Zenoss Datasources through the eyes of the Python Collector ZenPack

Monitoring Windows devices with SNMP

February 2015

Jane Curry

Skills 1st Ltd

www.skills-1st.co.uk

DRAFT

Jane Curry
Skills 1st Ltd
2 Cedar Chase
Taplow
Maidenhead
SL6 0EU
01628 782565

jane.curry@skills-1st.co.uk

Synopsis

Zenoss provides several mechanisms for collecting performance data from devices, using different protocols, driven by different Zenoss daemons. Many come as standard with Zenoss Core.

A simple way to run non-standard data collection is to use the zencommand daemon which will run any script - bash, python, whatever - and deliver data to the performance data files which, for Zenoss upto and including version 4, are held in Round Robin Database (rrd) format. Unfortunately the zencommand daemon has a large overhead if many scripts are run against many devices.

Zenoss introduced the zenpython daemon in the Python Collector ZenPack to streamline collecting custom data and to provide much more flexibility in both defining and handling that data.

The Python Collector ZenPack also offers a new collection method for configuration data. Modeler plugins can be constructed which use zenpython to gather configuration data in an efficient way.

This paper first describes the Zenoss architecture behind data collection. It then explores simple and more complex ways of building new datasources with the Python Collector ZenPack, both to gather data from a device and to collect data for components of devices. It also demonstrates building a Python-based modeler plugin to discover device components.

Performance data is gathered by writing Python code. The code has to process data that arrives at the zenpython daemon asynchronously. The Python “Twisted” libraries have to be used to achieve this - a major topic in itself. Examples using both scripts and SNMP are described.

All examples are included in ZenPacks.skills1st.WinSnmp. This ZenPack updates and combines the excellent work done by Ryan Matte with his ZenPacks.Nova.Windows.SNMPPerfMonitor ZenPacks.Nova.WinServiceSNMP.

This is not an introductory text. It is aimed at people who can write some Python code, are already comfortable with creating Zenoss ZenPacks and have a good working knowledge of Zenoss in general. For help getting to this starting point, see some of the items in the extensive reference section.

THIS IS CURRENTLY NOT COMPLETE - MORE TO FOLLOW..........

Notations

Throughout this paper, text to by typed, file names and menu options to be selected, are highlighted by *italics;* important points to take note of are shown in **bold.**

Points of particular note are highlighted by an icon.

Table of Contents

1 Introduction

This paper has a number of objectives:

- Provide documentation for developing and testing ZenPacks
- Describe in detail the ZenPack architecture for gathering performance data
- Explore the capabilities of Zenoss's Python Collector ZenPack
- Combine the functionality of Ryan Matte's ZenPacks.Nova.Windows.SNMPPerfMonitor ZenPacks.Nova.WinServiceSNMP to produce a single ZenPack for monitoring Windows devices using the SNMP protocol.

2 Architecture

2.1 Collecting data

There are some good architecture diagrams in Chapter 1 of the Zenoss Core 4 Administration Guide that help describe the interactions between the many Zenoss daemons.

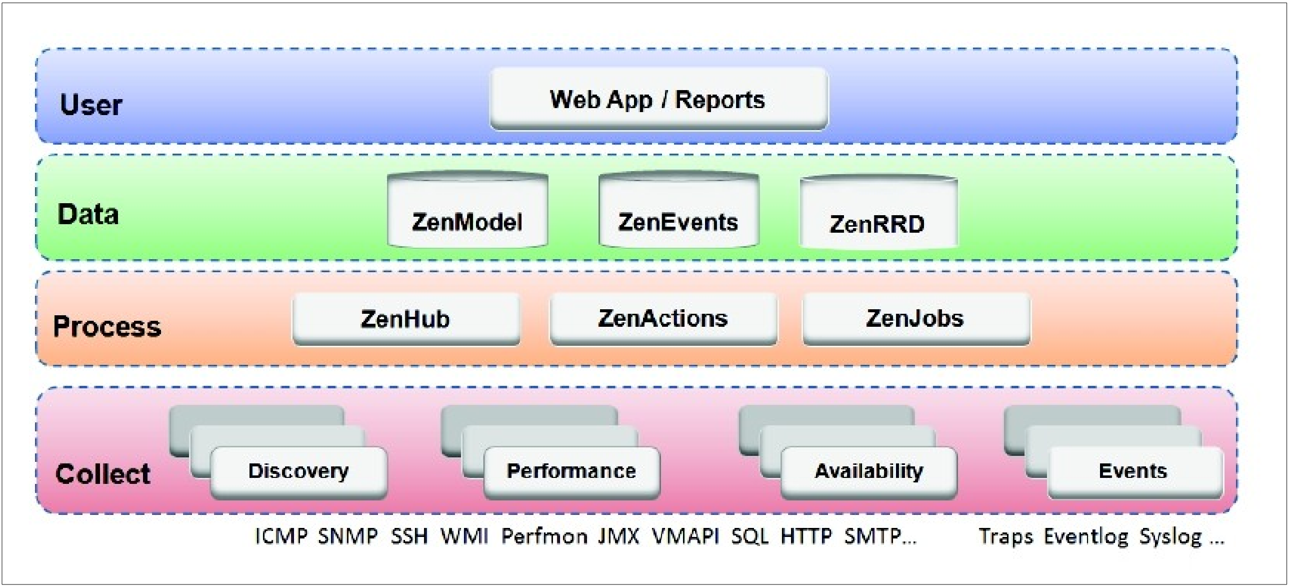


Figure 1: Zenoss Architecture

Fundamentally, data is collected by “collectors”. A Zenoss collector is always installed when you install Zenoss Core (or a Zenoss Enterprise hub). It will be called

localhost. It may be a logical entity in your single Zenoss server or you may install one or more separate Zenoss collector systems if you want to spread the load of data collection or avoid problems with firewalls. Each collector will have its own name, its own collection of devices that it is responsible for ,and its own set of Zenoss daemons. For the rest of this paper, the assumption is that there is a single localhost collector, co-located with the Zenoss hub and database.

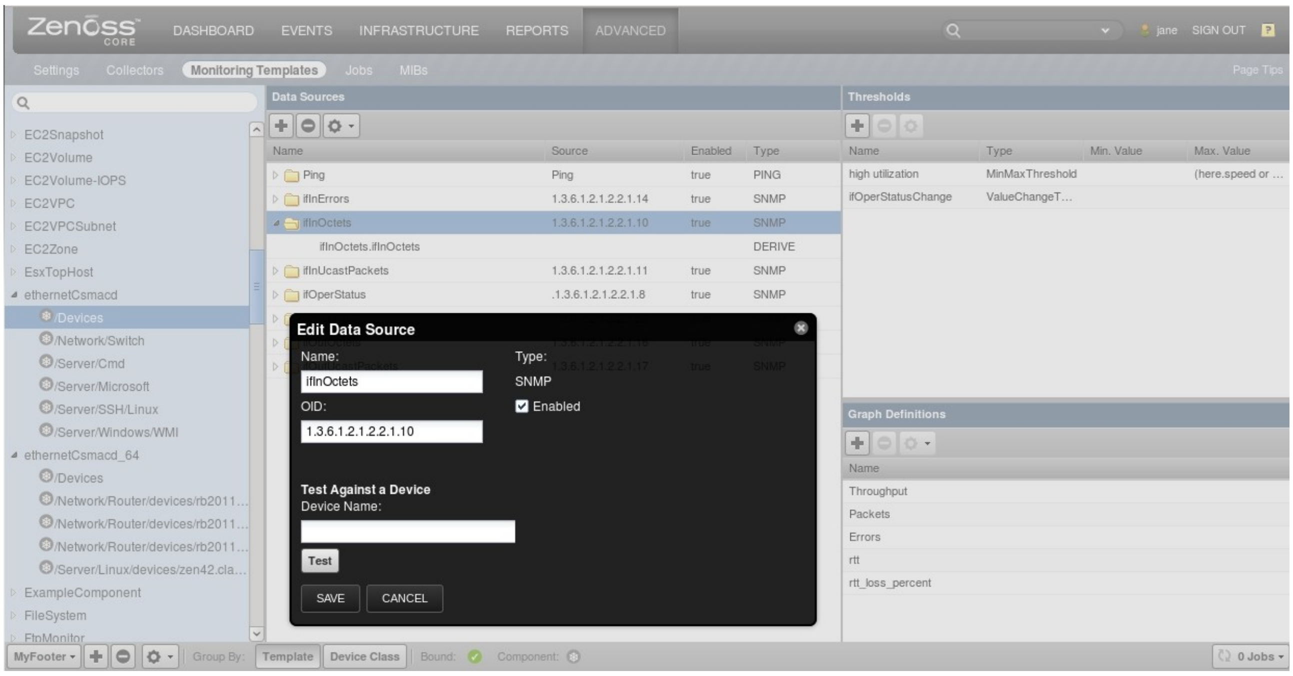
“Data” can be various types, shown in the diagram above:

- Discovery - this is typically **configuration** data. Examples for a device would be the number and type of network interfaces, the number of filesystems with their basic attributes, the number and names of monitored processes. zenmodeler is the daemon that collects this data and it typically runs every 12 hours. You do not expect configuration data to change minute-by-minute.
- Performance - this data often **is** relevant, at least at 5-minute intervals. So having discovered a device has several interfaces, including eth0, a performance daemon would then gather bytes in and bytes out for that interface, at 5 minute intervals. There are several Zenoss-supplied daemons that could do this. Typically the data is gathered using the SNMP protocol so zenperfsnmp would be the relevant daemon. If a device does not support SNMP, the data may be gathered using a command, perhaps over SSH; the collecting daemon would then be zencommand. If the device runs Windows, you may have one of the Zenoss-supplied ZenPacks to gather data using the WMI protocol using the zenwin daemon, or using the WinRM protocol which used the zenpython daemon.
- Availability data is typically collected by Zenoss daemons asking “are you still there” questions. zenping is a good example; by default, every device known to Zenoss is ping'ed every minute.
- Event data may come from the Zenoss daemons that gather external events information - zentrap gathers SNMP TRAP data; zensyslog gathers syslog data.

This paper will focus mainly on performance data and will touch on discovery data.

2.2 Performance templates

So how does a collector know what data to collect for what devices? **Performance templates** define what data is to be collected.



Data Sources

Name	Source	Enabled	Type
Ping	Ping	true	PING
ifInErrors	1.3.6.1.2.1.2.2.1.14	true	SNMP
ifInOctets	1.3.6.1.2.1.2.2.1.10	true	SNMP
ifInOctets.ifInOctets			DERIVE
ifInUcastPackets	1.3.6.1.2.1.2.2.1.11	true	SNMP
ifOperStatus	.1.3.6.1.2.1.2.2.1.8	true	SNMP

Thresholds

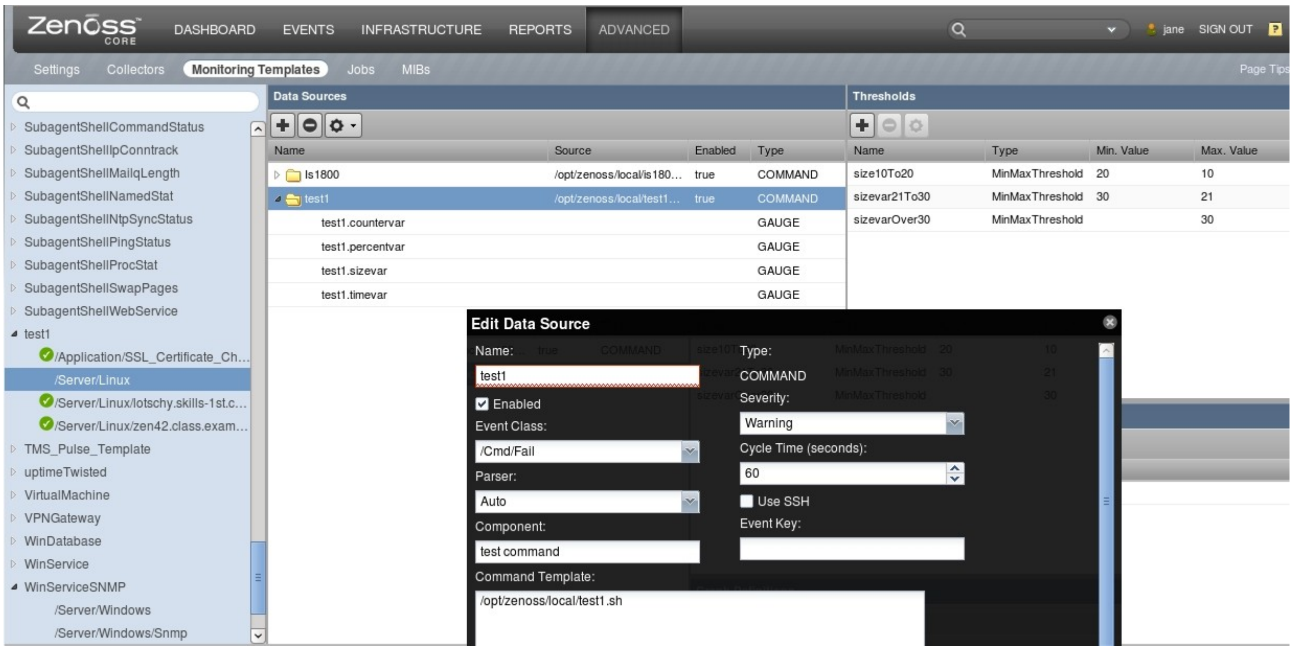
Name	Type	Min. Value	Max. Value
high utilization	MinMaxThreshold		(here.speed or …
ifOperStatusChange	ValueChangeT…		

Edit Data Source

Name: ifInOctets
Type: SNMP
OID: 1.3.6.1.2.1.2.2.1.10
☑ Enabled

Graph Definitions: Throughput, Packets, Errors, rtt, rtt_loss_percent

Figure 2: ethernetCsmacd SNMP template for /Devices



Data Sources

Name	Source	Enabled	Type
ls1800	/opt/zenoss/local/is180…	true	COMMAND
test1	/opt/zenoss/local/test1…	true	COMMAND
test1.countervar			GAUGE
test1.percentvar			GAUGE
test1.sizevar			GAUGE
test1.timevar			GAUGE

Thresholds

Name	Type	Min. Value	Max. Value
size10To20	MinMaxThreshold	20	10
sizevar21To30	MinMaxThreshold	30	21
sizevarOver30	MinMaxThreshold		30

Edit Data Source

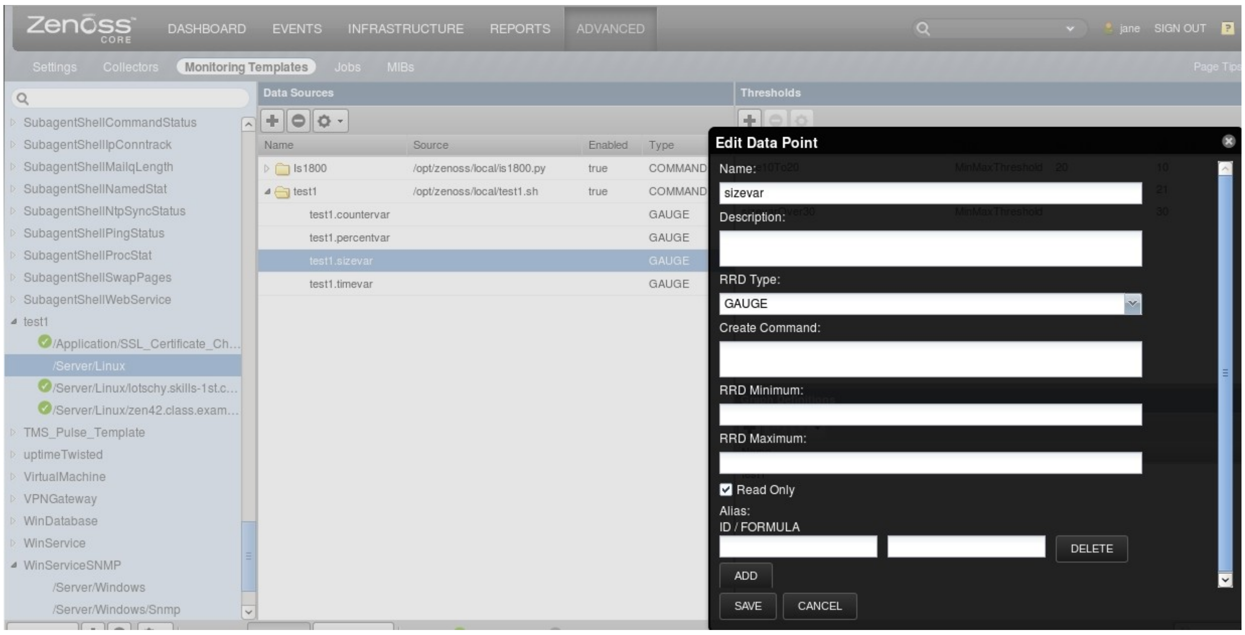
Name: test1
Type: COMMAND
☑ Enabled
Severity: Warning
Event Class: /Cmd/Fail
Cycle Time (seconds): 60
Parser: Auto
☐ Use SSH
Component: test command
Event Key:
Command Template: /opt/zenoss/local/test1.sh

Figure 3: test1 Command template for /Server/Linux

A template typically includes:

- One or more **datasources.**
  - The datasource has a name; good practice suggests it should be unique.
  - The data to be gathered. In Figure 2 above, it is an SNMP Object Id (OID). In Figure 3 the data is captured using a command.
  - Whether the datasource is enabled.

- ◦ An event class may be specified to be sent if the collection mechanism fails, with the severity of that event.
- ◦ The protocol / daemon used to gather information, though this is often inherent rather than explicitly stated.
- ◦ Depending on the collection mechanism, which may be introduced by a ZenPack, the datasource may have all sorts of other features.
- ◦



Name	Source	Enabled	Type
ls1800	/opt/zenoss/local/ls1800.py	true	COMMAND
test1	/opt/zenoss/local/test1.sh	true	COMMAND
test1.countervar			GAUGE
test1.percentvar			GAUGE
test1.sizevar			GAUGE
test1.timevar			GAUGE

Edit Data Point

Name:	sizevar
Description:	
RRD Type:	GAUGE
Create Command:	
RRD Minimum:	
RRD Maximum:	
☑ Read Only	
Alias: ID / FORMULA	

Figure 4: sizevar datapoint in test1 datasource

- •
- Each datasource may have one or more **datapoints**.
  - ◦ A datapoint has a name which must be unique within the datasource definition.
  - ◦ Figure 2 shows a single datapoint, ifInOctets, for the SNMP datasource ifInOctets. Because the datasource type was selected to be SNMP, a single datapoint is automatically created with the same name as the datasource. This is because an SNMP query typically returns a single piece of data for a given OID request.
  - ◦ Figure 3 shows the test1 datasource with 4 datapoints - countervar, sizevar, timevar and percentvar. A script can deliver as many values as you want from a single script. The datasource is of type COMMAND so we get more fields to define how the command is run. The trick with a COMMAND datasource is that the script must return "Nagios-style output" which typically echos to stdout a string like:

```
"This is a test - status OK | timevar=74s sizevar=9B percentvar=10% countervar=123c"
```

  - ◦ The summary of an event is populated by the text before the vertical bar; any data values are after the bar in the format <variable name>=<variable

value>. They may be comma or space separated. The trick is that, in the template, the datapoints need to be defined such that their **names exactly match the variable names** that are output.

- ◦ The datapoint has a type - COUNTER, GAUGE, DERIVE or ABSOLUTE.
- ◦ A datapoint may have a maximum and/or minimum value specified. Sample data outside this range will not be stored.
- ◦ The datapoint may have an alias.

When creating a datapoint, you specify its name. When the datapoint is used, its full name is <datasource>.<datapoint> (in some older versions of Zenoss, you may sometimes see <datasource>_<datapoint>). So, from Figure 4, we get:

- test1.countervar
- test1.sizevar
- test1.percentvar
- test1.timevar

When data has been gathered, it is stored in Round Robin Database (rrd) files. (Note that this is changing with Zenoss 5). It is zenrrd that actually populates the files. Data files are stored on the collector responsible for a device, under a directory structure starting $ZENHOME/perf/Devices/<device id>. Device-wide data, such as memory and load average are directly under this directory. If the template includes graphs, the graphs are seen from the left-hand *Graphs* menu for the device.

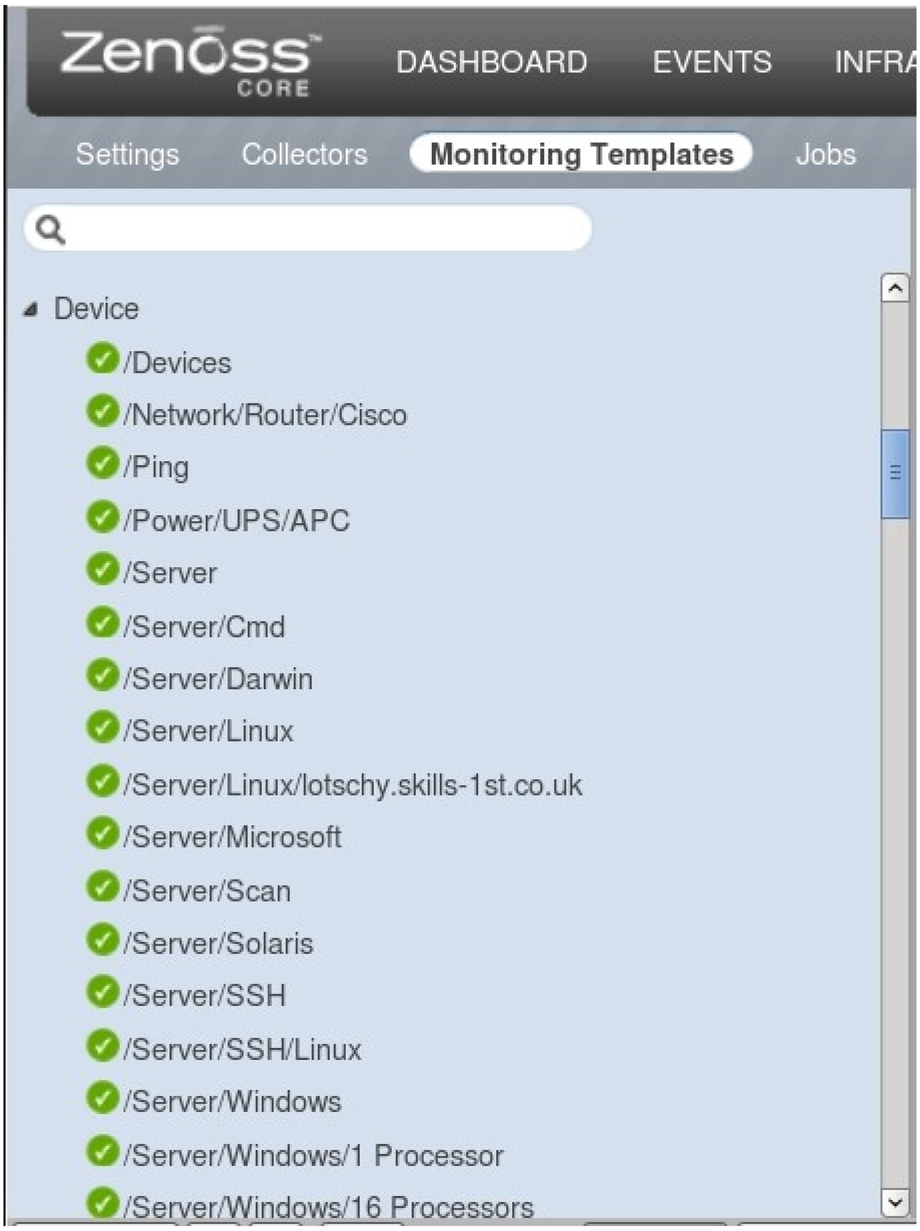
If the template is for component data, such as filesystems or interfaces, then the data files are in subdirectories such as *os/filesystems* or *os/interfaces*.

The datafile will end in *.rrd*. The main part of the filename is the datasource name concatenated with the datapoint name, separated by an underscore; so we get things like:

- test1_countervar.rrd
- os/interfaces/MS TCP Loopback interface/ifInOctets_ifInOctets.rrd

Note that these filenames do **not** include the template name. This is why it would be good practice to always have unique datasource names. If you have two templates, both with a datasource called test1, both of which have a datapoint called contervar, both of which are bound to a particular device, then you will have chaos with two templates both trying to write to the same file.

Templates are activated by **binding** them either to a device class or to a specific device. If bound to a class, the template is inherited down that class hierarchy and to the device instances. A template of the same name may be overridden at any point lower down the hierarchy, with changes. This is why there are lots of templates called Device, probably all different.



