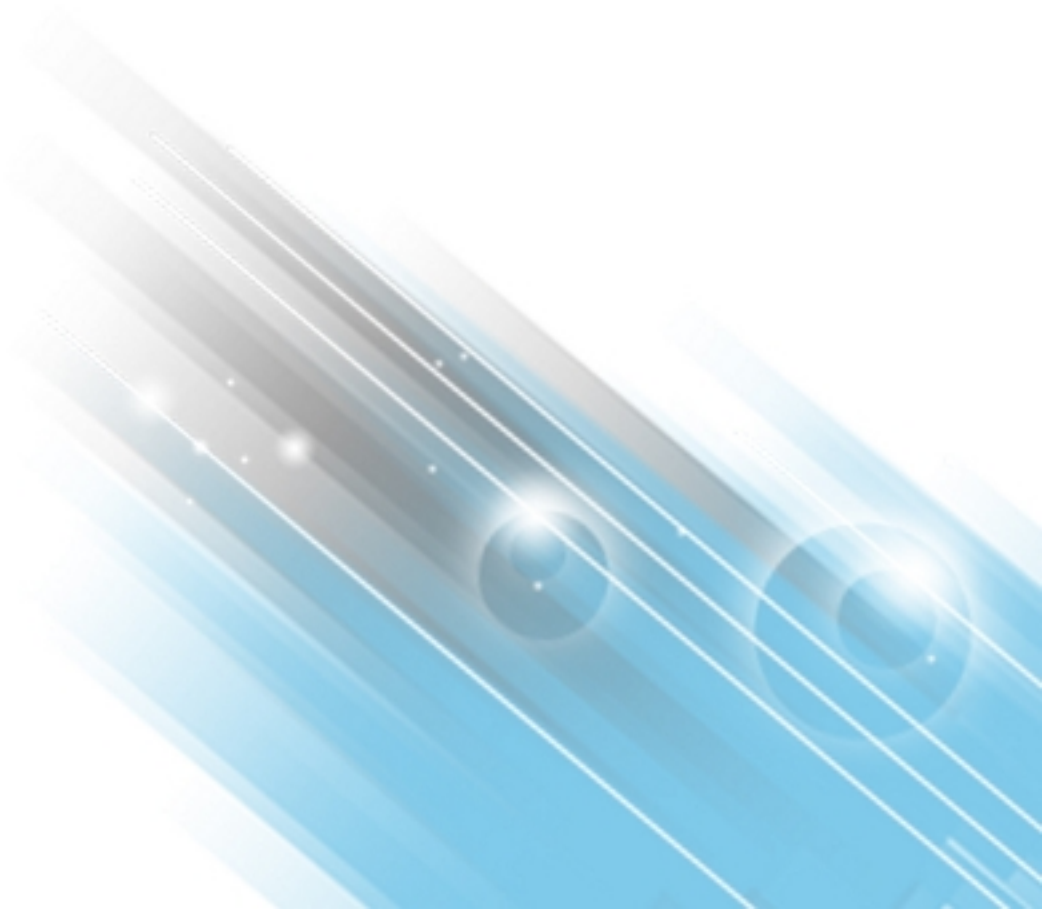
HP Diagnostics

Software Version: 9.24

Python Agent Guide



Document Release Date: January 2015
Software Release Date: January 2015

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

Java is a registered trademark of Oracle and/or its affiliates.

Oracle® is a registered trademark of Oracle and/or its affiliates.

Acknowledgements

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product includes software developed by the Spice Group (http://spice.codehaus.org).

For information about open source and third-party license agreements, see the *Open Source and Third-Party Software License Agreements* document in the Documentation directory on the product installation media.

Documentation Updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to: https://softwaresupport.hp.com/group/softwaresupport/search-result?keyword=.

This site requires an HP Passport account. If you do not have one, click the **Create an account** button on the HP Passport Sign in page.

Support

Visit the HP Software Support web site at: **https://softwaresupport.hp.com**

This web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software Support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support web site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts

- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract. To register for an HP Passport ID, go to **https://softwaresupport.hp.com** and click **Register**.

To find more information about access levels, go to: **https://softwaresupport.hp.com/web/softwaresupport/access-levels**

HP Software Solutions & Integrations and Best Practices

Visit HP Software Solutions Now at **https://h20230.www2.hp.com/sc/solutions/index.jsp** to explore how the products in the HP Software catalog work together, exchange information, and solve business needs.

Visit the Cross Portfolio Best Practices Library at **https://hpln.hp.com/group/best-practices-hpsw** to access a wide variety of best practice documents and materials.

Contents

Welcome to This Guide

Welcome to the HP Diagnostics Python Agent Guide. This guide describes how to install, configure and use the Diagnostics Python Agent.

The HP Diagnostics application comes with the following documentation:

- Diagnostics User Guide and Online Help
- Diagnostics Server Installation and Administration Guide
- Diagnostics FAQ document
- Diagnostics Data Model and Query API Guide
- Diagnostics Release Notes
- Diagnostics Java Agent Guide
- Diagnostics .NET Agent Guide
- Diagnostics Python Agent Guide

Diagnostics Documentation

HP Diagnostics includes the following documentation. Unless specified otherwise, the guides are in PDF format only and are available as downloads from the HP Software Support site (at https://softwaresupport.hp.com).

Diagnostics User Guide and Online Help. Explains how to choose and interpret the Diagnostics views in the Diagnostics Enterprise UI to analyze your monitored applications. To access the online help for Diagnostics, choose **Help** > **Help** in the Diagnostics Enterprise UI. If Diagnostics is integrated with another HP Software product the online help is also available through that product's Help menu. The User Guide is a PDF version of the online help and their content is identical. The User Guide is available from the Diagnostics online help Home page, from the Windows Start menu (**Start > Programs > HP Diagnostics Server > User Guide)**, or from the Diagnostics Server installation directory.

Diagnostics Server Installation and Administration Guide. Explains how to plan a Diagnostics deployment, and how to install and maintain a Diagnostics Server.

The following Agent guides contain content that supports agent installation, setup and configuration.

- **Diagnostics Java Agent Guide.** Describes how to install, configure, and use the Diagnostics Java Agent and the Diagnostics Profiler for Java.

- **Diagnostics .NET Agent Guide.** Describes how to install, configure, and use the Diagnostics .NET Agent and Diagnostics Profiler for .NET.
- **Diagnostics Python Agent Guide.** Describes how to install, configure, and use the Diagnostics Python Agent. The Guide (in PDF format) is also available from the Profiler UI help.

Diagnostics Collector Guide. Explains how to install and configure a Diagnostics Collector.

