THE EXPERT'S VOICE® IN PYTHON

SECOND EDITION

Pro Python System Administration

Rytis Sileika

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

The role of the system administrator has grown dramatically over the years. The number of systems supported by a single engineer has also increased. As such, it is impractical to handcraft each installation, and there is a need to automate as many tasks as possible. The structure of systems varies from organization to organization, therefore system administrators must be able to create their own management tools. Historically, the most popular programming languages for these tasks were UNIX shell and Perl. They served their purposes well, and I doubt they will ever cease to exist. However, the complexity of current systems requires new tools, and the Python programming language is one of them.

Python is an object-oriented programming language suitable for developing large-scale applications. Its syntax and structure make it very easy to read—so much so that the language is sometimes referred to as "executable pseudocode." The Python interpreter allows for interactive execution, so in some situations an administrator can use it instead of a standard UNIX shell. Although Python is primarily an object-oriented language, it is easily adopted for procedural and functional styles of programming. Given all that, Python makes a perfect fit as a new language for implementing system administration applications. There are a large number of Linux system utilities already written in Python, such as the Yum package manager and Anaconda, the Linux installation program.

The Prerequisites for Using this Book

This book is about using the Python programming language to solve specific system administration tasks. We look at the four distinctive system administration areas: network management, web server and web application management, database system management, and system monitoring. Although I explain in detail most of the technologies used in this book, bear in mind that the main goal here is to display the practical application of the Python libraries so as to solve rather specific issues. Therefore, I assume that you are a seasoned system administrator. You should be able to find additional information yourself; this book gives you a rough guide for how to reach your goal, but you must be able to work out how to adapt it to your specific system and environment.

As we discuss the examples, you will be asked to install additional packages and libraries. In most cases, I provide the commands and instructions to perform these tasks on a Fedora system, but you should be ready to adopt the instructions to the Linux distribution that you are going to use. Most of the examples also work without many modification on a recent OS X release (10.10.X).

I also assume that you have a background in the Python programming language. I introduce the specific libraries that are used in system administration tasks, as well as some lesser known or less often discussed language functionality, such as the generator functions or the class internal methods, but the basic language syntax is not explained here. If you want to refresh your Python skills, I recommend the following books: *Pro Python* by Marty Alchin and J. Burton Browning (Apress, 2012; but watch for a new edition due to be released in early 2015); *Python Programming for the Absolute Beginner* by Mike Dawson (Course Technology PTR, 2010); and *Core Python Applications Programming* by Wesley Chun (Prentice Hall, 2012)

All examples presented in this book assume the Python version 2.7. This is mostly dictated by the libraries that are used in the examples. Some libraries have been ported to Python 3; however, some have not. So if you need to run Python 3, make sure you check that the required libraries have Python 3 support.

The Structure of this Book

This book contains 14 chapters, and each chapter solves a distinctive problem. Some examples span multiple chapters, but even then, each chapter deals with a specific aspect of the particular problem.

In addition to the chapters, several other organizational layers characterize this book. First, I grouped the chapters by the problem type. Chapters 1 to 4 deal with network management issues; Chapters 5 to 7 talk about the Apache web server and web application management; Chapters 8 to 11 are dedicated to monitoring and statistical calculations; and Chapters 12 and 13 focus on database management issues.

Second, I maintain a common pattern in all chapters. I start with the problem statement and then move on to gather requirements and proceed through the design phase before moving into the implementation section.

Third, each chapter focuses on one or more technologies and the Python libraries that provide the language interface for the particular technology. Examples of such technologies could be the SOAP protocol, application plug-in architecture, or cloud computing concepts.

More specifically, here's a breakdown of the chapters:

Chapter 1: Reading and Collecting Performance Data Using SNMP

Most network-attached devices expose the internal counters via the Simple Network Management Protocol (SNMP). This chapter explains basic SNMP principles and the data structure. We then look at the Python libraries that provide the interface to SNMP–enabled devices. We also investigate the round robin database, which is the de facto standard for storing the statistical data. Finally, we look at the Jinja2 template framework, which allows us to generate simple web pages.

Chapter 2: Managing Devices Using the SOAP API

Complicated tasks, such as managing the device configuration, cannot be easily done by using SNMP because the protocol is too simplistic. Therefore, advanced devices, such as the Citrix Netscaler load balancers, provide the SOAP API interface to the device management system. In this chapter, we investigate the SOAP API structure and the libraries that enable the SOAP–based communication from the Python programming language. We also look at the basic logging functionality using the built-in libraries. This second edition of the book includes examples of how to use the new REST API to manage the load balancer devices.

Chapter 3: Creating a Web Application for IP Address Accountancy

In this chapter, we build a web application that maintains the list of the assigned IP addresses and the address ranges. We learn how to create web applications using the Django framework. I show you the way the Django application should be structured, tell how to create and configure the application settings, and explain the URL structure. We also investigate how to deploy the Django application using the Apache web server.