Device
- /Devices
- /Network/Router/Cisco
- /Ping
- /Power/UPS/APC
- /Server
- /Server/Cmd
- /Server/Darwin
- /Server/Linux
- /Server/Linux/lotschy.skills-1st.co.uk
- /Server/Microsoft
- /Server/Scan
- /Server/Solaris
- /Server/SSH
- /Server/SSH/Linux
- /Server/Windows
- /Server/Windows/1 Processor
- /Server/Windows/16 Processors

Figure 5: Device template with different versions for device classes and specific devices

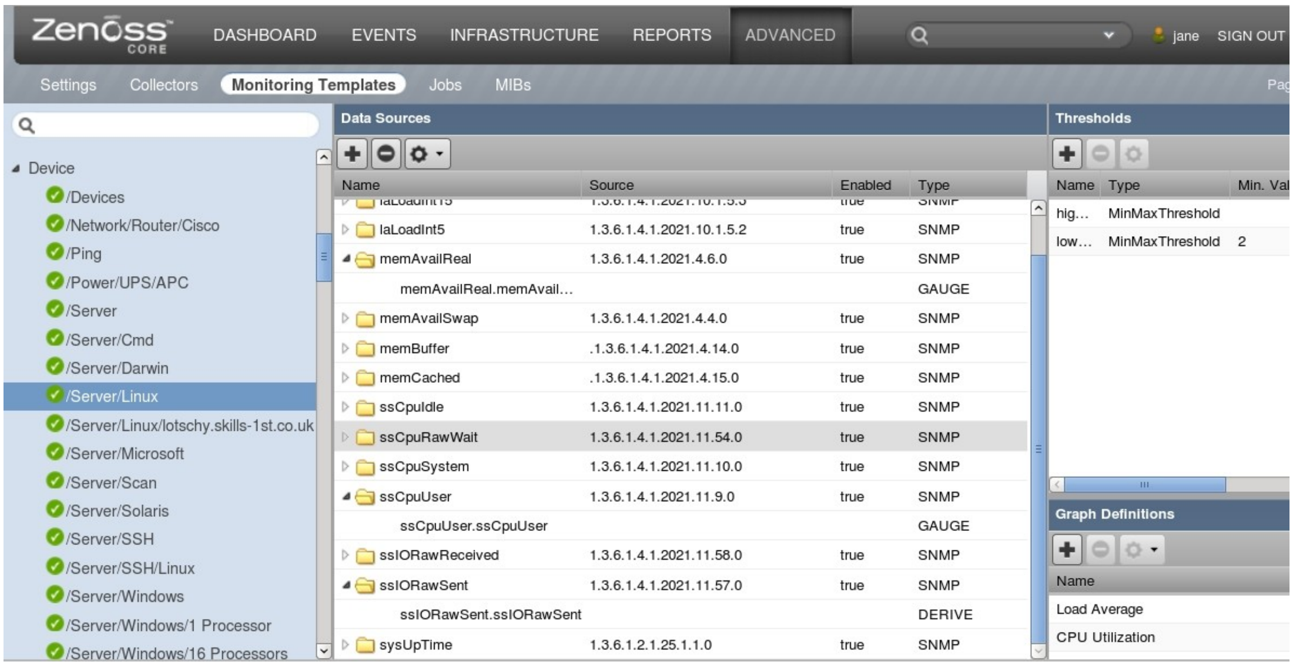
The template at the lowest point in the hierarchy is the one actually applied.

2.2.1 Device templates vs component templates

Some templates ask questions about an overall "device", Device (Server/Linux) has SNMP datasources that query cpu and memory statistics.

The "question" is asked once of the whole device. In SNMP terminology, the OID that is requested is usually a **scalar**, ie. a single number and it nearly always ends in a **.0** . Data is retrieved and stored directly under $ZENHOME/perf/Devices/<device id>.

These templates must be bound to a device class or device.



Device
- /Devices
- /Network/Router/Cisco
- /Ping
- /Power/UPS/APC
- /Server
- /Server/Cmd
- /Server/Darwin
- /Server/Linux
- /Server/Linux/lotschy.skills-1st.co.uk
- /Server/Microsoft
- /Server/Scan
- /Server/Solaris
- /Server/SSH
- /Server/SSH/Linux
- /Server/Windows
- /Server/Windows/1 Processor
- /Server/Windows/16 Processors

Data Sources

Name	Source	Enabled	Type
laLoadInt15	1.3.6.1.4.1.2021.10.1.5.3	true	SNMP
laLoadInt5	1.3.6.1.4.1.2021.10.1.5.2	true	SNMP
memAvailReal	1.3.6.1.4.1.2021.4.6.0	true	SNMP
memAvailReal.memAvail…			GAUGE
memAvailSwap	1.3.6.1.4.1.2021.4.4.0	true	SNMP
memBuffer	.1.3.6.1.4.1.2021.4.14.0	true	SNMP
memCached	.1.3.6.1.4.1.2021.4.15.0	true	SNMP
ssCpuIdle	1.3.6.1.4.1.2021.11.11.0	true	SNMP
ssCpuRawWait	1.3.6.1.4.1.2021.11.54.0	true	SNMP
ssCpuSystem	1.3.6.1.4.1.2021.11.10.0	true	SNMP
ssCpuUser	1.3.6.1.4.1.2021.11.9.0	true	SNMP
ssCpuUser.ssCpuUser			GAUGE
ssIORawReceived	1.3.6.1.4.1.2021.11.58.0	true	SNMP
ssIORawSent	1.3.6.1.4.1.2021.11.57.0	true	SNMP
ssIORawSent.ssIORawSent			DERIVE
sysUpTime	1.3.6.1.2.1.25.1.1.0	true	SNMP

Thresholds

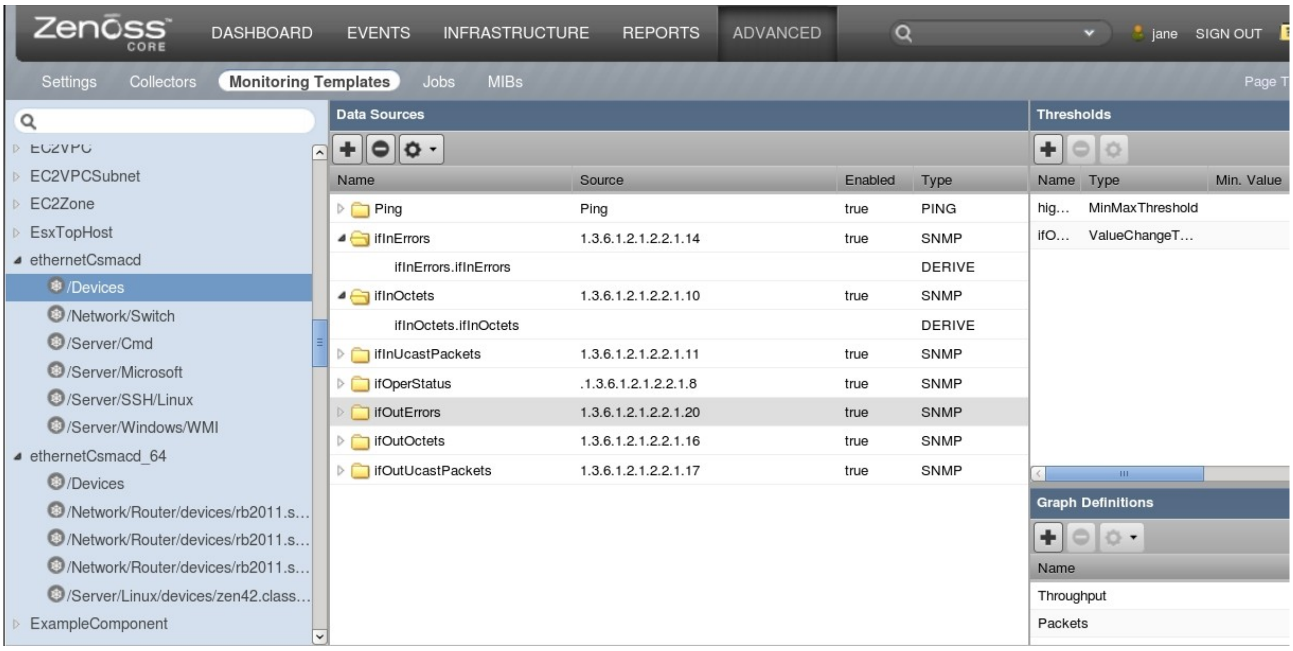
Name	Type	Min. Val
hig…	MinMaxThreshold	
low…	MinMaxThreshold	2

Graph Definitions

Name
Load Average
CPU Utilization

Figure 6: Device (/Server/Linux) template with scalar SNMP datasources and datapoints

A **component** template is rather different. Typically it gathers data for each instance of a component. The ethernetCsmacd (/Devices) template gathers several datapoints for **every** interface. With SNMP, typically the datasource specifies an OID for an SNMP **table** rather than a scalar, where the last digit of the OID refers to the instance within the table.



- EC2VPC
- EC2VPCSubnet
- EC2Zone
- EsxTopHost
- ethernetCsmacd
  - /Devices
  - /Network/Switch
  - /Server/Cmd
  - /Server/Microsoft
  - /Server/SSH/Linux
  - /Server/Windows/WMI
- ethernetCsmacd_64
  - /Devices
  - /Network/Router/devices/rb2011.s…
  - /Network/Router/devices/rb2011.s…
  - /Network/Router/devices/rb2011.s…
  - /Server/Linux/devices/zen42.class…
- ExampleComponent

Data Sources

Name	Source	Enabled	Type
Ping	Ping	true	PING
ifInErrors	1.3.6.1.2.1.2.2.1.14	true	SNMP
ifInErrors.ifInErrors			DERIVE
ifInOctets	1.3.6.1.2.1.2.2.1.10	true	SNMP
ifInOctets.ifInOctets			DERIVE
ifInUcastPackets	1.3.6.1.2.1.2.2.1.11	true	SNMP
ifOperStatus	.1.3.6.1.2.1.2.2.1.8	true	SNMP
ifOutErrors	1.3.6.1.2.1.2.2.1.20	true	SNMP
ifOutOctets	1.3.6.1.2.1.2.2.1.16	true	SNMP
ifOutUcastPackets	1.3.6.1.2.1.2.2.1.17	true	SNMP

Thresholds

Name	Type	Min. Value
hig…	MinMaxThreshold	
ifO…	ValueChangeT…	

Graph Definitions

Name
Throughput
Packets

Figure 7: ethernetCsmacd template showing OIDs for tables (no .0 on the end)

ifInOctets is one of the datasource/datapoints in the ethernetCsmacd template. Note in Figure 7 that the OID does not end in .0 . Also note in the templates dialogue that in the left-hand menu it has a grey “snowflake” icon against it; this denotes a component template.

To see what the datasource returns, use the snmpwalk utility to get the OID:

```
snmpwalk -v 2c -c public win2003net.class.example.org .1.3.6.1.2.1.2.2.1.10
        IF-MIB::ifInOctets.1 = Counter32: 1139
        IF-MIB::ifInOctets.2 = Counter32: 12594398
        IF-MIB::ifInOctets.3 = Counter32: 1453709
```

A table of values is returned, one per interface. It is the job of the datasource to work out how to assign the correct data to the correct interface.

Component templates **must not be bound by humans**! They are bound automatically when the zenmodeler daemon discovers instances of a component during the configuration cycle.

The association is made by the template (not datasource or datapoint - **template**) having **exactly** the same name as the object defining the component (it may be different if you code it that way - as with interface templates - but this is a good starting rule). Typically, you only find this information by digging in the code. The FileSystem component template is bound **automatically** to components of type FileSystem (have a look in $ZENHOME/Products/ZenModel/FileSystem.py for the object definition).

2.3 The role of zenhub in data collection

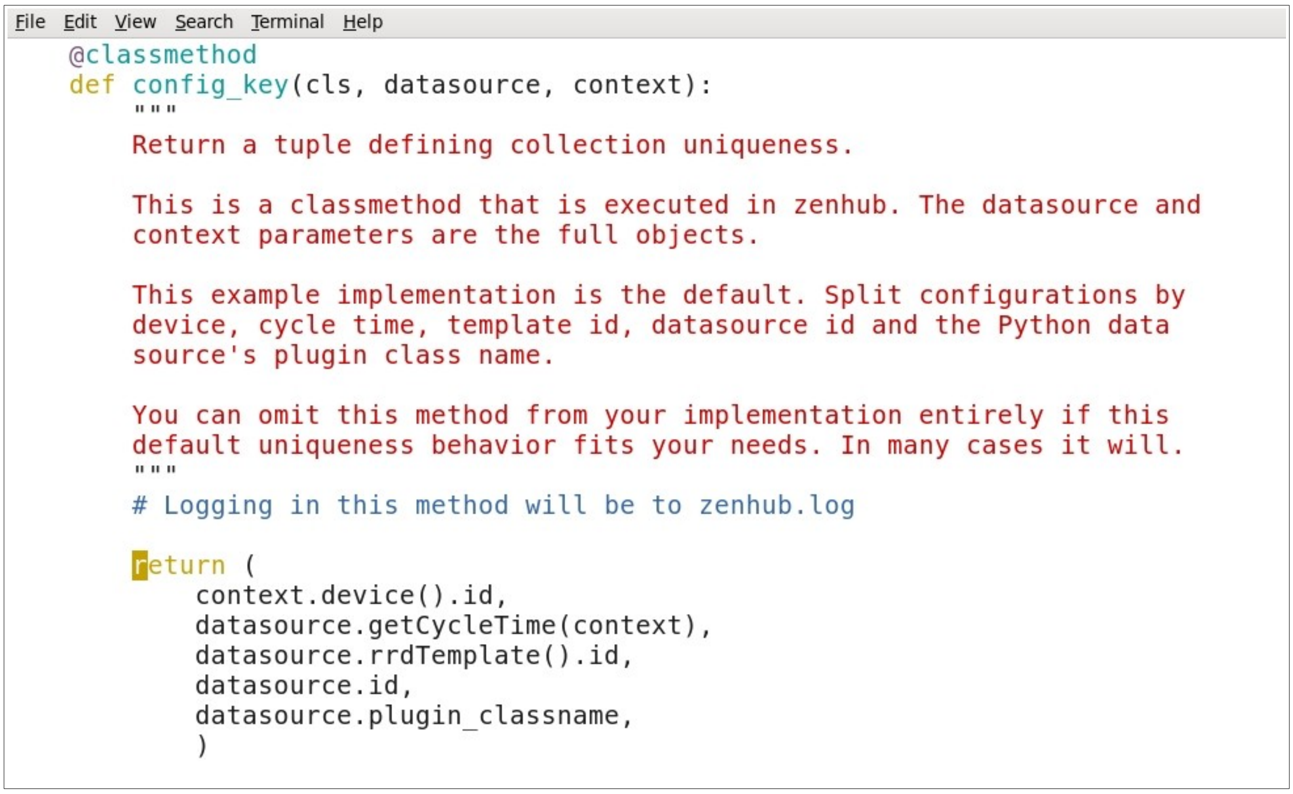
zenhub is the central Zenoss coordination daemon. Performance templates are usually defined through the graphical interface, included with the Core code, or shipped with ZenPacks. However they are defined, they are stored in the Zope database (zodb) which is implemented in MySql. In a simple implementation the MySql instance is co-located with the zenhub though it can be separated off to a separate server with both Zenoss Core and Enterprise. Fundamentally, zenhub directs access to the ZODB database.

Data collection is performed by various daemons in one or more Zenoss collectors. zenhub knows which devices are supported by which collectors. Part of the role of zenhub is to decide what template information should be sent to which collectors; the collector daemons can then build task lists of what data to gather from where. Whenever a template is changed, zenhub needs to push configuration changes to the appropriate collector(s); this now happens automatically when a template or device attribute is updated; in the past, the *Push Changes* menu option was used.

It would be wildly inefficient to individually pass every datapoint for every datasource for each object to the collectors. Each type of datasource is defined as an object class and each class is associated with a DataSourcePlugin class. The DataSourcePlugin code includes methods for zenhub to determine what data needs passing to which collectors.

2.3.1 config_key method

This method will be discussed in more detail later but for now, the comments in Figure 8 are helpful.



```
    @classmethod
    def config_key(cls, datasource, context):
        """
        Return a tuple defining collection uniqueness.

        This is a classmethod that is executed in zenhub. The datasource and
        context parameters are the full objects.

        This example implementation is the default. Split configurations by
        device, cycle time, template id, datasource id and the Python data
        source's plugin class name.

        You can omit this method from your implementation entirely if this
        default uniqueness behavior fits your needs. In many cases it will.
        """
        # Logging in this method will be to zenhub.log

        return (
            context.device().id,
            datasource.getCycleTime(context),
            datasource.rrdTemplate().id,
            datasource.id,
            datasource.plugin_classname,
            )
```

Figure 8: config_key method for a datasource plugin class

The method is run by zenhub.

It is passed the datasource object and the context object; “context” is the item that the datasource template is applied to, typically a device or a component.

It returns unique configuration “batches” where you specify what makes a config unique. The default is shown here. Note that the device id is included but not the context.id. This means that, by default, a single component datasource definition will serve for all similar components of the device, with the same datasource template, cycle time, datasource name and datasource plugin.

2.3.2 params method

A datasource may also implement a params method. Remember that data collection may be in a remote collector which does not have direct access to the zodb database. If the collection daemon needs access to zodb data, then it has to be fetched by zenhub and included in the configuration that is passed to the collection daemon.

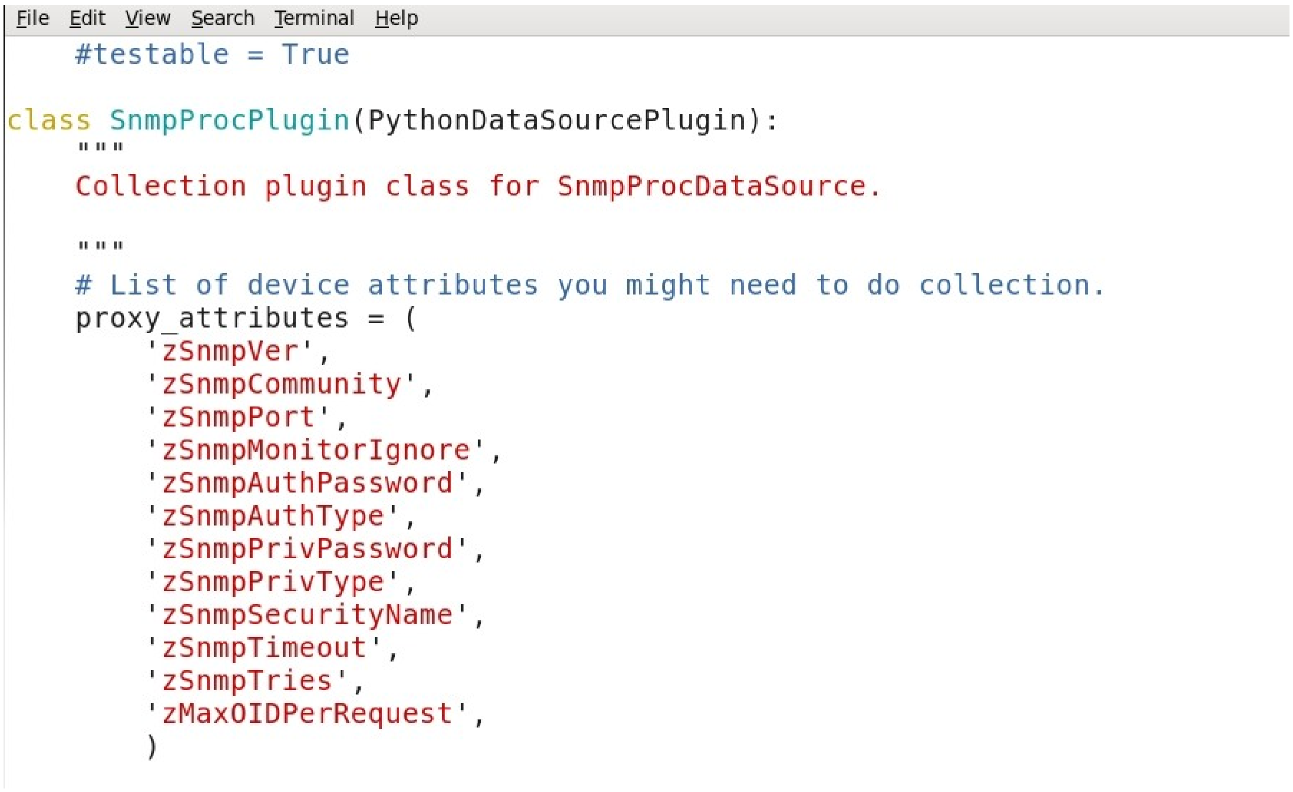
```
def params(cls, datasource, context):
        """
        Return params dictionary needed for this plugin.

        This is a classmethod that is executed in zenhub. The datasource and
        context parameters are the full objects.

        You have access to the dmd object database here and any attributes
        and methods for the context (either device or component).

        You can omit this method from your implementation if you don't require
        any additional information on each of the datasources of the config
        parameter to the collect method below. If you only need extra
        information at the device level it is easier to just use
        proxy_attributes as mentioned above.
        """
        params = {}
        params['snmpVer'] = datasource.talesEval(datasource.snmpVer, context)
        params['snmpCommunity'] = datasource.talesEval(datasource.snmpCommunity, context)

        # Get path to executable file, starting from this file
        #    which is in ZenPack base dir/datasources
        # Executables are in ZenPack base dir / libexec

        thisabspath = os.path.abspath(__file__)
        (filedir, tail) = os.path.split(thisabspath)
        libexecdir = filedir + '/../libexec'

        # script is winmem.py taking 3 parameters, hostname or IP, zSnmpVer, zSnmpCommunity
        cmdparts = [ os.path.join(libexecdir, 'winmem.py') ]

        # context is the object that the template is applied to  - either a device or a component
        #  In this case it is a device with all the attributes and methods of a device

        if context.titleOrId():
            cmdparts.append(context.titleOrId())
        elif context.manageIp:
            cmdparts.append(context.manageIp)
        else:
            cmdparts.append('UnknownHostOrIp')
```

Items of particular note in the params method above are:

- `params['snmpVer'] = datasource.talesEval(datasource.snmpVer, context)`
  - The datasource dialogue provides elements in the GUI for you to specify data. This "datasource.talesEval" construct provides access to that data (the snmpVer attribute in this case). More on this later.
- `if context.titleOrId():`
  - context is the object the template is applied to, typically a device or component. All attributes and methods of that object are available at this time. For a device, look in $ZENHOME/Products/ZenModel/Device.py for the object definition. This line uses the titleOrId() method.
- `elif context.manageIp:`
  - Similarly, this gets the manageIp attribute of the device or component.

2.3.3 proxy attributes

The proxy method provides great flexibility for accessing zodb attributes and methods of devices and components. If the requirements of the data collector are simpler - just attributes of a device - then they can be specified in a proxy_attributes statement of the DataSourcePlugin class. They are then accessed by zenhub and passed as part of the datasource configuration, to the collector.



```
    #testable = True

class SnmpProcPlugin(PythonDataSourcePlugin):
    """
    Collection plugin class for SnmpProcDataSource.

    """
    # List of device attributes you might need to do collection.
    proxy_attributes = (
        'zSnmpVer',
        'zSnmpCommunity',
        'zSnmpPort',
        'zSnmpMonitorIgnore',
        'zSnmpAuthPassword',
        'zSnmpAuthType',
        'zSnmpPrivPassword',
        'zSnmpPrivType',
        'zSnmpSecurityName',
        'zSnmpTimeout',
        'zSnmpTries',
        'zMaxOIDPerRequest',
        )
```

Figure 9: proxy_attributes for a DataSourcePlugin

3 The Python Collector ZenPack

Zenoss released the Python Collector ZenPack in 2013 - see http://wiki.zenoss.org/ZenPack:PythonCollector . I believe it only works with Zenoss 4 releases, not Zenoss 3 or earlier. There is some documentation provided on the wiki page but it is rather sparse.

The ZenPack provides:

- A new collector daemon that implements Python code - **zenpython**
- A new datasource base class - **PythonDataSource**
- With associated info and interfaces classes - **PythonDataSourceInfo** and **IPythonDataSourceInfo** to help define the layout of the dialogue for the datasource.
- A new DataSourcePlugin class - **PythonDataSourcePlugin** that actually specifies the work to be done in collecting data.
- **PythonDataSourceConfig** and **PythonConfig** classes (in the services directory) as base classes to define how zenhub handles Python datasource configurations.

Fundamentally, this daemon implements **your** Python code. Out-of-the-box, the Python Collector ZenPack does not immediately deliver anything useful.

From the wiki page.....

> “The goal of the *Python* data source type is to replicate some of the standard *COMMAND* data source type's functionality without requiring a new shell and shell subprocess to be spawned each time the data source is collected. The *COMMAND* data source type is infinitely flexible, but because of the shell and subprocess spawning, it's performance and ability to pass data into the collection script are limited. The *Python* data source type circumvents the need to spawn subprocesses by forcing the collection code to be asynchronous using the Twisted library. It circumvents the problem with passing data into the collection logic by being able to pass any basic Python data type without the need to worry about shell escaping issues. “
>
> “The *Python* data source type is intended to be used in one of two ways. The first way is directly through the creation of Python data sources through the web interface or in a ZenPack. When used in this way, it is the responsibility of the data source creator to implement the required Python class specified in the data source's *Python Class Name* property field. The second way the *Python* data source can be used is as a base class for another data source type. Used in this way, the ZenPack author will create a subclass of *PythonDataSource* to provide a higher-level functionality to the user. The user is then not responsible for writing a Python class to collect and process data. “

In practise, this means you have to create your own ZenPack to make use of zenpython. The second paragraph from the wiki talks about "creation of Python data sources through the web interface or in a ZenPack" but fundamentally you have to code a PythonDataSourcePlugin to perform the data collection, even if you then build a datasource of type *Python* through the GUI which then references your PythonDataSourcePlugin. This "first method" does avoid coding the datasource definition; that is, the GUI dialogue that allows you to specify relevant fields like Event Class and Severity, Cycle Time and Component but the downside is that you also lose that flexibility.