Diagnostics System Requirements and Support Matrixes Guide. Describes the system requirements for the various Diagnostics components.

Release Notes. Provides last-minute new information and known issues about each version of Diagnostics. The PDF file is also located in the Diagnostics installation disk root directory.

Diagnostics Data Model and Query API. Describes the Diagnostics data model and the query API you can use to access the data. The guide is also available from the Diagnostics online help Home page.

Diagnostics Frequently Asked Questions (FAQ). Gives answers to frequently asked questions. The FAQ is also available from the Diagnostics online help Home page.

Chapter 1: Introduction

This chapter provides an overview of the Python agent.

Diagnostics Python Agent Overview

The HP Diagnostics Python Agent install package includes the software necessary to capture events such as method invocations, server requests, and system metrics from Python applications. The Python Agent package must be installed on the systems to be monitored that are running Python applications. Each instrumented application results in a unique probe entity which can be independently configured for data collection.

Chapter 2: Installing Python Agents

This chapter describes how to install a Python Agent and gives you information about the setup and configuration of the Python Agent.

This chapter contains the following sections:

Requirements for the Diagnostics Python Agent Host

For details of the system requirements for the Diagnostics Python Agent Host, see "Requirements for the Diagnostics Python Agent Host" in the Diagnostics System Requirements and Support Matrixes Guide located on the HP Software Support site. Access requires an HP Passport login (register for an HP Passport).

Overview of the Python Agent Installation

The Python Agent must be installed on all systems running Python application that you wish to monitor. Installation involves simply unzipping the install package and running the setup.py script on each system. After this is completed, it is then necessary to define the points that you wish to monitor within your applications. If you wish to monitor OpenStack or Django, configuration files and scripts have been supplied that will allow you monitor these applications.

Installing the Python Agent

The Diagnostics Python Agent is distributed for all platforms in a single zip file named **HPDiagPythonAgt_<release number>.zip**.

Perform the following steps to install the Python Agent:

1. Unzip the **HPDiagPythonAgt_<version>.zip** file to a new directory on the target host. Avoid spaces in the directory name.

   This directory is referred to as <agent_install_directory> throughout this guide**.**

2. Change directory to **<agent_install_directory>**.

3. Execute the "probe_setup.py" script using the Python interpreter that is used for the monitored application:

   - For Linux

     ```
     /<path to python>/python  probe_setup.py
     ```

   - For Windows

     ```
     <path to python>\python.exe  probe_setup.py
     ```

   where <path_to_python> is the path to your Python executable such as **/usr/bin/python** on Linux or **C:\Python26\python.exe** on Windows.

The probe_setup.py Script

The script **probe_setup.py** is used to install, upgrade or remove the HP Diagnostics Python Agent.

Usage:

```
probe_setup.py [-h|--help] [-u|--update] [-r|--remove] [-d|--dont_ask]
```

Options:

Option	Description
-h, --help	Show this help message and exit.
-u, --update	Update or upgrade the Python Probe.
-r, --remove	Remove the Python Probe.
-d, --dont_ask	Install or remove the Python Probe without asking.

The **probe_setup.py** script accomplishes the following steps during the installation:

1. Install the **hpdiag modules** in the **site-packages** or **dist-packages** directory of the Python installation (see "Directory Structure" on the next page for details on where files are installed).

2. Install the **hpdiag scripts** in the Python **bin** (Linux) or **Scripts** (Windows) directory.
3. Install the PythonProbe configuration files to the **hpdiag/etc** directory.
4. Install the systemmetrics binary to the **hpdiag/bin** directory.
5. Create the PythonProbe log directory **/var/log/hpdiag** (Linux) or **%PROGRAMDATA%\Hewlett-Packard\hpdiag\log** (Windows).
6. Store a list of installed files in **hpdiag/backups/installed_files**.

Directory Structure

The Python Agent uses the following directory structure.

Python Modules

The **hpdiag** Python modules are stored in the Python **site-packages** or **dist-packages** directory as follows:

- On Linux: **/path/to/lib/python[python_version]/site-packages/hpdiag**
- On Windows: **\path\to\lib\python[python_version]\site-packages\hpdiag**

Scripts

- On Linux, the **hpdiag** Python scripts are copied into the **bin** directory, where the Python executable also resides.
- On Windows the scripts are installed into the **Scripts** directory under the Python installation directory.

hpdiag Directory

The HP Diagnostics Python Agent requires a dedicated directory for its configuration and binary files. The location of this directory differs based on the platform and in the case of Windows, is based on the Windows version as well.

- On Linux: **/opt/hpdiag**
- On Windows: **%PROGRAMDATA%\Hewlett-Packard\hpdiag**

Binaries

The binaries are stored in: **<hpdiag_dir>/bin**

Configuration Files

The configuration files are stored in: **<hpdiag_dir>/etc**

Log Files

The HP Diagnostics Python Agent creates the following directories for the Python Agent to place its log files:

- On Linux: **/var/log/hpdiag**
- On Windows: **%PROGRAMDATA%\Hewlett-Packard\hpdiag\log**

Removing the Python Agent

During the installation of the HP Diagnostics Python Agent, the installation script was copied into the **hpdiag** directory with the name **probe_deinstall.py.** Executing this script will remove the Python Agent.

Please make sure that no application is instrumented and that no probe is running when the probe will be removed. If a probe is still running on Windows, then the rename of the **hpdiag** directory will fail and an error is returned.On a Linux system we cannot detect if a probe is still running during uninstall. This may lead to unpredictable results.

On Linux:

```
/path/to/python /opt/hpdiag/probe_deinstall.py
```

On Windows:

```
cd %PROGRAMDATA%\Hewlett-Packard
```

```
\path\to\python hpdiag\probe_deinstall.py
```

Please note that on Windows it is necessary to call this script from outside of the **hpdiag** directory, because the hpdiag directory will be renamed during the de-installation. This rename fails when the console is opened in the hpdiag directory.

The deinstallation script will perform the following steps:

1. Remove the probe Python files from the Python **site-packages** directory.
2. Remove the **.egg-info** file.
3. Rename the hpdiag directory to **hpdiag.<date>-<time>**.

Chapter 3: Instrumenting a Python Application

There are multiple ways to instrument a Python application, and each is explained below.

Using the hpdiag_instrument.py Wrapper Script

The HP Diagnostics Python Agent provides a script to instrument and start an application: **hpdiag_instrument.py**.

No source code change is required in the Python application using this approach. If the main script of the monitored Python application is called "app_main.py", for example, then the instrumented application is run by the following command:

```
hpdiag_instrument.py --config app_main.conf --point app_main.point app_main.py
```

The script **hpdiag_instrument.py** initializes the Python probe and reads the capture points from the given point file. Afterwards, it starts the main script of the application via Python's **execfile** function. When the monitored application exits, this script closes all resources of the running probe.