Chapter 4: Integrating the IP Address Application with DHCP

This chapter expands on the previous chapter, and we implement the DHCP address range support. We also look at some advanced Django programming techniques, such as customizing the response MIME type and serving AJAX calls. This second edition adds new functionality to manage dynamic DHCP leases using OMAPI protocol.

Chapter 5: Maintaining a List of Virtual Hosts in an Apache Configuration File

This is another Django application that we develop in this book, but this time our focus is on the Django administration interface. While building the Apache configuration management application, you learn how to customize the default Django administration interface with your own views and functions.

Chapter 6: Gathering and Presenting Statistical Data from Apache Log Files

In this chapter, the goal is to build an application that parses and analyses the Apache web server log files. Instead of taking the straightforward but inflexible approach of building a monolithic application, we look at the design principles involved in building plug-in applications. You learn how to use the object and class type discovery functions and how to perform a dynamic module loading. This second edition of the book shows you how to perform data visualization based on the gathered data.

Chapter 7: Performing Complex Searches and Reporting on Application Log Files

This chapter also deals with the log file parsing, but this time I show you how to parse complex, multi-line log file entries. We investigate the functionality of the open-source log file parser tool called Exctractor, which you can download from http://exctractor.sourceforge.net/.

Chapter 8: A Web Site Availability Check Script for Nagios

Nagios is one of the most popular open-source monitoring systems, because its modular structure allows users to implement their own check scripts and thus customize the tool to meet their needs. In this chapter, we create two scripts that check the functionality of a website. We investigate how to use the Beautiful Soup HTML parsing library to extract the information from the HTML web pages.

Chapter 9: Management and Monitoring Subsystem

This chapter starts a three-chapter series in which we build a complete monitoring system. The goal of this chapter is not to replace mature monitoring systems such as Nagios or Zenoss but to show the basic principles of the distributed application programming. We look at database design principles such as data normalization. We also investigate how to implement the communication mechanisms between network services using the RPC calls.

Chapter 10: Remote Monitoring Agents

This is the second chapter in the monitoring series, where we implement the remote monitoring agent components. In this chapter, I also describe how to decouple the application from its configuration using the ConfigParser module.

Chapter 11: Statistics Gathering and Reporting

This is the last part of the monitoring series, where I show you how to perform basic statistical analysis on the collected performance data. We use scientific libraries: NumPy to perform the calculations and matplotlib to create the graphs. You learn how to find which performance readings fall into the comfort zone and how to calculate the boundaries of that zone. We also do the basic trend detection, which provides a good insight for the capacity planning.

Chapter 12: Distributed Message Processing System

This is a new chapter for the second edition of the book. In this chapter I show you how to convert the distributed management system to use Celery, a remote task execution framework.

Chapter 13: Automatic MySQL Database Performance Tuning

In this chapter, I show you how to obtain the MySQL database configuration variables and the internal status indicators. We build an application that makes a suggestion on how to improve the database engine performance based on the obtained data.

Chapter 14: Amazon EC2/S3 as a Data Warehouse Solution

This chapter shows you how to utilize the Amazon Elastic Compute Cloud (EC2) and offload the infrequent computation tasks to it. We build an application that automatically creates a database server where you can transfer data for further analysis. You can use this example as a basis to build an on-demand data warehouse solution.

The Example Source Code

The source code of all the examples in this book, along with any applicable sample data, can be downloaded from the Apress website by following instructions at www.apress.com/source-code/. The source code stored at this location contains the same code that is described in the book.

Most of the prototypes described in this book are also available as open-source projects. You can find these projects at the author's website, http://www.sysadminpy.com/.

CHAPTER 1

Reading and Collecting Performance Data Using SNMP

Most devices that are connected to a network report their status using SNMP (the Simple Network Management Protocol). This protocol was designed primarily for managing and monitoring network-attached hardware devices, but some applications also expose their statistical data using this protocol. In this chapter we will look at how to access this information from your Python applications. We are going to store the obtained data in an RRD (round robin database), using RRDTool—a widely known and popular application and library, which is used to store and plot the performance data. Finally we'll investigate the Jinja2 template system, which we'll use to generate simple web pages for our application.

Application Requirements and Design

The topic of system monitoring is very broad and usually encompasses many different areas. A complete monitoring system is rather complex and often is made up of multiple components working together. We are not going to develop a complete, self-sufficient system here, but we'll look into two important areas of a typical monitoring system: information gathering and representation. In this chapter we'll implement a system that queries devices using an SNMP protocol and then stores the data using the RRDTool library, which is also used to generate the graphs for visual data representation. All this is tied together into simple web pages using the Jinja2 templating library. We'll look at each of these components in more detail as we go along through the chapter.

Specifying the Requirements

Before we start designing our application we need to come up with some requirements for our system. First of all we need to understand the functionality we expect our system to provide. This will help us to create an effective (and we hope easy-to-implement) system design. In this chapter we are going to create a system that monitors network-attached devices, such as network switches and routers, using the SNMP protocol. So the first requirement is that the system be able to query any device using SNMP.