The "second way" to use the data source that is described in the above paragraph, does impose the responsibility of some coding to define the datasource GUI dialogue. The last sentence about "not responsible for writing a Python class to collect and process data" really is not valid. Although a skeletal PythonDataSourcePlugin class exists, all the actual collection methods are null code that you need to override in your ZenPack.

Another good reference is the Zenoss Labs ZenPack Development documentation at http://zenosslabs.readthedocs.org/en/latest/zenpack_development/monitoring_http_api/index.html . This "Monitoring an HTTP API" section provides examples of coding for the Python Collector ZenPack and uses the twisted.web.client libraries. The ZenPack code example can be found here - https://github.com/zenoss/ZenPacks.training.WeatherUnderground .

3.1 Programming with Twisted

The wiki also refers to "forcing the collection code to be asynchronous using the Twisted library" and then in the example has:

> "return somethingThatReturnsADeferred(

without giving real example code. Programming with Python "Twisted" libraries is an acquired taste and somewhat of a black art.

Fundamentally, Twisted provides methods that allow you to define a data collection routine but expects the data to be returned at some time in the future. The program does not have to wait, pending this data; it is handled asynchronously, at a later time, when the data arrives. Typically it uses a callback mechanism to link the data request with the returned data. Writing twisted code is non-trivial and debugging it is harder; printing or logging statements are often inappropriate as the data will not yet be available.

This paper is not a Twisted tutorial but it does provide examples of using both SNMP and commands through the Twisted libraries.

4 ZenPacks.Nova.Windows.SNMPPerfMonitor

To provide several different examples of using Python datasources, this paper describes the ZenPacks.skills1st.WinSnmp ZenPack. A second objective is to deliver a working ZenPack to monitor Windows devices using the SNMP protocol.

Ryan Matte, well known to the Zenoss community, has two long-established ZenPacks that provide this functionality but some of the data collection is performed through zencommand; this will be replaced by zenpython datasources. The example ZenPack will combine the functionality from both of Ryan's ZenPacks.

Note that when accessing ZenPacks through the Zenoss wiki site at http://wiki.zenoss.org/Category:ZenPacks , some of the github links may be broken. This is often because the link starts with git: rather than https: . For example:

- git://github.com/zenoss/ZenPacks.Nova.Windows.SNMPPerfMonitor.git

  can be corrected to

- https://github.com/zenoss/ZenPacks.Nova.Windows.SNMPPerfMonitor.git

4.1 Obtaining code and documentation for the ZenPack

Ryan Matte's ZenPacks.Nova.Windows.SNMPPerfMonitor is available at https://github.com/zenoss/ZenPacks.Nova.Windows.SNMPPerfMonitor . Note that this is version 1.6 and does not have process and paging monitoring. The latest 1.7 version I can only find in egg format referenced from the Zenoss wiki page at http://wiki.zenoss.org/ZenPack:Windows_SNMP_Performance_Monitor_%28Advanced%29 and the download link for the 1.7 egg is http://dmon.org/downloads/zenoss/zenpacks/zenoss4/ZenPacks.Nova.Windows.SNMPPerfMonitor-1.7-py2.7.egg .

Documentation for the ZenPack can be found at http://community.zenoss.org/docs/DOC-3386 . Note that it used to be called Windows SNMP Performance Monitor (Advanced). The description on this page (which applies to version 1.6 of the ZenPack) gives the following.

> **Description**:
> This ZenPack allows you to monitor performance data (CPU and Memory) of Windows hosts running the standard SNMP Service without needing snmp-informant installed. This ZenPack is still under development but is functional.
>
> This ZenPack adds organizers to **/Server/Windows** as follows:
>
> **/Server/Windows/1 Processor**
> **/Server/Windows/2 Processors**
> **/Server/Windows/4 Processors**
> **/Server/Windows/8 Processors**
> **/Server/Windows/16 Processors**

> Each one has a template that monitors a certain number of processors. New templates may be created if you need more than 16 processors, though the defaults should be ideal in most cases.
>
> To determine which group to place a server in perform an snmpwalk command for hrProcessorLoad as follows:
>
> ```
> snmpwalk -v1 -c <snmp string> <host> hrProcessorLoad
> ```
>
> The number of lines corresponds to the number of CPUs. Place the device in the appropriate group. This ZenPack allows each individual CPU/Core to be monitored as well as the total CPU usage. It also calculates total memory usage in percentage to make thresholding much easier. There are thresholds available for each individual CPU/Core as well, but these can be disabled if desired.

Version 1.7 of the ZenPack adds paging and number of processes to the data collected.

Note that there is a clear warning about the removal of the ZenPack:

> If you are upgrading from an older version of the pack I recommend removing it and then reinstalling it as I've made significant changes to the template which don't appear to merge well when upgrading. **In the case of this Advanced ZenPack, move all of your Windows devices out of the processor organizers to /Server/Windows so that they don't get deleted when the pack is removed. After removing and reinstalling the pack simply move the devices back in to their respective organizers.**

This comment applies to any ZenPack which creates and deletes Device Classes. If the ZenPack remove deletes a device class then all devices that have been allocated to that class will also be deleted. This does **not** apply if you reinstall the ZenPack; in that case, existing devices are preserved.

The documentation also suggests that an event transform be applied to the /Perf/Memory event class to transform the *summary* and *message* fields to report the **percentage** of memory or paging used, when a threshold is breached.

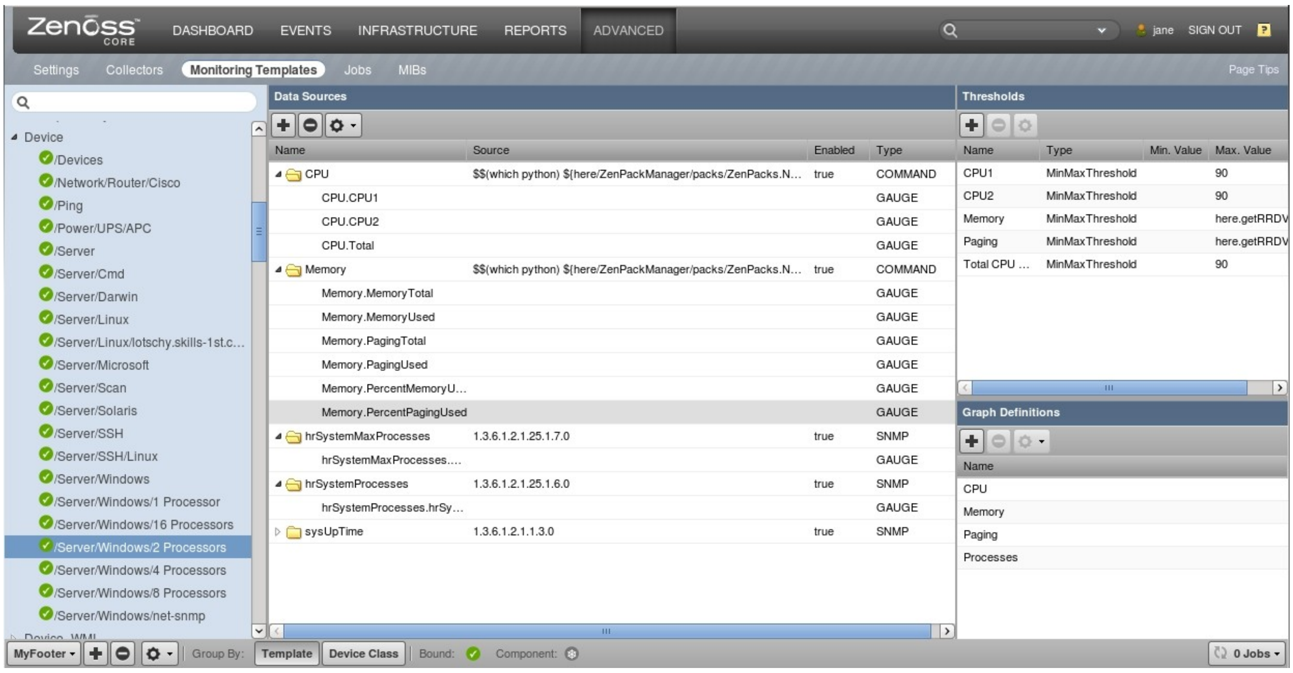
4.2 ZenPack implementation details

This is a simple ZenPack. It is really just a large number of objects in the *objects/objects.xml* file. The objects create:

- The Device classes documented above
- Performance templates for each of these device classes

All the templates are variants of a template called *Device*. The templates that gather cpu, memory and paging use zencommand to run scripts provided in the libexec directory of the ZenPack:

- wincpu.py
- winmem.py



Data Sources

Name	Source	Enabled	Type
CPU	$$(which python) $(here/ZenPackManager/packs/ZenPacks.N…	true	COMMAND
CPU.CPU1			GAUGE
CPU.CPU2			GAUGE
CPU.Total			GAUGE
Memory	$$(which python) $(here/ZenPackManager/packs/ZenPacks.N…	true	COMMAND
Memory.MemoryTotal			GAUGE
Memory.MemoryUsed			GAUGE
Memory.PagingTotal			GAUGE
Memory.PagingUsed			GAUGE
Memory.PercentMemoryU…			GAUGE
Memory.PercentPagingUsed			GAUGE
hrSystemMaxProcesses	1.3.6.1.2.1.25.1.7.0	true	SNMP
hrSystemMaxProcesses.…			GAUGE
hrSystemProcesses	1.3.6.1.2.1.25.1.6.0	true	SNMP
hrSystemProcesses.hrSy…			GAUGE
sysUpTime	1.3.6.1.2.1.1.3.0	true	SNMP

Thresholds

Name	Type	Min. Value	Max. Value
CPU1	MinMaxThreshold		90
CPU2	MinMaxThreshold		90
Memory	MinMaxThreshold		here.getRRDV
Paging	MinMaxThreshold		here.getRRDV
Total CPU …	MinMaxThreshold		90

Graph Definitions

Name
CPU
Memory
Paging
Processes

Figure 10: Device template from SNMPPerfMonitor ZenPack for /Server/Windows/2 Processors

Fundamentally all the data is collected using the SNMP protocol; however, as is not uncommon, SNMP doesn't quite provide an OID for the exact data required. The cpu, memory, and process data all comes from the standard HOST-RESOURCES mib.

For memory and paging we need total available, total used and percent used. What the hrStorage part of the host resources mib provides us with, as a table, is:

- SNMP OIDs are table values - hrStorageEntry 1.3.6.1.2.1.25.2.3.1
- The .2 column gives hrStorageType with values like hrStorageRam and hrStorageVirtualMemory
- The .4 column gives hrStorageAllocationUnits (ie block size in bytes)
- The .5 column gives hrStorageSize (ie total number of allocation units)
- The .6 column gives hrStorageUsed (ie number of allocation units used)

winmem.py

- Gathers this tabular data
- From the .2 hrStorageType it determines the OID instance that represents hrStorageRam and hrStorageVirtualMemory
- For these instances, it gathers the .4, .5 and .6 values and uses them to calculate the required datapoint values. For example:

    - MemoryTotal = (hrStorageAllocationUnits * hrStorageSize) /1.024
    - MemoryUsed = (hrStorageAllocationUnits * hrStorageUsed) / 1.024
    - PercentMemoryUsed = (MemoryUsed / MemoryTotal) * 100
- Data is gathered using the snmpwalk utility, taking parameters for the host to collect from, the SNMP version to use and the SNMP community name.
- The script echos an output line to stdout in Nagios format, with each of the datapoints after a vertical bar, in <var name> = <value> format

cpu utilisation is also provided by the HOST-RESOURCES mib where data is provided per cpu. This is why the ZenPack provides various different device classes to accommodate templates with datapoints for different hardware configurations.

wincpu.py is shown below. It provides a value for hrProcessorLoad for each CPU and an overall total.

```
#!/usr/bin/python
# Originally from Ryan Matte's ZenPacks.Nova.Windows.SNMPPerfMonitor
# Uses a command to run an snmpwalk to get cpu utilisation
# SNMP OID is actually a table of values - 1 value per processor
#   hrProcessorLoad     .1.3.6.1.2.1.25.3.3.1.2
#   For multiple CPUs, simply sum up the values
import sys
import commands

host = sys.argv[1]
snmp_ver = sys.argv[2]
community_string = sys.argv[3]

total = 0
cpu_index = 1

cpu_command = "snmpwalk -" + snmp_ver + " -c " + community_string + \
             " " + host + " .1.3.6.1.2.1.25.3.3.1.2"
(status, cpu_output) = commands.getstatusoutput(cpu_command)
if status == 0:
    cpu_list = cpu_output.split('\n')
    num_cpus = len(cpu_list)

    if num_cpus > 0:
        sys.stdout.write("OK|")
        for cpu in cpu_list:
            value = int(cpu.split(' ')[-1])
            total = total + value
            sys.stdout.write("CPU" + str(cpu_index) + "=" + str(value) + " "),
            cpu_index += 1
        sys.stdout.write("Total=" + str((total / num_cpus)) + "\n")
        sys.stdout.flush()
    else:
        print "Unknown"
else:
    print "Unknown"
```

When this script is used in a command template, the correct number of datapoints need to be provided to match the number of CPUs. See the template in Figure 10 above.

5 ZenPacks.Nova.WinServiceSNMP

5.1 Obtaining code and documentation for the ZenPack

Ryan Matte's ZenPacks.Nova.WinServiceSNMP can be found on the Zenoss wiki at http://wiki.zenoss.org/ZenPack:Windows_SNMP_Service_Monitor . The egg download is at http://dmon.org/downloads/zenoss/zenpacks/zenoss4/ZenPacks.Nova.WinServiceSNMP-1.1-py2.7.egg . I have not found the source code on github; however,I have built a development-mode tarball of the 1.7 code that is available at ?????

The documentation on the wiki page says:

> **DESCRIPTION:**
>
> The Windows SNMP Service Monitor ZenPack allows Zenoss to monitor Windows Services via SNMP. It automatically links to the /Server/Windows device class. After installing the pack you can simply remodel devices in /Server/Windows and Zenoss will discover any services running on them.
>
> Make sure that you lock the services to prevent them from being removed during a remodel while a service is down. Modeling will only pick up services that are running.
>
> **INSTALLATION:**
>
> During installation and removal the ZenPack rebuilds device relations for all devices within the /Server/Windows device class. Depending on the number of devices that you have in that class, it can take a long time. You will notice some

errors in the UI while the relations are being rebuilt, which is normal. Please be patient and allow it to complete. After the relations have been rebuilt Zenoss should be restarted. Make sure that the zenwinsrvsnmp daemon is running after the restart is performed.

ZPROPERTIES:

- zWinServiceSNMPIgnoreNames: Place the full names of any services that you want to ignore in this line by line.
- zWinServiceSNMPMonitorNames: Place the full names of any services which you explicitly want to monitor (ignoring all others) in this line by line.
- zWinServiceSNMPMonitorNamesEnable: This enables/disables the use of zWinServiceSNMPMonitorNames

Note that you need to remodel your devices for the above to take effect.

Keep in mind that zWinServiceSNMPIgnoreNames is constantly in use. If you put the same service name in both zWinServiceSNMPIgnoreNames and zWinServiceSNMPMonitorNames it will be ignored.

DAEMON:

- zenwinsrvsnmp: Make sure this daemon is running or service monitoring won't work.

TEMPLATE:

- WinServiceSNMP in /Server/Windows: This template is required for monitoring services. Do not bind this template to the device. Make sure the template is in the class that the device is in (or a higher class). The template will automatically be used for the windows services components.

MODELER PLUGIN:

- community.snmp.WinServiceMap: This plugin is required during modeling.

Note that this ZenPack only detects Windows services that are actually running. Unlike both the old and new Zenoss-provided Windows ZenPacks that use WMI and WinRM protocols respectively, there is no concept of the “Start Mode” of a service (because it is not provided by any SNMP OID).

Also note that a **component** template, WinServiceSNMP, is provided.

Note that this ZenPack modifies all existing devices under /Server/Windows when the ZenPack is installed.

5.2 ZenPack implementation details

This is a much more complex ZenPack. In addition to the WinServiceSNMP component template, code is provided to implement:

- A new device object type - WinServiceSNMPDevice
- A new component for the WinServiceSNMPDevice called WinServiceSNMP
- A new daemon to gather component data - zenwinsrvsnmp
- A new datasource, WinServiceSNMPDataSource, that runs an snmpwalk command to gather service data
- A new modeler plugin - WinServiceMap
- A new event class, /Status/WinServiceSNMP which includes a transform
- New zProperties zWinServiceSNMPIgnoreNames, zWinServiceSNMPMonitorNames and zWinServiceSNMPMonitorNamesEnable which are declared in the __init__.py of the main ZenPack directory.
- A new left-hand menu for a device, called *Services,* which shows monitoring, locking and status details of Windows services. This is implemented in a combination of the __init__.py and files under the skins subdirectory hierarchy.

When the ZenPack is installed, all devices under /Server/Windows have their zPythonClass property set to the new device type defined by ZenPacks.Nova.WinServiceSNMP.WinServiceSNMPDevice . This is only done if zPythonClass is **not** currently set for a device (ie. it won't override a value already configured).

The modeler plugin and the datasource both use SNMP to get data from the Lan Manager MIB table that delivers service information. The scSvcTable Entry (1.3.6.1.4.1.77.1.2.3.1) has entries for:

- .1 svSvcName
- .3 svSvcOperatingState where
  - active(1)
  - continue-pending(2)
  - pause-pending(3)
  - paused(4)

Both the standard Windows SNMP agent and the net-snmp Windows agent appear to support the Lan Manager MIB, in addition to the standard Mib-2 and HOST-RESOURCES mibs.

6 ZenPacks.skills1st.WinSnmp

The primary objective of this paper is to explore using the Python Collector. The various datasources and datapoints implemented in Ryan's two ZenPacks will be reimplemented to be driven by the zenpython daemon.

Some of the examples in this section you would not normally convert to a PythonDataSource; those datasources that gather simple SNMP OID values may as well be done by the zenperfsnmp daemon and its associated datasource type; however the examples here all demonstrate particular points, even though you may not choose to implement them this way in a production environment.

Every effort has been made to ensure that Ryan's ZenPacks and this Skills 1st ZenPack can coexist; hence many objects have had their name slightly changed (typically by adding "Python") to ensure there are no name clashes.

6.1 Creating a ZenPack

Do ensure that the ZenPack is in "development mode". My method for creating this ZenPack would be:

1. Create a new ZenPack through the GUI. Make a note of the name - you will need to use it in the new datasource files. This ZenPack is *ZenPacks.skills1st.WinSnmp* .
2. Do a recursive copy of the directory under $ZENHOME/ZenPacks to a local working directory and then reinstall in development mode. For example:

   ```
   cd $ZENHOME/ZenPacks
   cp -R ZenPacks.skills1st.WinSnmp /opt/zenoss/local  (assuming this exists)
   cd /opt/zenoss/local
   zenpack --link --install  ZenPacks.skills1st.WinSnmp
   ```

   - This gets you a development mode ZenPack in a working directory
3. You should restart zenhub and zopectl at this point.
4. If you inspect the directory hierarchy of the ZenPack, there will be skeleton files for many things, many of which you don't need. The main working directory of the ZenPack is:

   ```
   ZenPacks.skills1st.SnmpWin/ZenPacks/skills1st/SnmpWin
   ```
5. Under here will be directories such as:
   - datasources
   - modeler
   - libexec
   - objects

6.2 Building a Python datasource to run a script

There are two main parts to defining a datasource:

- The code that builds the GUI dialogue for your new datasource type
- The code that then gathers values to populate the datasource's datapoints. This code has several methods some of which are actually implemented by zenhub to build configurations for collectors and some are run by the zenpython daemon to collect the data.

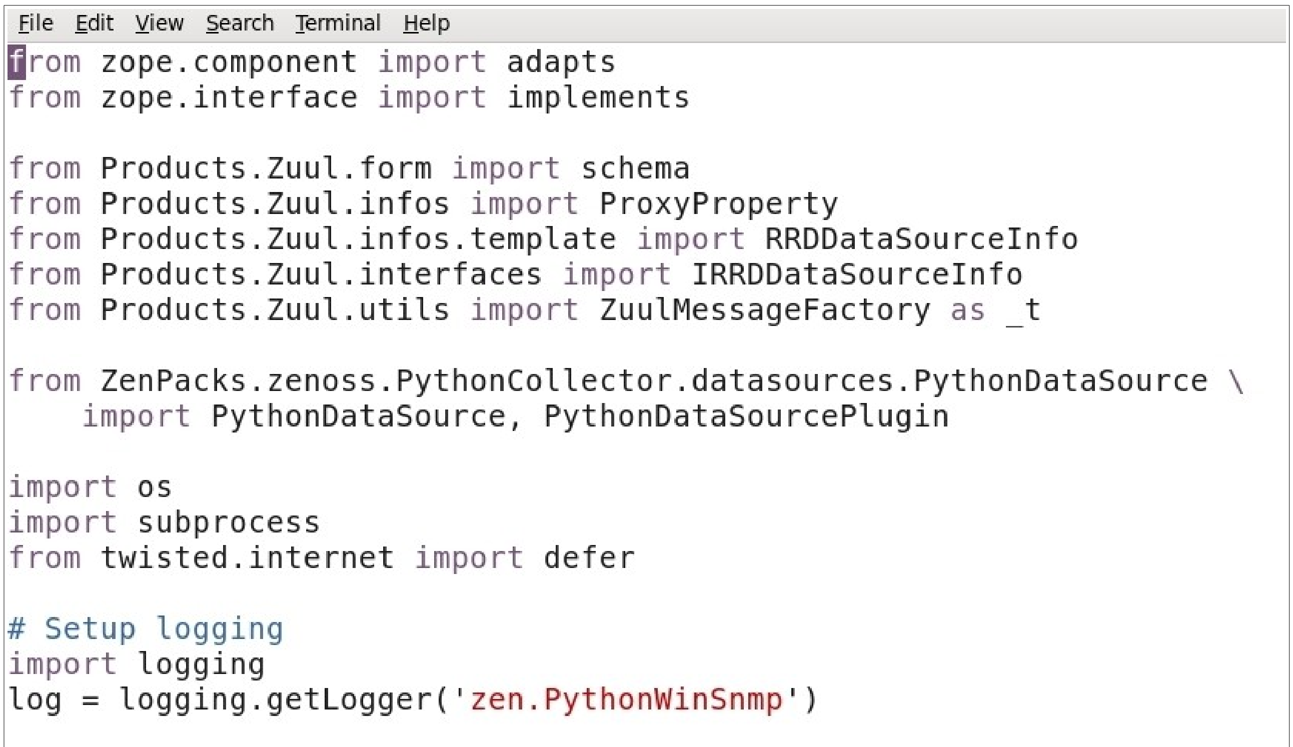
These sections are generally common whatever daemon the datasource is written for.

This first example is going to reimplement the zencommand datasource that gathers memory and paging data using a script but the script will be driven by the zenpython daemon, rather than by zencommand; however, the same *winmem.py* file will be used.

Datasource files are created in the **datasources** directory of a ZenPack.

6.2.1 Imports and logging

First there is some necessary preamble, importing prerequisite objects and methods and setting up logging.



```
from zope.component import adapts
from zope.interface import implements

from Products.Zuul.form import schema
from Products.Zuul.infos import ProxyProperty
from Products.Zuul.infos.template import RRDDataSourceInfo
from Products.Zuul.interfaces import IRRDDataSourceInfo
from Products.Zuul.utils import ZuulMessageFactory as _t

from ZenPacks.zenoss.PythonCollector.datasources.PythonDataSource \
    import PythonDataSource, PythonDataSourcePlugin

import os
import subprocess
from twisted.internet import defer

# Setup logging
import logging
log = logging.getLogger('zen.PythonWinSnmp')
```

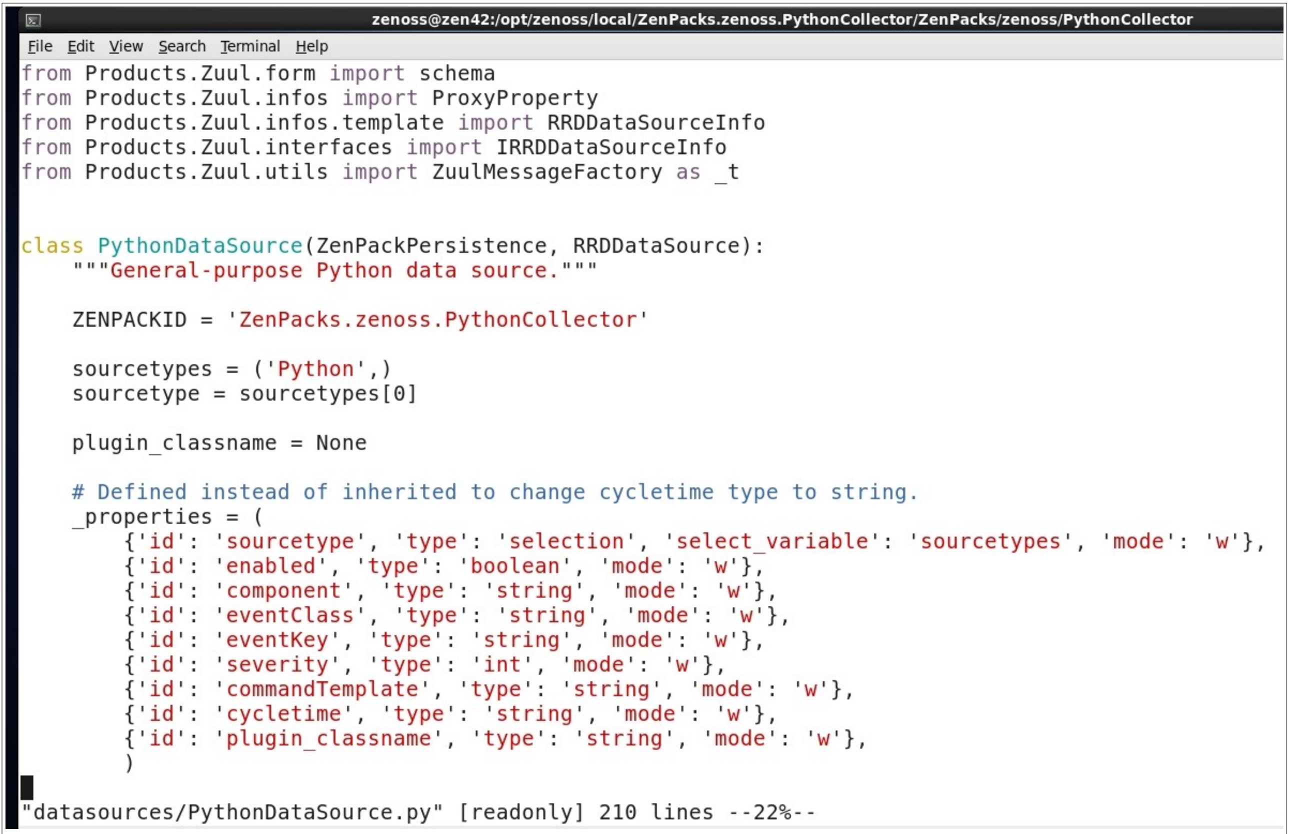
Figure 11: Preamble for the CmdSnmpMemDataSource.py file

Note the import of the PythonDataSource and PythonDataSourcePlugin from the Python Collector ZenPack.

