencies that rely on this (and only this) contact

`email`
This is the %s attribute for object definition

`get_effective_contactgroups`()
Get a list of all Contactgroup that are hooked to this contact

`get_effective_hosts`()
Get a list of all Host that are hooked to this Contact

`get_effective_services`()
Get a list of all Service that are hooked to this Contact

`host_notification_commands`
This is the %s attribute for object definition

`host_notification_options`
This is the %s attribute for object definition

`host_notification_period`
This is the %s attribute for object definition

`host_notifications_enabled`
This is the %s attribute for object definition

`object_type` = 'contact'

`objects` = <pynag.Model.ObjectFetcher object at 0x7fb0e593f2d0>

`pager`
This is the %s attribute for object definition

`remove_from_contactgroup`(*contactgroup*)

`rename`(*shortname*)
Renames this object, and triggers a change in related items as well.

Args: shortname: New name for this object

Returns: None

`retain_nonstatus_information`
This is the %s attribute for object definition

`retain_status_information`
This is the %s attribute for object definition

`service_notification_commands`
This is the %s attribute for object definition

`service_notification_options`
This is the %s attribute for object definition

`service_notification_period`
This is the %s attribute for object definition

`service_notifications_enabled`
This is the %s attribute for object definition

class `pynag.Model.`**`Contactgroup`**(*item=None*, *filename=None*, ***kwargs*)
Bases: `pynag.Model.ObjectDefinition`

`add_contact`(*contact_name*)
Adds one specific contact to this contactgroup.

`alias`
This is the %s attribute for object definition

`contactgroup_members`
This is the %s attribute for object definition

`contactgroup_name`
This is the %s attribute for object definition

`delete`(*recursive=False*, *cleanup_related_items=True*)
Delete this contactgroup and optionally remove references in hosts/services

Works like ObjectDefinition.delete() except:

Arguments: cleanup_related_items – If True, remove all references to this group in hosts,services,etc.
recursive – If True, remove dependant escalations.

`get_effective_contactgroups``()`
Returns a list of every Contactgroup that is a member of this Contactgroup

`get_effective_contacts``()`
Returns a list of every Contact that is a member of this Contactgroup

`get_effective_hosts``()`
Return every Host that belongs to this contactgroup

`get_effective_services``()`
Return every Host that belongs to this contactgroup

`members`
This is the %s attribute for object definition

`object_type` **= 'contactgroup'**

`objects` **= <pynag.Model.ObjectFetcher object at 0x7fb0e557ba10>**

`remove_contact``(`*contact_name*`)`
Remove one specific contact from this contactgroup

`rename``(`*shortname*`)`
Renames this object, and triggers a change in related items as well.

Args: shortname: New name for this object

Returns: None

class `pynag.Model.`**`Host`**`(`*item=None*, *filename=None*, ***kwargs*`)`
Bases: `pynag.Model.ObjectDefinition`

`2d_coords`
This is the %s attribute for object definition

`3d_coords`
This is the %s attribute for object definition

`acknowledge``(`*sticky=1*, *notify=1*, *persistent=0*, *author='pynag'*, *comment='acknowledged by pynag'*, *recursive=False*, *timestamp=None*`)`

`action_url`
This is the %s attribute for object definition

`active_checks_enabled`
This is the %s attribute for object definition

`add_to_contactgroup``(`*contactgroup*`)`

`add_to_hostgroup``(`*hostgroup_name*`)`
Add host to a hostgroup

`address`
This is the %s attribute for object definition

`alias`
This is the %s attribute for object definition

`check_command`
This is the %s attribute for object definition

`check_freshness`
This is the %s attribute for object definition

`check_interval`
This is the %s attribute for object definition

check_period
This is the %s attribute for object definition

contact_groups
This is the %s attribute for object definition

contacts
This is the %s attribute for object definition

copy (*recursive=False*, *filename=None*, ***args*)
Same as ObjectDefinition.copy() except can recursively copy services

delete (*recursive=False*, *cleanup_related_items=True*)
Delete this host and optionally its services

Works like ObjectDefinition.delete() except for:

Arguments: cleanup_related_items – If True, remove references found in hostgroups and escalations recursive – If True, also delete all services of this host

display_name
This is the %s attribute for object definition

downtime (*start_time=None*, *end_time=None*, *trigger_id=0*, *duration=7200*, *author=None*, *comment='Downtime scheduled by pynag'*, *recursive=False*)
Put this object in a schedule downtime.

Arguments: start_time – When downtime should start. If None, use time.time() (now) end_time – When scheduled downtime should end. If None use start_time + duration duration – Alternative to end_time, downtime lasts for duration seconds. Default 7200 seconds. trigger_id – trigger_id>0 means that this downtime should trigger another downtime with trigger_id. author – name of the contact scheduling downtime. If None, use current system user comment – Comment that will be put in with the downtime recursive – Also schedule same downtime for all service of this host.

Returns: None because commands sent to nagios have no return values

Raises: PynagError if this does not look an active object.

event_handler
This is the %s attribute for object definition

event_handler_enabled
This is the %s attribute for object definition

first_notification_delay
This is the %s attribute for object definition

flap_detection_enabled
This is the %s attribute for object definition

flap_detection_options
This is the %s attribute for object definition

freshness_threshold
This is the %s attribute for object definition

get_current_status ()
Returns a dictionary with status data information for this object

get_effective_check_command ()
Returns a Command object as defined by check_command attribute

Raises KeyError if check_command is not found or not defined.

`get_effective_contact_groups``()`
: Returns a list of all Contactgroup that belong to this Host

`get_effective_contacts``()`
: Returns a list of all Contact that belong to this Host

`get_effective_hostgroups``()`
: Returns a list of all Hostgroup that belong to this Host

`get_effective_network_children``(`*recursive=False*`)`
: Get all objects that depend on this one via "parents" attribute

  Arguments: recursive - If true include grandchildren in list to be returned

  Returns: a list of ObjectDefinition objects

`get_effective_network_parents``(`*recursive=False*`)`
: Get all objects this one depends on via "parents" attribute

  Arguments: recursive - If true include grandparents in list to be returned

  Returns: a list of ObjectDefinition objects

`get_effective_services``()`
: Returns a list of all Service that belong to this Host

`get_related_objects``()`

`high_flap_threshold`
: This is the %s attribute for object definition

`host_name`
: This is the %s attribute for object definition

`hostgroups`
: This is the %s attribute for object definition

`icon_image`
: This is the %s attribute for object definition

`icon_image_alt`
: This is the %s attribute for object definition

`initial_state`
: This is the %s attribute for object definition

`low_flap_threshold`
: This is the %s attribute for object definition

`max_check_attempts`
: This is the %s attribute for object definition

`notes`
: This is the %s attribute for object definition

`notes_url`
: This is the %s attribute for object definition

`notification_interval`
: This is the %s attribute for object definition

`notification_options`
: This is the %s attribute for object definition

`notification_period`
This is the %s attribute for object definition

`notifications_enabled`
This is the %s attribute for object definition

`object_type` = 'host'

`objects` = <pynag.Model.ObjectFetcher object at 0x7fb0e557b890>

`obsess_over_host`
This is the %s attribute for object definition

`parents`
This is the %s attribute for object definition

`passive_checks_enabled`
This is the %s attribute for object definition

`process_perf_data`
This is the %s attribute for object definition

`remove_from_contactgroup`(*contactgroup*)

`remove_from_hostgroup`(*hostgroup_name*)
Removes host from specified hostgroup

`rename`(*shortname*)
Rename this host, and modify related objects

`retain_nonstatus_information`
This is the %s attribute for object definition

`retain_status_information`
This is the %s attribute for object definition

`retry_interval`
This is the %s attribute for object definition

`stalking_options`
This is the %s attribute for object definition

`statusmap_image`
This is the %s attribute for object definition

`vrml_image`
This is the %s attribute for object definition

class `pynag.Model.`**`HostDependency`**(*item=None*, *filename=None*, ***kwargs*)
Bases: `pynag.Model.ObjectDefinition`

`dependency_period`
This is the %s attribute for object definition

`dependent_host_name`
This is the %s attribute for object definition

`dependent_hostgroup_name`
This is the %s attribute for object definition

`execution_failure_criteria`
This is the %s attribute for object definition

`host_name`
This is the %s attribute for object definition

hostgroup_name
This is the %s attribute for object definition

inherits_parent
This is the %s attribute for object definition

notification_failure_criteria
This is the %s attribute for object definition

object_type = 'hostdependency'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e593f550>

class pynag.Model.**HostEscalation**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

contact_groups
This is the %s attribute for object definition

contacts
This is the %s attribute for object definition

escalation_options
This is the %s attribute for object definition

escalation_period
This is the %s attribute for object definition

first_notification
This is the %s attribute for object definition

host_name
This is the %s attribute for object definition

hostgroup_name
This is the %s attribute for object definition

last_notification
This is the %s attribute for object definition

notification_interval
This is the %s attribute for object definition

object_type = 'hostescalation'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e591dc50>

class pynag.Model.**Hostgroup**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

action_url
This is the %s attribute for object definition

add_host(*host_name*)
Adds host to this group. Behaves like Hostgroup._add_member_to_group

alias
This is the %s attribute for object definition

delete(*recursive=False*, *cleanup_related_items=True*)
Delete this hostgroup and optionally remove references in hosts and services

Works like ObjectDefinition.delete() except:

Arguments: cleanup_related_items – If True, remove all references to this group in hosts/services,escalations,etc recursive – If True, remove services and escalations that bind to this (and only this) hostgroup

downtime(*start_time=None*, *end_time=None*, *trigger_id=0*, *duration=7200*, *author=None*, *comment='Downtime scheduled by pynag'*, *recursive=False*)
Put every host and service in this hostgroup in a schedule downtime.

Arguments: start_time – When downtime should start. If None, use time.time() (now) end_time – When scheduled downtime should end. If None use start_time + duration duration – Alternative to end_time, downtime lasts for duration seconds. Default 7200 seconds. trigger_id – trigger_id>0 means that this downtime should trigger another downtime with trigger_id. author – name of the contact scheduling downtime. If None, use current system user comment – Comment that will be put in with the downtime recursive – For compatibility with other downtime commands, recursive is always assumed to be true

Returns: None because commands sent to nagios have no return values

Raises: PynagError if this does not look an active object.

get_effective_hostgroups()
Returns a list of every Hostgroup that is a member of this Hostgroup

get_effective_hosts()
Returns a list of all Host that belong to this hostgroup

get_effective_services()
Returns a list of all Service that belong to this hostgroup

hostgroup_members
This is the %s attribute for object definition

hostgroup_name
This is the %s attribute for object definition

members
This is the %s attribute for object definition

notes
This is the %s attribute for object definition

notes_url
This is the %s attribute for object definition

object_type = 'hostgroup'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e58f85d0>

remove_host(*host_name*)
Remove host from this group. Behaves like Hostgroup._remove_member_from_group

rename(*shortname*)
Rename this hostgroup, and modify hosts if required

pynag.Model.**Object**
alias of HostEscalation

class pynag.Model.**ObjectDefinition**(*item=None*, *filename=None*, ***kwargs*)
Bases: object

Holds one instance of one particular Object definition

Example:

```
>>> objects = ObjectDefinition.objects.all
>>> my_object = ObjectDefinition( dict )
```

attribute_appendfield(*attribute_name*, *value*)
Convenient way to append value to an attribute with a comma seperated value

Example:

```
>>> myservice = Service()
>>> myservice.attribute_appendfield(attribute_name="contact_groups", value="alladmins")
>>> myservice.contact_groups
'+alladmins'
>>> myservice.attribute_appendfield(attribute_name="contact_groups", value='webmasters')
>>> print myservice.contact_groups
+alladmins,webmasters
```

attribute_is_empty(*attribute_name*)
Check if the attribute is empty

Parameters **attribute_name** – A attribute such as *host_name*

Returns True or False

attribute_removefield(*attribute_name*, *value*)
Convenient way to remove value to an attribute with a comma seperated value

Example:

```
>>> myservice = Service()
>>> myservice.contact_groups = "+alladmins,localadmins"
>>> myservice.attribute_removefield(attribute_name="contact_groups", value='localadmins'
>>> print myservice.contact_groups
+alladmins
>>> myservice.attribute_removefield(attribute_name="contact_groups", value="alladmins")
>>> print myservice.contact_groups
None
```

attribute_replacefield(*attribute_name*, *old_value*, *new_value*)
Convenient way to replace field within an attribute with a comma seperated value

Example:

```
>>> myservice = Service()
>>> myservice.contact_groups = "+alladmins,localadmins"
>>> myservice.attribute_replacefield(attribute_name="contact_groups", old_value='localadm
>>> print myservice.contact_groups
+alladmins,webmasters
```

copy(*recursive=False*, *filename=None*, ***args*)
Copies this object definition with any unsaved changes to a new configuration object

Arguments: filename: If specified, new object will be saved in this file. recursive: If true, also find any related children objects and copy those **args: Any argument will be treated a modified attribute in the new definition.