The information gathered from the devices needs to be stored for future reference and analysis. Let's make some assumptions about the use of this information. First, we don't need to store it indefinitely. (I'll talk more about permanent information storage in Chapters 9–11.) This means that the information is stored only for a predefined period of time, and once it becomes obsolete it will be erased. This presents our second requirement: the information needs to be deleted after it has "expired."

Second, the information needs to be stored so that graphs can be produced. We are not going to use it for anything else, and therefore the data store should be optimized for the data representation tasks.

Finally, we need to generate the graphs and represent this information on easily accessible web pages. The information needs to be structured by the device names only. For example, if we are monitoring several devices for CPU and network interface utilization, this information needs to be presented on a single page. We don't need to present this information on multiple time scales; by default the graphs should show the performance indicators for the last 24 hours.

High-Level Design Specification

Now that we have some ideas about the functionality of our system, let's create a simple design, which we'll use as a guide in the development phase. The basic approach is that each of the requirements we specified earlier should be covered by one or more design decisions.

The first requirement is that we need to monitor the network-attached devices, and we need to do so using SNMP. This means that we have to use appropriate Python library that deals with the SNMP objects. The SNMP module is not included in the default Python installation, so we'll have to use one of the external modules. I recommend using the PySNMP library (available at `http://pysnmp.sourceforge.net/`), which is readily available on most of the popular Linux distributions.

The perfect candidate for the data store engine is RRDTool (available at `http://oss.oetiker.ch/rrdtool/`). The round robin database means that the database is structured in such a way that each "table" has a limited length, and once the limit is reached, the oldest entries are dropped. In fact they are not dropped; the new ones are simply written into their position.

The RRDTool library provides two distinct functionalities: the database service and the graph-generation toolkit. There is no native support for RRD databases in Python, but there is an external library available that provides an interface to the RRDTool library.

Finally, to generate the web page we will use the Jinja2 templating library (available at `http://jinja.pocoo.org`, or on GitHub: `https://github.com/mitsuhiko/jinja2`), which lets us create sophisticated templates and decouple the design and development tasks.

We are going to use a simple Windows INI-style configuration file to store the information about the devices we will be monitoring. This information will include details such as the device address, SNMP object reference, and access control details.

The application will be split into two parts: the first part is the information-gathering tool that queries all configured devices and stores the data in the RRDTool database, and the second part is the report generator, which generates the web site structure along with all required images. Both components will be instantiated from the standard UNIX scheduler application, cron. These two scripts will be named `snmp-manager.py` and `snmp-pages.py`, respectively.

Introduction to SNMP

SNMP (Simple Network Management Protocol) is a UDP-based protocol used mostly for managing network-attached devices, such as routers, switches, computers, printers, video cameras, and so on. Some applications also allow access to internal counters via the SNMP protocol.

SNMP not only allows you to read performance statistics from the devices, it can also send control messages to instruct a device to perform some action—for example, you can restart a router remotely by using SNMP commands.

There are three main components in a system managed by SIMPLE NETWORK MANAGEMENT PROTOCOL (SNMP):

- The management system which is responsible for managing all devices
- The managed devices, which are all devices managed by the management system
- The SNMP agent, which is an application that runs on each of the managed devices and interacts with the management system

This relationship is illustrated in Figure 1-1.

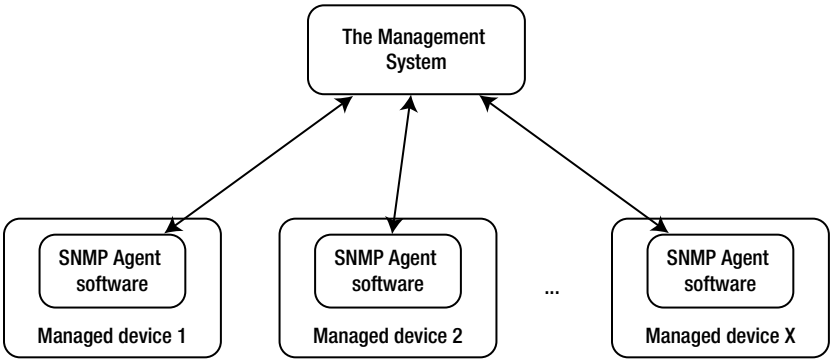

Figure 1-1. *The SNMP network components*

This approach is rather generic. The protocol defines seven basic commands, of which the most interesting to us are `get`, `get bulk`, and `response`. As you may have guessed, the former two are the commands that the management system issues to the agent, and the latter is a response from the agent software.

How does the management system know what to look for? The protocol does not define a way of exchanging this information, and therefore the management system has no way to interrogate the agents to obtain the list of available variables.