It is good practice (and strongly advised) to set up logging. Use something unique but common throughout the ZenPack. *zen.PythonWinSnmp* is used here.

6.2.2 The datasource class

The second part defines a new DataSource class, inheriting from the PythonDataSource. Inspect the code in the Python Collector ZenPack, under datasources/PythonDataSource.py, to see the properties that are inherited by default.


zenoss@zen42:/opt/zenoss/local/ZenPacks.zenoss.PythonCollector/ZenPacks/zenoss/PythonCollector
File Edit View Search Terminal Help

```
from Products.Zuul.form import schema
from Products.Zuul.infos import ProxyProperty
from Products.Zuul.infos.template import RRDDataSourceInfo
from Products.Zuul.interfaces import IRRDDataSourceInfo
from Products.Zuul.utils import ZuulMessageFactory as _t


class PythonDataSource(ZenPackPersistence, RRDDataSource):
    """General-purpose Python data source."""

    ZENPACKID = 'ZenPacks.zenoss.PythonCollector'

    sourcetypes = ('Python',)
    sourcetype = sourcetypes[0]

    plugin_classname = None

    # Defined instead of inherited to change cycletime type to string.
    _properties = (
        {'id': 'sourcetype', 'type': 'selection', 'select_variable': 'sourcetypes', 'mode': 'w'},
        {'id': 'enabled', 'type': 'boolean', 'mode': 'w'},
        {'id': 'component', 'type': 'string', 'mode': 'w'},
        {'id': 'eventClass', 'type': 'string', 'mode': 'w'},
        {'id': 'eventKey', 'type': 'string', 'mode': 'w'},
        {'id': 'severity', 'type': 'int', 'mode': 'w'},
        {'id': 'commandTemplate', 'type': 'string', 'mode': 'w'},
        {'id': 'cycletime', 'type': 'string', 'mode': 'w'},
        {'id': 'plugin_classname', 'type': 'string', 'mode': 'w'},
        )

"datasources/PythonDataSource.py" [readonly] 210 lines --22%--
```

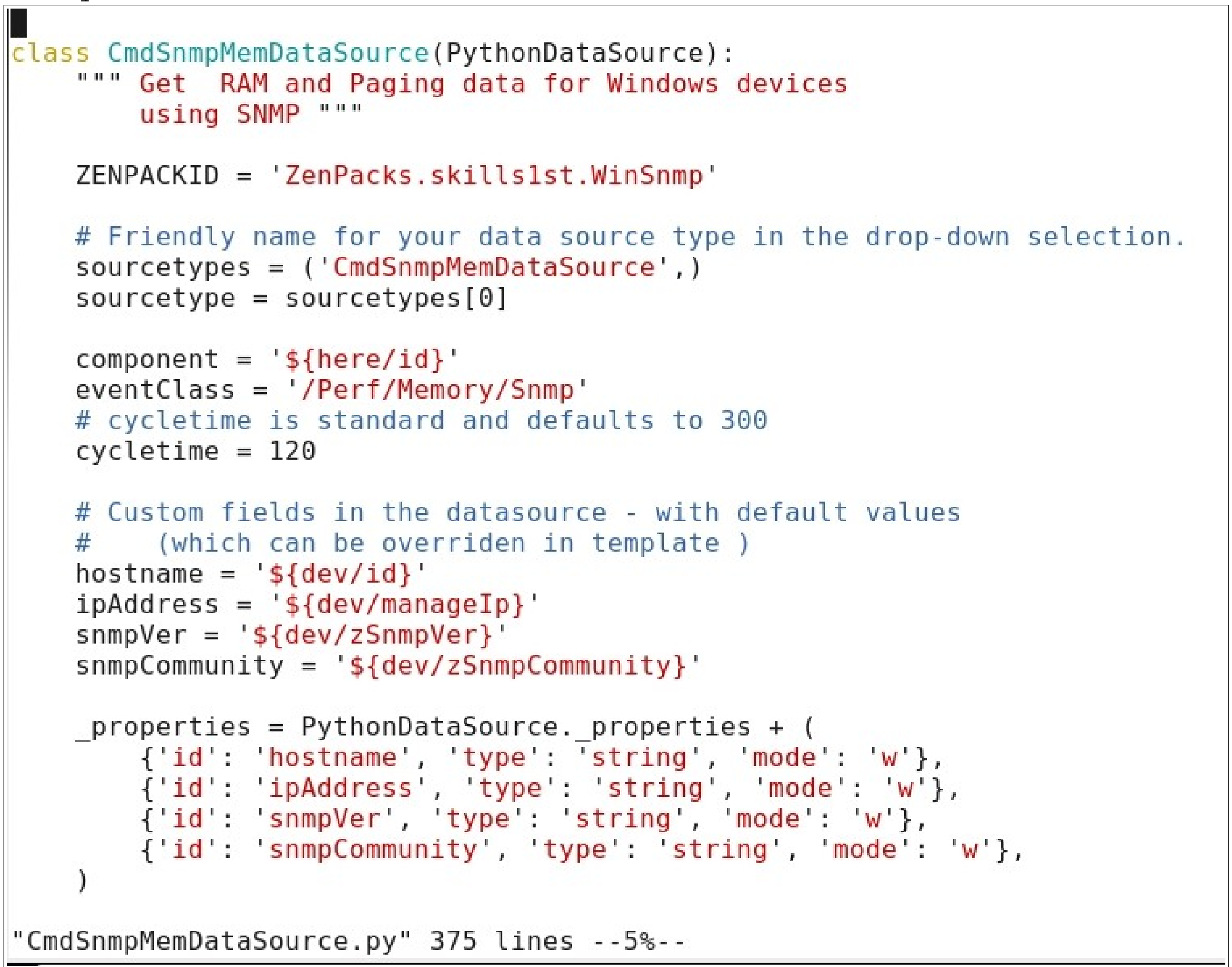
Figure 12: Python Collector ZenPack definition of PythonDataSource class

The first few lines set a ZENPACKID variable that can be used later and then defines the new source type name.

```
    ZENPACKID = 'ZenPacks.skills1st.WinSnmp'
    # Friendly name for your data source type in the drop-down selection.
    sourcetypes = ('CmdSnmpMemDataSource',)
    sourcetype = sourcetypes[0]
```

Standard attributes inherited from the parent PythonDataSource, can be given default values.

```
    component = '${here/id}'
    eventClass = '/Perf/Memory/Snmp'
    # cycletime is standard and defaults to 300
    cycletime = 120
```

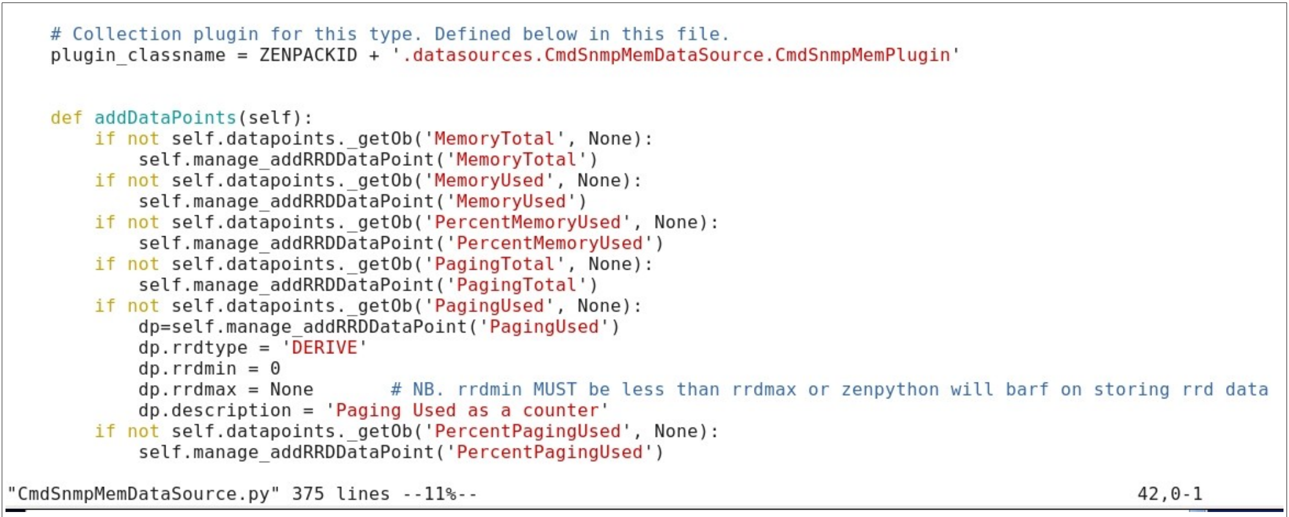


```
class CmdSnmpMemDataSource(PythonDataSource):
    """ Get  RAM and Paging data for Windows devices
        using SNMP """

    ZENPACKID = 'ZenPacks.skills1st.WinSnmp'

    # Friendly name for your data source type in the drop-down selection.
    sourcetypes = ('CmdSnmpMemDataSource',)
    sourcetype = sourcetypes[0]

    component = '${here/id}'
    eventClass = '/Perf/Memory/Snmp'
    # cycletime is standard and defaults to 300
    cycletime = 120

    # Custom fields in the datasource - with default values
    #    (which can be overriden in template )
    hostname = '${dev/id}'
    ipAddress = '${dev/manageIp}'
    snmpVer = '${dev/zSnmpVer}'
    snmpCommunity = '${dev/zSnmpCommunity}'

    _properties = PythonDataSource._properties + (
        {'id': 'hostname', 'type': 'string', 'mode': 'w'},
        {'id': 'ipAddress', 'type': 'string', 'mode': 'w'},
        {'id': 'snmpVer', 'type': 'string', 'mode': 'w'},
        {'id': 'snmpCommunity', 'type': 'string', 'mode': 'w'},
    )

"CmdSnmpMemDataSource.py" 375 lines --5%--
```

Figure 13: Start of new CmdSnmpMemDataSource class, inheriting from PythonDataSource

You can specify other fields that you want to see in the GUI dialogue for this datasource.

```
# Custom fields in the datasource - with default values
    #    (which can be overridden in template )
    hostname = '${dev/id}'
    ipAddress = '${dev/manageIp}'
    snmpVer = '${dev/zSnmpVer}'
    snmpCommunity = '${dev/zSnmpCommunity}'

    _properties = PythonDataSource._properties + (
        {'id': 'hostname', 'type': 'string', 'mode': 'w'},
        {'id': 'ipAddress', 'type': 'string', 'mode': 'w'},
        {'id': 'snmpVer', 'type': 'string', 'mode': 'w'},
        {'id': 'snmpCommunity', 'type': 'string', 'mode': 'w'},
    )
```

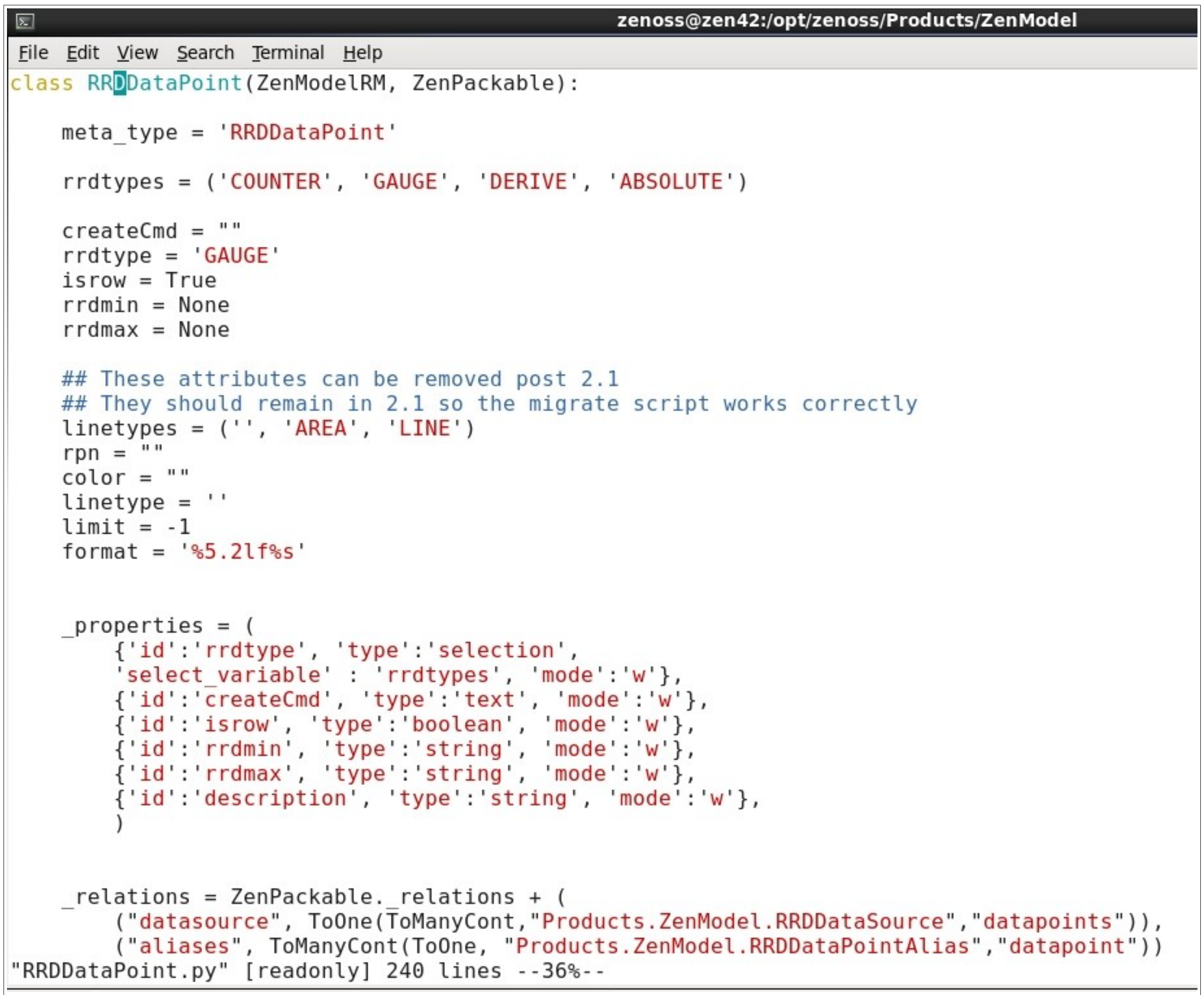


```
    # Collection plugin for this type. Defined below in this file.
    plugin_classname = ZENPACKID + '.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin'


    def addDataPoints(self):
        if not self.datapoints._getOb('MemoryTotal', None):
            self.manage_addRRDDataPoint('MemoryTotal')
        if not self.datapoints._getOb('MemoryUsed', None):
            self.manage_addRRDDataPoint('MemoryUsed')
        if not self.datapoints._getOb('PercentMemoryUsed', None):
            self.manage_addRRDDataPoint('PercentMemoryUsed')
        if not self.datapoints._getOb('PagingTotal', None):
            self.manage_addRRDDataPoint('PagingTotal')
        if not self.datapoints._getOb('PagingUsed', None):
            dp=self.manage_addRRDDataPoint('PagingUsed')
            dp.rrdtype = 'DERIVE'
            dp.rrdmin = 0
            dp.rrdmax = None       # NB. rrdmin MUST be less than rrdmax or zenpython will barf on storing rrd data
            dp.description = 'Paging Used as a counter'
        if not self.datapoints._getOb('PercentPagingUsed', None):
            self.manage_addRRDDataPoint('PercentPagingUsed')

"CmdSnmpMemDataSource.py" 375 lines --11%--                                                        42,0-1
```

Figure 14: plugin_classname and datapoints for the new datasource

The plugin_classname specifies the object path to the plugin; typically this will be in this same file.

Datapoints for the datasource can be defined using the manage_addRRDDataPoint() method. It is perfectly possible to override the default values of the datapoint attributes as shown in Figure 14 for *PagingUsed*. The fundamental definition of rrd datapoints is in $ZENHOME/Products/ZenModel/RRDDatapoint.py.



zenoss@zen42:/opt/zenoss/Products/ZenModel
File Edit View Search Terminal Help

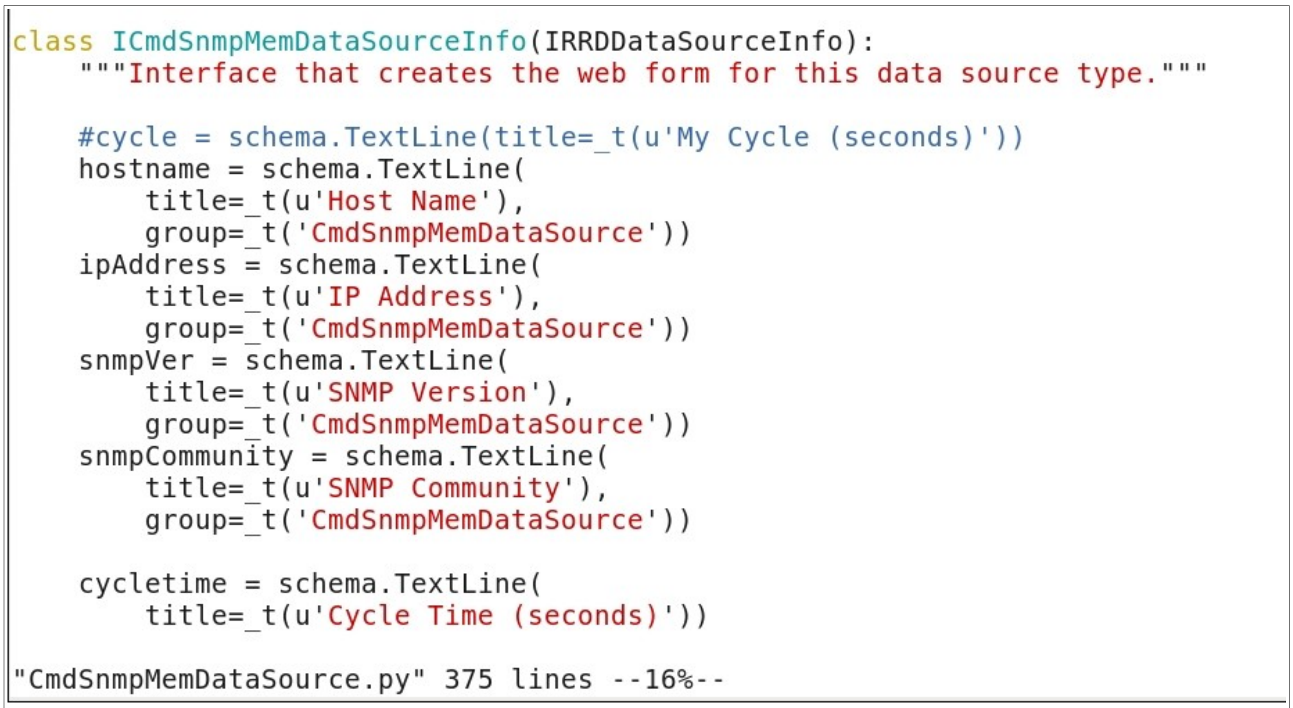
```
class RRDDataPoint(ZenModelRM, ZenPackable):

    meta_type = 'RRDDataPoint'

    rrdtypes = ('COUNTER', 'GAUGE', 'DERIVE', 'ABSOLUTE')

    createCmd = ""
    rrdtype = 'GAUGE'
    isrow = True
    rrdmin = None
    rrdmax = None

    ## These attributes can be removed post 2.1
    ## They should remain in 2.1 so the migrate script works correctly
    linetypes = ('', 'AREA', 'LINE')
    rpn = ""
    color = ""
    linetype = ''
    limit = -1
    format = '%5.2lf%s'


    _properties = (
        {'id':'rrdtype', 'type':'selection',
        'select_variable' : 'rrdtypes', 'mode':'w'},
        {'id':'createCmd', 'type':'text', 'mode':'w'},
        {'id':'isrow', 'type':'boolean', 'mode':'w'},
        {'id':'rrdmin', 'type':'string', 'mode':'w'},
        {'id':'rrdmax', 'type':'string', 'mode':'w'},
        {'id':'description', 'type':'string', 'mode':'w'},
        )


    _relations = ZenPackable._relations + (
        ("datasource", ToOne(ToManyCont,"Products.ZenModel.RRDDataSource","datapoints")),
        ("aliases", ToManyCont(ToOne, "Products.ZenModel.RRDDataPointAlias","datapoint"))
"RRDDataPoint.py" [readonly] 240 lines --36%--
```

Figure 15: RRDDatapoint.py in $ZENHOME/Products/ZenModel defines the attributes of a datapoint

6.2.3 Info and Interface definitions

Since we are defining a panel that will be supplied by the Zenoss GUI, we have to provide interface and info definitions. Some ZenPack writers choose to put such definitions in separate info.py and interfaces.py files. Here the code is included in the datasource file.

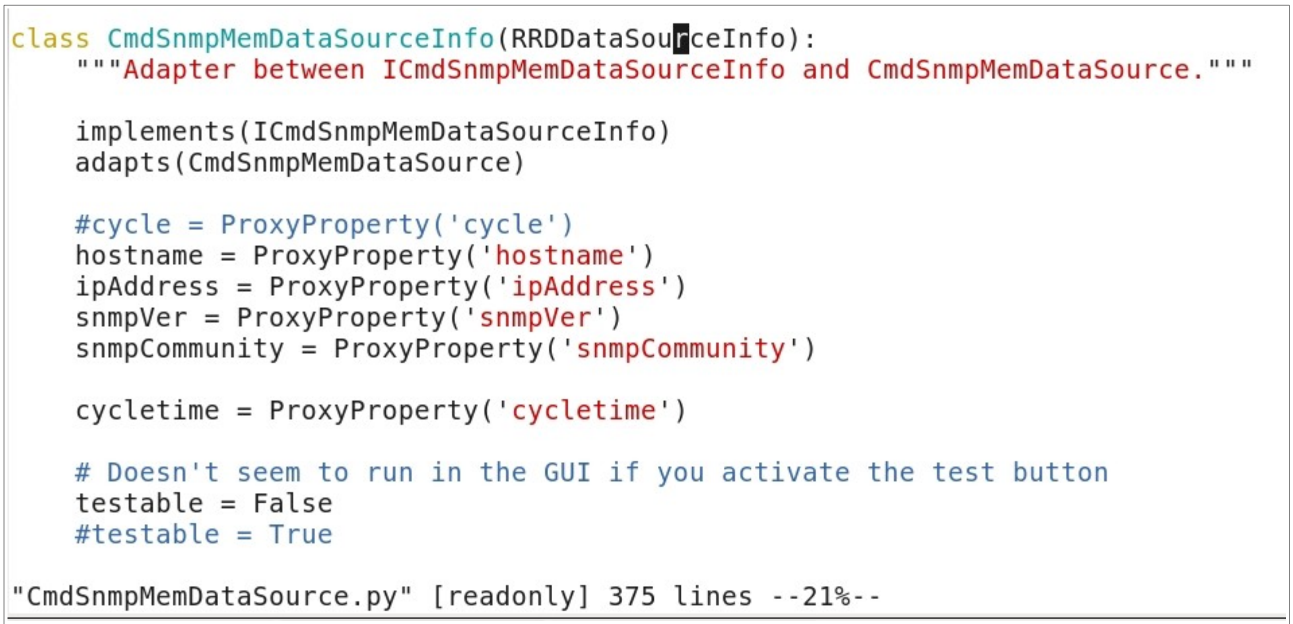
The interface information defines the text label that will appear above the relevant box in the new datasource GUI dialogue.



```
class ICmdSnmpMemDataSourceInfo(IRRDDataSourceInfo):
    """Interface that creates the web form for this data source type."""

    #cycle = schema.TextLine(title=_t(u'My Cycle (seconds)'))
    hostname = schema.TextLine(
        title=_t(u'Host Name'),
        group=_t('CmdSnmpMemDataSource'))
    ipAddress = schema.TextLine(
        title=_t(u'IP Address'),
        group=_t('CmdSnmpMemDataSource'))
    snmpVer = schema.TextLine(
        title=_t(u'SNMP Version'),
        group=_t('CmdSnmpMemDataSource'))
    snmpCommunity = schema.TextLine(
        title=_t(u'SNMP Community'),
        group=_t('CmdSnmpMemDataSource'))

    cycletime = schema.TextLine(
        title=_t(u'Cycle Time (seconds)'))

"CmdSnmpMemDataSource.py" 375 lines --16%--
```

Figure 16: Interface class definition for the new datasource

The info class links defines the new attributes as proxy properties.