Examples: myhost = Host.objects.get_by_shortname('myhost.example.com')

# Copy this host to a new one myhost.copy(host_name="newhost.example.com", address="127.0.0.1")

# Copy this host and all its services: myhost.copy(recursive=True, host_name="newhost.example.com", address="127.0.0.1")

Returns:

- A copy of the new ObjectDefinition
- A list of all copies objects if recursive is True

`delete`(*recursive=False*, *cleanup_related_items=True*)
Deletes this object definition from its configuration files.

Parameters

- **recursive** – If True, look for items that depend on this object and delete them as well (for example, if you delete a host, delete all its services as well)
- **cleanup_related_items** – If True, look for related items and remove references to this one. (for example, if you delete a host, remove its name from all hostgroup.members entries)

`get`(*value*, *default=None*)
self.get(x) == self[x]

`get_all_macros`()
Returns {macroname:macrovalue} hash map of this object's macros

`get_attribute`(*attribute_name*)
Get one attribute from our object definition

Parameters **attribute_name** – A attribute such as *host_name*

`get_attribute_tuple`()
Returns all relevant attributes in the form of:

(attribute_name,defined_value,inherited_value)

`get_description`()
Returns a human friendly string describing current object.

It will try the following in order: * return self.name (get the generic name) * return self get_shortname() * return "Untitled $object_type"

`get_effective_children`(*recursive=False*)
Get a list of all objects that inherit this object via "use" attribute

Parameters **recursive** – If true, include grandchildren as well

Returns A list of ObjectDefinition objects

`get_effective_command_line`(*host_name=None*)
Return a string of this objects check_command with all macros (i.e. $HOSTADDR$) resolved

`get_effective_notification_command_line`(*host_name=None*, *contact_name=None*)
Get this objects notifications with all macros (i.e. $HOSTADDR$) resolved

Parameters

- **host_name** – Simulate notification using this host. If None: Use first valid host (used for services)
- **contact_name** – Simulate notification for this contact. If None: use first valid contact for the service

Returns string of this objects notifications

get_effective_parents(*recursive=False*, *cache_only=False*)
Get all objects that this one inherits via "use" attribute

Arguments: recursive - If true include grandparents in list to be returned

Returns: a list of ObjectDefinition objects

get_filename()
Get name of the config file which defines this object

get_id()
Return a unique ID for this object

get_macro(*macroname*, *host_name=None*, *contact_name=None*)
Take macroname (e.g. $USER1$) and return its actual value

Arguments: macroname – Macro that is to be resolved. For example $HOSTADDRESS$ host_name – Optionally specify host (use this for services that

– don't define host specifically for example ones that only – define hostgroups

Returns: (str) Actual value of the macro. For example "$HOSTADDRESS$" becomes "127.0.0.1"

get_parents()
Out-dated, use get_effective_parents instead. Kept here for backwards compatibility

get_related_objects()
Returns a list of ObjectDefinition that depend on this object

Object can "depend" on another by a 'use' or 'host_name' or similar attribute

Returns: List of ObjectDefinition objects

get_shortname()
Returns shortname of an object in string format.

For the confused, nagios documentation refers to shortnames usually as <object_type>_name.

- In case of Host it returns host_name
- In case of Command it returns command_name
- etc
- Special case for services it returns "host_name/service_description"

Returns None if no attribute can be found to use as a shortname

get_suggested_filename()
Get a suitable configuration filename to store this object in

Returns filename, eg str('/etc/nagios/pynag/templates/hosts.cfg')

has_key(*key*)
Same as key in self

is_defined(*attribute_name*)
Returns True if attribute_name is defined in this object

is_dirty()
Returns true if any attributes has been changed on this object, and therefore it needs saving

is_registered()
Returns true if object is enabled (registered)

items()

keys()

move(*filename*)
Move this object definition to a new file. It will be deleted from current file.

This is the same as running:

```
>>> self.copy(filename=filename)
>>> self.delete()
```

Returns The new object definition

name
This is the %s attribute for object definition

object_type = **None**

objects = **<pynag.Model.ObjectFetcher object at 0x7fb0e58d5d90>**

register
This is the %s attribute for object definition

reload_object()
Re-applies templates to this object (handy when you have changed the use attribute

rename(*shortname*)
Change the shortname of this object

Most objects that inherit this one, should also be responsible for updating related objects about the rename.

Args: shortname: New name for this object

Returns: None

rewrite(**args*, ***kw*)

run_check_command(*host_name=None*)
Run the check_command defined by this service. Returns return_code,stdout,stderr

save(**args*, ***kw*)

set_attribute(*attribute_name*, *attribute_value*)
Set (but does not save) one attribute in our object

Parameters

- **attribute_name** – A attribute such as *host_name*
- **attribute_value** – The value you would like to set

set_filename(*filename*)
Set name of the config file which this object will be written to on next save.

set_macro(*macroname*, *new_value*)
Update a macro (custom variable) like $ARG1$ intelligently

Returns: None

Notes: You are responsible for calling .save() after modifying the object

Examples:

```
>>> s = Service()
>>> s.check_command = 'okc-execute!arg1!arg2'
>>> s.set_macro('$ARG1$', 'modified1')
>>> s.check_command
'okc-execute!modified1!arg2'
>>> s.set_macro('$ARG5$', 'modified5')
>>> s.check_command
'okc-execute!modified1!arg2!!!modified5'
>>> s.set_macro('$_SERVICE_TEST$', 'test')
>>> s['__TEST']
'test'
```

unregister(*recursive=True*)
Short for self['register'] = 0 ; self.save()

use
This is the %s attribute for object definition

class pynag.Model.**ObjectFetcher**(*object_type*)
Bases: object

This class is a wrapper around pynag.Parsers.config. Is responsible for fetching dict objects from config.data and turning into high ObjectDefinition objects

Internal variables:

- _cached_objects = List of every ObjectDefinition
- _cached_id[o.get_id()] = o
- _cached_shortnames[o.object_type][o.get_shortname()] = o
- _cached_names[o.object_type][o.name] = o
- _cached_object_type[o.object_type].append(o)

all

filter(***kwargs*)
Returns all objects that match the selected filter

Example:

Get all services where host_name is examplehost.example.com

```
>>> Service.objects.filter(host_name='examplehost.example.com')
```

Get service with host_name=examplehost.example.com and service_description='Ping'

```
>>> Service.objects.filter(host_name='examplehost.example.com',
...                        service_description='Ping')
```

Get all services that are registered but without a host_name

```
>>> Service.objects.filter(host_name=None,register='1')
```

Get all hosts that start with 'exampleh'

```
>>> Host.objects.filter(host_name__startswith='exampleh')
```

Get all hosts that end with 'example.com'

```
>>> Service.objects.filter(host_name__endswith='example.com')
```

Get all contactgroups that contain 'dba'

```
>>> Contactgroup.objects.filter(host_name__contains='dba')
```

Get all hosts that are not in the 'testservers' hostgroup

```
>>> Host.objects.filter(hostgroup_name__notcontains='testservers')
```

Get all services with non-empty name

```
>>> Service.objects.filter(name__isnot=None)
```

Get all hosts that have an address:

```
>>> Host.objects.filter(address_exists=True)
```

`get_all`(**args*, ***kw*)

`get_by_id`(*id*, *cache_only=False*)
Get one specific object

Returns ObjectDefinition

Raises ValueError if object is not found

`get_by_name`(*object_name*, *cache_only=False*)
Get one specific object by its object_name (i.e. name attribute)

Returns ObjectDefinition

Raises ValueError if object is not found

`get_by_shortname`(*shortname*, *cache_only=False*)
Get one specific object by its shortname (i.e. host_name for host, etc)

Parameters

- **shortname** – shortname of the object. i.e. host_name, command_name, etc.
- **cache_only** – If True, dont check if configuration files have changed since last parse

Returns ObjectDefinition

Raises ValueError if object is not found

`get_object_types`()
Returns a list of all discovered object types

`needs_reload`(**args*, ***kw*)

`reload_cache`(**args*, ***kw*)

class `pynag.Model.`**`ObjectRelations`**
Bases: `object`

Static container for objects and their respective neighbours

`command_host` **= defaultdict(<type 'set'>, {})**

`command_service` **= defaultdict(<type 'set'>, {})**

`contact_contactgroups` **= defaultdict(<type 'set'>, {})**

contact_hosts = defaultdict(<type 'set'>, {})

contact_services = defaultdict(<type 'set'>, {})

contactgroup_contactgroups = defaultdict(<type 'set'>, {})

contactgroup_contacts = defaultdict(<type 'set'>, {})

contactgroup_hosts = defaultdict(<type 'set'>, {})

contactgroup_services = defaultdict(<type 'set'>, {})

contactgroup_subgroups = defaultdict(<type 'set'>, {})

host_contact_groups = defaultdict(<type 'set'>, {})

host_contacts = defaultdict(<type 'set'>, {})

host_hostgroups = defaultdict(<type 'set'>, {})

host_services = defaultdict(<type 'set'>, {})

hostgroup_hostgroups = defaultdict(<type 'set'>, {})

hostgroup_hosts = defaultdict(<type 'set'>, {})

hostgroup_services = defaultdict(<type 'set'>, {})

hostgroup_subgroups = defaultdict(<type 'set'>, {})

static reset()
Runs clear() on every member attribute in ObjectRelations

static resolve_contactgroups()
Update all contactgroup relations to take into account contactgroup.contactgroup_members

static resolve_hostgroups()
Update all hostgroup relations to take into account hostgroup.hostgroup_members

static resolve_regex()
If any object relations are a regular expression, then expand them into a full list

static resolve_servicegroups()
Update all servicegroup relations to take into account servicegroup.servicegroup_members

service_contact_groups = defaultdict(<type 'set'>, {})

service_contacts = defaultdict(<type 'set'>, {})

service_hostgroups = defaultdict(<type 'set'>, {})

service_hosts = defaultdict(<type 'set'>, {})

service_servicegroups = defaultdict(<type 'set'>, {})

servicegroup_members = defaultdict(<type 'set'>, {})

servicegroup_servicegroups = defaultdict(<type 'set'>, {})

servicegroup_services = defaultdict(<type 'set'>, {})

servicegroup_subgroups = defaultdict(<type 'set'>, {})

use = defaultdict(<function <lambda> at 0x7fb0e556cf50>, {})

class pynag.Model.**Service**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

acknowledge(*sticky=1*, *notify=1*, *persistent=0*, *author='pynag'*, *comment='acknowledged by pynag'*, *timestamp=None*)

action_url
This is the %s attribute for object definition

active_checks_enabled
This is the %s attribute for object definition

add_to_contactgroup(*contactgroup*)

add_to_servicegroup(*servicegroup_name*)
Add this service to a specific servicegroup

check_command
This is the %s attribute for object definition

check_freshness
This is the %s attribute for object definition

check_interval
This is the %s attribute for object definition

check_period
This is the %s attribute for object definition

contact_groups
This is the %s attribute for object definition

contacts
This is the %s attribute for object definition

display_name
This is the %s attribute for object definition

downtime(*start_time=None*, *end_time=None*, *trigger_id=0*, *duration=7200*, *author=None*, *comment='Downtime scheduled by pynag'*, *recursive=False*)
Put this object in a schedule downtime.

Arguments: start_time – When downtime should start. If None, use time.time() (now) end_time – When scheduled downtime should end. If None use start_time + duration duration – Alternative to end_time, downtime lasts for duration seconds. Default 7200 seconds. trigger_id – trigger_id>0 means that this downtime should trigger another downtime with trigger_id. author – name of the contact scheduling downtime. If None, use current system user comment – Comment that will be put in with the downtime recursive – Here for compatibility. Has no effect on a service.