The issue is resolved by using a Management Information Base (or MIB). Each device usually has an associated MIB, which describes the structure of the management data on that system. Such a MIB would list in hierarchical order all object identifiers (OIDs) that are available on the managed device. The OID effectively represents a node in the object tree. It contains numerical identifiers of all nodes leading to the current OID starting from the node at the top of the tree. The node IDs are assigned and regulated by the IANA (Internet Assigned Numbers Authority). An organization can apply for an OID node, and when it is assigned it is responsible for managing the OID structure below the allocated node.

Figure 1-2 illustrates a portion of the OID tree.

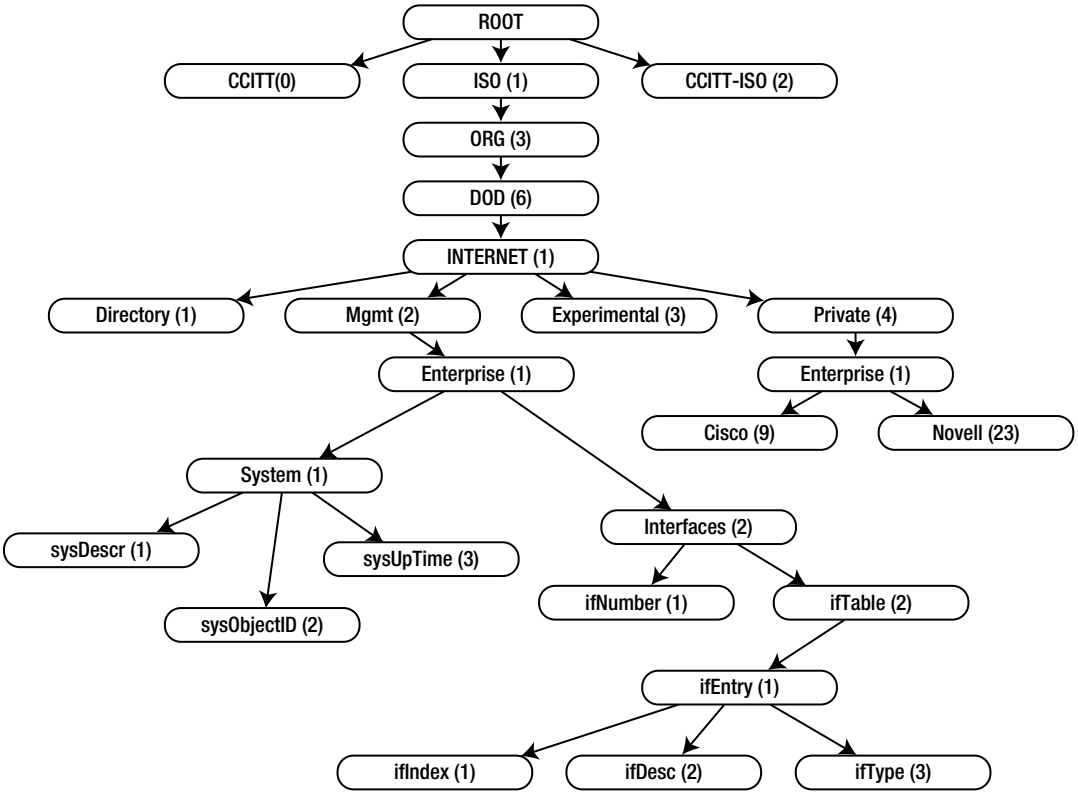

Figure 1-2. *The SNMP OID tree*

Let's look at some example OIDs. The OID tree node that is assigned to the Cisco organization has a value of 1.3.6.1.4.1.9, which means that all proprietary OIDs that are associated with the Cisco manufactured devices will start with these numbers. Similarly, the Novell devices will have their OIDs starting with 1.3.6.1.4.1.23.

I deliberately emphasized proprietary OIDs because some properties are expected to be present (if and where available) on all devices. These are under the 1.3.6.1.2.1.1 (System SNMP Variables) node, which is defined by RFC1213. For more details on the OID tree and its elements, visit `http://www.alvestrand.no/objectid/top.html`. This website allows you to browse the OID tree and it contains quite a large collection of the various OIDs.

The System SNMP Variables Node

In most cases the basic information about a device will be available under the System SNMP Variables OID node subtree. Therefore let's have a close look at what you can find there.

This OID node contains several additional OID nodes. Table 1-1 provides a description for most of the subnodes.