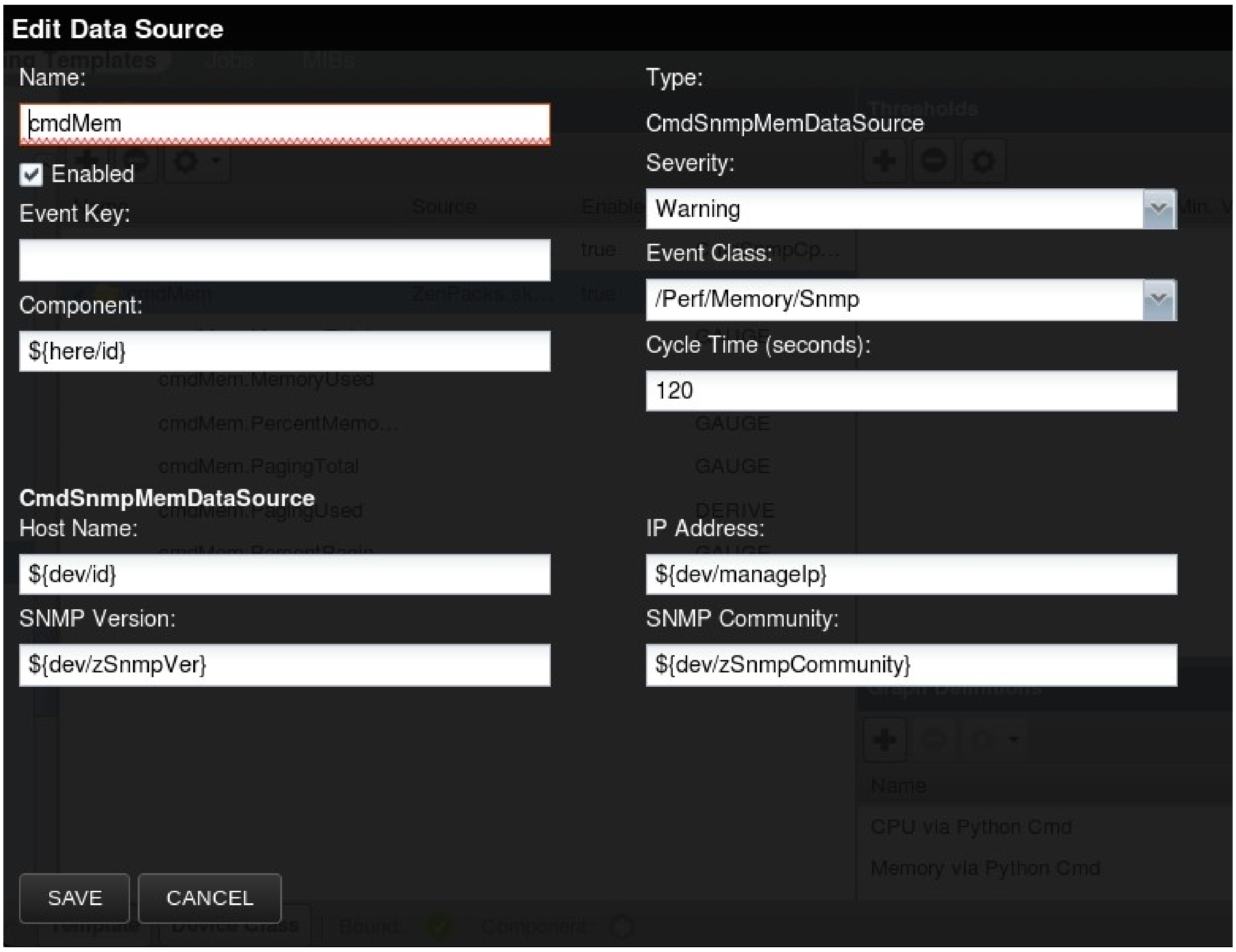


```
class CmdSnmpMemDataSourceInfo(RRDDataSourceInfo):
    """Adapter between ICmdSnmpMemDataSourceInfo and CmdSnmpMemDataSource."""

    implements(ICmdSnmpMemDataSourceInfo)
    adapts(CmdSnmpMemDataSource)

    #cycle = ProxyProperty('cycle')
    hostname = ProxyProperty('hostname')
    ipAddress = ProxyProperty('ipAddress')
    snmpVer = ProxyProperty('snmpVer')
    snmpCommunity = ProxyProperty('snmpCommunity')

    cycletime = ProxyProperty('cycletime')

    # Doesn't seem to run in the GUI if you activate the test button
    testable = False
    #testable = True

"CmdSnmpMemDataSource.py" [readonly] 375 lines --21%--
```

Figure 17: Info class definition for the new datasource

Note the “testable” attribute. This controls whether the GUI window has a “Test” button. In practise, this never seems to work for a PythonDataSource so it has been coded with a “False” value.

This completes the definition of the GUI window to define the new datasource. The result can be seen in Figure 18.



Name: cmdMem

Enabled: ✓

Event Key:

Component: ${here/id}

Type: CmdSnmpMemDataSource

Severity: Warning

Event Class: /Perf/Memory/Snmp

Cycle Time (seconds): 120

CmdSnmpMemDataSource

Host Name: ${dev/id}

IP Address: ${dev/manageIp}

SNMP Version: ${dev/zSnmpVer}

SNMP Community: ${dev/zSnmpCommunity}

Figure 18: GUI dialogue box created for the CmdSnmpMemDataSource

6.2.4 The PythonDataSourcePlugin class

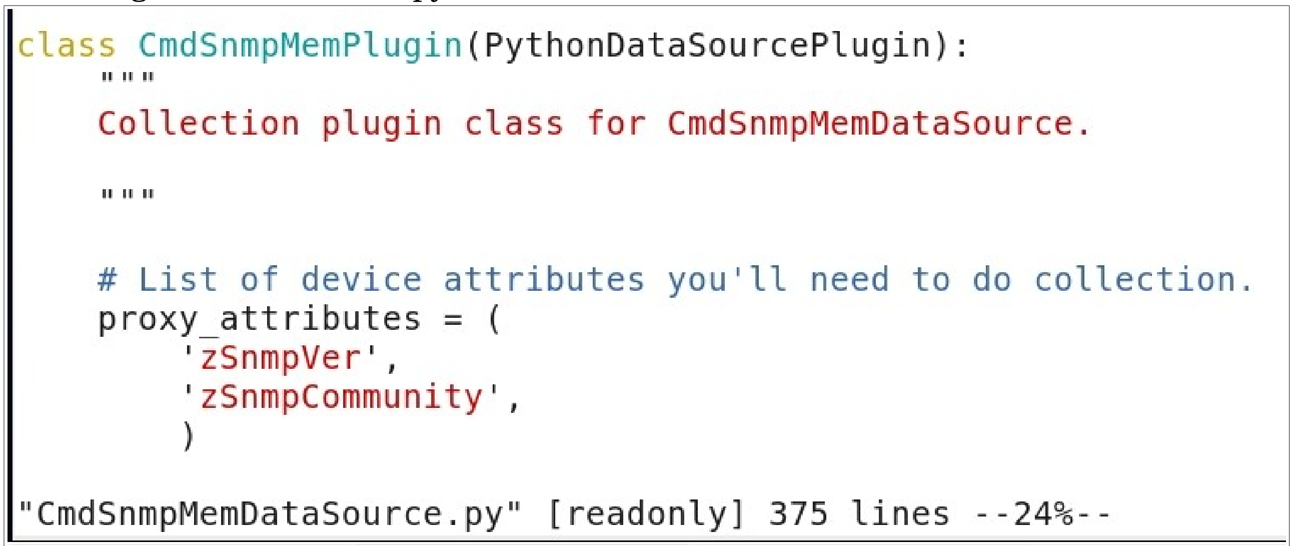
The rest of the CmdSnmpMemDataSource.py file defines the PythonDataSourcePlugin class that gathers the data.

The parent PythonDataSourcePlugin class can be inspected in the Python collector ZenPack's *datasources/PythonDataSource.py* file. A number of methods are defined, some of which are effectively dummies in this parent class, that are to be overridden in your new class:

- def config_key(cls, datasource, context):
    - Return list that is used to split configurations at the collector. This is a classmethod that is executed in zenhub. The datasource and context parameters are the full objects. Check zenhub.log for errors.
- def params(cls, datasource, context):
    - Return params dictionary needed for this plugin. This is a classmethod that is executed in zenhub. The datasource and context parameters are the full objects.

- def __init__(self, config=None):
  - Initialize the plugin with a configuration.   New in version 1.3.
- def collect(self, config):
  - No default collect behaviour. You must implement this method. This method must return a Twisted deferred. The deferred results will be sent to the onResult then either onSuccess or onError callbacks below. This method really is run by zenpython daemon. Check zenpython.log for any log messages.
- def onSuccess(self, result, config):
  - Called only on success. After onResult, before onComplete. You should return a data structure with zero or more events, values and maps. Note that values is a dictionary and events and maps are lists.
- def onError(self, result, config):
  - Called only on error. After onResult, before onComplete. You can omit this method if you want the error result of the collect method to be used without further processing. It recommended to implement this method to capture errors.
- def onComplete(self, result, config):
  - Called last for success and error. You can omit this method if you want the result of either the onSuccess or onError method to be used without further processing.
- def cleanup(self, config):
  - Called when collector exits, or task is deleted or recreated. May be omitted.

The first element in the plugin code specifies any proxy_attributes that you want to use from the Zope database. They will be accessed by zenhub and passed as part of the configuration to the zenpython daemon on the collector.



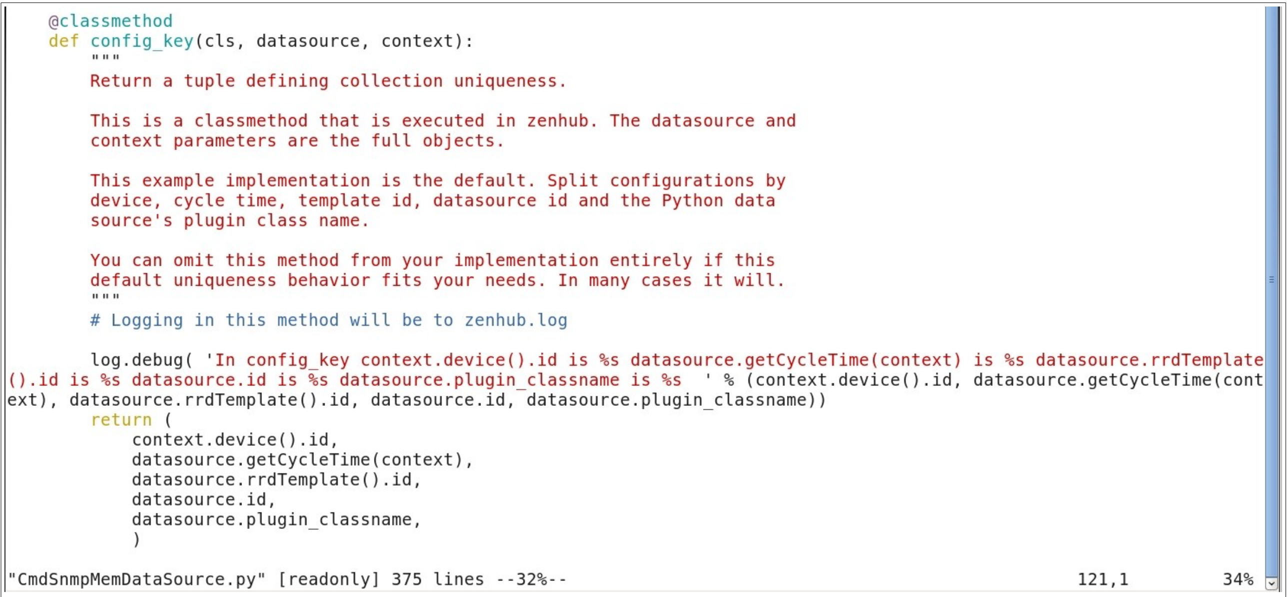
```
class CmdSnmpMemPlugin(PythonDataSourcePlugin):
    """
    Collection plugin class for CmdSnmpMemDataSource.

    """

    # List of device attributes you'll need to do collection.
    proxy_attributes = (
        'zSnmpVer',
        'zSnmpCommunity',
        )

"CmdSnmpMemDataSource.py" [readonly] 375 lines --24%--
```

Figure 19: proxy_attributes for the PythonDataSourcePlugin class

Since we shall use the SNMP protocol to gather data, the zSnmpVer and zSnmpCommunity configuration properties will be useful. This is nothing to do with the new fields that are defined in the datasource GUI. We are simply providing alternatives here.

The config_key method decides how to determine a "unique" configuration for this datasource. It is run by zenhub. The only modification from the default is to include a logging line, which will appear in zenhub.log if the zenhub daemon is run with debugging turned on. You can turn debugging on for a running zenhub, as the zenoss user, from a command prompt, with:

```
zenhub debug
less $ZENHOME/log/zenhub.log                to inspect the log file
```



```
    @classmethod
    def config_key(cls, datasource, context):
        """
        Return a tuple defining collection uniqueness.

        This is a classmethod that is executed in zenhub. The datasource and
        context parameters are the full objects.

        This example implementation is the default. Split configurations by
        device, cycle time, template id, datasource id and the Python data
        source's plugin class name.

        You can omit this method from your implementation entirely if this
        default uniqueness behavior fits your needs. In many cases it will.
        """
        # Logging in this method will be to zenhub.log

        log.debug( 'In config_key context.device().id is %s datasource.getCycleTime(context) is %s datasource.rrdTemplate
().id is %s datasource.id is %s datasource.plugin_classname is %s  ' % (context.device().id, datasource.getCycleTime(cont
ext), datasource.rrdTemplate().id, datasource.id, datasource.plugin_classname))
        return (
            context.device().id,
            datasource.getCycleTime(context),
            datasource.rrdTemplate().id,
            datasource.id,
            datasource.plugin_classname,
            )

"CmdSnmpMemDataSource.py" [readonly] 375 lines --32%--                                        121,1         34%
```

Figure 20: config_key method determines configuration uniqueness

The params method may be used to gather information from the Zope database for either a device or a component. You have access to object methods as well as attributes. params is run by zenhub.

The params method coded here is more complex than necessary but demonstrates several alternative ways of accessing information. Basically, we have two alternatives (in addition to the device proxy_attributes described above):

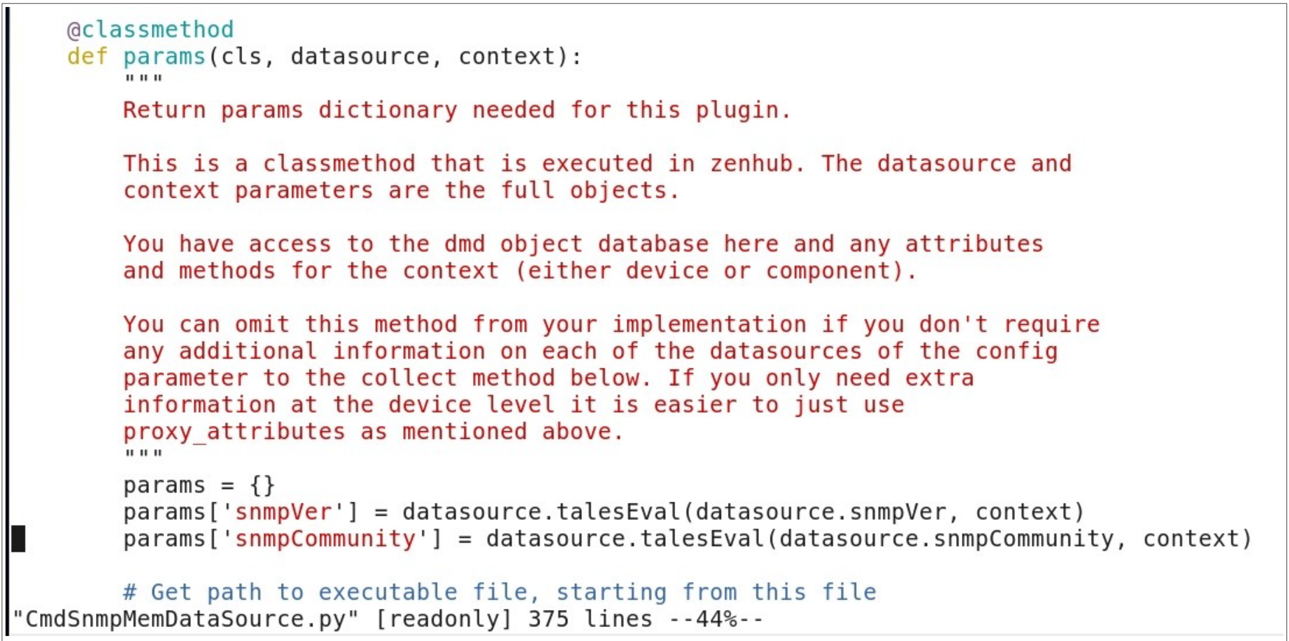
- Data values provided in the datasource GUI dialogue
- Data values retrieved from zodb

The params method returns a dictionary (typically called params).

The

`params['snmpVer'] = datasource.talesEval(datasource.snmpVer, context)` construct

provides access to the value in the datasource.



```
    @classmethod
    def params(cls, datasource, context):
        """
        Return params dictionary needed for this plugin.

        This is a classmethod that is executed in zenhub. The datasource and
        context parameters are the full objects.

        You have access to the dmd object database here and any attributes
        and methods for the context (either device or component).

        You can omit this method from your implementation if you don't require
        any additional information on each of the datasources of the config
        parameter to the collect method below. If you only need extra
        information at the device level it is easier to just use
        proxy_attributes as mentioned above.
        """
        params = {}
        params['snmpVer'] = datasource.talesEval(datasource.snmpVer, context)
        params['snmpCommunity'] = datasource.talesEval(datasource.snmpCommunity, context)

        # Get path to executable file, starting from this file
"CmdSnmpMemDataSource.py" [readonly] 375 lines --44%--
```

Figure 21: params method gathers data values from the datasource GUI dialogue

This datasource plugin is going to run the same winmem.py file that Ryan's ZenPack called. We need to build that command. The next section of the params method delivers the command, based on the assumption that winmem.py is in the ZenPack's libexec directory.

The first part finds the path to winmem.py and creates the filepath:

```
# Get path to executable file, starting from this file
        #    which is in ZenPack base dir/datasources
        # Executables are in ZenPack base dir / libexec


        thisabspath = os.path.abspath(__file__)
        (filedir, tail) = os.path.split(thisabspath)
        libexecdir = filedir + '/../libexec'


        # script is winmem.py taking 3 parameters, hostname or IP, zSnmpVer, zSnmpCommunity
        cmdparts = [ os.path.join(libexecdir, 'winmem.py') ]
```

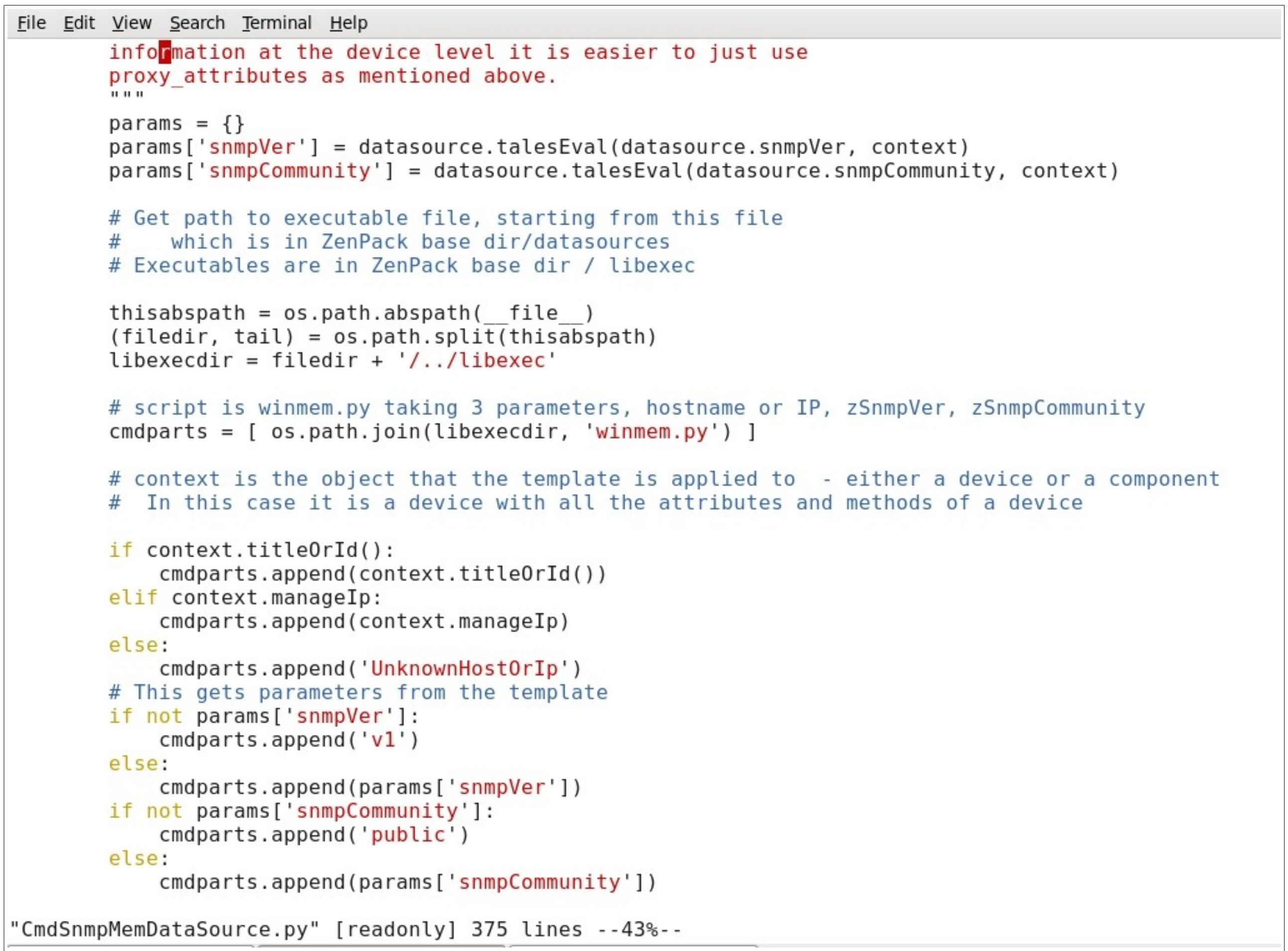
The next section provides the hostname or IP address, the snmp version and the snmp community parameters. This section uses a mixture of techniques as a demonstration. You probably wouldn't do it this way in production.