The modules used by the python application are instrumented at runtime when they are imported. The probe uses the custom import hook **sys.meta_path** as described in the PEP 302 of the Python language. This might conflict with applications that also use this import hook. See "Decorate the Functions and Classes of the Monitored Application" on page 16 for an alternative.

Usage:

```
hpdiag_instrument.py [--config_dir <config dir>] [--bin_dir <bin_dir>] \
        [--config <config_file>] --point <point_file> \
        [--single] <target_script> [<target_script_args>]
```

Options:

Option	Description
-h, --help	Show this help message and exit.

Option	Description
-d CONFIGDIR, --config_ dir=CONFIGDIR	Configuration directory of the Python Agent.
-b BINDIR, --bin_dir=BINDIR	Binary directory of the Python Agent.
-c FILE, --config=FILE	Python probe configuration file [default = probe.conf
-p FILE, --point=FILE	Configuration of methods to measure.
-s, --single	Instrument the target_script as well as any modules it loads. By default, only modules referenced in the target_script are instrumented.

Parameters:

The parameters `--config_dir`, `--bin_dir` and `--config` are optional and are only needed when it is desired to use different settings than the defaults.

Option	Default	Environment Variable	Description
-d, --config_dir	/opt/hpdiag/etc; %PROGRAMDATA%\Hewlett-Packard\hpdiag\etc	$PYPROBE_ CONFIG_DIR	Directory containing the configuration files.
-b, --bin_dir	/opt/hpdiag/bin; %PROGRAMDATA%\Hewlett-Packard\hpdiag\bin	$PYPROBE_BIN_ DIR	Directory containing the binary files like 'systemmetrics'
-c, --config	probe.conf	N/A	File containing the probe configuration.

Note: The specification of the directories as parameter for **hpdiag_instrument.py** has a higher priority than the environment variable settings. The environment variable settings have a higher priority than the defaults.

Several examples for starting your application are shown below:

Example 1:

```
hpdiag_instrument.py --point webapp.point webapp.py
```

Example 2:

```
hpdiag_instrument.py --config my_probe.conf --point webapp.point webapp.py
```

Example 3:

```
hpdiag_instrument.py -d /path/to/my/config/data \
 -p other_weapp.point \
 webapp.py
```

On Windows, the path_to_python\python.exe must be added in front of hpdiag_instrument.py.

Instrument the Main Script of the Monitored Application

It is also possible to initialize and shutdown the Python probe directly from the main script of the Python application (similar to what hpdiag_instrument.py does). The code below shows this approach:

```
try:
  from hpdiag import pyprobe
except ImportError:
  class PyProbeDummy(object):
    @staticmethod
    def init(*args, **kws):
      print "Warning: Cannot initialize HP Diagnostics Python Agent. Failed to
import 'hpdiag.pyprobe' in file '%s'" % __file__
    @staticmethod
    def shutdown():
      print "Warning: Cannot shutdown HP Diagnostics Python Agent. Failed to
import 'hpdiag.pyprobe' in file '%s'" % __file__
  pyprobe = PyProbeDummy

pyprobe.init(config_file = "app_probe.conf", point_file = "app.point")

try:
  def main():
  # call the application entry point here

  if __name__ == '__main__':
  main()
finally:
  pyprobe.shutdown()
```

There are only a few lines to be added into the main script:

1. The statement to import the module "hpdiag.pyprobe"
2. The initialize function "pyprobe.init()" at the beginning

3. The shutdown function at the end
4. The try-finally block around the original code. This is optional, but highly recommended.

Note: In WSGI scripts, only the first two lines are needed. Adding the shutdown function at the end will cause the probe to not function properly. See below for more details.

The initialize function takes up to four parameters:

- **config_file**: The configuration file for the probe.
- **point_file**: The point file containing the capture points for the instrumented Python application.
- **config_dir**: The directory where the configuration files (probe configuration and point file) are located, if different from the default location.
- **bin_dir**: The directory where the executables (systemmetrics, ...) are located, if different from the default location.

Note: Please be sure to always specify the parameter for the pyprobe.init() function using keywords like "point_file = app.point". This allows the parameters to be listed in any order, and also allows for parameters to remain unset so that they will get the default values.

All APIs of the python probe are in the module **hpdiag.pyprobe**. Only the functions and classes defined in this module should be used to instrument the monitored application! Functions and classes in all other modules of the Python probe may change without notice at any time!

Decorate the Functions and Classes of the Monitored Application

It is also possible to create the capture points in the Python source at run-time by using the following decorator functions from the module hpdiag.pyprobe: func_point, method_point, and class_point. They are used as decorators directly in the Python source above the instrumented function, method, or class. The supported arguments for these decorators are exactly the same arguments as those for the capture points in the point file. For example:

```
from hpdiag import pyprobe

@pyprobe.class_point(method = "^fib$|rfib ", layer = "fiboLayer")
class Fibo(object):
  # the implementation of the Fibo class
```

This decorator creates one capture point for the class **Fibo**. The method's argument specifies that the method fib and the method rfib should be instrumented. Please note that Python regular expressions

are used here. The regular expression ^fib$ means that only the method fib is instrumented whereas the regular expression rfib means that any method that has a sub-string rfib in its name will be instrumented (for example, also rfib_seq).

The argument **layer** defines the layer for all instrumented methods of this class. The other mandatory arguments **class** and **module** are automatically determined by the decorator.

It is also possible to decorate a single function or method using the **func_point** and **method_point** decorator. For example:

```
class Fibo(object):
  @pyprobe.method_point(clazz = "Fibo", layer = "fiboLayer", args = "0")
  def fib_seq2(self, n):
    # the implementation of the method
```

Even though the decorator is executed in the context of the method, it is necessary to specify the name of the class because the class is not yet defined (and so cannot be automatically determined) at the time the decorator is executed.

Note: Please also note that the argument name is clazz because class is a Python key word which cannot be used.

If the instrumented function or method already has other decorators (for example, it is a @staticmethod or @classmethod), then the decorator that creates the capture point for the probe must be written directly above the function or method (if not it might cause problems). For example:

```
@_DecoMemoize
@pyprobe.func_point(layer = "fiboLayer", args = 0)
def mfib(n):
  # the implementation of the function
```

Please note that all three decorator functions are executed at import time of the module and create just the capture point. The automatic instrumentation of the module via the above described import hook is performed after the module was loaded. Thus, the decorated functions, methods, and classes are treated like any capture point read from the point file.