Table 1-1. *System SNMP OIDs*

OID String	OID Name	Description
1.3.6.1.2.1.1.1	`sysDescr`	A string containing a short description of the system or device. Usually contains the hardware type and operating system details.
1.3.6.1.2.1.1.2	`sysObjectID`	A string containing the vendor-specific device OID node. For example, if the organization has been assigned an OID node 1.3.6.1.4.1.8888 and this specific device has been assigned a .1.1 OID space under the organization's space, this field would contain a value of 1.3.6.1.4.1.8888.1.1.
1.3.6.1.2.1.1.3	`sysUpTime`	A number representing the time in hundreds of a second from the time when the system was initialized.
1.3.6.1.2.1.1.4	`sysContact`	An arbitrary string containing information about the contact person who is responsible for this system.
1.3.6.1.2.1.1.5	`sysName`	A name that has been assigned to the system. Usually this field contains a fully qualified domain name.
1.3.6.1.2.1.1.6	`sysLocation`	A string describing the physical location of the system.
1.3.6.1.2.1.1.7	`sysServices`	A number that indicates which services are offered by this system. The number is a bitmap representation of all OSI protocols, with the lowest bit representing the first OSI layer. For example, a switching device (operating on layer 2) would have this number set to $2^2 = 4$. This field is rarely used now.
1.3.6.1.2.1.1.8	`sysLastChange`	A number containing the value of `sysUpTime` at the time of a change to any of the system SNMP objects.
1.3.6.1.2.1.1.9	`sysTable`	A node containing multiple `sysEntry` elements. Each element represents a distinct capability and the corresponding OID node value.

The Interfaces SNMP Variables Node

Similarly, the basic interface statistics can be obtained from the Interfaces SNMP Variables OID node subtree. The OID for the interfaces variables is 1.3.6.1.2.1.2 and contains two subnodes:

- An OID containing the total number of network interfaces. The OID value for this entry is 1.3.6.1.2.1.2.1; and it is usually referenced as `ifNumber`. There are no subnodes available under this OID.
- An OID node that contains all interface entries. Its OID is 1.3.6.1.2.1.2.2 and it is usually referenced as `ifTable`. This node contains one or more entry nodes. An entry node (1.3.6.1.2.1.2.2.1, also known as `ifEntry`) contains the detailed information about that particular interface. The number of entries in the list is defined by the `ifNumber` node value.

You can find detailed information about all `ifEntry` subnodes in Table 1-2.

Table 1-2. *Interface entry SNMP OIDs*

OID String	OID Name	Description
1.3.6.1.2.1.2.2.1.1	`ifIndex`	A unique sequence number assigned to the interface.
1.3.6.1.2.1.2.2.1.2	`ifDescr`	A string containing the interface name and other available information, such as the hardware manufacturer's name.
1.3.6.1.2.1.2.2.1.3	`ifType`	A number representing the interface type, depending on the interface's physical link and protocol.
1.3.6.1.2.1.2.2.1.4	`ifMtu`	The largest network datagram that this interface can transmit.
1.3.6.1.2.1.2.2.1.5	`ifSpeed`	The estimated current bandwidth of the interface. If the current bandwidth cannot be calculated, this number should contain the maximum possible bandwidth for the interface.
1.3.6.1.2.1.2.2.1.6	`ifPhysAddress`	The physical address of the interface, usually a MAC address on Ethernet interfaces.
1.3.6.1.2.1.2.2.1.7	`ifAdminStatus`	This OID allows setting the new state of the interface. Usually limited to the following values: 1 (Up), 2 (Down), 3 (Testing).
1.3.6.1.2.1.2.2.1.8	`ifOperStatus`	The current state of the interface. Usually limited to the following values: 1 (Up), 2 (Down), 3 (Testing).
1.3.6.1.2.1.2.2.1.9	`ifLastChange`	The value containing the system uptime (`sysUpTime`) reading when this interface entered its current state. May be set to zero if the interface entered this state before the last system reinitialization.
1.3.6.1.2.1.2.2.1.10	`ifInOctets`	The total number of bytes (octets) received on the interface.
1.3.6.1.2.1.2.2.1.11	`ifInUcastPkts`	The number of unicast packets forwarded to the device's network stack.
1.3.6.1.2.1.2.2.1.12	`ifInNUcastPkts`	The number of non-unicast packets delivered to the device's network stack. Non-unicast packets are usually either broadcast or multicast packets.
1.3.6.1.2.1.2.2.1.13	`ifInDiscards`	The number of dropped packets. This does not indicate a packet error, but may indicate that the receive buffer was too small to accept the packets.
1.3.6.1.2.1.2.2.1.14	`ifInErrors`	The number of received invalid packets.
1.3.6.1.2.1.2.2.1.15	`ifInUnknownProtos`	The number of packets that were dropped because the protocol is not supported on the device interface.
1.3.6.1.2.1.2.2.1.16	`ifOutOctets`	The number of bytes (octets) transmitted out of the interface.
1.3.6.1.2.1.2.2.1.17	`ifOutUcastPkts`	The number of unicast packets received from the device's network stack. This number also includes the packets that were discarded or not sent.