- `context.titleOrId()`
    - Uses the titleOrId() method of the device object
- `context.manageIp`
    - Looks up the manageIp attribute of the device in the zodb

      ```
      if not params['snmpVer']:
          cmdparts.append('v1')
      ```

```
    else:
        cmdparts.append(params['snmpVer'])
    if not params['snmpCommunity']:
        cmdparts.append('public')
    else:
        cmdparts.append(params['snmpCommunity'])
```

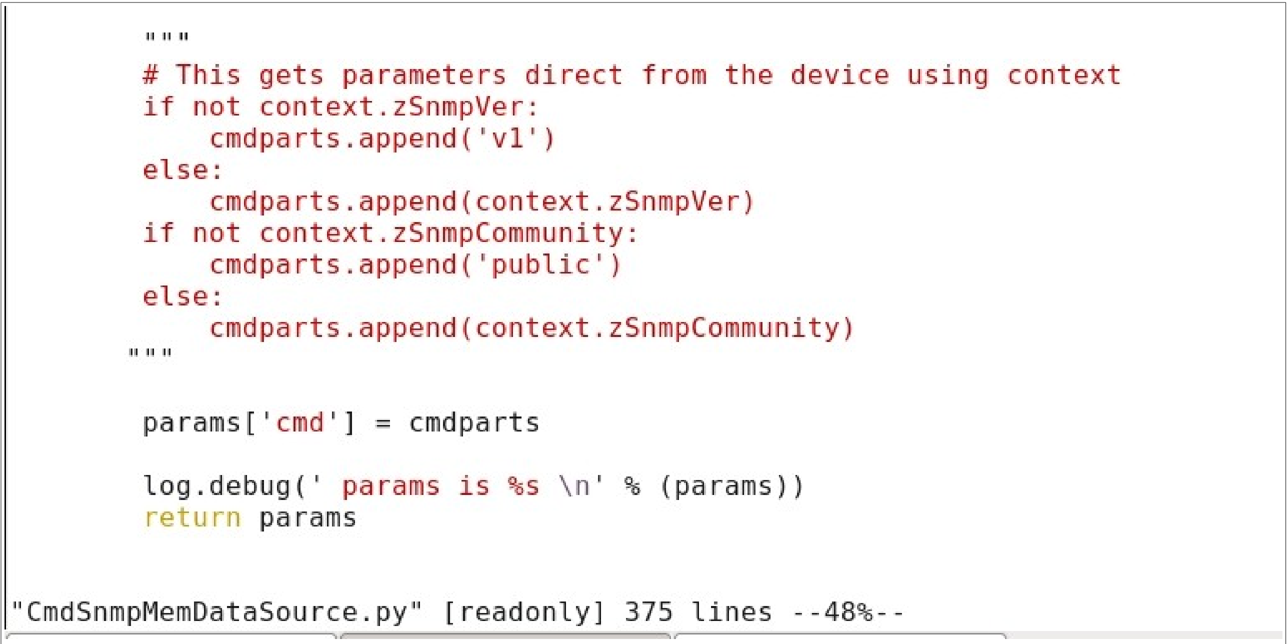
- This is the contrived part. It demonstrates that you can access the snmpCommunity and snmpVer elements of the params dictionary that you have already setup, gathering data from the datasource.



```
        information at the device level it is easier to just use
        proxy_attributes as mentioned above.
        """
        params = {}
        params['snmpVer'] = datasource.talesEval(datasource.snmpVer, context)
        params['snmpCommunity'] = datasource.talesEval(datasource.snmpCommunity, context)

        # Get path to executable file, starting from this file
        #    which is in ZenPack base dir/datasources
        # Executables are in ZenPack base dir / libexec

        thisabspath = os.path.abspath(__file__)
        (filedir, tail) = os.path.split(thisabspath)
        libexecdir = filedir + '/../libexec'

        # script is winmem.py taking 3 parameters, hostname or IP, zSnmpVer, zSnmpCommunity
        cmdparts = [ os.path.join(libexecdir, 'winmem.py') ]

        # context is the object that the template is applied to  - either a device or a component
        #  In this case it is a device with all the attributes and methods of a device

        if context.titleOrId():
            cmdparts.append(context.titleOrId())
        elif context.manageIp:
            cmdparts.append(context.manageIp)
        else:
            cmdparts.append('UnknownHostOrIp')
        # This gets parameters from the template
        if not params['snmpVer']:
            cmdparts.append('v1')
        else:
            cmdparts.append(params['snmpVer'])
        if not params['snmpCommunity']:
            cmdparts.append('public')
        else:
            cmdparts.append(params['snmpCommunity'])

"CmdSnmpMemDataSource.py" [readonly] 375 lines --43%--
```

Figure 22: params method - building the cmd key in the params dictionary

Rather than using params['snmpVer'], you may prefer to use the next ,commented-out section which gathers zProperty data from zodb.



```
        """
        # This gets parameters direct from the device using context
        if not context.zSnmpVer:
            cmdparts.append('v1')
        else:
            cmdparts.append(context.zSnmpVer)
        if not context.zSnmpCommunity:
            cmdparts.append('public')
        else:
            cmdparts.append(context.zSnmpCommunity)
       """

        params['cmd'] = cmdparts

        log.debug(' params is %s \n' % (params))
        return params


"CmdSnmpMemDataSource.py" [readonly] 375 lines --48%--
```

Figure 23: params method using zProperties of the device to create command parameters

One way or another, the cmd key of the params dictionary should look something like:

```
['/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/WinSnmp/dataso
urces/../libexec/winmem.py', 'win2003net.class.example.org', 'v2c', 'public']
```

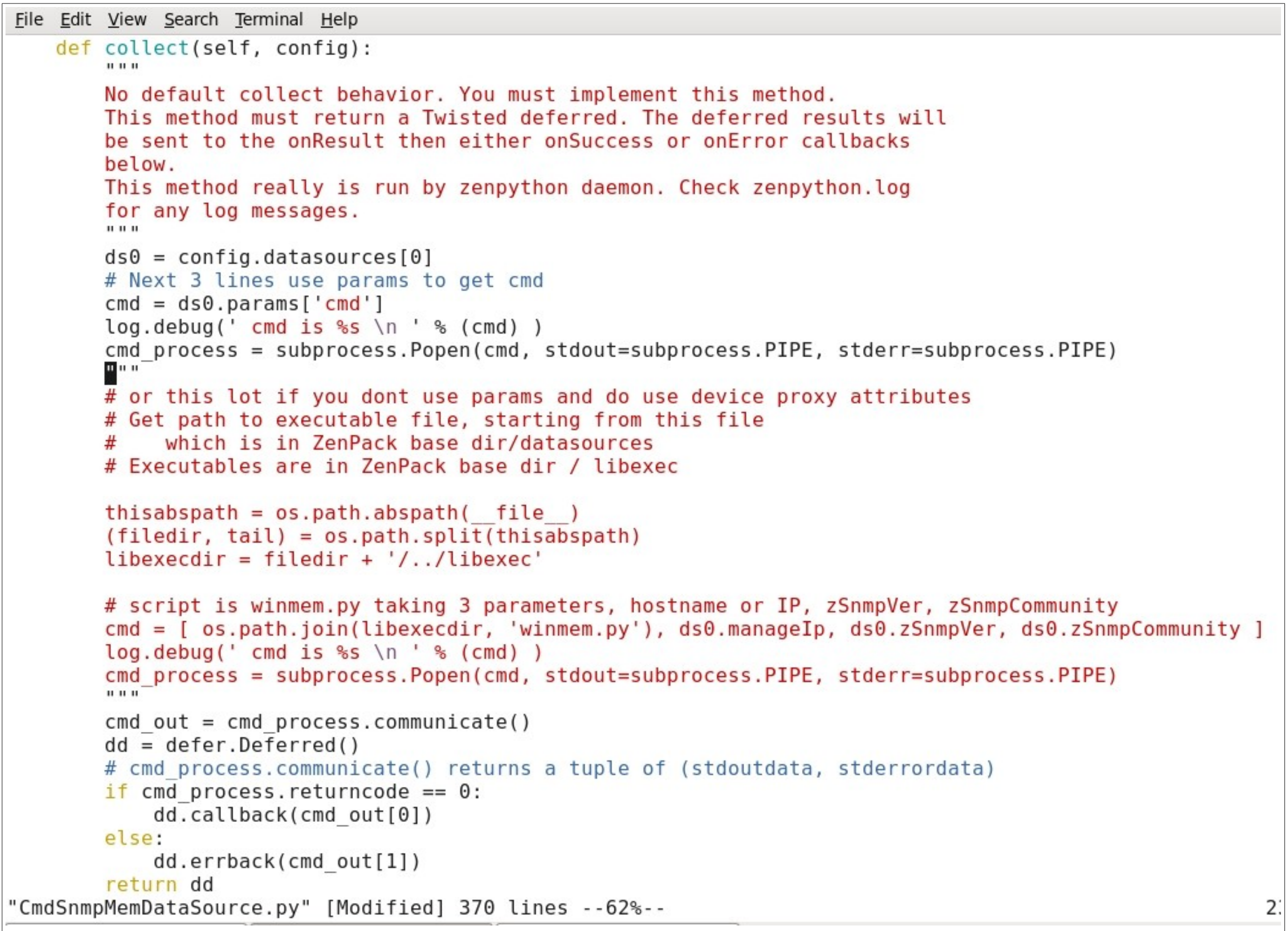
Note that cmdparts is a Python list with the command and its parameters as separate elements of the list. This is the format required to use in the subprocess.pOpen method in the collect method that follows.

The next method in the new PythonDataSourcePlugin is *collect* - the one that actually gets the data. Again, it provides two alternative ways to do things, one of them commented out.

```
    def collect(self, config):

            ds0 = config.datasources[0]

            # Next 3 lines use params to get cmd

            cmd = ds0.params['cmd']

            log.debug(' cmd is %s \n ' % (cmd) )

            cmd_process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

The collect method is run by zenpython and is passed the datasource config structure as its parameter..

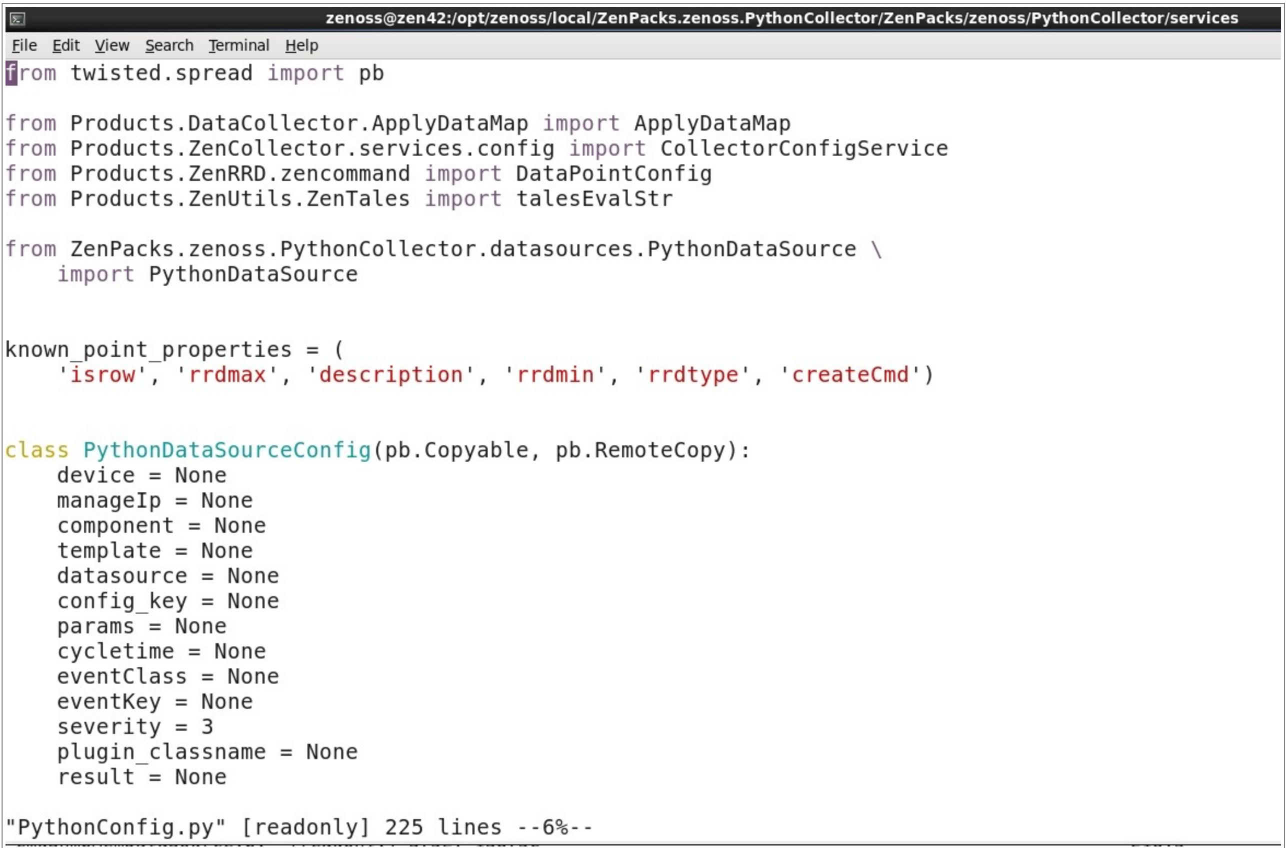
This code uses the params dictionary that we carefully built in the params method and then calls the Python subprocess.Popen method to gather data.



```
    def collect(self, config):
        """
        No default collect behavior. You must implement this method.
        This method must return a Twisted deferred. The deferred results will
        be sent to the onResult then either onSuccess or onError callbacks
        below.
        This method really is run by zenpython daemon. Check zenpython.log
        for any log messages.
        """
        ds0 = config.datasources[0]
        # Next 3 lines use params to get cmd
        cmd = ds0.params['cmd']
        log.debug(' cmd is %s \n ' % (cmd) )
        cmd_process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        """
        # or this lot if you dont use params and do use device proxy attributes
        # Get path to executable file, starting from this file
        #    which is in ZenPack base dir/datasources
        # Executables are in ZenPack base dir / libexec

        thisabspath = os.path.abspath(__file__)
        (filedir, tail) = os.path.split(thisabspath)
        libexecdir = filedir + '/../libexec'

        # script is winmem.py taking 3 parameters, hostname or IP, zSnmpVer, zSnmpCommunity
        cmd = [ os.path.join(libexecdir, 'winmem.py'), ds0.manageIp, ds0.zSnmpVer, ds0.zSnmpCommunity ]
        log.debug(' cmd is %s \n ' % (cmd) )
        cmd_process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        """
        cmd_out = cmd_process.communicate()
        dd = defer.Deferred()
        # cmd_process.communicate() returns a tuple of (stdoutdata, stderrordata)
        if cmd_process.returncode == 0:
            dd.callback(cmd_out[0])
        else:
            dd.errback(cmd_out[1])
        return dd
"CmdSnmpMemDataSource.py" [Modified] 370 lines --62%--                                         2:
```

Figure 24: collect method for new datasource plugin

The alternative, commented-out code completely ignores the work done in the params method. We use the same code to build the filepath for the command, directly in the collect method. The parameters come from the proxy_attributes we defined at the start of the plugin method (zSnmpVer and zSnmpCommunity). We did not specify the manageIp attribute but some attributes of the data source are passed by default in the config parameter. The defaults can be seen in the code in the Python Collector ZenPack under services/PythonConfig.py.


zenoss@zen42:/opt/zenoss/local/ZenPacks.zenoss.PythonCollector/ZenPacks/zenoss/PythonCollector/services
File Edit View Search Terminal Help

```
from twisted.spread import pb

from Products.DataCollector.ApplyDataMap import ApplyDataMap
from Products.ZenCollector.services.config import CollectorConfigService
from Products.ZenRRD.zencommand import DataPointConfig
from Products.ZenUtils.ZenTales import talesEvalStr

from ZenPacks.zenoss.PythonCollector.datasources.PythonDataSource \
    import PythonDataSource


known_point_properties = (
    'isrow', 'rrdmax', 'description', 'rrdmin', 'rrdtype', 'createCmd')


class PythonDataSourceConfig(pb.Copyable, pb.RemoteCopy):
    device = None
    manageIp = None
    component = None
    template = None
    datasource = None
    config_key = None
    params = None
    cycletime = None
    eventClass = None
    eventKey = None
    severity = 3
    plugin_classname = None
    result = None

"PythonConfig.py" [readonly] 225 lines --6%--
```

Figure 25: Default attributes of a PythonDataSourceConfig class

The last part of the collect method arranges for the command to be run and return deferred output.

```
        cmd_out = cmd_process.communicate()
        dd = defer.Deferred()
        # cmd_process.communicate() returns a tuple of (stdoutdata, stderrordata)
        if cmd_process.returncode == 0:
            dd.callback(cmd_out[0])
        else:
            dd.errback(cmd_out[1])
        return dd
```

The trick here is that collect must return a "Twisted deferred". So we specify our output to be delivered to cmd_out - and then defer it! Data is delivered asynchronously at a later stage. If the returncode is 0 (successful) then we continue with the success callback; if it is not successful we continue with the error callback.

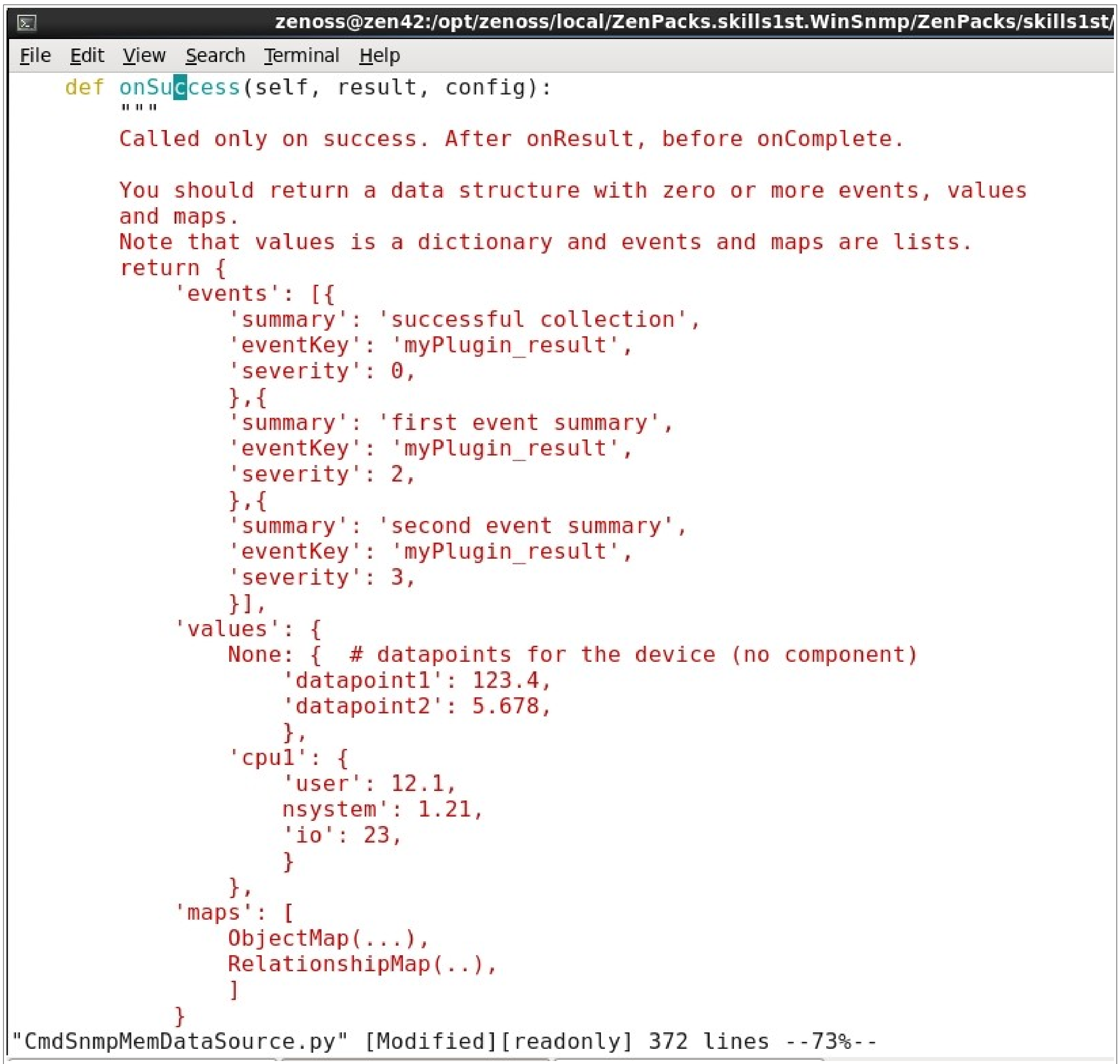
The onResult method of the plugin can be omitted. In this case, it is just used to provide logging (to zenpython.log).

```
    def onResult(self, result, config):
        """
        Called first for success and error.

        You can omit this method if you want the result of the collect method
        to be used without further processing.
        """
        log.debug( 'result is %s ' % (result))

        return result
```

onSuccess is the last major method to implement. It need to process the data that was successfully returned by the collect method. It must deliver a data structure with zero or more events, values and maps. The data in the values dictionary is then used to populate the rrd files.

The trick with the values dictionary is that it is a dictionary of dictionaries with one dictionary per component datapoint, where the keys are the components and the values are the data. Data values for a device (rather than a component) have the *None* key.

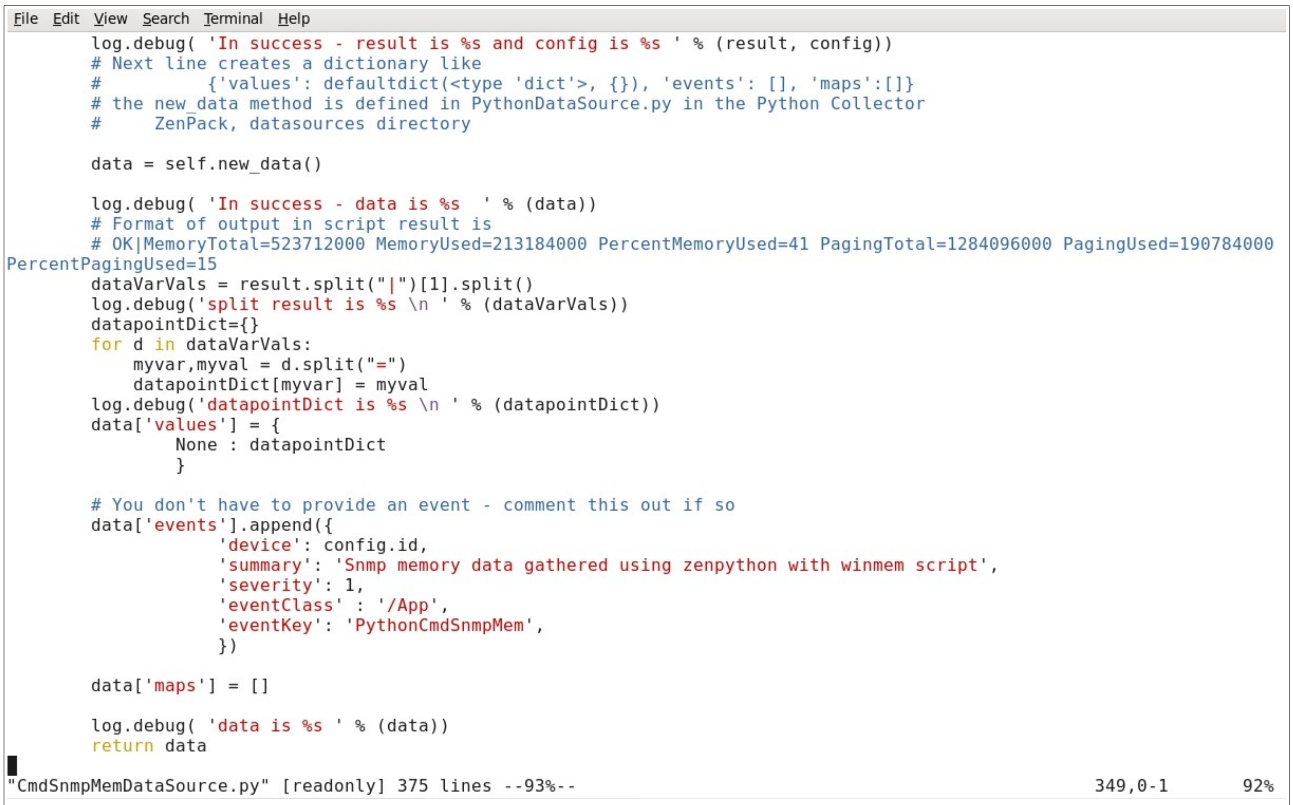
Note that the events and maps data structures are lists, not dictionaries. You do not have to return an events list; in the onSuccess method you may prefer not to; it will default to an empty list.


zenoss@zen42:/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/
File Edit View Search Terminal Help

```
    def onSuccess(self, result, config):
        """
        Called only on success. After onResult, before onComplete.

        You should return a data structure with zero or more events, values
        and maps.
        Note that values is a dictionary and events and maps are lists.
        return {
            'events': [{
                'summary': 'successful collection',
                'eventKey': 'myPlugin_result',
                'severity': 0,
                },{
                'summary': 'first event summary',
                'eventKey': 'myPlugin_result',
                'severity': 2,
                },{
                'summary': 'second event summary',
                'eventKey': 'myPlugin_result',
                'severity': 3,
                }],
            'values': {
                None: {  # datapoints for the device (no component)
                    'datapoint1': 123.4,
                    'datapoint2': 5.678,
                    },
                'cpu1': {
                    'user': 12.1,
                    nsystem': 1.21,
                    'io': 23,
                    }
                },
            'maps': [
                ObjectMap(...),
                RelationshipMap(..),
                ]
            }
"CmdSnmpMemDataSource.py" [Modified][readonly] 372 lines --73%--
```

Figure 26: onSuccess method for the new data plugin - what you need to return

The maps list can be used to modify attributes of the device or component in the zodb database, in the light of the performance data retrieved.