In Code Creation of Capture Points

If you do not want to add the decorators in all the source files of the monitored application (or if the sources are not available at all), it is also possible to create all the capture points in one place within your application.

```
from hpdiag import pyprobe
pyprobe.init(config_file = "probe.conf", point_file = "app.point")
pl = pyprobe.PointList()
pl.create_method_point("func_name", "module_name", <point arguments>)
pl.create_method_point("method_name", "module_name", "class_name", <point
```

```
arguments>)
pl.create_class_point(<class instance>, <point arguments>)
pl.create_point(<point arguments>)
```

The point arguments are the same as the options of a point in the point file, for example, layer="Database", detail="is_sql_statement".

Once all points are created, it is possible to trigger the instrumentation by calling:

```
pyprobe.instrument(pl)

# call the actual application entry point


pyprobe.shutdown()
```

Passing the point list to the instrument function ensures that only the newly created points are instrumented. Capture points that were read from the point file (passed as second parameter to the init function) are, by default, automatically instrumented at import time of the module (using the custom import hook describe above).

Please note that the point file that is passed to the init function is optional! If not specified, only the capture points created by the decorator functions are used by default, that is, if the automatic instrumentation at import time is enabled.

It is possible to disable the automatic instrumentation at import time. Use the following argument in the probe section of the probe config file to do this:

```
[Probe]

auto_instrument = True
```

If this argument is set to False (the default is True), the modules imported by the monitored application are not automatically instrumented. Instead, it is possible to trigger the instrumentation any time at runtime by calling:

```
pyprobe.instrument()
```

Because no point list is passed as parameter to the function, it will use all capture points that were created so far at runtime and/or read by the init function from the point file to instrument the currently loaded modules.

Instrumenting a Single Script

A single script is characterized by the fact that it is not imported by another script. Thus it is more difficult to instrument such a script. If a script can be instrumented or not depends on the availability of classes and methods inside the script.

Prerequisites

The hpdiag_instrument.py tool allows the execution of instrumented single scripts by loading it as a module and calling its main() method. This means that the existence of a main() function in the script is a prerequisite. Simple Python scripts often have no main() method, but look like this:

```
if __name__ == '__main__':

instance = MyClass()
:
xyz = helper_function()
```

In most cases this can be easily changed to:

```
if __name__ == '__main__':
instance = MyClass()
:
xyz = helper_function()
In most cases this can be easily changed to:
def main():
instance = MyClass()
:
xyz = helper_function()
if __name__ == '__main__':
main()
```

This allows access to the main() method by the hpdiag_instrument.py tool, and thus to instrument this single script.

Point Definitions

The script is imported with its file name as module name, so that its name is referenced in the point file as module name to define the instrumentation points. For example when the script name is myScript.py then this is imported as 'myScript' and might be referenced in the point file as follows:

```
[myScript_1]
module = myScript
class = MyClass
method = class_method
layer = myscript
```

Note: Because single scripts are imported as module, the file name must not contain any dots ('.'). For example myScript-0.2.py does not work because dots are not allowed in module names. Correct is myScript.py or my_script.py.

Calling hpdiag_instrument.py

The hpdiag_instrument.py tool has the parameter '-s | --single' to indicate that the called Python script is a single script:

```
hpdiag_instrument.py --config myScript.conf --point myScript.point \
        --single /path/to/myScript.py --script_par1 ...
```

Chapter 4: Configuring the Python Agent

The file **<hpdiag_dir>/etc/probe.conf** drives the basic agent behavior. The probe.conf file has section/namespaces. Configuration parameters are defined within these namespaces.

The following sections give detailed descriptions of the configuration parameters in the probe.conf file. Also included are two sections that give details on some specific URI replace pattern configurations in the probe.conf file.

Namespace [Mediator]

hostname: The Diagnostics mediator host name.

port: The Diagnostics mediator port number.

channeltype: One of synchronous, threaded, or multiprocess. This value configures how events are sent to the mediator. Python has a very peculiar threading behavior, so testing may be necessary to determine the optimal settings for your application.

- synchronous: The events are sent as part of the business application thread. This might slow down the business application.
- threaded: The events are sent in a separate thread, but in the same process as the business application. This is the default.
- multiprocess: The events are sent in a separate process.

reconnect_timeout: The timeout in seconds before the next reconnect, in case the connection to the mediator has been lost. Server requests that complete while the mediator connection is unavailable are dropped silently.

keep_alive_interval: Interval in seconds at which the probe will send keep alive messages to the registrar on the mediator.

Namespace [Logging]

class: Specify the logging (handler) type. There are two types supported:

- TimedRotatingFileHandler: It supports rotation of disk log files at certain timed intervals.
- RotatingFileHandler: It supports rotation of disk log files based on file size limits.

file: The absolute path to the log file.

level: The default logging level: CRITICAL, ERROR, WARNING, INFO or DEBUG.

level_exceptions: Specify exceptions to the default logging level of the Python probe. These exceptions are specified as Python dictionary with a Python pattern as key and the logging level as value (in the form of a string). The probe iterates through all keys (patterns) of the dictionary and will use the first one that matches. The order is not defined, however.

The example below sets the DEBUG level to all loggers in modules that start with hpdiag.location. Likewise, it sets the INFO level to all loggers in modules that start with hpdiag.importhook:

```
level_exceptions = {r'hpdiag\.location.*' : 'DEBUG', r'hpdiag\.importhook.*' :
'INFO'}
```

backup_count: If nonzero, at most backup_count files will be kept. If more would be created when roll-over occurs, the oldest one is deleted.

max_file_size: For RotatingFileHandler: The maximum size of the log file in MB.

when: For TimedRotatingFileHandler: Rotating happens based on the product of when and interval. Possible values are:

'S' Seconds, 'M' Minutes, 'H' Hours, 'D' Days, 'W#' Week day (# = 0 - 6 with 0 = Monday), or 'midnight' Roll over at midnight.

interval: For TimedRotatingFileHandler: The roll-over interval. Example: If when is set to '1#' (= Tuesday) and interval is set to '2', then the log file will be rolled over every second Tuesday.

utc: Use times in UTC (default is local time).

Namespace [Probe]

probe_id: The name of the probe instance. Add %0 to the probe_id to get a unique probe name if several instances of the same probe are running on the same system.

registered_hostname: The hostname to be used if DNS/IP lookups don't work reliably.

probe_group: Probe Group name (used in the same manner as in the Java and .NET probes).

system_group: System Group name (used in the same manner as in the Java and .NET probes).

auto_instrument: Enable/disable automatic instrumentation at import time (default: True).

instrument_loaded_modules: Instrument modules that have been loaded before pyprobe.init() is called (default: False).

instrument_pyprobe_threads: Instrument points found in the probe threads, e.g. monitor the probe itself (default: False).

error_on_duplicate_location: An exception is thrown whenever the same location is instrumented multiple times (default: False).

sql_parsing_mode: Parsing mode of SQL queries.