(*continued*)

Table 1-2. (*continued*)

OID String	OID Name	Description
1.3.6.1.2.1.2.2.1.18	`ifNUcastPkts`	The number of non-unicast packets received from the device's network stack. This number also includes the packets that were discarded or not sent.
1.3.6.1.2.1.2.2.1.19	`ifOutDiscards`	The number of valid packets that were discarded. It's not an error, but it may indicate that the send buffer is too small to accept all packets.
1.3.6.1.2.1.2.2.1.20	`ifOutErrors`	The number of outgoing packets that couldn't be transmitted because of the errors.
1.3.6.1.2.1.2.2.1.21	`ifOutQLen`	The length of the outbound packet queue.
1.3.6.1.2.1.2.2.1.22	`ifSpecific`	Usually contains a reference to the vendor-specific OID describing this interface. If such information is not available the value is set to an OID 0.0, which is syntactically valid, but is not pointing to anything.

Authentication in SNMP

Authentication in earlier SNMP implementations is somewhat primitive and is prone to attacks. An SNMP agent defines two community strings: one for read-only access and the other for read/write access. When the management system connects to the agent, it must authenticate with one of those two strings. The agent accepts commands only from a management system that has authenticated with valid community strings.

Querying SNMP from the Command Line

Before we start writing our application, let's quickly look at how to query SNMP from the command line. This is particularly useful if you want to check whether the information returned by the SNMP agent is correctly accepted by your application.

The command-line tools are provided by the Net-SNMP-Utils package, which is available for most Linux distributions. This package includes the tools to query and set SNMP objects. Consult your Linux distribution documentation for the details on installing this package. For example, on a RedHat-based system you can install these tools with the following command:

```
$ sudo yum install net-snmp-utils
```

On a Debian-based system the package can be installed like this:

```
$ sudo apt-get install snmp
```

The most useful command from this package is `snmpwalk`, which takes an OID node as an argument and tries to discover all subnode OIDs. This command uses the SNMP operation `getnext`, which returns the next node in the tree and effectively allows you to traverse the whole subtree from the indicated node. If no OID has been specified, `snmpwalk` will use the default SNMP system OID (1.3.6.1.2.1) as the starting point. Listing 1-1 demonstrates the `snmpwalk` command issued against a laptop running Fedora Linux.