Returns: None because commands sent to nagios have no return values

Raises: PynagError if this does not look an active object.

event_handler
This is the %s attribute for object definition

event_handler_enabled
This is the %s attribute for object definition

first_notification_delay
This is the %s attribute for object definition

flap_detection_enabled
This is the %s attribute for object definition

flap_detection_options
This is the %s attribute for object definition

freshness_threshold
This is the %s attribute for object definition

get_current_status()
: Returns a dictionary with status data information for this object

get_effective_check_command()
: Returns a Command object as defined by check_command attribute

  Raises KeyError if check_command is not found or not defined.

get_effective_contact_groups()
: Returns a list of all Contactgroup that belong to this Service

get_effective_contacts()
: Returns a list of all Contact that belong to this Service

get_effective_hostgroups()
: Returns a list of all Hostgroup that belong to this Service

get_effective_hosts()
: Returns a list of all Host that belong to this Service

get_effective_servicegroups()
: Returns a list of all Servicegroup that belong to this Service

get_shortname()

high_flap_threshold
: This is the %s attribute for object definition

host_name
: This is the %s attribute for object definition

hostgroup_name
: This is the %s attribute for object definition

icon_image
: This is the %s attribute for object definition

icon_image_alt
: This is the %s attribute for object definition

initial_state
: This is the %s attribute for object definition

is_volatile
: This is the %s attribute for object definition

low_flap_threshold
: This is the %s attribute for object definition

max_check_attempts
: This is the %s attribute for object definition

merge_with_host()
: Moves a service from its original file to the same file as the first effective host

notes
: This is the %s attribute for object definition

notes_url
: This is the %s attribute for object definition

notification_interval
: This is the %s attribute for object definition

`notification_options`
This is the %s attribute for object definition

`notification_period`
This is the %s attribute for object definition

`notifications_enabled`
This is the %s attribute for object definition

`object_type` = 'service'

`objects` = <pynag.Model.ObjectFetcher object at 0x7fb0e557b950>

`obsess_over_service`
This is the %s attribute for object definition

`passive_checks_enabled`
This is the %s attribute for object definition

`process_perf_data`
This is the %s attribute for object definition

`remove_from_contactgroup`(*contactgroup*)

`remove_from_servicegroup`(*servicegroup_name*)
remove this service from a specific servicegroup

`rename`(*shortname*)
Not implemented. Do not use.

`retain_nonstatus_information`
This is the %s attribute for object definition

`retain_status_information`
This is the %s attribute for object definition

`retry_interval`
This is the %s attribute for object definition

`service_description`
This is the %s attribute for object definition

`servicegroups`
This is the %s attribute for object definition

`stalking_options`
This is the %s attribute for object definition

class `pynag.Model.`**`ServiceDependency`**(*item=None*, *filename=None*, ***kwargs*)
Bases: `pynag.Model.ObjectDefinition`

`dependency_period`
This is the %s attribute for object definition

`dependent_host_name`
This is the %s attribute for object definition

`dependent_hostgroup_name`
This is the %s attribute for object definition

`dependent_service_description`
This is the %s attribute for object definition

`execution_failure_criteria`
This is the %s attribute for object definition

host_name
This is the %s attribute for object definition

hostgroup_name
This is the %s attribute for object definition

inherits_parent
This is the %s attribute for object definition

notification_failure_criteria
This is the %s attribute for object definition

object_type = 'servicedependency'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e58f81d0>

service_description
This is the %s attribute for object definition

class pynag.Model.**ServiceEscalation**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

contact_groups
This is the %s attribute for object definition

contacts
This is the %s attribute for object definition

escalation_options
This is the %s attribute for object definition

escalation_period
This is the %s attribute for object definition

first_notification
This is the %s attribute for object definition

host_name
This is the %s attribute for object definition

hostgroup_name
This is the %s attribute for object definition

last_notification
This is the %s attribute for object definition

notification_interval
This is the %s attribute for object definition

object_type = 'serviceescalation'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e557b9d0>

service_description
This is the %s attribute for object definition

class pynag.Model.**Servicegroup**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

action_url
This is the %s attribute for object definition

add_service(*shortname*)
Adds service to this group. Behaves like _add_object_to_group(object, group)

alias
This is the %s attribute for object definition

downtime(*start_time=None*, *end_time=None*, *trigger_id=0*, *duration=7200*, *author=None*, *comment='Downtime scheduled by pynag'*, *recursive=False*)
Put every host and service in this servicegroup in a schedule downtime.

Arguments: start_time – When downtime should start. If None, use time.time() (now) end_time – When scheduled downtime should end. If None use start_time + duration duration – Alternative to end_time, downtime lasts for duration seconds. Default 7200 seconds. trigger_id – trigger_id>0 means that this downtime should trigger another downtime with trigger_id. author – name of the contact scheduling downtime. If None, use current system user comment – Comment that will be put in with the downtime recursive – For compatibility with other downtime commands, recursive is always assumed to be true

Returns: None because commands sent to nagios have no return values

Raises: PynagError if this does not look an active object.

get_effective_servicegroups()
Returns a list of every Servicegroup that is a member of this Servicegroup

get_effective_services()
Returns a list of all Service that belong to this Servicegroup

members
This is the %s attribute for object definition

notes
This is the %s attribute for object definition

notes_url
This is the %s attribute for object definition

object_type = 'servicegroup'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e557ba90>

remove_service(*shortname*)
remove service from this group. Behaves like _remove_object_from_group(object, group)

servicegroup_members
This is the %s attribute for object definition

servicegroup_name
This is the %s attribute for object definition

class pynag.Model.**Timeperiod**(*item=None*, *filename=None*, ***kwargs*)
Bases: pynag.Model.ObjectDefinition

alias
This is the %s attribute for object definition

exclude
This is the %s attribute for object definition

object_type = 'timeperiod'

objects = <pynag.Model.ObjectFetcher object at 0x7fb0e58d5350>

timeperiod_name
This is the %s attribute for object definition

pynag.Model.**eventhandlers = []**
eventhandlers – A list of Model.EventHandlers object.

`all_attributes` Module

`macros` Module

This file contains a dict object that maps Nagios Standard macronames to specific values.

i.e. macros['$HOSTADDR$'] should return 'address'

Subpackages

EventHandlers Package

`EventHandlers` Package This module is experimental.

The idea is to create a mechanism that allows you to hook your own events into an ObjectDefinition instance.

This enables you for example to log to file every time an object is rewritten.

class `pynag.Model.EventHandlers.`**`BaseEventHandler`**(*debug=False*)

`debug`(*object_definition*, *message*)
Used for any particual debug notifications

`pre_save`(*object_definition*, *message*)
Called at the beginning of save()

`save`(*object_definition*, *message*)
Called when objectdefinition.save() has finished

`write`(*object_definition*, *message*)
Called whenever a modification has been written to file

exception `pynag.Model.EventHandlers.`**`EventHandlerError`**(*message*, *errorcode=None*, *errorstring=None*)
Bases: `exceptions.Exception`

class `pynag.Model.EventHandlers.`**`FileLogger`**(*logfile='/var/log/pynag.log'*, *debug=False*)
Bases: `pynag.Model.EventHandlers.BaseEventHandler`

Handler that logs everything to file

`debug`(*object_definition*, *message*)
Used for any particular debug notifications

`save`(*object_definition*, *message*)
Called when objectdefinition.save() has finished

`write`(*object_definition*, *message*)
Called whenever a modification has been written to file

class `pynag.Model.EventHandlers.`**`GitEventHandler`**(*gitdir*, *source*, *modified_by*, *auto_init=False*, *ignore_errors=False*)
Bases: `pynag.Model.EventHandlers.BaseEventHandler`

`debug`(*object_definition*, *message*)

`get_uncommited_files`()
Returns a list of files that are have unstaged changes

`is_commited`()
Returns True if all files in git repo are fully commited

pre_save (*object_definition*, *message*)
Commits object_definition.get_filename() if it has any changes

save (*object_definition*, *message*)

write (*object_definition*, *message*)

class `pynag.Model.EventHandlers.`**NagiosReloadHandler** (*nagios_init*, **args*, ***kwargs*)
Bases: `pynag.Model.EventHandlers.BaseEventHandler`

This handler reloads nagios every time that a change is made. This is only meant for small environments

debug (*object_definition*, *message*)
Used for any particual debug notifications

pre_save (*object_definition*, *message*)
Called at the beginning of save()

save (*object_definition*, *message*)
Called when objectdefinition.save() has finished

write (*object_definition*, *message*)
Called whenever a modification has been written to file

class `pynag.Model.EventHandlers.`**PrintToScreenHandler** (*debug=False*)
Bases: `pynag.Model.EventHandlers.BaseEventHandler`

Handler that prints everything to stdout

debug (*object_definition*, *message*)
Used for any particual debug notifications

save (*object_definition*, *message*)
Called when objectdefinition.save() has finished

write (*object_definition*, *message*)
Called whenever a modification has been written to file

2.2.3 Parsers Package

`Parsers` Package

This module contains low-level Parsers for nagios configuration and status objects.

Hint: If you are looking to parse some nagios configuration data, you probably want pynag.Model module instead.

The highlights of this module are:

class Config: For Parsing nagios local nagios configuration files class Livestatus: To connect to MK-Livestatus class StatusDat: To read info from status.dat (not used a lot, migrate to mk-livestatus) class LogFiles: To read nagios log-files class MultiSite: To talk with multiple Livestatus instances

class `pynag.Parsers.`**Config** (*cfg_file=None*, *strict=False*)
Bases: `object`

Parse and write nagios config files

abspath (*path*)
Return the absolute path of a given relative path.

The current working directory is assumed to be the dirname of nagios.cfg

Args:

path: relative path to be transformed into absolute path. (string)

Returns:

Absolute path of given relative path.

Example:

```
>>> c = config(cfg_file="/etc/nagios/nagios.cfg")
>>> c.abspath('nagios.cfg')
'/etc/nagios/nagios.cfg'
>>> c.abspath('/etc/nagios/nagios.cfg')
'/etc/nagios/nagios.cfg'
```

access (**args*, ***kwargs*)
Wrapper around os.access

cleanup ()
Remove configuration files that have no configuration items

commit ()
Write any changes that have been made to it's appropriate file

compareObjects (*item1*, *item2*)
Compares two items. Returns true if they are equal

Compares every key: value pair for both items. If anything is different, the items will not be considered equal.

Args: item1, item2: Items to be compared.

Returns:

True – Items are equal

False – Items are not equal

delete_host (*object_name*, *user_key=None*)
Delete a host from its configuration files

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

True on success.

delete_hostgroup (*object_name*, *user_key=None*)
Delete a hostgroup from its configuration files

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

True on success.

delete_object (*object_type*, *object_name*, *user_key=None*)
Delete object from configuration files

Args:

object_type: Type of the object to delete from configuration files.

object_name: Name of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

True on success.

`delete_service`(*service_description*, *host_name*)

Delete service from configuration files

Args:

service_description: service_description field value of the object to delete from configuration files.

host_name: host_name field value of the object to delete from configuration files.

Returns:

True on success.

`edit_object`(*item*, *field_name*, *new_value*)

Modifies a (currently existing) item.

Changes are immediate (i.e. there is no commit)

Args:

item: Item to modify.

field_name: Field that will be updated.

new_value: Updated value of field *field_name*

Example Usage: edit_object(item, field_name="host_name", new_value="examplehost.example.com")

Returns: True on success

Warning: THIS FUNCTION IS DEPRECATED. USE item_edit_field() instead

`edit_service`(*target_host*, *service_description*, *field_name*, *new_value*)

Edit a service's attributes

Takes a host, service_description pair to identify the service to modify and sets its field *field_name* to *new_value*.

Args:

target_host: name of the host to which the service is attached to. (string)

service_description: Service description of the service to modify. (string)

field_name: Field to modify. (string)

new_value: Value to which the *field_name* field will be updated (string)

Returns:

True on success

Raises:

`ParserError` if the service is not found

exists(**args*, ***kwargs*)
Wrapper around os.path.exists

extended_parse()
This parse is used after the initial parse() command is run.

It is only needed if you want extended meta information about hosts or other objects

flag_all_commit()
Flag every item in the configuration to be committed This should probably only be used for debugging purposes

get_cfg_dirs()
Parses the main config file for configuration directories

Returns:

List of all cfg directories used in this configuration

Example:

```
print(get_cfg_dirs())
['/etc/nagios/hosts','/etc/nagios/objects',...]
```

get_cfg_files()
Return a list of all cfg files used in this configuration

Filenames are normalised so that if nagios.cfg specifies relative filenames we will convert it to fully qualified filename before returning.

Returns:

List of all configurations files used in the configuration.

Example:

print(get_cfg_files()) ['/etc/nagios/hosts/host1.cfg','/etc/nagios/hosts/host2.cfg',...]

get_cfg_value(*key*)
Returns one specific value from your nagios.cfg file, None if value is not found.