1 = just methods, no SQL queries

2 = main categories for SQL queries (select/update/insert/delete/...)

3 = a measurement per whole SQL query aggregating similar statements into a single measurements

4 = a measurement per whole SQL query aggregating only identical statements

Agent side trimming:

maximum_stack_depth: Don't capture any data about methods called at a depth greater than this. For example, if maximum_stack_depth is 3, and "/login.do" calls a() calls b() calls c(), only login.do, a, and b will be captured. Setting a low maximum_stack_depth can somewhat reduce the overhead of capture. Setting this to a very high value disables depth trimming. This is dangerous if potentially recursive methods are instrumented as it can lead to nearly infinite call-trees. This will consume a lot of memory. Setting this value above 100 is strongly discouraged. The default is 25.

minimum_method_latency: Latency trimming - never capture any data about regular methods that execute faster than this number of milliseconds. Depending upon your platform & whether hi-res time stamps are being used, it may not be useful to specify this value in increments of less than 10ms. It defaults to 51 milliseconds.

minimum_fragment_latency: If an entire server request takes less than this number of milliseconds, it will not be captured, unless a threshold has been set on that server request. The default value is 51ms.

maximum_method_calls: Tree size trimming - never capture more than this number of methods per instance tree. This is regardless of latency and depth trimming. It defaults to 1000. Note that this applies to all methods, including outbound calls.

minimum_sql_latency: If an SQL statement takes less than this amount of time, it will not be trended, until it does exceed this time. It defaults to 1000 milliseconds (one second).

httpserver_port: Port to use for python probe http server.

http_client_show_url: Enables/disables the inclusion of the URL as part of the identity of an outbound call. The value should be set to false for REST service client applications.

uri_replace_pattern: A comma-separated list of pattern replacement operations to attempt on each URI (see "URI Truncation and Mapping" on the next page).

uri_pathsegments: Number of URI path segments to allow (see "URI Path Segment Trimming" on page 26).

username: User name used to authenticate the mediator with the probe http server. If it is empty, a default user name will be used.

password: Password used to authenticate the mediator with the probe http server. Use the utility **hpdiag_encodepassword.py** to encode your password before adding it there. If it is empty, a default password will be used.

Namespace [SystemMetricsCollector]

enabled: True or False, decides whether the system metrics collector is active.

sampling_rate: How fast should a metric be locally sampled. Uses time string values, like 5s.

metrics_group: What group should system metrics be associated with? This value may be the same as an existing probe group, or completely independent.

udp_port: Port to use for system metrics UDP control port. Do NOT modify this unless there is a conflict with another application. All Diagnostics agents on a system MUST be configured to use the same port.

mediator_port: Port on the mediator used to deliver metrics to.

udp_retry_interval: How often should the metrics collector try to open the UDP port in case it is in use by another program. Uses time string values, like 10min.

username: User name used to authenticate the system metrics collector with the mediator.

password: Password used to authenticate the system metrics collector with the mediator. Use the utility hpdiag_encodepassword.py to encode your password before entering it here.

Namespace [SystemMetrics]

This namespace contains the system metrics to collect.

These system metrics collector entries use the same layout as the ones for the Java Agent (see HP Diagnostics Java Agent Guide chapter on "Java Agent - System Metrics Capture") with the exception that the collector name is not available in the Python agent.

URI Truncation and Mapping

It is possible to truncate or change the URI of a request using Python regular expressions. This is specified in the **probe.conf** file in the option **uri_replace_pattern**. This is a comma-separated list of pattern replacement operations to attempt on each URI. This is useful to replace many server request URIs with one simplified server request URI that aggregates them. The truncation or mapping of URIs is done using the 's/pattern/replace/' syntax, which is the only supported syntax for the URI replacement patterns.

How and Where are the Patterns Used

This functionality is applied after before:code custom functions, args:name or args:n were applied. The output of before:code or args:x is used as input for the URI replacement patterns.

If more than one pattern is specified, all patterns will be applied. The patterns are applied in order. The output of a previous matched pattern will be used as input for the next pattern. The resulting string is used in the Diagnostics GUI for the request name.

Characteristics

Because s/pattern/replace is not Python syntax, it is necessary to use '#' instead of '/' in the configuration file

```
s/pattern/replace/
```

must be written as

```
s#pattern#replace#
```

s/pattern/replace/ is used to be comparable with the syntax in Perl or on the Unix shell. It is also possible to omit the s and write #pattern#replace#.

Examples

Truncate before a string, match the string and any characters that follow it and leave replace empty. In this example $ matches end-of-line.

```
uri_replace_pattern = s#string.*$##
```

Truncate after a string. Match the string in a grouping and use \group-number to put the string into the replacement.

```
uri_replace_pattern = s#(string).*$#\1#
```

Use a comma separated list to perform multiple operations. The operations will all be performed in order. This would change every foo to bar and then change every bar back to foo.

```
uri_replace_pattern = s#foo#bar#,s#bar#foo#
```

Truncate before any semicolon.

```
uri_replace_pattern = s#;.*$##
```

Truncate before any /! or !. This uses ? to say that the slash is optional.

```
uri_replace_pattern = s#/?\!.*$##
```

Truncate before any ';' or '/!' or '!'.

```
uri_replace_pattern = s#(;|/?\!).*$##
```

Map /django/portal/ and /django/myportal/ to Django Portal.

```
uri_replace_pattern = s#^/django/(my)?portal/#Django Portal#
```

Other examples:

```
uri_replace_pattern = s#(;|/?\!).*$##,s#.*\.
(js|css|jpg|gif|png|pdf|html|jar|class)$#Static Content#

uri_replace_pattern = s#.*/([a-zA-Z0-9_ ]*)\.py#\1#
```

URI Path Segment Trimming

The URI path can be trimmed by the definition of **uri_pathsegments** in the **probe.conf** file. uri_ pathsegments is set to the number of URI path segments to allow - everything after this point will be trimmed. For example, with a setting of 2, URLs like /foo/bar/1, /foo/bar/2 will be trimmed to /foo/bar. A value of -1 or 0 will disable the path trimming.

Chapter 5: The Points File

The points file specifies which Python classes, methods and functions are monitored.

This chapter contains the following sections:

Description of the Parameters in the Points File

The points file specifies which Python classes, methods and functions are monitored.

The syntax of the points file is the same as for the Java probe. Therefore see the Java probe documentation for details.