Listing 1-1. An Example of the `snmpwalk` Command

```
$ snmpwalk -v2c -c public -On 192.168.1.68
.1.3.6.1.2.1.1.1.0 = STRING: Linux fedolin.example.com 2.6.32.11-99.fc12.i686 #1➥
SMP Mon Apr 5 16:32:08 EDT 2010 i686
.1.3.6.1.2.1.1.2.0 = OID: .1.3.6.1.4.1.8072.3.2.10
.1.3.6.1.2.1.1.3.0 = Timeticks: (110723) 0:18:27.23
.1.3.6.1.2.1.1.4.0 = STRING: Administrator (admin@example.com)
.1.3.6.1.2.1.1.5.0 = STRING: fedolin.example.com
.1.3.6.1.2.1.1.6.0 = STRING: MyLocation, MyOrganization, MyStreet, MyCity, MyCountry
.1.3.6.1.2.1.1.8.0 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.2.1 = OID: .1.3.6.1.6.3.10.3.1.1
.1.3.6.1.2.1.1.9.1.2.2 = OID: .1.3.6.1.6.3.11.3.1.1
.1.3.6.1.2.1.1.9.1.2.3 = OID: .1.3.6.1.6.3.15.2.1.1
.1.3.6.1.2.1.1.9.1.2.4 = OID: .1.3.6.1.6.3.1
.1.3.6.1.2.1.1.9.1.2.5 = OID: .1.3.6.1.2.1.49
.1.3.6.1.2.1.1.9.1.2.6 = OID: .1.3.6.1.2.1.4
.1.3.6.1.2.1.1.9.1.2.7 = OID: .1.3.6.1.2.1.50
.1.3.6.1.2.1.1.9.1.2.8 = OID: .1.3.6.1.6.3.16.2.2.1
.1.3.6.1.2.1.1.9.1.3.1 = STRING: The SNMP Management Architecture MIB.
.1.3.6.1.2.1.1.9.1.3.2 = STRING: The MIB for Message Processing and Dispatching.
.1.3.6.1.2.1.1.9.1.3.3 = STRING: The management information definitions for the➥
SNMP User-based Security Model.
.1.3.6.1.2.1.1.9.1.3.4 = STRING: The MIB module for SNMPv2 entities
.1.3.6.1.2.1.1.9.1.3.5 = STRING: The MIB module for managing TCP implementations
.1.3.6.1.2.1.1.9.1.3.6 = STRING: The MIB module for managing IP and ICMP➥
implementations
.1.3.6.1.2.1.1.9.1.3.7 = STRING: The MIB module for managing UDP implementations
.1.3.6.1.2.1.1.9.1.3.8 = STRING: View-based Access Control Model for SNMP.
.1.3.6.1.2.1.1.9.1.4.1 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.2 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.3 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.4 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.5 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.6 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.7 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.1.9.1.4.8 = Timeticks: (3) 0:00:00.03
.1.3.6.1.2.1.2.1.0 = INTEGER: 5
.1.3.6.1.2.1.2.2.1.1.1 = INTEGER: 1
.1.3.6.1.2.1.2.2.1.1.2 = INTEGER: 2
.1.3.6.1.2.1.2.2.1.1.3 = INTEGER: 3
.1.3.6.1.2.1.2.2.1.1.4 = INTEGER: 4
.1.3.6.1.2.1.2.2.1.1.5 = INTEGER: 5
.1.3.6.1.2.1.2.2.1.2.1 = STRING: lo
.1.3.6.1.2.1.2.2.1.2.2 = STRING: eth0
.1.3.6.1.2.1.2.2.1.2.3 = STRING: wlan1
.1.3.6.1.2.1.2.2.1.2.4 = STRING: pan0
.1.3.6.1.2.1.2.2.1.2.5 = STRING: virbr0
.1.3.6.1.2.1.2.2.1.3.1 = INTEGER: softwareLoopback(24)
.1.3.6.1.2.1.2.2.1.3.2 = INTEGER: ethernetCsmacd(6)
.1.3.6.1.2.1.2.2.1.3.3 = INTEGER: ethernetCsmacd(6)
.1.3.6.1.2.1.2.2.1.3.4 = INTEGER: ethernetCsmacd(6)
```

```
.1.3.6.1.2.1.2.2.1.3.5 = INTEGER: ethernetCsmacd(6)
.1.3.6.1.2.1.2.2.1.4.1 = INTEGER: 16436
.1.3.6.1.2.1.2.2.1.4.2 = INTEGER: 1500
.1.3.6.1.2.1.2.2.1.4.3 = INTEGER: 1500
.1.3.6.1.2.1.2.2.1.4.4 = INTEGER: 1500
.1.3.6.1.2.1.2.2.1.4.5 = INTEGER: 1500
.1.3.6.1.2.1.2.2.1.5.1 = Gauge32: 10000000
.1.3.6.1.2.1.2.2.1.5.2 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.5.3 = Gauge32: 10000000
.1.3.6.1.2.1.2.2.1.5.4 = Gauge32: 10000000
.1.3.6.1.2.1.2.2.1.5.5 = Gauge32: 10000000
.1.3.6.1.2.1.2.2.1.6.1 = STRING:
.1.3.6.1.2.1.2.2.1.6.2 = STRING: 0:d:56:7d:68:b0
.1.3.6.1.2.1.2.2.1.6.3 = STRING: 0:90:4b:64:7b:4d
.1.3.6.1.2.1.2.2.1.6.4 = STRING: 4e:e:b8:9:81:3b
.1.3.6.1.2.1.2.2.1.6.5 = STRING: d6:f9:7c:2c:17:28
.1.3.6.1.2.1.2.2.1.7.1 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.7.2 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.7.3 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.7.4 = INTEGER: down(2)
.1.3.6.1.2.1.2.2.1.7.5 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.8.1 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.8.2 = INTEGER: down(2)
.1.3.6.1.2.1.2.2.1.8.3 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.8.4 = INTEGER: down(2)
.1.3.6.1.2.1.2.2.1.8.5 = INTEGER: up(1)
.1.3.6.1.2.1.2.2.1.9.1 = Timeticks: (0) 0:00:00.00
.1.3.6.1.2.1.2.2.1.9.2 = Timeticks: (0) 0:00:00.00
.1.3.6.1.2.1.2.2.1.9.3 = Timeticks: (0) 0:00:00.00
.1.3.6.1.2.1.2.2.1.9.4 = Timeticks: (0) 0:00:00.00
.1.3.6.1.2.1.2.2.1.9.5 = Timeticks: (0) 0:00:00.00
.1.3.6.1.2.1.2.2.1.10.1 = Counter32: 89275
.1.3.6.1.2.1.2.2.1.10.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.10.3 = Counter32: 11649462
.1.3.6.1.2.1.2.2.1.10.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.10.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.11.1 = Counter32: 1092
.1.3.6.1.2.1.2.2.1.11.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.11.3 = Counter32: 49636
.1.3.6.1.2.1.2.2.1.11.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.11.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.12.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.12.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.12.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.12.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.12.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.13.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.13.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.13.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.13.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.13.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.14.1 = Counter32: 0
```

```
.1.3.6.1.2.1.2.2.1.14.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.14.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.14.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.14.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.15.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.15.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.15.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.15.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.15.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.16.1 = Counter32: 89275
.1.3.6.1.2.1.2.2.1.16.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.16.3 = Counter32: 922277
.1.3.6.1.2.1.2.2.1.16.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.16.5 = Counter32: 3648
.1.3.6.1.2.1.2.2.1.17.1 = Counter32: 1092
.1.3.6.1.2.1.2.2.1.17.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.17.3 = Counter32: 7540
.1.3.6.1.2.1.2.2.1.17.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.17.5 = Counter32: 17
.1.3.6.1.2.1.2.2.1.18.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.18.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.18.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.18.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.18.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.19.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.19.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.19.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.19.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.19.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.20.1 = Counter32: 0
.1.3.6.1.2.1.2.2.1.20.2 = Counter32: 0
.1.3.6.1.2.1.2.2.1.20.3 = Counter32: 0
.1.3.6.1.2.1.2.2.1.20.4 = Counter32: 0
.1.3.6.1.2.1.2.2.1.20.5 = Counter32: 0
.1.3.6.1.2.1.2.2.1.21.1 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.21.2 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.21.3 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.21.4 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.21.5 = Gauge32: 0
.1.3.6.1.2.1.2.2.1.22.1 = OID: .0.0
.1.3.6.1.2.1.2.2.1.22.2 = OID: .0.0
.1.3.6.1.2.1.2.2.1.22.3 = OID: .0.0
.1.3.6.1.2.1.2.2.1.22.4 = OID: .0.0
.1.3.6.1.2.1.2.2.1.22.5 = OID: .0.0
.1.3.6.1.2.1.25.1.1.0 = Timeticks: (8232423) 22:52:04.23
.1.3.6.1.2.1.25.1.1.0 = No more variables left in this MIB View (It is past the end↩
of the MIB tree)
```