```
        log.debug( 'In success - result is %s and config is %s ' % (result, config))
        # Next line creates a dictionary like
        #           {'values': defaultdict(<type 'dict'>, {}), 'events': [], 'maps':[]}
        # the new_data method is defined in PythonDataSource.py in the Python Collector
        #     ZenPack, datasources directory

        data = self.new_data()

        log.debug( 'In success - data is %s  ' % (data))
        # Format of output in script result is
        # OK|MemoryTotal=523712000 MemoryUsed=213184000 PercentMemoryUsed=41 PagingTotal=1284096000 PagingUsed=190784000
PercentPagingUsed=15
        dataVarVals = result.split("|")[1].split()
        log.debug('split result is %s \n ' % (dataVarVals))
        datapointDict={}
        for d in dataVarVals:
            myvar,myval = d.split("=")
            datapointDict[myvar] = myval
        log.debug('datapointDict is %s \n ' % (datapointDict))
        data['values'] = {
                None : datapointDict
                }

        # You don't have to provide an event - comment this out if so
        data['events'].append({
                    'device': config.id,
                    'summary': 'Snmp memory data gathered using zenpython with winmem script',
                    'severity': 1,
                    'eventClass' : '/App',
                    'eventKey': 'PythonCmdSnmpMem',
                    })

        data['maps'] = []

        log.debug( 'data is %s ' % (data))
        return data

"CmdSnmpMemDataSource.py" [readonly] 375 lines --93%--                          349,0-1       92%
```

Figure 27: The main body of the onSuccess method, processing the returned data

First we use the new_data() method to initiate a new, empty datastructure to return.

This datasource is running the winmem.py script to gather device-wide data - single scalar values. The output is in the format:

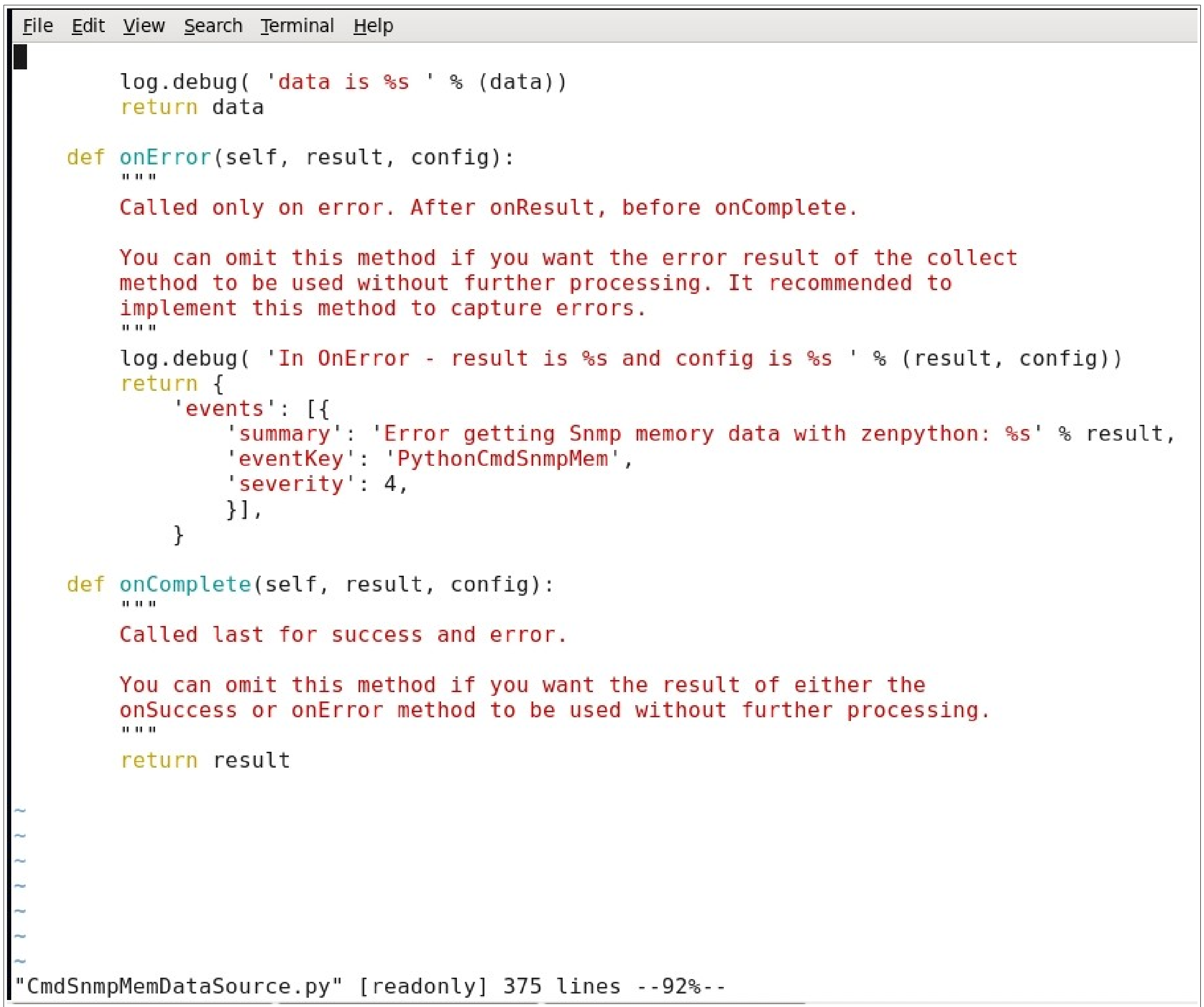
```
OK|MemoryTotal=523712000 MemoryUsed=212800000 PercentMemoryUsed=41
PagingTotal=1284096000 PagingUsed=204032000 PercentPagingUsed=16
```

Note that the variable names in the section after the vertical bar, exactly match the datapoint names defined earlier in this datasource file.

We use Python code to split the output into sections before and after the vertical bar. The variables sections are then split on the "=" symbol and parsed into a new dictionary, *datapointDict*, where the key is the variable name and the the value is the data value. This is now in the correct format to return as the value part of the "None" key (device-wide values), in the returned data dictionary.

As a demonstration, an event is also generated with Info severity, of event class /App. config.id refers to the device itself so is passed as the device attribute of the event.

The onError and onComplete methods are fairly trivial. onError delivers an event, similar to the one above but with an Error severity.


File Edit View Search Terminal Help

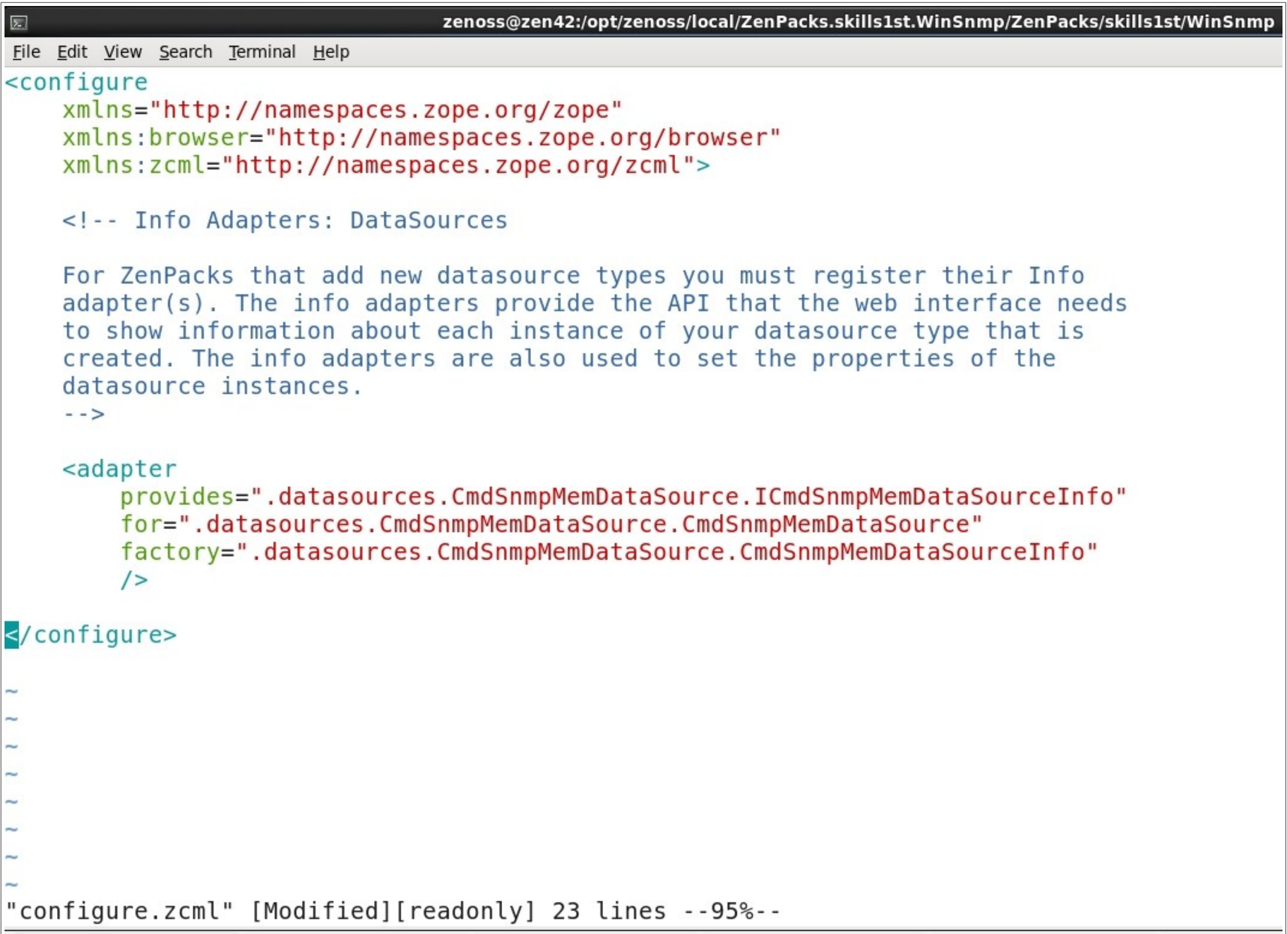
```
        log.debug( 'data is %s ' % (data))
        return data

    def onError(self, result, config):
        """
        Called only on error. After onResult, before onComplete.

        You can omit this method if you want the error result of the collect
        method to be used without further processing. It recommended to
        implement this method to capture errors.
        """
        log.debug( 'In OnError - result is %s and config is %s ' % (result, config))
        return {
            'events': [{
                'summary': 'Error getting Snmp memory data with zenpython: %s' % result,
                'eventKey': 'PythonCmdSnmpMem',
                'severity': 4,
                }],
            }

    def onComplete(self, result, config):
        """
        Called last for success and error.

        You can omit this method if you want the result of either the
        onSuccess or onError method to be used without further processing.
        """
        return result
~
~
~
~
~
~
~
"CmdSnmpMemDataSource.py" [readonly] 375 lines --92%--
```

Figure 28: onError and onComplete methods for the new datasource plugin

That's all there is to it!

6.3 configure.zcml

There is one other thing you need to do before you can test the datasource. The configure.zcml file in the main directory of the ZenPack, needs lines added to define the adapter for the new datasource.


zenoss@zen42:/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/WinSnmp
File Edit View Search Terminal Help

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:browser="http://namespaces.zope.org/browser"
    xmlns:zcml="http://namespaces.zope.org/zcml">

    <!-- Info Adapters: DataSources

    For ZenPacks that add new datasource types you must register their Info
    adapter(s). The info adapters provide the API that the web interface needs
    to show information about each instance of your datasource type that is
    created. The info adapters are also used to set the properties of the
    datasource instances.
    -->

    <adapter
        provides=".datasources.CmdSnmpMemDataSource.ICmdSnmpMemDataSourceInfo"
        for=".datasources.CmdSnmpMemDataSource.CmdSnmpMemDataSource"
        factory=".datasources.CmdSnmpMemDataSource.CmdSnmpMemDataSourceInfo"
        />

</configure>

~
~
~
~
~
~
~
~
"configure.zcml" [Modified][readonly] 23 lines --95%--
```

Figure 29: configure.zcml with adapter for new datasource

At this stage, with a datasource file and a configure.zcml, reinstall the ZenPack again. I would also completely restart zenoss, for safety, with *zenoss stop; zenoss start*.

Subsequent “tweaks” to the ZenPack files you can probably get away with just restarting zenhub, zopectl and zenpython:

```
zenhub restart; zopectl restart; zenpython restart
```

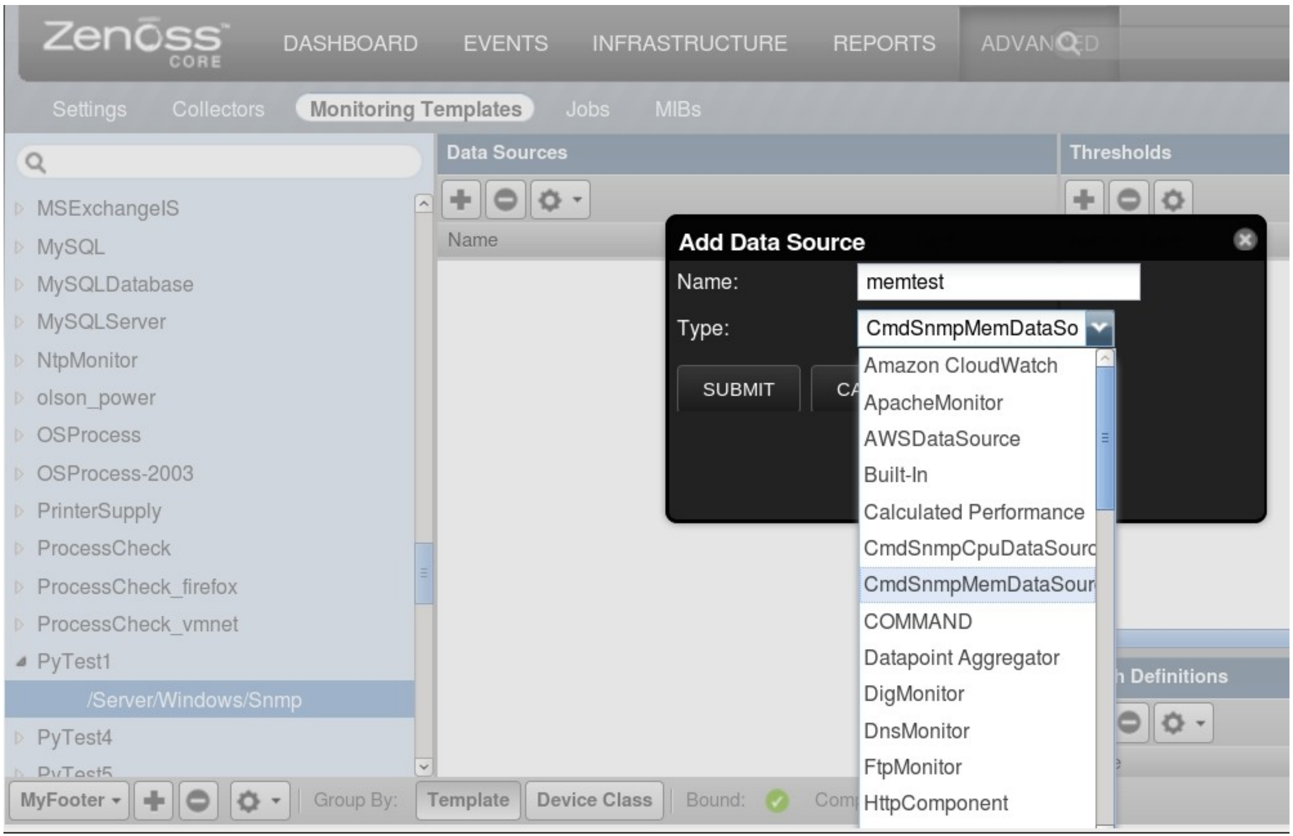
6.4 Testing the new datasource

Ultimately the new ZenPack is going to create some new device classes, starting with /Server/Windows/Snmp. For now create this device class by hand. Put a test device into this class; the example used here will be *win2003net.class.example.org*.

6.4.1 Defining a template to utilise a new datasource

The first test of the new datasource is to create a new template; select */Server/Windows/Snmp* as the template path.

Within that template, create a new datasource. You should find the new datasource in the list of datasource types.



Add Data Source

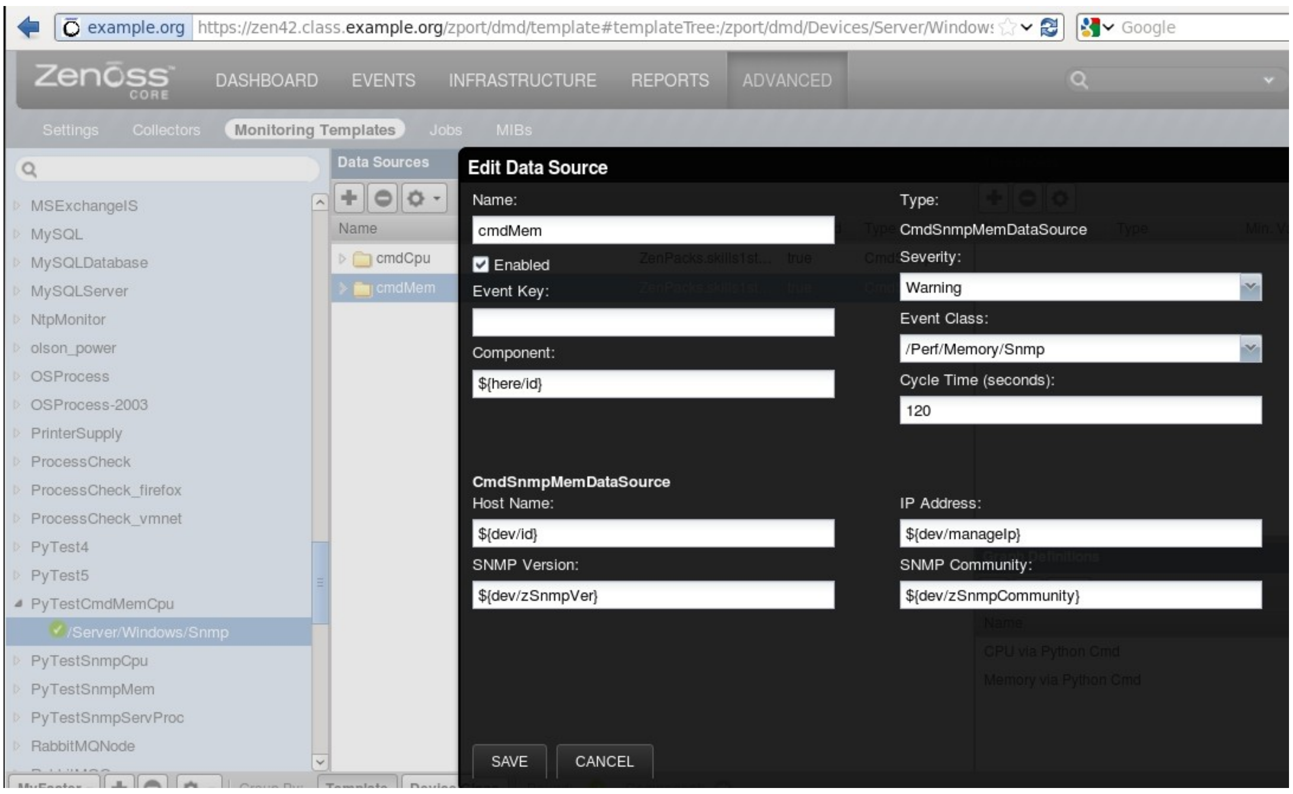
Name: memtest

Type: CmdSnmpMemDataSo

Figure 30: Creating a new datasource of type CmdSnmpMemDataSource

When you have created it, you should also find that the datapoints you defined have automatically been created for you.

Double-click the datasource and check that the correct fields appear in the datasource dialogue.



Edit Data Source

Name: cmdMem

Type: CmdSnmpMemDataSource

✓ Enabled

Severity: Warning

Event Key:

Event Class: /Perf/Memory/Snmp

Component: ${here/id}

Cycle Time (seconds): 120

CmdSnmpMemDataSource

Host Name: ${dev/id}

IP Address: ${dev/manageIp}

SNMP Version: ${dev/zSnmpVer}

SNMP Community: ${dev/zSnmpCommunity}

Figure 31: Datasource dialogue

At this stage, it is probably optimistic to start creating thresholds and graphs!

6.4.2 Testing the datasource configuration

As this is a device-level template, it needs binding. Bind it to the device class */Server/Windows/Snmp;* this class should already have you test device in it.

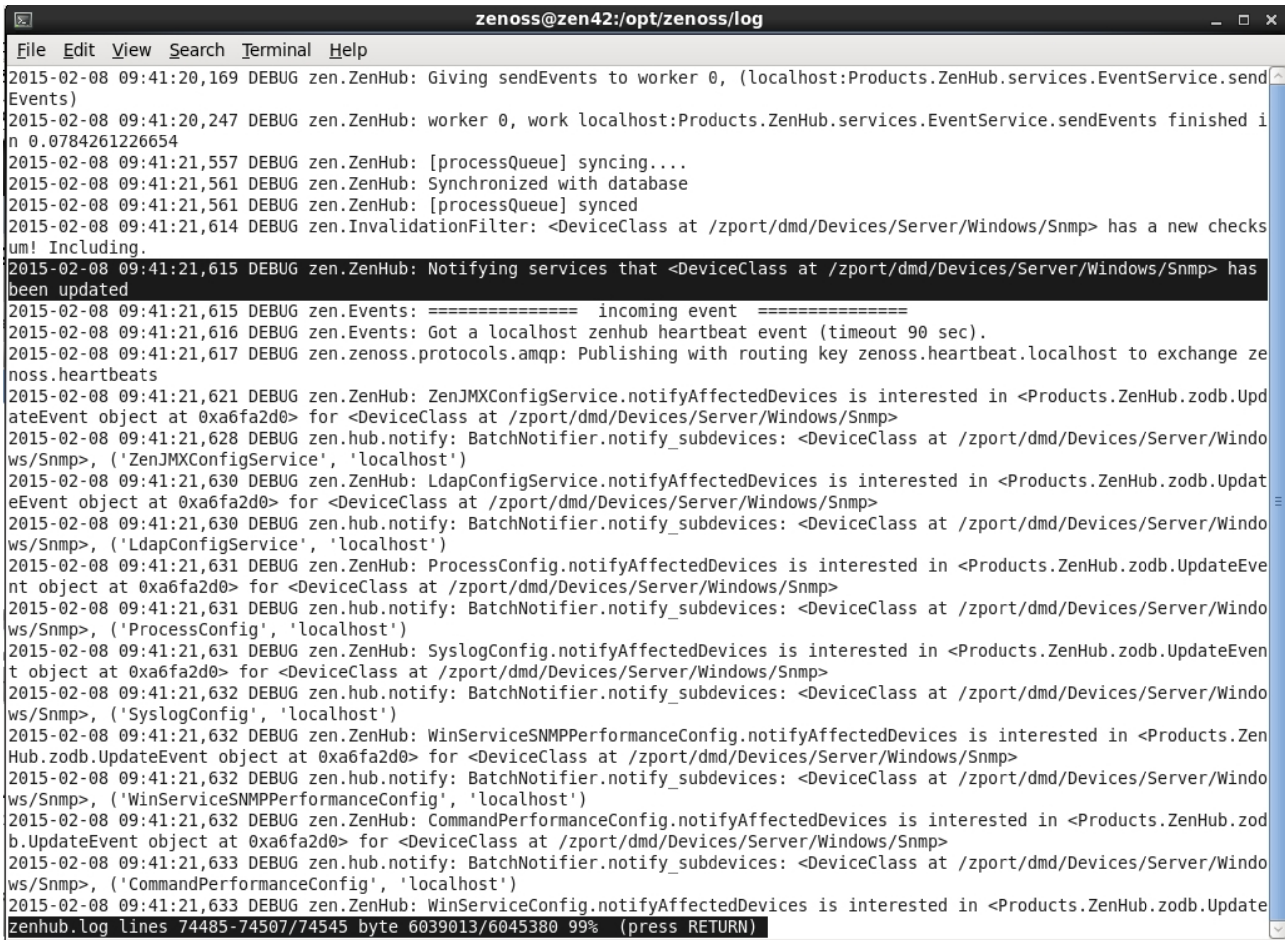
This is the point at which the *config_key* and *params* methods of the datasource plugin are exercised. For now, unbind the template again.

As the zenoss user, turn debugging on to zenhub. This is a toggle switch that you can apply to any running zenoss daemon.

```
        zenhub debug
```

Now use less to show $ZENHOME/log/zenhub.log.