Arguments:

key: what attribute to fetch from nagios.cfg (example: "command_file")

Returns:

String of the first value found for

Example:

```
>>> c = Config()
>>> log_file = c.get_cfg_value('log_file')
# Should return something like "/var/log/nagios/nagios.log"
```

get_command(*object_name*, *user_key=None*)
Return a Command object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_contact(*object_name*, *user_key=None*)

Return a Contact object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_contactgroup(*object_name*, *user_key=None*)

Return a Contactgroup object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_host(*object_name*, *user_key=None*)

Return a host object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_hostdependency(*object_name*, *user_key=None*)

Return a hostdependency object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_hostgroup(*object_name*, *user_key=None*)

Return a hostgroup object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_new_item(*object_type*, *filename*)

Returns an empty item with all necessary metadata

Creates a new item dict and fills it with usual metadata:

- object_type : object_type (arg)
- filename : filename (arg)
- template_fields = []
- needs_commit = None
- delete_me = None
- defined_attributes = {}
- inherited_attributes = {}
- raw_definition = "define %s {nn} % object_type"

Args:

object_type: type of the object to be created (string)

filename: Path to which the item will be saved (string)

Returns:

A new item with default metadata

get_object(*object_type*, *object_name*, *user_key=None*)
Return a complete object dictionary

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: User defined key. Default None. (string)

Returns:

The item found to match all the criterias.

None if object is not found

get_object_types()
Returns a list of all discovered object types

get_resource(*resource_name*)
Get a single resource value which can be located in any resource.cfg file

Arguments:

resource_name: Name as it appears in resource file (i.e. $USER1$)

Returns:

String value of the resource value.

Raises:

`KeyError` if resource is not found

`ParserError` if resource is not found and you do not have permissions

get_resources()
Returns a list of every private resources from nagios.cfg

get_service(*target_host*, *service_description*)
Return a service object

Args:

target_host: host_name field of the service to be returned. This is the host to which is attached the service.

service_description: service_description field of the service to be returned.

Returns:

The item found to match all the criterias.

get_servicedependency(*object_name*, *user_key=None*)
Return a servicedependency object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_servicegroup(*object_name*, *user_key=None*)
Return a Servicegroup object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_timeperiod(*object_name*, *user_key=None*)
Return a Timeperiod object

Args:

object_name: object_name field value of the object to delete from configuration files.

user_key: user_key to pass to `get_object()`

Returns:

The item found to match all the criterias.

get_timestamps()
Returns hash map of all nagios related files and their timestamps

guess_cfg_file()
Returns a path to any nagios.cfg found on your system

Use this function if you don't want specify path to nagios.cfg in your code and you are confident that it is located in a common location

Checked locations are as follows:

- /etc/nagios/nagios.cfg
- /etc/nagios3/nagios.cfg
- /usr/local/nagios/etc/nagios.cfg
- /nagios/etc/nagios/nagios.cfg
- ./nagios.cfg

- ./nagios/nagios.cfg
- /etc/icinga/icinga.cfg
- /usr/local/icinga/etc/icinga.cfg
- ./icinga.cfg
- ./icinga/icinga.cfg
- /etc/naemon/naemon.cfg
- /usr/local/naemon/etc/naemon.cfg
- ./naemon.cfg
- ./naemon/naemon.cfg
- /etc/shinken/shinken.cfg

Returns:

  str. Path to the nagios.cfg or equivalent file

  None if couldn't find a file in any of these locations.

`guess_nagios_binary``()`

Returns a path to any nagios binary found on your system

Use this function if you don't want specify path to the nagios binary in your code and you are confident that it is located in a common location

Checked locations are as follows:

- /usr/bin/nagios
- /usr/sbin/nagios
- /usr/local/nagios/bin/nagios
- /nagios/bin/nagios
- /usr/bin/icinga
- /usr/sbin/icinga
- /usr/bin/naemon
- /usr/sbin/naemon
- /usr/local/naemon/bin/naemon.cfg
- /usr/bin/shinken
- /usr/sbin/shinken

Returns:

  str. Path to the nagios binary

  None if could not find a binary in any of those locations

`guess_nagios_directory``()`

Returns a path to the nagios configuration directory on your system

Use this function for determining the nagios config directory in your code

Returns:

  str. directory containing the nagios.cfg file

Raises:

`pynag.Parsers.ConfigFileNotFound` if cannot guess config file location.

`isdir`(**args*, ***kwargs*)
Wrapper around os.path.isdir

`isfile`(**args*, ***kwargs*)
Wrapper around os.path.isfile

`islink`(**args*, ***kwargs*)
Wrapper around os.path.islink

`item_add`(*item*, *filename*)
Adds a new object to a specified config file.

Args:

item: Item to be created

filename: Filename that we are supposed to write the new item to. This is the path to the file. (string)

Returns:

True on success

Raises:

`IOError` on failed save

`item_edit_field`(*item*, *field_name*, *new_value*)
Modifies one field of a (currently existing) object.

Changes are immediate (i.e. there is no commit)

Args:

item: Item to be modified. Its field *field_name* will be set to *new_value*.

field_name: Name of the field that will be modified. (str)

new_value: Value to which will be set the field *field_name*. (str)

Example usage:: edit_object(item, field_name="host_name", new_value="examplehost.example.com") # doctest: +SKIP

Returns: True on success

Raises:

`ValueError` if object is not found

`IOError` if save fails

`item_remove`(*item*)
Delete one specific item from its configuration files

Args:

item: Item that is to be rewritten

str_new_item: string representation of the new item

Examples:: item_remove(item, "define service {n name example-service n register 0 n }n")

Returns:

True on success

Raises:

`ValueError` if object is not found

`IOError` if save fails

`item_remove_field`(*item*, *field_name*)

Removes one field of a (currently existing) object.

Changes are immediate (i.e. there is no commit)

Args:

item: Item to remove field from.

field_name: Field to remove. (string)

Example usage:: item_remove_field(item, field_name="contactgroups")

Returns: True on success

Raises:

`ValueError` if object is not found

`IOError` if save fails

`item_rename_field`(*item*, *old_field_name*, *new_field_name*)

Renames a field of a (currently existing) item.

Changes are immediate (i.e. there is no commit).

Args:

item: Item to modify.

old_field_name: Name of the field that will have its name changed. (string)

new_field_name: New name given to *old_field_name* (string)

Example usage:: item_rename_field(item, old_field_name="normal_check_interval", new_field_name="check_interval")

Returns: True on success

Raises:

`ValueError` if object is not found

`IOError` if save fails

`item_rewrite`(*item*, *str_new_item*)

Completely rewrites item with string provided.

Args:

item: Item that is to be rewritten

str_new_item: str representation of the new item

Examples:: item_rewrite(item, "define service {n name example-service n register 0 n }n")

Returns:

True on success

Raises:

`ValueError` if object is not found

`IOError` if save fails

listdir(**args*, ***kwargs*)
Wrapper around os.listdir

needs_reload()
Checks if the Nagios service needs a reload.

Returns:

True if Nagios service needs reload of cfg files

False if reload not needed or Nagios is not running

needs_reparse()
Checks if the Nagios configuration needs to be reparsed.

Returns:

True if any Nagios configuration file has changed since last parse()

open(*filename*, **args*, ***kwargs*)
Wrapper around global open()

Simply calls global open(filename, *args, **kwargs) and passes all arguments as they are received. See global open() function for more details.

parse(**args*, ***kw*)

parse_file(*filename*)
Parses a nagios object configuration file and returns lists of dictionaries.

This is more or less a wrapper around `config.parse_string()`, so reading documentation there is useful.

Args:

filename: Path to the file to parse (string)

Returns:

A list containing elements parsed by `parse_string()`

parse_maincfg(**args*, ***kw*)

parse_string(*string*, *filename='None'*)
Parses a string, and returns all object definitions in that string

Args:

string: A string containing one or more object definitions

filename (optional): If filename is provided, it will be referenced when raising exceptions

Examples:

```
>>> test_string = "define host {\nhost_name examplehost\n}\n"
>>> test_string += "define service {\nhost_name examplehost\nservice_description example ser
>>> c = config()
>>> result = c.parse_string(test_string)
>>> for i in result: print i.get('host_name'), i.get('service_description', None)
examplehost None
examplehost example service
```

Returns:

A list of dictionaries, that look like self.data

Raises:

`ParserError`

`print_conf`(*item*)
Return a string that can be used in a configuration file

Args:

item: Item to be dumped as a string.

Returns:

String representation of item.

`readlink`(*selfself*, **args*, ***kwargs*)
Wrapper around os.readlink

`remove`(**args*, ***kwargs*)
Wrapper around os.remove

`reset`()
Reinitializes the data of a parser instance to its default values.

`stat`(**args*, ***kwargs*)
Wrapper around os.stat

`write`(**args*, ***kw*)

exception `pynag.Parsers.`**`ConfigFileNotFound`**(*message*, *item=None*)
Bases: `pynag.Parsers.ParserError`

This exception is thrown if we cannot locate any nagios.cfg-style config file.

class `pynag.Parsers.`**`ExtraOptsParser`**(*section_name=None*, *config_file=None*)
Bases: `object`

Get Nagios Extra-Opts from a config file as specified by http://nagiosplugins.org/extra-opts

We could ALMOST use pythons ConfParser but nagios plugin team thought it would be a good idea to support multiple values per key, so a dict datatype no longer works.

Its a shame because we have to make our own "ini" parser as a result

Usage:

```
# cat /etc/nagios/plugins.ini
[main]
host_name = localhost
[other section]
host_name = example.com
# EOF

e = ExtraOptsParser(section_name='main', config_file='/etc/nagios/plugins.ini')
e.get('host_name')  # returns "localhost"
e.get_values()  # Returns a dict of all the extra opts
e.getlist('host_name')  # returns all values of host_name (if more than one were specified) in a
```

`get`(*option_name*, *default=<object object at 0x7fb0e9daa580>*)

Return the value of one specific option

Args:

option_name: The value set to this option will be returned

Returns:

The value of *option_name*

Raises:

`ValueError` when *option_name* cannot be found in options

`get_default_config_file`()

Return path to first readable extra-opt config-file found

According to the nagiosplugins extra-opts spec the search method is as follows:

1. Search for nagios.ini or nagios-plugins.ini in : splitted variable NAGIOS_CONFIG_PATH
2. Search in a predefined list of files
3. Return None if no config file is found

The method works as follows:

To quote the spec on NAGIOS_CONFIG_PATH:

"To use a custom location, set a NAGIOS_CONFIG_PATH environment variable to the set of directories that should be checked (this is a colon-separated list just like PATH). The first plugins.ini or nagios-plugins.ini file found in these directories will be used."

`get_default_section_name`()

According to extra-opts standard, the default should be filename of check script being run

`get_values`()

Returns a dict with all extra-options with the granted section_name and config_file

Results are in the form of:

```
{
  'key': ["possible","values"]
}
```

`getlist`(*option_name*, *default=<object object at 0x7fb0e9daa580>*)

Return a list of all values for option_name

Args:

option_name: All the values set to this option will be returned

Returns:

List containing all the options set to *option_name*

Raises:

`ValueError` when *option_name* cannot be found in options

`parse_file`(*filename*)

Parses an ini-file and returns a dict of the ini values.

The datatype returned is a list of sections where each section is a dict of values.

Args:

filename: Full path to the ini-file to be parsed.

Example the following the file:

```
[main]
name = this is a name
key = value
key = value2
```

Would return:

```
[
  {'main':
    {
      'name': ['this is a name'],
      'key': [value, value2]
    }
  },
]
```

parse_string (*string*)

Parses a string that is supposed to be ini-style format.

See `parse_file()` for more info

Args:

string: String to be parsed. Should be in ini-file format.

Returns:

Dictionnary containing all the sections of the ini-file and their respective data.

Raises:

`ParserError` when line does not follow the ini format.

standard_locations = ['/etc/nagios/plugins.ini', '/usr/local/nagios/etc/plugins.ini', '/usr/local/etc/nagios/plugins.ini', '

class pynag.Parsers.**Livestatus** (*livestatus_socket_path=None*, *nagios_cfg_file=None*, *authuser=None*)

Bases: `object`

Wrapper around MK-Livestatus

Example usage:

```
s = Livestatus()
for hostgroup s.get_hostgroups():
    print(hostgroup['name'], hostgroup['num_hosts'])
```

get (*table*, **args*, ***kwargs*)

Same as self.query('GET %s' % (table,))

Extra arguments will be appended to the query.