The following arguments are supported:

Argument	Description	Mandatory
module	A Python regular expression	yes
class	A Python regular expression	no
method	A Python regular expression	yes
layer	The name of the layer	yes
layer_type	One of the following 4 values: • method (the default) • trended_method • portlet • sql	no

Argument	Description	Mandatory
detail	Specifies more specific capture instructions. It is a comma-separated list of the following: • before:code:<name>: execute the custom code with filename <name> before the instrumented method/function • after:code:<name>: execute the custom code with filename <name> after the instrumented method/function • args:name: uses the string representation of the instance on which the instrumented method was called as call argument • args:n: uses the string representation of the argument on index 'n' as call argument in the GUI (see more details below) • is_sql_statement: marks methods/functions that execute SQL statements • inbound: marks a method/function as inbound call that is used to track cross-VM transactions • outbound: marks a method/function as outbound call that is used to track cross-VM transactions • method_trim: indicates that every invocation of the method instrumented by this point should be “trimmed”, that is, not reported. However, side-effects of the corresponding code-snippets, if any, take place normally. • method_no_trim: indicates that no latency-based trimming should take place when a method instrumented by this point is executed. • no_layer_recurse: prohibits recording of any methods called from the method instrumented by this point, unless the callee belongs to a different layer.	no

For example:

```
[httplibHTTPConnectionOutbound]
module = httplib
class = HTTPConnection
method = request
layer = Sending
detail = outbound,before:code:httpconnection_outbound
```

To distinguish a method of a class from a function within a module, the Python agent introduces the additional argument “module” and considers the class argument as optional. Thus, a point describes

either a set of module functions or a set of class methods. If both functions as well as class methods within the same module should be captured, it is necessary to specify two different points.

Including Points Files

The point file referenced during the instrumentation can include other point files. This is done by using the special point IncludePoints. The file references have to be relative to the location of the main point file.

For example:

```
[IncludePoints]
1 = ../../etc/httprequest.point
2 = httpserver.point
3 = others/database.point
```

Chapter 6: Description of Custom Code

Custom code are Python functions that can be executed before or after the monitored method or function is executed. These functions are stored in files in the Python agent custom_code directory. The custom code functions used are defined in the points file and are specified separately for each monitored function. The custom code functions are referenced by file name.

The following sections gives details about custom code.

The Purpose of Custom Code

Custom code can be used in the Python Probe to extract data from the arguments passed into an instrumented function or method. If this data is returned by the custom code, it will be displayed as an argument of the method in the Diagnostics GUI (in the call profile). With custom code, it is even possible to modify the arguments of an instrumented function or method. Custom code is also used to track calls between multiple probe installations (cross VM calls).

Custom code can be called two times: before the instrumented method is called (before:code) and after it was called (after:code).

before:code

The **before:code** is used to extract data from the argument list of the instrumented method. If this extracted data (for example, a URI) is returned by the custom function, it will be displayed in the Diagnostics GUI as call argument.

The custom code functions are also used to intercept information that is needed for correct display in the Diagnostics UI. The custom code function can return a string (used as the argument of the call, as explained above), a dictionary, or a tuple of both. In the dictionary, the following entries are used by the probe to report data to the server:

Key	Meaning
uri	URI of an incoming http service request.

Key	Meaning
inbound_coloring	Coloring token of an inbound call used to track cross-VM transactions.
remote_ip	Caller IP address of an inbound call.
diag_arg	The diag argument required for both incoming and outgoing calls.

Server requests are reported as inbound to the Diagnostics server if a coloring token has been reported by any method in the call stack.

If any method reported an URI, the server request type is reported as 'HTTP'; otherwise it will show up as "Pseudo"

Check the files in **etc/custom_code** for syntax and usage examples of custom code, especially the way the coloring tokens are injected and retrieved from the calls.

after:code

The **after:code** can be used to do any processing that might be useful after the instrumented method was called.

Custom Functions

All custom code needs to be written as a function with the name custom_fct_before(...) or custom_fct_after(...). The custom function that is used for before:code takes the following argument list: custom_fct_before(instance, location, args, kws)

- **instance**: the class instance on which the instrumented method is called. It is None for instrumented module functions.
- **location**: the python probe location object that identifies the instrumented function/method.
- **args**: the tuple of positional arguments passed to the instrumented function/method.
- **kws**: the dictionary of keyword arguments passed to the instrumented function/method.

It can return the following values:

- **method argument**: the argument string for the instrumented method as displayed by the Diagnostics GUI in the call profile.
- **a dictionary**: a dictionary of key value pairs. This dictionary is passed to the before:after function after the instrumented method got called. There are two special keys in this dictionary, however. If custom_fct_before adds the keys "method_args" and/or "method_kws" to this dictionary, it is assumed that they represent the modified argument list of the instrumented function/method being called. The value for key "method_args" must be of type 'tuple' and the value of key "method_kws" must be of type 'dict'. If the instrumented method is an outbound call, then this dictionary has to

contain the key "diag_arg". If it is an inbound call, it has to contain the keys "diag_arg", "inbound_coloring" and "remote_ip".

- **a tuple**: both values described above wrapped into a tuple.
- **None**: the custom code function may also return None.

The custom function for after:code takes the following parameters: custom_fct_after(instance, location, method_return_value, code_dict).

- **instance**: the class instance on which the instrumented method is called. It is None for instrumented module functions.
- **location**: the python probe location object that identifies the instrumented function/method.
- **return value**: the return value of the instrumented function or method.
- **a dictionary**: the dictionary that was returned by the before:code function.

Example for before:code in the file custom_code/cust_example_before.py:

```
# Used by [DiagShop]

from urlparse import urlparse

def custom_fct_before(instance, location, args, kws):

  ret_val = None
  purl = urlparse(str(args[0]))
  if len(purl.scheme) == 0:
    ret_val = ''
  else:
  ret_val = purl.path

return ret_val
```

The file name cust_example_before is used as reference of the custom code to be used in the point file. The function name is always custom_fct_before(instance, location, args, kws). This code would be referenced in the point file via the following:

```
detail = before:code:cust_example_before
```

Example for after:code in the file custom_code/cust_example_after.py:

```
def custom_fct_after(instance, location, method_return_value, code_dict):

  print "CustomAfter: Custom code executed - does not return anything."
```

It is possible to define a custom_fct_before(...) function and a custom_fct_after(...) function in the same file and reference it using the same name. Which function is be used is defined in the detail section in the point file.