Rebind the template to the device class and check updates in zenhub.log.



```
zenoss@zen42:/opt/zenoss/log
File Edit View Search Terminal Help
2015-02-08 09:41:20,169 DEBUG zen.ZenHub: Giving sendEvents to worker 0, (localhost:Products.ZenHub.services.EventService.send
Events)
2015-02-08 09:41:20,247 DEBUG zen.ZenHub: worker 0, work localhost:Products.ZenHub.services.EventService.sendEvents finished i
n 0.0784261226654
2015-02-08 09:41:21,557 DEBUG zen.ZenHub: [processQueue] syncing....
2015-02-08 09:41:21,561 DEBUG zen.ZenHub: Synchronized with database
2015-02-08 09:41:21,561 DEBUG zen.ZenHub: [processQueue] synced
2015-02-08 09:41:21,614 DEBUG zen.InvalidationFilter: <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp> has a new checks
um! Including.
2015-02-08 09:41:21,615 DEBUG zen.ZenHub: Notifying services that <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp> has
been updated
2015-02-08 09:41:21,615 DEBUG zen.Events: ===============  incoming event  ===============
2015-02-08 09:41:21,616 DEBUG zen.Events: Got a localhost zenhub heartbeat event (timeout 90 sec).
2015-02-08 09:41:21,617 DEBUG zen.zenoss.protocols.amqp: Publishing with routing key zenoss.heartbeat.localhost to exchange ze
noss.heartbeats
2015-02-08 09:41:21,621 DEBUG zen.ZenHub: ZenJMXConfigService.notifyAffectedDevices is interested in <Products.ZenHub.zodb.Upd
ateEvent object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,628 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('ZenJMXConfigService', 'localhost')
2015-02-08 09:41:21,630 DEBUG zen.ZenHub: LdapConfigService.notifyAffectedDevices is interested in <Products.ZenHub.zodb.Updat
eEvent object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,630 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('LdapConfigService', 'localhost')
2015-02-08 09:41:21,631 DEBUG zen.ZenHub: ProcessConfig.notifyAffectedDevices is interested in <Products.ZenHub.zodb.UpdateEve
nt object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,631 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('ProcessConfig', 'localhost')
2015-02-08 09:41:21,631 DEBUG zen.ZenHub: SyslogConfig.notifyAffectedDevices is interested in <Products.ZenHub.zodb.UpdateEven
t object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,632 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('SyslogConfig', 'localhost')
2015-02-08 09:41:21,632 DEBUG zen.ZenHub: WinServiceSNMPPerformanceConfig.notifyAffectedDevices is interested in <Products.Zen
Hub.zodb.UpdateEvent object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,632 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('WinServiceSNMPPerformanceConfig', 'localhost')
2015-02-08 09:41:21,632 DEBUG zen.ZenHub: CommandPerformanceConfig.notifyAffectedDevices is interested in <Products.ZenHub.zod
b.UpdateEvent object at 0xa6fa2d0> for <DeviceClass at /zport/dmd/Devices/Server/Windows/Snmp>
2015-02-08 09:41:21,633 DEBUG zen.hub.notify: BatchNotifier.notify_subdevices: <DeviceClass at /zport/dmd/Devices/Server/Windo
ws/Snmp>, ('CommandPerformanceConfig', 'localhost')
2015-02-08 09:41:21,633 DEBUG zen.ZenHub: WinServiceConfig.notifyAffectedDevices is interested in <Products.ZenHub.zodb.Update
zenhub.log lines 74485-74507/74545 byte 6039013/6045380 99%  (press RETURN)
```

Figure 32: zenhub.log when a configuration change is received

Look carefully for error messages.

6.4.3 Testing the target device

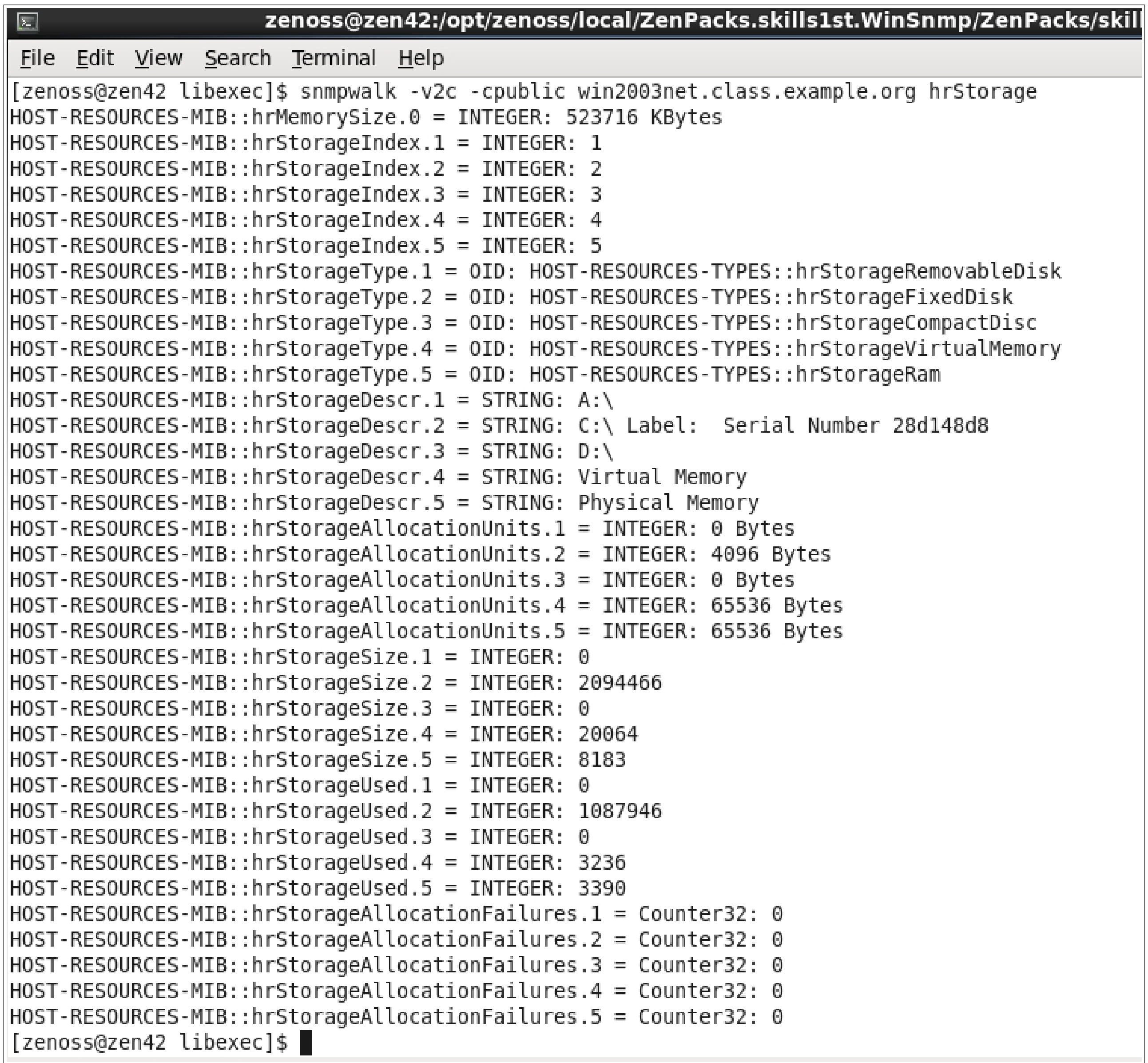
At this stage, it would be prudent to ensure that the target test device is up and responds to the SNMP OIDs that will be required.

I have tested with both the standard Microsoft SNMP agent and with the windows version of net-snmp; both seem to respond well to standard Mib-2, HOST-RESOURCES requests and Lan Manager OIDs.

For testing, I am using SNMP v2 and a community name of public. Use snmpwalk to test. The winmem.py script is requesting data from the hrStorage table (.1.3.6.1.2.1.25.2.3.1) so try:

```
snmpwalk -v2c -cpublic win2003net.class.example.org .1.3.6.1.2.1.25.2.3.1
snmpwalk -v2c -cpublic win2003net.class.example.org hrStorage
```

You should get responses.


zenoss@zen42:/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skil
File Edit View Search Terminal Help

```
[zenoss@zen42 libexec]$ snmpwalk -v2c -cpublic win2003net.class.example.org hrStorage
HOST-RESOURCES-MIB::hrMemorySize.0 = INTEGER: 523716 KBytes
HOST-RESOURCES-MIB::hrStorageIndex.1 = INTEGER: 1
HOST-RESOURCES-MIB::hrStorageIndex.2 = INTEGER: 2
HOST-RESOURCES-MIB::hrStorageIndex.3 = INTEGER: 3
HOST-RESOURCES-MIB::hrStorageIndex.4 = INTEGER: 4
HOST-RESOURCES-MIB::hrStorageIndex.5 = INTEGER: 5
HOST-RESOURCES-MIB::hrStorageType.1 = OID: HOST-RESOURCES-TYPES::hrStorageRemovableDisk
HOST-RESOURCES-MIB::hrStorageType.2 = OID: HOST-RESOURCES-TYPES::hrStorageFixedDisk
HOST-RESOURCES-MIB::hrStorageType.3 = OID: HOST-RESOURCES-TYPES::hrStorageCompactDisc
HOST-RESOURCES-MIB::hrStorageType.4 = OID: HOST-RESOURCES-TYPES::hrStorageVirtualMemory
HOST-RESOURCES-MIB::hrStorageType.5 = OID: HOST-RESOURCES-TYPES::hrStorageRam
HOST-RESOURCES-MIB::hrStorageDescr.1 = STRING: A:\
HOST-RESOURCES-MIB::hrStorageDescr.2 = STRING: C:\ Label:  Serial Number 28d148d8
HOST-RESOURCES-MIB::hrStorageDescr.3 = STRING: D:\
HOST-RESOURCES-MIB::hrStorageDescr.4 = STRING: Virtual Memory
HOST-RESOURCES-MIB::hrStorageDescr.5 = STRING: Physical Memory
HOST-RESOURCES-MIB::hrStorageAllocationUnits.1 = INTEGER: 0 Bytes
HOST-RESOURCES-MIB::hrStorageAllocationUnits.2 = INTEGER: 4096 Bytes
HOST-RESOURCES-MIB::hrStorageAllocationUnits.3 = INTEGER: 0 Bytes
HOST-RESOURCES-MIB::hrStorageAllocationUnits.4 = INTEGER: 65536 Bytes
HOST-RESOURCES-MIB::hrStorageAllocationUnits.5 = INTEGER: 65536 Bytes
HOST-RESOURCES-MIB::hrStorageSize.1 = INTEGER: 0
HOST-RESOURCES-MIB::hrStorageSize.2 = INTEGER: 2094466
HOST-RESOURCES-MIB::hrStorageSize.3 = INTEGER: 0
HOST-RESOURCES-MIB::hrStorageSize.4 = INTEGER: 20064
HOST-RESOURCES-MIB::hrStorageSize.5 = INTEGER: 8183
HOST-RESOURCES-MIB::hrStorageUsed.1 = INTEGER: 0
HOST-RESOURCES-MIB::hrStorageUsed.2 = INTEGER: 1087946
HOST-RESOURCES-MIB::hrStorageUsed.3 = INTEGER: 0
HOST-RESOURCES-MIB::hrStorageUsed.4 = INTEGER: 3236
HOST-RESOURCES-MIB::hrStorageUsed.5 = INTEGER: 3390
HOST-RESOURCES-MIB::hrStorageAllocationFailures.1 = Counter32: 0
HOST-RESOURCES-MIB::hrStorageAllocationFailures.2 = Counter32: 0
HOST-RESOURCES-MIB::hrStorageAllocationFailures.3 = Counter32: 0
HOST-RESOURCES-MIB::hrStorageAllocationFailures.4 = Counter32: 0
HOST-RESOURCES-MIB::hrStorageAllocationFailures.5 = Counter32: 0
[zenoss@zen42 libexec]$
```

Figure 33: snmpwalk test for hrStorage OIDs

6.4.4 Testing with zenpython

The best way to test that data is gathered by the new datasource is to run zenpython from the command line with full debugging. The datasource code has been well scattered with *log.debug* statements exactly for this purpose.

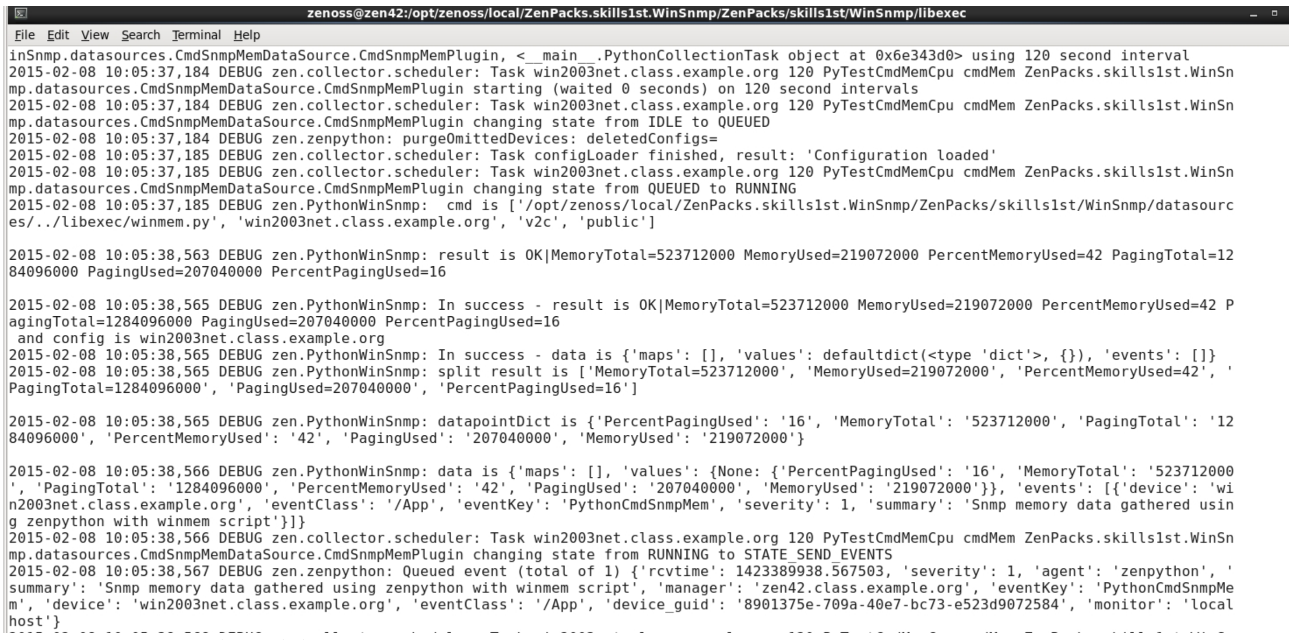
```
zenpython run -v 10 -d win2003net.class.example.org
```

Try to ensure that there are no other Python templates active against your test device to reduce the output. If there is too much to scan, redirect the out put to a file:

```
zenpython run -v 10 -d win2003net.class.example.org > /tmp/fred 2>&1
```

If you see a message saying it cannot find the configuration for your test device "Is that the right name", this means that there is a problem with the configuration section. Check zenhub.log and inspect the config_keys and params methods of the plugin.

The output should show a number of phases:


zenoss@zen42:/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/WinSnmp/libexec

```
inSnmp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin, <__main__.PythonCollectionTask object at 0x6e343d0> using 120 second interval
2015-02-08 10:05:37,184 DEBUG zen.collector.scheduler: Task win2003net.class.example.org 120 PyTestCmdMemCpu cmdMem ZenPacks.skills1st.WinSn
mp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin starting (waited 0 seconds) on 120 second intervals
2015-02-08 10:05:37,184 DEBUG zen.collector.scheduler: Task win2003net.class.example.org 120 PyTestCmdMemCpu cmdMem ZenPacks.skills1st.WinSn
mp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin changing state from IDLE to QUEUED
2015-02-08 10:05:37,184 DEBUG zen.zenpython: purgeOmittedDevices: deletedConfigs=
2015-02-08 10:05:37,185 DEBUG zen.collector.scheduler: Task configLoader finished, result: 'Configuration loaded'
2015-02-08 10:05:37,185 DEBUG zen.collector.scheduler: Task win2003net.class.example.org 120 PyTestCmdMemCpu cmdMem ZenPacks.skills1st.WinSn
mp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin changing state from QUEUED to RUNNING
2015-02-08 10:05:37,185 DEBUG zen.PythonWinSnmp:  cmd is ['/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/WinSnmp/datasourc
es/../libexec/winmem.py', 'win2003net.class.example.org', 'v2c', 'public']

2015-02-08 10:05:38,563 DEBUG zen.PythonWinSnmp: result is OK|MemoryTotal=523712000 MemoryUsed=219072000 PercentMemoryUsed=42 PagingTotal=12
84096000 PagingUsed=207040000 PercentPagingUsed=16

2015-02-08 10:05:38,565 DEBUG zen.PythonWinSnmp: In success - result is OK|MemoryTotal=523712000 MemoryUsed=219072000 PercentMemoryUsed=42 P
agingTotal=1284096000 PagingUsed=207040000 PercentPagingUsed=16
 and config is win2003net.class.example.org
2015-02-08 10:05:38,565 DEBUG zen.PythonWinSnmp: In success - data is {'maps': [], 'values': defaultdict(<type 'dict'>, {}), 'events': []}
2015-02-08 10:05:38,565 DEBUG zen.PythonWinSnmp: split result is ['MemoryTotal=523712000', 'MemoryUsed=219072000', 'PercentMemoryUsed=42', '
PagingTotal=1284096000', 'PagingUsed=207040000', 'PercentPagingUsed=16']

2015-02-08 10:05:38,565 DEBUG zen.PythonWinSnmp: datapointDict is {'PercentPagingUsed': '16', 'MemoryTotal': '523712000', 'PagingTotal': '12
84096000', 'PercentMemoryUsed': '42', 'PagingUsed': '207040000', 'MemoryUsed': '219072000'}

2015-02-08 10:05:38,566 DEBUG zen.PythonWinSnmp: data is {'maps': [], 'values': {None: {'PercentPagingUsed': '16', 'MemoryTotal': '523712000
', 'PagingTotal': '1284096000', 'PercentMemoryUsed': '42', 'PagingUsed': '207040000', 'MemoryUsed': '219072000'}}, 'events': [{'device': 'wi
n2003net.class.example.org', 'eventClass': '/App', 'eventKey': 'PythonCmdSnmpMem', 'severity': 1, 'summary': 'Snmp memory data gathered usin
g zenpython with winmem script'}]}
2015-02-08 10:05:38,566 DEBUG zen.collector.scheduler: Task win2003net.class.example.org 120 PyTestCmdMemCpu cmdMem ZenPacks.skills1st.WinSn
mp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin changing state from RUNNING to STATE_SEND_EVENTS
2015-02-08 10:05:38,567 DEBUG zen.zenpython: Queued event (total of 1) {'rcvtime': 1423389938.567503, 'severity': 1, 'agent': 'zenpython', '
summary': 'Snmp memory data gathered using zenpython with winmem script', 'manager': 'zen42.class.example.org', 'eventKey': 'PythonCmdSnmpMe
m', 'device': 'win2003net.class.example.org', 'eventClass': '/App', 'device_guid': '8901375e-709a-40e7-bc73-e523d9072584', 'monitor': 'local
host'}
```

Figure 34: zenpython debugging output

1. You should see the cycletime for the datasource - 120 seconds in this case
2. You should see a task with the template name (PyTestCmdMemCpu), the datasource id (cmdMem) and the object path to the plugin (ZenPacks.skills1st.WinSnmp.datasources.CmdSnmpMemDataSource.CmdSnmpMemPlugin). The task should change from IDLE to QUEUED to RUNNING.
3. You should see the cmd variable - remember this is the list that will be passed to subprocess.Popen().

   ```
   cmd is
   ['/opt/zenoss/local/ZenPacks.skills1st.WinSnmp/ZenPacks/skills1st/WinSnmp/
   datasources/../libexec/winmem.py', 'win2003net.class.example.org', 'v2c',
   'public']
   ```
4. Check that the command parameters for device, snmp version and snmp community have substituted correctly.
5. The code has a log.debug that reports the result in the onResult method. This is immediately after data collection has taken place.

   ```
   result is OK|MemoryTotal=523712000 MemoryUsed=219072000
   PercentMemoryUsed=42 PagingTotal=1284096000 PagingUsed=207040000
   PercentPagingUsed=16
   ```
6. You should see the task changing state from RUNNING to STATE_SEND_EVENTS and see the event that you constructed in the onSuccess method.

7. You should see the task changing state from STATE_SEND_EVENTS to STORE_PERF_DATA. This is where the data actually gets saved into rrd files. You can see the file names and the values.
8. You should then see the task changing state from STORE_PERF_DATA to IDLE.

6.4.5 Adding graphs and threshold to the template

The ultimate test of the datasource is that you can create graphs with the datapoints and also thresholds. If the zenpython test runs successfully, then it is worth creating graph and threshold definitions to use the data.

6.5 Datasource to collect cpu utilisation with a command

Ryan's ZenPacks.Nova.Windows.SNMPPerfMonitor gathers memory and cpu utilisation, both using a script to drive SNMP. The ....

6.6 Building a datasource to gather a single SNMP scalar value

6.7 Building a datasource to gather SNMP table values

6.8 Building a Python modeler plugin

Remember....

If plugin doesn't show in the list to be run, suspect syntax errors in the plugin itself

plugin classname must be same as filename

Change modeler then bounce zenhub and zopectl

Clear browser cache

portal_type = meta_type = 'WinServiceSNMPPython' - must match component type

6.9 The rest of the ZenPack

6.9.1 Device and component objects

6.9.2 The __init__.py file

- Menus
-

7 Conclusions

References

1. “Creating Zenoss ZenPacks for Zenoss 3” by Jane Curry, January 2011. http://www.skills-1st.co.uk/papers/jane/zenpacks/zenpacks3.pdf
2. Zenoss Core 4 Administration Guide - http://wiki.zenoss.org/Zenoss_Core_4.2.x
3. Zenoss Developer's Guide 3 -
4. ZenPack development Guide - http://zenosslabs.readthedocs.org/en/latest/zenpack_development/index.html
   a) In particular, check the “Monitoring an HTTP API” section - http://zenosslabs.readthedocs.org/en/latest/zenpack_development/monitoring_http_api/index.html
5. ZenPacks.zenoss.PythonCollector
   - Documentation http://wiki.zenoss.org/ZenPack:PythonCollector
   - Github https://github.com/zenoss/ZenPacks.zenoss.PythonCollector.git
6. ZenPacks which use the Python Collector ZenPack
   a) ZenPacks.zenoss.XenServer
      - Documentation http://wiki.zenoss.org/ZenPack:XenServer
      - Github https://github.com/zenoss/ZenPacks.zenoss.XenServer.git

   b) ZenPacks.TwoNMS.Rancid by mwallraf (Python modeler)
      - Github https://github.com/mwallraf/ZenPacks.TwoNMS.Rancid

   c) ZenPacks.zenoss.MySqlMonitor (Python modeler)
      - Documentation http://wiki.zenoss.org/ZenPack:MySQL_Database_Monitor_%28Core%29
      - Github https://github.com/zenoss/ZenPacks.zenoss.MySqlMonitor.git

   d) ZenPacks.zenoss.PostgreSQl (Python modeler)
      - Documentation http://wiki.zenoss.org/ZenPack:PostgreSQL
      - Github https://github.com/zenoss/ZenPacks.zenoss.PostgreSQL.git

   e) ZenPacks.zenoss.AWS
      - Documentation http://wiki.zenoss.org/ZenPack:Amazon_Web_Services
      - Github https://github.com/zenoss/ZenPacks.zenoss.AWS.git

   f) ZenPacks.zenoss.Hadoop (dsplugins.py example)
      - Documentation http://wiki.zenoss.org/ZenPack:Hadoop
      - Github https://github.com/zenoss/ZenPacks.zenoss.Hadoop.git

   g) ZenPacks.zenoss.HBase (dsplugins directory example)

- Documentation http://wiki.zenoss.org/ZenPack:HBase
- Github https://github.com/zenoss/ZenPacks.zenoss.HBase.git

h) ZenPacks.zenoss.OpenStackInfrastructure

- Documentation http://wiki.zenoss.org/ZenPack:OpenStack_%28Provider_View%29
- Github https://github.com/zenoss/ZenPacks.zenoss.OpenStackInfrastructure

i) ZenPacks.training.NetBotz referenced in the Zenoss Labs ZenPack development documentation:

- https://github.com/cluther/ZenPacks.training.NetBotz

j) ZenPacks.training.WeatherUnderground referenced in the Zenoss Labs ZenPack development documentation:

- https://github.com/zenoss/ZenPacks.training.WeatherUnderground

k)

7. Ryan Matte's ZenPacks.Nova.Windows.SNMPPerfMonitor - https://github.com/zenoss/ZenPacks.Nova.Windows.SNMPPerfMonitor . Note that this is version 1.6 and does not have process and paging monitoring. The latest 1.7 version I can only find in egg format referenced from the wiki page at http://wiki.zenoss.org/ZenPack:Windows_SNMP_Performance_Monitor_%28Advanced%29 and the download link for the 1.7 egg is http://dmon.org/downloads/zenoss/zenpacks/zenoss4/ZenPacks.Nova.Windows.SNMPPerfMonitor-1.7-py2.7.egg
8. Ryan Matte's ZenPacks.Nova.WinServiceSNMP - the wiki page is at http://wiki.zenoss.org/ZenPack:Windows_SNMP_Service_Monitor . The egg download is at http://dmon.org/downloads/zenoss/zenpacks/zenoss4/ZenPacks.Nova.WinServiceSNMP-1.1-py2.7.egg . I have built a development-mode tarball of the 1.7 code that is available at ?????
9. ZenPacks.zenoss.CalculatedPerformance ZenPack - http://wiki.zenoss.org/ZenPack:Calculated_Performance
10. Nagios format - see section 6.3 of the Zenoss Administration Guide. For detailed information on the format, follow the reference to https://nagios-plugins.org/doc/guidelines.html
11. Lan Manager MIB (LanMgr-Mib-II) can be downloaded from http://www.mibsearch.com/vendors/LAN%20Manager/download/LanMgr-Mib-II-MIB
12.
13.

Acknowledgements

Several people have contributed either actively or passively to this paper:

- Ryan Matte for the Windows SNMP ZenPacks
- Chet Luther for countless hints and tips
- agmenut on the Zenoss IRC at UC Regents (agmenut@ucdevis.edu) for ZenPacks.ucdavis.PureStorage - https://bitbucket.org/ucdavis/zenpacks.ucdavis.purestorage/src/ec2a69f9061eadfd81164ee62b0eb1353d953d54/ZenPacks/ucdavis/PureStorage/dsplugins.py?at=master
- Jason Stanley (jstanley or jls on Zenoss IRC), jstanley734@gmail.com , for various samples and comments
- Doug Syer on github - https://github.com/dougsyer/Zenoss-Examples-and-Tools
-