Args:

table: Table from which the data will be retrieved

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Example:

```
get('contacts', 'Columns: name alias')
```

Returns:

Answer from livestatus in python format.

get_contact(*contact_name*)

Performs a GET query for a particular contact

This performs:

```
'''GET contacts
Filter: contact_name = %s''' % contact_name
```

Args:

contact_name: name of the contact to obtain livestatus data from

Returns:

Answer from livestatus in python format.

get_contactgroup(*name*)

Performs a GET query for a particular contactgroup

This performs:

```
'''GET contactgroups
Filter: contactgroup_name = %s''' % contactgroup_name
```

Args:

contactgroup_name: name of the contactgroup to obtain livestatus data from

Returns:

Answer from livestatus in python format.

get_contactgroups(**args*, ***kwargs*)

Performs a GET query for all contactgroups

This performs:

```
'''GET contactgroups
%s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

get_contacts(**args*, ***kwargs*)

Performs a GET query for all contacts

This performs:

```
'''GET contacts
%s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

get_host (*host_name*)

Performs a GET query for a particular host

This performs:

```
'''GET hosts
Filter: host_name = %s''' % host_name
```

Args:

host_name: name of the host to obtain livestatus data from

Returns:

Answer from livestatus in python format.

get_hostgroup (*name*)

Performs a GET query for a particular hostgroup

This performs:

```
'''GET hostgroups
Filter: hostgroup_name = %s''' % hostgroup_name
```

Args:

hostgroup_name: name of the hostgroup to obtain livestatus data from

Returns:

Answer from livestatus in python format.

get_hostgroups (**args*, ***kwargs*)

Performs a GET query for all hostgroups

This performs:

```
'''GET hostgroups
%s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

get_hosts (**args*, ***kwargs*)

Performs a GET query for all hosts

This performs:

```
'''GET hosts %s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

get_service (*host_name*, *service_description*)

Performs a GET query for a particular service

This performs:

```
'''GET services
Filter: host_name = %s
Filter: service_description = %s''' % (host_name, service_description)
```

Args:

host_name: name of the host the target service is attached to.

service_description: Description of the service to obtain livestatus data from.

Returns:

Answer from livestatus in python format.

get_servicegroup(*name*)

Performs a GET query for a particular servicegroup

This performs:

```
'''GET servicegroups
Filter: servicegroup_name = %s''' % servicegroup_name
```

Args:

servicegroup_name: name of the servicegroup to obtain livestatus data from

Returns:

Answer from livestatus in python format.

get_servicegroups(**args*, ***kwargs*)

Performs a GET query for all servicegroups

This performs:

```
'''GET servicegroups
%s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

get_services(**args*, ***kwargs*)

Performs a GET query for all services

This performs:

```
'''GET services
%s %s''' % (*args, **kwargs)
```

Args:

args, kwargs: These will be appendend to the end of the query to perform additionnal instructions.

Returns:

Answer from livestatus in python format.

query(*query*, **args*, ***kwargs*)

Performs LQL queries the livestatus socket

Queries are corrected and convienient default data are added to the query before sending it to the socket.

Args:

query: Query to be passed to the livestatus socket (string)

args, kwargs: Additionnal parameters that will be sent to `pynag.Utils.grep_to_livestatus()`. The result will be appended to the query.

Returns:

Answer from livestatus. It will be in python format unless specified otherwise.

Raises:

`ParserError` if problems connecting to livestatus.

`test`(*raise_error=True*)

Test if connection to livestatus socket is working

Args:

raise_error: If set to True, raise exception if test fails,otherwise return False

Raises:

ParserError if raise_error == True and connection fails

Returns:

True – Connection is OK False – there are problems and raise_error==False

exception `pynag.Parsers.`**`LivestatusNotConfiguredException`**(*message*, *item=None*)

Bases: `pynag.Parsers.ParserError`

This exception is raised if we tried to autodiscover path to livestatus and failed

class `pynag.Parsers.`**`LogFiles`**(*maincfg=None*)

Bases: `object`

Parses Logfiles defined in nagios.cfg and allows easy access to its content

Content is stored in python-friendly arrays of dicts. Output should be more or less compatible with mk_livestatus log output

`get_flap_alerts`(***kwargs*)

Same as `get_log_entries()`, except return timeperiod transitions.

Takes same parameters.

`get_log_entries`(*start_time=None*, *end_time=None*, *strict=True*, *search=None*, ***kwargs*)

Get Parsed log entries for given timeperiod.

Args: start_time: unix timestamp. if None, return all entries from today

end_time: If specified, only fetch log entries older than this (unix timestamp)

strict: If True, only return entries between start_time and end_time, if False, then return entries that belong to same log files as given timeset

search: If provided, only return log entries that contain this string (case insensitive)

kwargs: All extra arguments are provided as filter on the log entries. f.e. host_name=”localhost”

Returns:

List of dicts

get_logfiles()

Returns a list with the fullpath to every log file used by nagios.

Lists are sorted by modification times. Newest logfile is at the front of the list so usually nagios.log comes first, followed by archivelogs

Returns:

List of strings

get_notifications(***kwargs*)

Same as get_log_entries(), except return only notifications. Takes same parameters.

get_state_history(*start_time=None*, *end_time=None*, *host_name=None*, *strict=True*, *service_description=None*)

Returns a list of dicts, with the state history of hosts and services.

Args:

start_time: unix timestamp. if None, return all entries from today

end_time: If specified, only fetch log entries older than this (unix timestamp)

host_name: If provided, only return log entries that contain this string (case insensitive)

service_description: If provided, only return log entries that contain this string (case insensitive)

Returns:

List of dicts with state history of hosts and services

class pynag.Parsers.**MultiSite**(**args*, ***kwargs*)

Bases: pynag.Parsers.Livestatus

Wrapps around multiple Livesatus instances and aggregates the results of queries.

Example:

```
>>> m = MultiSite()
>>> m.add_backend(path='/var/spool/nagios/livestatus.socket', name='local')
>>> m.add_backend(path='127.0.0.1:5992', name='remote')
```

add_backend(*path*, *name*)

Add a new livestatus backend to this instance.

Arguments: path (str): Path to file socket or remote address name (str): Friendly shortname for this backend

get_backend(*backend_name*)

Return one specific backend that has previously been added

get_backends()

Returns a list of mk_livestatus instances

Returns: list. List of mk_livestatus instances

get_contact(*contact_name*, *backend=None*)

Same as Livestatus.get_contact()

get_contactgroup(*contactgroup_name*, *backend=None*)

Same as Livestatus.get_contact()

get_host(*host_name*, *backend=None*)

Same as Livestatus.get_host()

get_hostgroup(*hostgroup_name*, *backend=None*)
Same as Livestatus.get_hostgroup()

get_service(*host_name*, *service_description*, *backend=None*)
Same as Livestatus.get_service()

get_servicegroup(*servicegroup_name*, *backend=None*)
Same as Livestatus.get_servicegroup()

query(*query*, **args*, ***kwargs*)
Behaves like mk_livestatus.query() except results are aggregated from multiple backends

Arguments: backend (str): If specified, fetch only data from this backend (see add_backend()) *args: Passed directly to mk_livestatus.query() **kwargs: Passed directly to mk_livestatus.query()

class pynag.Parsers.**ObjectCache**(*cfg_file=None*, *strict=False*)
Bases: pynag.Parsers.Config

Loads the configuration as it appears in objects.cache file

get_cfg_files()

exception pynag.Parsers.**ParserError**(*message*, *item=None*)
Bases: exceptions.Exception

ParserError is used for errors that the Parser has when parsing config.

Typical usecase when there is a critical error while trying to read configuration.

filename = None

line_start = None

message = None

class pynag.Parsers.**RetentionDat**(*filename=None*, *cfg_file=None*)
Bases: object

Easy way to parse the content of retention.dat

After calling parse() contents of retention.dat are kept in self.data

Example Usage:

```
r = retention()
r.parse()
print r
print r.data['info']
```

parse()
Parses your status.dat file and stores in a dictionary under self.data

Returns:

None

Raises:

ParserError: if problem arises while reading status.dat

ParserError: if status.dat is not found

IOError: if status.dat cannot be read

class pynag.Parsers.**SshConfig**(*host*, *username*, *password=None*, *cfg_file=None*)
Bases: pynag.Parsers.Config

Parse object configuration files from remote host via ssh

Uses python-paramiko for ssh connections.

access(**args*, ***kwargs*)
Wrapper around os.access only, via ssh connection

add_to_tar(*path*)

exists(*path*)
Wrapper around os.path.exists only, via ssh connection

get_cfg_files()

is_cached(*filename*)

isdir(*path*)
Behaves like os.path.isdir only, via ssh connection

isfile(*path*)
Behaves like os.path.isfile only, via ssh connection

islink(*path*)
Behaves like os.path.islink only, via ssh connection

listdir(**args*, ***kwargs*)
Wrapper around os.listdir but via ssh connection

open(*filename*, **args*, ***kwargs*)
Behaves like file.open only, via ssh connection

readlink(*path*)
Behaves like os.readlink only, via ssh connection

stat(**args*, ***kwargs*)
Wrapper around os.stat only, via ssh connection

class pynag.Parsers.**StatusDat**(*filename=None*, *cfg_file=None*)
Bases: pynag.Parsers.RetentionDat

Easy way to parse status.dat file from nagios

After calling parse() contents of status.dat are kept in status.data Example usage:

```
>>> s = status()
>>> s.parse()
>>> keys = s.data.keys()
>>> 'info' in keys
True
>>> 'programstatus' in keys
True
>>> for service in s.data.get('servicestatus',[]):
...     host_name=service.get('host_name', None)
...     description=service.get('service_description',None)
```

get_contactstatus(*contact_name*)
Returns a dictionary derived from status.dat for one particular contact

Args:

contact_name: *contact_name* field of the contact's status.dat data to parse and return as a dict.

Returns:

dict derived from status.dat for the contact.

Raises:

ValueError if object is not found

Example:

```
>>> s = status()
>>> s.get_contactstatus(contact_name='invalid_contact')
ValueError('invalid_contact',)
>>> first_contact = s.data['contactstatus'][0]['contact_name']
>>> s.get_contactstatus(first_contact)['contact_name'] == first_contact
True
```

get_hoststatus(*host_name*)

Returns a dictionary derived from status.dat for one particular contact

Args:

host_name: *host_name* field of the host's status.dat data to parse and return as a dict.

Returns:

dict derived from status.dat for the host.

Raises:

ValueError if object is not found

get_servicestatus(*host_name*, *service_description*)

Returns a dictionary derived from status.dat for one particular service

Args:

service_name: *service_name* field of the host's status.dat data to parse and return as a dict.

Returns:

dict derived from status.dat for the service.

Raises:

ValueError if object is not found

class pynag.Parsers.**config**(*cfg_file=None*, *strict=False*)

Bases: pynag.Parsers.Config

This class is here only for backwards compatibility. Use Config instead.

class pynag.Parsers.**mk_livestatus**(*livestatus_socket_path=None*, *nagios_cfg_file=None*, *authuser=None*)

Bases: pynag.Parsers.Livestatus

This class is here only for backwards compatibility. Use Livestatus instead.

class pynag.Parsers.**object_cache**(*cfg_file=None*, *strict=False*)

Bases: pynag.Parsers.ObjectCache

This class is here only for backwards compatibility. Use ObjectCache instead.

class pynag.Parsers.**retention**(*filename=None*, *cfg_file=None*)

Bases: pynag.Parsers.RetentionDat

This class is here only for backwards compatibility. Use RetentionDat instead.

class pynag.Parsers.**status**(*filename=None*, *cfg_file=None*)

Bases: pynag.Parsers.StatusDat

This class is here only for backwards compatibility. Use StatusDat instead.

2.2.4 Plugins Package

`Plugins` Package

Python Nagios extensions

class `pynag.Plugins.`**`PluginHelper`**

PluginHelper takes away some of the tedious work of writing Nagios plugins. Primary features include:

- Keep a collection of your plugin messages (queue for both summary and longoutput)
- Keep record of exit status
- Keep a collection of your metrics (for both perfdata and thresholds)
- Automatic Command-line arguments
- Make sure output of your plugin is within Plugin Developer Guidelines

Usage: p = PluginHelper() p.status(warning) p.add_summary('Example Plugin with warning status') p.add_metric('cpu load', '90') p.exit()

`add_long_output`(*message*)

Appends message to the end of Plugin long_output. Message does not need a suffix

Examples:

```
>>> p = PluginHelper()
>>> p.add_long_output('Status of sensor 1')
>>> p.add_long_output('* Temperature: OK')
>>> p.add_long_output('* Humidity: OK')
>>> p.get_long_output()
u'Status of sensor 1\n* Temperature: OK\n* Humidity: OK'
```

`add_metric`(*label=u'', value=u'', warn=u'', crit=u'', min=u'', max=u'', uom=u'', perfdatastring=None*)

Add numerical metric (will be outputted as nagios performanca data)

Examples:

```
>>> p = PluginHelper()
>>> p.add_metric(label="load1", value="7")
>>> p.add_metric(label="load5", value="5")
>>> p.add_metric(label="load15",value="2")
>>> p.get_perfdata()
"'load1'=7;;;; 'load5'=5;;;; 'load15'=2;;;;"
```