Note: The Python Probe imports the custom code files as Python modules. This means that all limitations regarding the file names for Python modules also apply to the custom code files. For example the characters (<space>) or ('-') are not allowed in file names

Using Sub-directories

Because the custom code files are handled as Python modules by the Python Probe, it is also possible to categorize custom code files in sub-directories (modules). If this is desired, each sub-directory needs to have a Python special file in it - this is the file **_init_.py**. This file can be empty, but must be there to be able to import custom code from a sub-directory. Example:

```
pyapp_code
|- get_http_request.py
|- get_request_2.py
|- pyapp_controller
| |- __init__.py
| |- get_details.py
| |- do_something.py
|- pyapp_scheduler
| |- __init__.py
| |- get_request_from_queue.py
| |- get_service_request.py
```

With these files in place, the files of this structure can be referenced in the point file for example via

```
detail = before:code:get_http_request
```

from the custom code base directory or for the pyapp controller from the pyapp_controller sub-directory:

```
detail = before:code:pyapp_controller.get_details
```

Note: A **_init_.py** file is not needed in the custom code base directory, because the files in this directory are not regarded as Python modules.

Returning HTTP Request Status Codes

For each HTTP request the HTTP server returns a status code. The custom code can be used to report this status code to the HP Diagnostics server. To do this, the location object, passed to the before and after functions, implements the method add_request_attribute. It takes the attribute name and the attribute value as parameters. At the moment, only the following four attributes are supported by the HP Diagnostics server:

- WS_consumer_id
- HTTP_status_code
- HTTP_status_desc
- tcp_server_port

The following example shows how to extract the HTTP status code of requests to django applications and have it sent to the HP Diagnostics server:

```
def custom_fct_after(instance, location, method_return_value, custom_code_dict):
  from django.core.handlers import wsgi

  try:
    status_text = wsgi.STATUS_CODE_TEXT[method_return_value.status_code]
  except KeyError:
    status_text = 'UNKNOWN STATUS CODE'

  if method_return_value.status_code >= 500 and method_return_value.status_code
<=699:
    location.add_request_attribute("HTTP_status_code", str(method_return_
value.status_code))
    location.add_request_attribute("HTTP_status_desc", status_text)

   return None
```

Cross VM Server Requests

Outbound Calls

To enable HP Diagnostics to connect calls made from one instrumented application to another, a unique identifier (coloring) needs to be added to the data sent to the called application. This can be done with custom code.

The following example is used to instrument the request method of the python httplib.HTTPConnectionOutbound class. It shows how to get the coloring from the probe using the location.get_outbound_coloring call which takes the called target as parameter. The next step is to add it to the data which will get sent to the called application. location.create_diag_envelope will either add it to the data to be sent (passed as second parameters) or will return an encoded version of the coloring if no data is passed. In the latter case, you have to add the coloring to the request yourself. The data to be sent has to be a str for the enveloping to work! This example adds it as an additional HTTP header called X-Mercury-Diag-HTTP-Color.

Then a string called diag_arg needs to get constructed which must be passed back to the Python probe via a dictionary (using the dictionary key "diag_arg"). In case the arguments of the instrumented methods are modified within the custom code, they also have to be passed back to the probe via the

returned dictionary (using the keys "method_args" for the positional arguments and "method_kws" for the keyword arguments).

File httpconnection_outbound.py:

```
# Used by [httplibHTTPConnectionOutbound]

import httplib

def custom_fct_before(instance, location, args, kws):

  if isinstance(instance, httplib.HTTPSConnection):
    url = "https://%s:%s/%s"
  elif isinstance(instance, httplib.HTTPConnection):
    url = "http://%s:%s/%s"
  else:
    url = "request://%s:%s/%s"

outbound_coloring = location.get_outbound_coloring(url % (instance.host,
instance.port, args[1]))
outbound_coloring = location.create_diag_envelope(outbound_coloring, "")

if (args[3]):
  args[3]['X-Mercury-Diag-HTTP-Color'] = outbound_coloring
else:
  args[3] = {'X-Mercury-Diag-HTTP-Color' : outbound_coloring}

param_dict = {'name': '{0}:{1}'.format(instance.host, instance.port), 'target':
'{0}:{1}'.format(instance.host, instance.port)}
diag_arg = location.create_diag_arg('http', param_dict)
result = {}
result['diag_arg'] = diag_arg

result['method_kws'] = kws
result['method_args'] = args

return result
```

Inbound Calls

In inbound calls, the custom code is used to remove the coloring from the request received and pass it to the python probe.

The following example is used to instrument the WSGI handler of the Django framework. It removes the coloring from the request, passed as the X-Mercury-Diag-HTTP-Color parameter using the location.get_coloring_from_diag_envelope method. The coloring is then returned to the python probe. In addition to the coloring, a diag_arg string and the IP address of the calling application and the called URI needs to get returned.

File basehttprequesthandler_inbound.py:

```
# Used by [BaseHTTPServerBaseHTTPRequestHandlerInbound]

import BaseHTTPServer, socket

def custom_fct_before(instance, location, args, kws):

  result = {}
  path = None

  if 'X-Mercury-Diag-HTTP-Color' in instance.headers:
    inbound_coloring = location.get_coloring_from_diag_envelope(instance.headers
['X-Mercury-Diag-HTTP-Color'])
    del (instance.headers['X-Mercury-Diag-HTTP-Color'])
    result['inbound_coloring'] = inbound_coloring

    if isinstance(instance, BaseHTTPServer.BaseHTTPRequestHandler):
      host, port = instance.client_address[:2]

    param_dict = {'name': instance.path, 'target': instance.headers['host']}
    diag_arg = location.create_diag_arg('http', param_dict)

    path = instance.path
    result['diag_arg'] = diag_arg
    result['remote_ip'] = host
    result['uri'] = path

return (path, result)
```

Argument Extraction

args:name

args:name uses the string representation of the instance on which the instrumented method was called as call argument. For class or static methods or a module function, it returns the doc string of the instrumented function. If no doc string exists, it returns the module name.

args:n

args:n uses the string representation of the argument on index 'n' as call argument in the GUI. 'n' can be in the range 0 - 254.

args:name and args:n can be used together with an after:code custom function, but not together with a before:code custom function. If a before:code function is referenced and args is used, it is undefined as to which one will be used.

Chapter 7: Available Out-of-the-Box Configurations

The Python Agent comes with a number of out-of-the-box configurations as ready-to-use configuration or as starting point for own configurations. Currently available is instrumentation for:

- "OpenStack Instrumentation" below
- "Django and WSGI Instrumentation" on page 40

OpenStack Instrumentation

The Python Agent provides configuration for the instrumentation of the OpenStack cloud computing platform (Diablo and Essex Release).

For OpenStack, the following is provided in addition to the standard python agent:

- Points files for every component of OpenStack
- Setup scripts and configuration files for OpenStack

Point Files

For every component of OpenStack one or more ready-made point files can be installed and used.