```
>>> p = PluginHelper()
>>> p.add_metric(perfdatastring="load1=6;;;;")
>>> p.add_metric(perfdatastring="load5=4;;;;")
>>> p.add_metric(perfdatastring="load15=1;;;;")
>>> p.get_perfdata()
"'load1'=6;;;; 'load5'=4;;;; 'load15'=1;;;;"
```

`add_option`(**args, **kwargs*)

Same as self.parser.add_option()

`add_status`(*new_status=None*)

Update exit status of the nagios plugin. This function will keep history of the worst status added

Examples: >>> p = PluginHelper() >>> p.add_status(0) # ok >>> p.add_status(2) # critical >>> p.add_status(1) # warning >>> p.get_status() # 2

```
>>> p = PluginHelper()
>>> p.add_status('warning')
>>> p.add_status('ok')
>>> p.get_status()
1
>>> p.add_status('okay')
Traceback (most recent call last):
...
Exception: Invalid status supplied "okay"
```

add_summary(*message*)
Adds message to Plugin Summary

arguments = None

check_all_metrics()
Checks all metrics (add_metric() against any thresholds set in self.options.thresholds or with –threshold from commandline)

check_metric(*metric_name*, *thresholds*)
Check one specific metric against a list of thresholds. Updates self.status() and writes to summary or longout as appropriate.

Arguments: metric_name – A string representing the name of the metric (the label part of the performance data) thresholds – a list in the form of [(level,range)] where range is a string in the format of "start..end"

Examples: >>> p = PluginHelper() >>> thresholds = [(warning,'2..5'), (critical,'5..inf')] >>> p.get_plugin_output() u'Unknown -' >>> p.add_metric('load15', '3') >>> p.check_metric('load15',thresholds) >>> p.get_plugin_output() u"Warning - Warning on load15 | 'load15'=3;@2:5;~:5;;"

```
>>> p = PluginHelper()
>>> thresholds = [(warning,'2..5'), (critical,'5..inf')]
>>> p.add_metric('load15', '3')
>>> p.verbose = True
>>> p.check_metric('load15',thresholds)
>>> p.get_plugin_output()
u"Warning - Warning on load15 | 'load15'=3;@2:5;~:5;;\nWarning on load15"
```

Invalid metric: >>> p = PluginHelper() >>> p.add_status(ok) >>> p.add_summary('Everythings fine!') >>> p.get_plugin_output() u'OK - Everythings fine!' >>> thresholds = [(warning,'2..5'), (critical,'5..inf')] >>> p.check_metric('never_added_metric', thresholds) >>> p.get_plugin_output() u'Unknown - Everythings fine!. Metric never_added_metric not found'

Invalid threshold: >>> p = PluginHelper() >>> thresholds = [(warning, 'invalid'), (critical,'5..inf')] >>> p.add_metric('load1', '10') >>> p.check_metric('load1', thresholds) Traceback (most recent call last): ... SystemExit: 3

Returns: None

convert_perfdata(*perfdata*)
Converts new threshold range format to old one. Returns None.

Examples: x..y -> x:y inf..y -> :y -inf..y -> :y x..inf -> x: -inf..inf -> :

debug(*message*)

exit(*exit_code=None*, *summary=None*, *long_output=None*, *perfdata=None*)

Print all collected output to screen and exit nagios style, no arguments are needed except if you want to override default behavior.

Arguments: summary – Is this text as the plugin summary instead of self.get_summary() long_output – Use this text as long_output instead of self.get_long_output() perfdata – Use this text instead of self.get_perfdata() exit_code – Use this exit code instead of self.status()

get_default_values(*section_name=None*, *config_file=None*)
Returns an optionParser.Values instance of all defaults after parsing extra opts config file

The Nagios extra-opts spec we use is the same as described here: http://nagiosplugins.org/extra-opts

Arguments

get_long_output()
Returns all long_output that has been added via add_long_output

get_metric(*label*)
Return one specific metric (PerfdataMetric object) with the specified label. Returns None if not found.

Example: >>> p = PluginHelper() >>> p.add_metric(label="load1", value="7") >>> p.add_metric(label="load15",value="2") >>> p.get_metric("load1") 'load1'=7;;;; >>> p.get_metric("unknown") # Returns None

get_perfdata()
Get perfdatastring for all valid perfdatametrics collected via add_perfdata

Examples: >>> p = PluginHelper() >>> p.add_metric(label="load1", value="7", warn="-inf..10", crit="10..inf") >>> p.add_metric(label="load5", value="5", warn="-inf..7", crit="7..inf") >>> p.add_metric(label="load15",value="2", warn="-inf..5", crit="5..inf") >>> p.get_perfdata() "'load1'=7;10:;~:10;; 'load5'=5;7:;~:7;; 'load15'=2;5:;~:5;;"

Example with legacy output (show_legacy should be set with a cmdline option): >>> p.show_legacy = True >>> p.get_perfdata() "'load1'=7;10:;~:10;; 'load5'=5;7:;~:7;; 'load15'=2;5:;~:5;;"

get_plugin_output(*exit_code=None*, *summary=None*, *long_output=None*, *perfdata=None*)
Get all plugin output as it would be printed to screen with self.exit()

Examples of functionality: >>> p = PluginHelper() >>> p.get_plugin_output() u'Unknown -'

```
>>> p = PluginHelper()
>>> p.add_summary('Testing')
>>> p.add_long_output('Long testing output')
>>> p.add_long_output('More output')
>>> p.get_plugin_output(exit_code=0)
u'OK - Testing\nLong testing output\nMore output'

>>> p = PluginHelper()
>>> p.add_summary('Testing')
>>> p.add_status(0)
>>> p.get_plugin_output()
u'OK - Testing'

>>> p = PluginHelper()
>>> p.show_status_in_summary = False
>>> p.add_summary('Testing')
>>> p.add_metric(label="load1", value="7")
>>> p.add_metric(label="load5", value="5")
>>> p.add_metric(label="load15",value="2")
>>> p.get_plugin_output(exit_code=0)
u"Testing | 'load1'=7;;;; 'load5'=5;;;; 'load15'=2;;;;"
```

```
>>> p = PluginHelper()
>>> p.show_status_in_summary = False
>>> p.add_summary('Testing')
>>> p.add_long_output('Long testing output')
>>> p.add_long_output('More output')
>>> p.add_metric(label="load1", value="7")
>>> p.add_metric(label="load5", value="5")
>>> p.add_metric(label="load15",value="2")
>>> p.get_plugin_output(exit_code=0)
u"Testing | 'load1'=7;;;; 'load5'=5;;;; 'load15'=2;;;;\nLong testing output\nMore output"
```

get_status()
Returns the worst nagios status (integer 0,1,2,3) that has been put with add_status()

If status has never been added, returns 3 for UNKNOWN

get_summary()

options = None

parse_arguments(*argument_list=None*)
Parsers commandline arguments, prints error if there is a syntax error.

Creates: self.options – As created by OptionParser.parse() self.arguments – As created by OptionParser.parse()

Arguments: argument_list – By default use sys.argv[1:], override only if you know what you are doing.

Returns: None

run_function(*function*, **args*, ***kwargs*)
Executes "function" and exits Nagios style with status "unkown" if there are any exceptions. The stacktrace will be in long_output.

Example: >>> p = PluginHelper() >>> p.add_status('ok') >>> p.get_status() 0 >>> p.add_status('okay') Traceback (most recent call last): ... Exception: Invalid status supplied "okay" >>> p.run_function(p.add_status, 'warning') >>> p.get_status() 1 >>> p.run_function(p.add_status, 'okay') Traceback (most recent call last): ... SystemExit: 3

set_long_output(*message*)
Overwrite current long_output with message

Example: >>> s = PluginHelper() >>> s.add_long_output('first long output') >>> s.set_long_output('Fatal error') >>> s.get_long_output() u'Fatal error'

set_summary(*message*)
Overwrite current summary with message

Example: >>> s = PluginHelper() >>> s.add_summary('first summary') >>> s.set_summary('Fatal error') >>> s.get_summary() u'Fatal error'

set_timeout(*seconds=50*)
Configures plugin to timeout after seconds number of seconds

show_debug = False

show_legacy = False

show_longoutput = True

show_perfdata = True

show_status_in_summary = True

show_summary = True

status (*new_status=None*)
Same as get_status() if new_status=None, otherwise call add_status(new_status)

thresholds = None

timeout = 58

verbose = False

pynag.Plugins.**check_range** (*value*, *range_threshold=None*)
Returns True if value is within range_threshold.

Format of range_threshold is according to: http://nagiosplug.sourceforge.net/developer-guidelines.html#THRESHOLDFORMAT

Arguments: value – Numerical value to check (i.e. 70) range – Range to compare against (i.e. 0:90)

Returns: True – If value is inside the range False – If value is outside the range (alert if this happens) False – if invalid value is specified

10 < 0 or > 10, (outside the range of {0 .. 10}) 10: < 10, (outside {10 .. ∞}) ~:10 > 10, (outside the range of {-∞ .. 10}) 10:20 < 10 or > 20, (outside the range of {10 .. 20}) @10:20 10 and 20, (inside the range of {10 .. 20}) 10 < 0 or > 10, (outside the range of {0 .. 10}) ——————————————————

# Example runs for doctest, False should mean alert >>> check_range(78, "90") # Example disk is 78% full, threshold is 90 True >>> check_range(5, 10) # Everything between 0 and 10 is True True >>> check_range(0, 10) # Everything between 0 and 10 is True True >>> check_range(10, 10) # Everything between 0 and 10 is True True >>> check_range(11, 10) # Everything between 0 and 10 is True False >>> check_range(-1, 10) # Everything between 0 and 10 is True False >>> check_range(-1, "~:10") # Everything Below 10 True >>> check_range(11, "10:") # Everything above 10 is True True >>> check_range(1, "10:") # Everything above 10 is True False >>> check_range(0, "5:10") # Everything between 5 and 10 is True False >>> check_range(0, "@5:10") # Everything outside 5:10 is True True >>> check_range(None) # Return False if value is not a number False >>> check_range("10000000 PX") # What happens on invalid input False >>> check_range("10000000", "invalid:invalid") # What happens on invalid range Traceback (most recent call last): ... PynagError: Invalid threshold format: invalid:invalid

pynag.Plugins.**check_threshold** (*value*, *warning=None*, *critical=None*)
Checks value against warning/critical and returns Nagios exit code.

Format of range_threshold is according to: http://nagiosplug.sourceforge.net/developer-guidelines.html#THRESHOLDFORMAT

Returns (in order of appearance): UNKNOWN int(3) – On errors or bad input CRITICAL int(2) – if value is within critical threshold WARNING int(1) – If value is within warning threshold OK int(0) – If value is outside both tresholds

Arguments: value – value to check warning – warning range critical – critical range

# Example Usage: >>> check_threshold(88, warning="0:90", critical="0:95") 0 >>> check_threshold(92, warning=":90", critical=":95") 1 >>> check_threshold(96, warning=":90", critical=":95") 2

class pynag.Plugins.**simple** (*shortname=None*, *version=None*, *blurb=None*, *extra=None*, *url=None*, *license=None*, *plugin=None*, *timeout=15*, *must_threshold=True*)
Nagios plugin helper library based on Nagios::Plugin

Sample usage

from pynag.Plugins import WARNING, CRITICAL, OK, UNKNOWN, simple as Plugin

# Create plugin object np = Plugin() # Add arguments np.add_arg("d", "disk") # Do activate plugin np.activate() ... check stuff, np['disk'] to address variable assigned above... # Add a status message and severity

np.add_message(WARNING, "Disk nearing capacity") # Get parsed code and messages (code, message) = np.check_messages() # Return information and exit nagios_exit(code, message)

`activate()`
Parse out all command line options and get ready to process the plugin. This should be run after argument preps

`add_arg`(*spec_abbr*, *spec*, *help_text*, *required=1*, *action=u'store'*)
Add an argument to be handled by the option parser. By default, the arg is not required.

required = optional parameter action = [store, append, store_true]

`add_message`(*code*, *message*)
Add a message with code to the object. May be called multiple times. The messages added are checked by check_messages, following.

Only CRITICAL, WARNING, OK and UNKNOWN are accepted as valid codes.

`add_perfdata`(*label*, *value*, *uom=None*, *warn=None*, *crit=None*, *minimum=None*, *maximum=None*)
Append perfdata string to the end of the message

`check_messages`(*joinstr=u' '*, *joinallstr=None*)
Check the current set of messages and return an appropriate nagios return code and/or a result message. In scalar context, returns only a return code; in list context returns both a return code and an output message, suitable for passing directly to nagios_exit()

joinstr = string A string used to join the relevant array to generate the message string returned in list context i.e. if the 'critical' array is non-empty, check_messages would return:

joinstr.join(critical)

joinallstr = string By default, only one set of messages are joined and returned in the result message i.e. if the result is CRITICAL, only the 'critical' messages are included in the result; if WARNING, only the 'warning' messages are included; if OK, the 'ok' messages are included (if supplied) i.e. the default is to return an 'errors-only' type message.

If joinallstr is supplied, however, it will be used as a string to join the resultant critical, warning, and ok messages together i.e. all messages are joined and returned.

`check_perfdata_as_metric()`

`check_range`(*value*)
Check if a value is within a given range. This should replace change_threshold eventually. Exits with appropriate exit code given the range.

Taken from: http://nagiosplug.sourceforge.net/developer-guidelines.html Range definition

Generate an alert if x... 10 < 0 or > 10, (outside the range of {0 .. 10}) 10: < 10, (outside {10 .. #}) ~:10 > 10, (outside the range of {-# .. 10}) 10:20 < 10 or > 20, (outside the range of {10 .. 20}) @10:20 # 10 and # 20, (inside the range of {10 .. 20})

`code_string2int`(*code_text*)
Changes CRITICAL, WARNING, OK and UNKNOWN code_text to integer representation for use within add_message() and nagios_exit()

`nagios_exit`(*code_text*, *message*)
Exit with exit_code, message, and optionally perfdata

`perfdata_string()`

`send_nsca`(**args*, ***kwargs*)
Wrapper around pynag.Utils.send_nsca - here for backwards compatibility

`new_threshold_syntax` Module

These are helper functions and implementation of proposed new threshold format for nagios plugins according to: http://nagiosplugins.org/rfc/new_threshold_syntax

In short, plugins should implement a –threshold option which takes argument in form of: # metric={metric},ok={range},warn={range},crit={range},unit={unit}prefix={SI prefix}

Example: –treshold metric=load1,ok=0..5,warning=5..10,critical=10..inf

`pynag.Plugins.new_threshold_syntax.`**`check_range`**(*value*, *range*)
Returns True if value is within range, else False

Arguments: value – Numerical value to check, can be any number range – string in the format of "start..end"

Examples: >>> check_range(5, "0..10") True >>> check_range(11, "0..10") False

`pynag.Plugins.new_threshold_syntax.`**`check_threshold`**(*value*, *ok=None*, *warning=None*, *critical=None*)
Checks value against warning/critical and returns Nagios exit code.

Format of range_threshold is according to: http://nagiosplugins.org/rfc/new_threshold_syntax

This function returns (in order of appearance): int(0) - If no levels are specified, return OK int(3) - If any invalid input provided, return UNKNOWN int(0) - If an ok level is specified and value is within range, return OK int(2) - If a critical level is specified and value is within range, return CRITICAL int(1) - If a warning level is specified and value is within range, return WARNING int(2) - If an ok level is specified, return CRITICAL int(0) - Otherwise return OK

Arguments: value – value to check ok – ok range warning – warning range critical – critical range

# Example Usage: >>> check_threshold(88, warning="90..95", critical="95..100") 0 >>> check_threshold(92, warning="90..95", critical="95..100") 1 >>> check_threshold(96, warning="90..95", critical="95..100") 2

`pynag.Plugins.new_threshold_syntax.`**`parse_threshold`**(*threshold*)
takes a threshold string as an input and returns a hash map of options and values

Examples:

```
>>> parse_threshold('metric=disk_usage,ok=0..90,warning=90..95,critical=95.100')
{'thresholds': [(0, '0..90'), (1, '90..95'), (2, '95.100')], 'metric': 'disk_usage'}
```

2.2.5 Utils Package

`Utils` Package

Misc utility classes and helper functions for pynag

This module contains misc classes and conveninence functions that are used throughout the pynag library.

class `pynag.Utils.`**`AttributeList`**(*value=None*)
Bases: `object`

Parse a list of nagios attributes into a parsable format. (e. contact_groups)

This makes it handy to mangle with nagios attribute values that are in a comma seperated format.

Typical comma-seperated format in nagios configuration files looks something like this:

```
contact_groups     +group1,group2,group3
```

Example:

```
>>> i = AttributeList('+group1,group2,group3')
>>> i.operator
'+'
>>> i.fields
['group1', 'group2', 'group3']

# if your data is already in a list format you can use it directly:
>>> i = AttributeList(['group1', 'group2', 'group3'])
>>> i.fields
['group1', 'group2', 'group3']

# white spaces will be stripped from all fields
>>> i = AttributeList('+group1, group2')
>>> i
+group1,group2
>>> i.fields
['group1', 'group2']
```

append(*object*)

Same as list.append():

Args:

object: Item to append into self.fields (typically a string)

Example:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.append('group5')
>>> i.fields
['group1', 'group2', 'group3', 'group5']
```

count(*value*)

Same as list.count()

Args: value: Any object that might exist in self.fields (string)

Returns: The number of occurances that 'value' has in self.fields

Example:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.count('group3')
1
```

extend(*iterable*)

Same as list.extend()

Args: iterable: Any iterable that list.extend() supports

Example:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.extend(['group4', 'group5'])
>>> i.fields
['group1', 'group2', 'group3', 'group4', 'group5']
```

index(*value*, *start=0*, *stop=None*)

Same as list.index()

Args: value: object to look for in self.fields

start: start at this index point

stop: stop at this index point

Returns: The index of 'value' (integer)

Examples:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.index('group2')
1
>>> i.index('group3', 2, 5)
2
```

insert(*index*, *object*)

Same as list.insert()

Args:

object: Any object that will be inserted into self.fields (usually a string)

Example:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.insert(1, 'group4')
>>> i.fields
['group1', 'group4', 'group2', 'group3']
```

remove(*value*)

Same as list.remove()

Args: value: The object that is to be removed

Examples:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.remove('group3')
>>> i.fields
['group1', 'group2']
```

reverse()

Same as list.reverse()

Examples:

```
>>> i = AttributeList('group1,group2,group3')
>>> i.reverse()
>>> i.fields
['group3', 'group2', 'group1']
```

sort()

Same as list.sort()

Examples:

```
>>> i = AttributeList('group3,group1,group2')
>>> i.sort()
>>> print(i.fields)
['group1', 'group2', 'group3']
```

class `pynag.Utils.`**`GitRepo`**(*directory*, *auto_init=True*, *author_name='Pynag User'*, *author_email=None*)

Bases: `object`

`add`(*filename*)

Run git add on filename

Args: filename (str): name of one file to add,

Returns: str. The stdout from "git add" shell command.

`commit`(*message='commited by pynag'*, *filelist=None*, *author=None*)

Commit files with "git commit"

Args:

message (str): Message used for the git commit

filelist (list of strings): List of filenames to commit (if None, then commit all files in the repo)

author (str): Author to use for git commit. If any is specified, overwrite self.author_name and self.author_email

Returns: stdout from the "git commit" shell command.

`diff`(*commit_id_or_filename=None*)

Returns diff (as outputted by "git diff") for filename or commit id.

If commit_id_or_filename is not specified. show diff against all uncommited files.

Args: commit_id_or_filename (str): git commit id or file to diff with

Returns: str. git diff for filename or commit id

Raises: PynagError: Invalid commit id or filename was given

`get_uncommited_files`()

Returns a list of files that are have unstaged changes

Returns: List. All files that have unstaged changes.

`get_valid_commits`()

Returns a list of all commit ids from git log

Returns: List of all valid commit hashes

`init`()

Initilizes a new git repo (i.e. run "git init")

`is_dirty`(*filename*)

Returns True if filename needs to be committed to git

Args:

filename (str): file to check

`is_up_to_date`()

Returns True if all files in git repo are fully commited

Returns:

bool. Git repo is up-to-date True – All files are commited

False – At least one file is not commited

log(***kwargs*)

Returns a log of previous commits. Log is is a list of dict objects.

Any arguments provided will be passed directly to pynag.Utils.grep() to filter the results.

Args: kwargs: Arguments passed to pynag.Utils.grep()

Returns: List of dicts. Log of previous commits.

Examples: self.log(author_name='nagiosadmin')

self.log(comment__contains='localhost')

pre_save(*object_definition*, *message*)

Commits object_definition.get_filename() if it has any changes.

This function is called by `pynag.Model.EventHandlers` before calling `pynag.Utils.GitRepo.save()`

Args:

object_definition (pynag.Model.ObjectDefinition): object to commit changes

message (str): git commit message as specified in `git commit -m`

A message from the authors: *"Since this is still here, either i forgot to remove it, or because it is here for backwards compatibility, palli"*

revert(*commit*)

Revert some existing commits works like "git revert"

save(*object_definition*, *message*)

Commits object_definition.get_filename() if it has any changes. This function is called by `pynag.Model.EventHandlers`

Args:

object_definition (pynag.Model.ObjectDefinition): object to commit changes

message (str): git commit message as specified in `git commit -m`

show(*commit_id*)

Returns output from "git show" for a specified commit_id

Args: commit_id (str): Commit id of the commit to display (`git show`)

Returns: str. Output of `git show commit_id`

Raises: PynagError: Invalid commit_id was given

write(*object_definition*, *message*)

This method is called whenever `pynag.Model.EventHandlers` is called.

Args:

object_definition (pynag.Model.ObjectDefinition): Object to write to file.

message (str): git commit message as specified in `git commit -m`

class `pynag.Utils.`**`PerfData`**(*perfdatastring=''*)

Bases: `object`

Data Structure for a nagios perfdata string with multiple perfdata metric

Example string:

```
>>> perf = PerfData("load1=10 load2=10 load3=20 'label with spaces'=5")
>>> perf.metrics
['load1'=10;;;;, 'load2'=10;;;;, 'load3'=20;;;;, 'label with spaces'=5;;;;]
>>> for i in perf.metrics: print("%s %s" % (i.label, i.value))
load1 10
load2 10
load3 20
label with spaces 5
```

add_perfdatametric(*perfdatastring='', label='', value='', warn='', crit='', min='', max='', uom=''*)

Add a new perfdatametric to existing list of metrics.

Args:

perfdatastring (str): Complete perfdata string

label (str): Label section of the perfdata string

value (str): Value section of the perfdata string

warn (str): WARNING threshold

crit (str): CRITICAL threshold

min (str): Minimal value of control

max (str): Maximal value of control

uom (str): Measure unit (octets, bits/s, volts, ...)

Example:

```
>>> s = PerfData()
>>> s.add_perfdatametric("a=1")
>>> s.add_perfdatametric(label="utilization",value="10",uom="%")
```

get_perfdatametric(*metric_name*)

Get one specific perfdatametric

Args: metric_name (str): Name of the metric to return

Example:

```
>>> s = PerfData("cpu=90% memory=50% disk_usage=20%")
>>> my_metric = s.get_perfdatametric('cpu')
>>> my_metric.label, my_metric.value
('cpu', '90')
```

is_valid()

Returns True if the every metric in the string is valid

Example usage:

```
>>> PerfData("load1=10 load2=10 load3=20").is_valid()
True
>>> PerfData("10b").is_valid()
False
>>> PerfData("load1=").is_valid()
False
>>> PerfData("load1=10 10").is_valid()
False
```

reconsile_thresholds()
Convert all thresholds in new_threshold_syntax to the standard one

class pynag.Utils.**PerfDataMetric**(*perfdatastring=''*, *label=''*, *value=''*, *warn=''*, *crit=''*, *min=''*, *max=''*, *uom=''*)
Bases: object

Data structure for one single Nagios Perfdata Metric

Attributes:

perfdatastring (str): Complete perfdata string

label (str): Label section of the perfdata string

value (str): Value section of the perfdata string

warn (str): WARNING threshold

crit (str): CRITICAL threshold

min (str): Minimal value of control

max (str): Maximal value of control

uom (str): Measure unit (octets, bits/s, volts, ...)

crit = ''

get_dict()
Returns a dictionary which contains this class' attributes.

Returned dict example:

```
{
    'label': self.label,
    'value': self.value,
    'uom': self.uom,
    'warn': self.warn,
    'crit': self.crit,
    'min': self.min,
    'max': self.max,
}
```

get_status()
Return nagios-style exit code (int 0-3) by comparing

Example:

self.value with self.warn and self.crit

```
>>> PerfDataMetric("label1=10;20;30").get_status()
0
>>> PerfDataMetric("label2=25;20;30").get_status()
1
>>> PerfDataMetric("label3=35;20;30").get_status()
2
```

Invalid metrics always return unknown

```
>>> PerfDataMetric("label3=35;invalid_metric").get_status()
3
```

is_valid()
Returns True if all Performance data is valid. Otherwise False

Example Usage:

```
>>> PerfDataMetric("load1=2").is_valid()
True
>>> PerfDataMetric("load1").is_valid()
False
>>> PerfDataMetric('').is_valid()
False
>>> PerfDataMetric('invalid_value=invalid').is_valid()
False
>>> PerfDataMetric('invalid_min=0;0;0;min;0').is_valid()
False
>>> PerfDataMetric('invalid_min=0;0;0;0;max').is_valid()
False
>>> PerfDataMetric('label with spaces=0').is_valid()
False
>>> PerfDataMetric("'label with spaces=0'").is_valid()
False
```

label = ''

max = ''

min = ''

reconsile_thresholds()

Convert threshold from new threshold syntax to current one.

For backwards compatibility

split_value_and_uom(*value*)

Example:

get value="10M" and return (10,"M")

```
>>> p = PerfDataMetric()
>>> p.split_value_and_uom( "10" )
('10', '')
>>> p.split_value_and_uom( "10c" )
('10', 'c')
>>> p.split_value_and_uom( "10B" )
('10', 'B')
>>> p.split_value_and_uom( "10MB" )
('10', 'MB')
>>> p.split_value_and_uom( "10KB" )
('10', 'KB')
>>> p.split_value_and_uom( "10TB" )
('10', 'TB')
>>> p.split_value_and_uom( "10%" )
('10', '%')
>>> p.split_value_and_uom( "10s" )
('10', 's')
>>> p.split_value_and_uom( "10us" )
('10', 'us')
>>> p.split_value_and_uom( "10ms" )
('10', 'ms')
```

uom = ''

value = ''

warn = ''

class `pynag.Utils.`**`PluginOutput`**(*stdout*)

This class parses a typical stdout from a nagios plugin

It splits the output into the following fields:

- Summary
- Long Output
- Perfdata

Attributes:

summary (str): Summary returned by the plugin check

long_output (str)

perfdata (str): Data returned by the plugin as a string

parsed_perfdata: perfdata parsed and split

Example Usage:

```
>>> p = PluginOutput("Everything is ok | load1=15 load2=10")
>>> p.summary
'Everything is ok '
>>> p.long_output
''
>>> p.perfdata
'load1=15 load2=10'
>>> p.parsed_perfdata.metrics
['load1'=15;;;;, 'load2'=10;;;;]
```

`long_output` = **None**

`parsed_perfdata` = **None**

`perfdata` = **None**

`summary` = **None**

exception `pynag.Utils.`**`PynagError`**(*message*, *errorcode=None*, *errorstring=None*, **args*, ***kwargs*)

Bases: `exceptions.Exception`

The default pynag exception.

Exceptions raised within the pynag library should aim to inherit this one.

`pynag.Utils.`**`cache_only`**(*func*)

class `pynag.Utils.`**`defaultdict`**(*default_factory=None*, **a*, ***kw*)

Bases: `dict`

This is an alternative implementation of collections.defaultdict.

Used as a fallback if using python 2.4 or older.

Usage:

```
try:
    from collections import defaultdict
except ImportError:
    from pynag.Utils import defaultdict
```

`copy`()

pynag.Utils.**grep**(*objects*, ***kwargs*)

Returns all the elements from array that match the keywords in **kwargs

See documentation for pynag.Model.ObjectDefinition.objects.filter() for example how to use this.

Arguments:

objects (list of dict): list to be searched

kwargs (str): Any search argument provided will be checked against every dict

Examples:

```
array = [
{'host_name': 'examplehost', 'state':0},
{'host_name': 'example2', 'state':1},
]
grep_dict(array, state=0)
# should return [{'host_name': 'examplehost', 'state':0},]
```

pynag.Utils.**grep_to_livestatus**(**args*, ***kwargs*)

Converts from pynag style grep syntax to livestatus filter syntax.

Example:

```
>>> grep_to_livestatus(host_name='test')
['Filter: host_name = test']
>>> grep_to_livestatus(service_description__contains='serv')
['Filter: service_description ~ serv']
>>> grep_to_livestatus(service_description__isnot='serv')
['Filter: service_description != serv']
>>> grep_to_livestatus(service_description__contains=['serv','check'])
['Filter: service_description ~ serv']
>>> grep_to_livestatus(service_description__contains='foo', contacts__has_field='admin')
['Filter: contacts >= admin', 'Filter: service_description ~ foo']
>>> grep_to_livestatus(service_description__has_field='foo')
['Filter: service_description >= foo']
>>> grep_to_livestatus(service_description__startswith='foo')
['Filter: service_description ~ ^foo']
>>> grep_to_livestatus(service_description__endswith='foo')
['Filter: service_description ~ foo$']
```

pynag.Utils.**reconsile_threshold**(*threshold_range*)

Take threshold string as and normalize it to the format supported by plugin development team

The input (usually a string in the form of 'the new threshold syntax') is a string in the form of x..y

The output will be a compatible string in the older nagios plugin format @x:y

Examples:

```
>>> reconsile_threshold("0..5")
'@0:5'
>>> reconsile_threshold("inf..5")
'5:'
>>> reconsile_threshold("5..inf")
'~:5'
>>> reconsile_threshold("inf..inf")
'@~:'
>>> reconsile_threshold("^0..5")
'0:5'
>>> reconsile_threshold("10..20")
```

```
'@10:20'
>>> reconsile_threshold("10..inf")
'~:10'
```

pynag.Utils.**runCommand**(*command*, *raise_error_on_fail=False*, *shell=True*, *env=None*)

Run command from the shell prompt. Wrapper around subprocess.

Args:

command (str): string containing the command line to run

raise_error_on_fail (bool): Raise PynagError if returncode > 0

Returns:

str: stdout/stderr of the command run

Raises:

PynagError if returncode > 0

pynag.Utils.**send_nsca**(*code*, *message*, *nscahost*, *hostname=None*, *service=None*, *nscabin='send_nsca'*, *nscaconf=None*)

Send data via send_nsca for passive service checks

Args:

code (int): Return code of plugin.

message (str): Message to pass back.

nscahost (str): Hostname or IP address of NSCA server.

hostname (str): Hostname the check results apply to.

service (str): Service the check results apply to.

nscabin (str): Location of send_nsca binary. If none specified whatever is in the path will be used.

nscaconf (str): Location of the NSCA configuration to use if any.

Returns:

[result,stdout,stderr] of the command being run

pynag.Utils.**synchronized**(*lock*)

Synchronization decorator

Use this to make a multi-threaded method synchronized and thread-safe.

Use the decorator like so:

```
@pynag.Utils.synchronized(pynag.Utils.rlock)
```

class pynag.Utils.**CheckResult**(*nagios_result_dir*, *file_time=1406146591.95924*)

Bases: object

Methods for creating host and service checkresults for nagios processing

host_result(*host_name*, ***kwargs*)

Create a service checkresult

Any kwarg will be added to the checkresult

Args: host_name (str) service_descritpion (str)

Kwargs: check_type (int): active(0) or passive(1) check_options (int) scheduled_check (int) reschedule_check (int) latency (float) start_time (float) finish_time (float) early_timeout (int) exited_ok (int) return_code (int) output (str): plugin output

`service_result` (*host_name*, *service_description*, ***kwargs*)

Create a service checkresult

Any kwarg will be added to the checkresult

Args: host_name (str) service_descritpion (str)

Kwargs: check_type (int): active(0) or passive(1) check_options (int) scheduled_check (int) reschedule_check (int) latency (float) start_time (float) finish_time (float) early_timeout (int) exited_ok (int) return_code (int) output (str): plugin output

`submit` ()

Submits the results to nagios

The importer

General Utilities from importing nagios objects. Currently .csv files are supported

Either execute this script standalone from the command line or use it as a python library like so:

```
>>> from pynag.Utils import importer
>>> pynag_objects = importer.import_from_csv_file(filename='foo', seperator=',')
>>> for i in pynag_objects:
...     i.save()
```

`pynag.Utils.importer.dict_to_pynag_objects` (*dict_list*, *object_type=None*)

Take a list of dictionaries, return a list of pynag.Model objects.

Args: dict_list: List of dictionaries that represent pynag objects object_type: Use this object type as default, if it is not specified in dict_list

Returns: List of pynag objects

`pynag.Utils.importer.import_from_csv_file` (*filename*, *seperator=', '*, *object_type=None*)

Parses filename and returns a list of pynag objects.

Args: filename: Path to a file seperator: use this symbol to seperate columns in the file object_type: Assume this object_type if there is no object_type column

`pynag.Utils.importer.parse_arguments` ()

Parse command line arguments

`pynag.Utils.importer.parse_csv_file` (*filename*, *seperator=', '*)

Parse filename and return a dict representing its contents

`pynag.Utils.importer.parse_csv_string` (*csv_string*, *seperator=', '*)

Parse csv string and return a dict representing its contents

CHAPTER 3

The pynag command line

3.1 NAME

3.1.1 SYNOPSIS

pynag <sub-command> [options] [arguments]

3.1.2 DESCRIPTION

pynag is a command-line utility that can be used to view or change current nagios configuration.

3.1.3 sub-commands

list

print to screen nagios configuration objects as specified by a WHERE clause

pynag list [attribute1] [attribute2] [WHERE ...]

update

modify specific attributes of nagios objects as specified by a WHERE and SET clause

pynag update set attr1=value WHERE attr=value and attr=value

delete

Delete objects from nagios configuration as specified by a WHERE clause

pynag delete delete <WHERE ...>

add

Add a new object definition

pynag add <object_type> <attr1=value1> [attr2=value2]

copy

Copy objects, specifiying which attributes to change

pynag copy <WHERE ...> <SET attr1=value1 [attr2=value2] ...>

execute

Executes the currently configured check command for a host or a service

pynag execute <host_name> [service_description]

config

modify values in main nagios configuration file (nagios.cfg)

pynag config [–set <attribute=value>] [–old_value=attribute]
pynag config [–append <attribute=value>] [–old_value=attribute]
pynag config [–remove <attribute>] [–old_value=attribute]
pynag config [–get <attribute>]

3.1.4 WHERE statements

Some Subcommands use WHERE statements to filter which objects to work with. Where has certain similarity with SQL syntax.

Syntax:

WHERE <attr=value> [AND attr=value] [OR attr=value]
[another where statement]

where "attr" is any nagios attribute (i.e. host_name or service_description).

Example:

pynag list WHERE host_name=localhost and object_type=service
pynag list WHERE object_type=host or object_type=service

Any search attributes have the same syntax as the pynag filter. For example these work just fine:

pynag list WHERE host_name__contains=production
pynag list WHERE host_name__startswith=prod
pynag list WHERE host_name__notcontains=test
pynag list host_name address WHERE address__exists=True
pynag list host_name WHERE register__isnot=0

The pynag filter supports few parameters that are not just attributes.

Example:

- filename – The filename which the object belongs
- id – pynag unique identifier for the object
- effective_command_line – command which nagios will execute

Of course these can be combined with the pynag filter syntax:

pynag list where filename__startswith=/etc/nagios/conf.d/
pynag list host_name service_description effective_command_line

For detailed description of the filter see pydoc for pynag.Model.ObjectDefintion.filter()

3.1.5 SET statements

Subcommands that use SET statements (like update or copy) use them a list of attributes change for a specific object.

Syntax:

SET <attr1=value1> [attr2=value2] [...]

Example:

pynag update SET address=127.0.0.1 WHERE host_name=localhost and object_type=host

3.1.6 EXAMPLES

List all services that have "myhost" as a host_name

pynag list host_name service_description WHERE host_name=myhost and object_type=service

Set check_period to 24x7 on all services that belong to host "myhost"

pynag update set check_period=24x7 WHERE host_name=myhost

list examples

pynag list host_name address WHERE object_type=host
pynag list host_name service_description WHERE host_name=examplehost and object_type=service

update examples

pynag update SET host_name=newhostname WHERE host_name=oldhostname
pynag update SET address=127.0.0.1 WHERE host_name='examplehost.example.com' and object_type=host

copy examples

pynag copy SET host_name=newhostname WHERE host_name=oldhostname
pynag copy SET address=127.0.0.1 WHERE host_name='examplehost.example.com' and object_type=host

add examples

pynag add host host_name=examplehost use=generic-host address=127.0.0.1
pynag add service service_description="Test Service" use="check_nrpe" host_name="localhost"

delete examples

pynag delete where object_type=service and host_name='mydeprecated_host'
pynag delete where filename__startswith='/etc/nagios/myoldhosts'

execute examples

pynag execute localhost
pynag execute localhost "Disk Space

3.1.7 Additional Resources

See http://github.com/pynag/pynag.git for more information.