- common.point
- dashboard.point
- glance.point
- keystone.point
- nova-api.point
- nova-general-controller.point
- nova-queue-send.point
- nova-scheduler.point
- setup-openstack.conf
- setup-openstack.txt

- swift-common.point
- swift-account-server.point
- swift-container-server.point
- swift-object-server.point
- swift-proxy-server.point

Setup of the OpenStack Instrumentation

The startup scripts of the OpenStack components that need to be instrumented must be changed in order to start the instrumentation together with a particular configuration. This can be done using a setup script **hpdiag_setup_openstack.py**.

```
hpdiag_setup_openstack.py -i|--install <os_version> | \
     -u|--uninstall <os_version> \
    [-m|--mediator <hostname_fqdn>] [-h|--help]
```

Install OpenStack instrumentation:

-i --install <os_verson> Install the OpenStack instrumentation

-u --uninstall <os_version> Remove the OpenStack instrumentation

-m --mediator <mediator> Hostname of the Diagnostics Server

-h --help Display this message

OpenStack versions:

essex OpenStack 2012.1

diablo OpenStack 2011.3

The setup script uses the information about which component will be instrumented and where to find its start script that is provided in the **setup_openstack.conf** file.

The **setup_openstack.conf** file has the following syntax:

```
<probe id>:<absolute path to the start script>:<pyprobe.init call>
```

For example:

```
nova-compute:/usr/bin/nova-compute:pyprobe.init(config_
dir="/opt/hpdiag/etc/openstack", \
bin_dir="/opt/hpdiag/bin", config_file = "nova-compute.conf", \
point_file = "nova-general-controller.point")
```

In addition to setting up the instrumentation in the OpenStack start up scripts, the script **hpdiag_ setup_openstack.py** creates a configuration file for each component of OpenStack. It uses the default

configuration file **probe.conf** in **/opt/hpdiag/etc** as master and creates a copy for each component. Each configuration file contains the hostname of the Diagnostics server (mediator) and the probe ID which will be displayed in Diagnostics' user interface. The hostname and the probe ID are added to the component's configuration file automatically. The name of the configuration file is <**probe_id>.conf**.

The instrumentation steps are:

1. Stop all OpenStack processes

   For

   Diablo go into `/opt/hpdiag/etc/openstack-diablo`

   Essex go into `/opt/hpdiag/etc/openstack-essex`

2. Call **hpdiag_setup_openstack.py** to instrument the OpenStack components Swift, Nova, Glance, Keystone, and the dashboard. The script creates various ***.conf** files for the various Python probes that monitor OpenStack.

   ```
   > hpdiag_setup_openstack.py -m <diagnostics_server_name> -i essex|diablo
   ```

   When **-m** is omitted, then the hostname will be taken from the file **/opt/hpdiag/etc/probe.conf**. You may edit this file to set the HP Diagnostics server name for the OpenStack instrumentation.

3. Restart the OpenStack services.

   For all Swift servers only one Python source file is modified: **/usr/share/pyshared/swift/common/wsgi.py**. It is the central entry point for most Swift processes. The inserted **pyprobe.init** function call looks as follows:

   ```
   pyprobe.init(config_dir = "/opt/hpdiag/etc/openstack-essex",
   config_file = "swift-" + log_name + ".conf",
   point_file = "swift-" + log_name + ".point")
   ```

   As you can see, the name of the *.conf and *.point files is built based on the log_name variable. It must be "proxy-server", "account-server", "container-server", or "object-server" to match the generated files from **hpdiag_setup_openstack.py**. To ensure this, check the swift config files in **/etc/swift**. For example, the default log_name for the Swift proxy server in the Essex release is "swift-proxy". This does not match the generated *.conf files. Thus, edit the file **/etc/swift/proxy-server.conf** and change the value for log_name in the section [app:proxy-server] to "proxy-server". Alternatively, you can also rename the generated Swift *.conf and *.point files if you do not want to change the files in **/etc/swift**.

   The original OpenStack scripts are preserved in **/opt/hpdiag/backup**. The instrumentation can be removed with the command **hpdiag_setup_openstack.py \-u essex\|diablo**.

Django and WSGI Instrumentation

The Python Agent provides configuration for the instrumentation of the Django and WSGI. The provided point files can be used for that. The Django point file is:

```
django.point
```

This instruments a point in the WSGI handler, that provides the request information:

```
[DjangoWSGIHandlerInbound]
module = django.core.handlers.wsgi
class = WSGIHandler
method = __call__
layer = WSGIHandler
detail = inbound,before:code:django_wsgi_call_inbound
```

Setup of the Django Instrumentation

The Django WSGI script needs to be changed to instrument a Django application. The Python Probe initialization needs to be done in that script.

Example Script:

```
import os, sys

# ---- Start of PyProbe section
# Calculate the path based on the location of the WSGI script.
sys.path.append(os.path.dirname(__file__))
sys.path.append('<path_to_the_app>')

# Instrument the application
from hpdiag import pyprobe
pyprobe.init(config_dir = '/opt/hpdiag/etc/mysite',
  bin_dir = '/opt/hpdiag/bin',
  config_file="probe.conf",
  point_file="mysite.point")
# ---- End of PyProbe section

os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Note: It is recommended to put the original start-up code into a t**ry:** / **except:** / **finally:** block where the **finally:** executes **pyprobe.shutdown()**. This is not recommended for WSGI scripts because the WSGI scripts are executed and terminated for every request. Calling **pyprobe.shutdown()** would launch a new probe every time, which strongly reduces the correlation and presentation quality in the HP Diagnostics UI.

Chapter 8: Troubleshooting

- **Reconnect/Reinitialize Event Channel After Server Reboot**

  In case the Diagnostics server has been rebooted or shut down for some reason the python probe gets disconnected from the server. In this case everytime the probe wants to send data to the server it tries to reconnect first. In order to avoid that reconnection attempts occur too often, the probe only tries to reconnect to the server after a timeout. By default this timeout is set to 5 seconds. The value can be modified in the configuration file. See "Configuring the Python Agent" on page 21 for more information about this value. While the probe is disconnected from the server all collected data will be deleted. After the server is running the probe gets reconnected automatically and continues to send collected data. The maximum time needed for a reconnection after the server is up and running again, is the reconnection timeout mentioned above.

- **Rotating log files are known to result in errors on Windows.**

  The workaround is to set file size or the rotation interval in the **probe.conf** file to large values to ensure that rotation never happens.

Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on Python Agent Guide (Diagnostics 9.24)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to SW-doc@hp.com.

We appreciate your feedback!