As an exercise, try to identify some of the listed OIDs using Tables 1-1 and 1-2 and find out what they mean.

Querying SNMP Devices from Python

Now we know enough about SNMP to start working on our own management system, which will be querying the configured systems on regular intervals. First let's specify the configuration that we will be using in the application.

Configuring the Application

As we already know, we need the following information available for every check:

- An IP address or resolvable domain name of the system that runs the SNMP agent software
- The read-only community string that will be used to authenticate with the agent software
- The OID node's numerical representation

We are going to use the Windows INI-style configuration file because of its simplicity. Python includes a configuration parsing module by default, so it is also convenient to use. (Chapter 9 discusses the `ConfigParser` module in great detail; refer to that chapter for more information about the module.)

Let's go back to the configuration file for our application. There is no need to repeat the system information for every SNMP object that we're going to query, so we can define each system parameter once in a separate section and then refer to the system ID in each check section. The check section defines the OID node identifier string and a short description, as shown in Listing 1-2. Create a configuration file called `snmp-manage.cfg` with the contents from the listing below; don't forget to modify the IP and security details accordingly.

Listing 1-2. The Management System Configuration File

```
[system_1]
description=My Laptop
address=192.168.1.68
port=161
communityro=public

[check_1]
description=WLAN incoming traffic
oid=1.3.6.1.2.1.2.2.1.10.3
system=system_1

[check_2]
description=WLAN incoming traffic
oid=1.3.6.1.2.1.2.2.1.16.3
system=system_1
```

Make sure that the `system` and `check` section IDs are unique, or you may get unpredictable results.

We're going to create an `SnmpManager` class with two methods, one to add a system and the other to add a check. As the check contains the system ID string, it will automatically be assigned to that particular system. In Listing 1-3 you can see the class definition and also the initialization part that reads in the configuration and iterates through the sections and updates the class object accordingly. Create a file called `snmp-manage.py` with the contents shown in the listing below. We will work on adding new features to the script as we go along.

Listing 1-3. Reading and Storing the Configuration

```
import sys
from ConfigParser import SafeConfigParser

class SnmpManager:
    def __init__(self):
        self.systems = {}

    def add_system(self, id, descr, addr, port, comm_ro):
        self.systems[id] = {'description' : descr,
                            'address'     : addr,
                            'port'        : int(port),
                            'communityro' : comm_ro,
                            'checks'      : {}
                           }

    def add_check(self, id, oid, descr, system):
        oid_tuple = tuple([int(i) for i in oid.split('.')])
        self.systems[system]['checks'][id] = {'description': descr,
                                              'oid'        : oid_tuple,
                                             }

def main(conf_file=""):
    if not conf_file:
        sys.exit(-1)
    config = SafeConfigParser()
    config.read(conf_file)
    snmp_manager = SnmpManager()
    for system in [s for s in config.sections() if s.startswith('system')]:
        snmp_manager.add_system(system,
                                config.get(system, 'description'),
                                config.get(system, 'address'),
                                config.get(system, 'port'),
                                config.get(system, 'communityro'))
    for check in [c for c in config.sections() if c.startswith('check')]:
        snmp_manager.add_check(check,
                               config.get(check, 'oid'),
                               config.get(check, 'description'),
                               config.get(check, 'system'))

if __name__ == '__main__':
    main(conf_file='snmp-manager.cfg')
```

As you see in the example, we first have to iterate through the system sections and update the object before proceeding with the check sections.

■ **Note** This order is important, because if we try to add a check for a system that hasn't been inserted yet, we'll get a dictionary index error.

Also note that we are converting the OID string to a tuple of integers. You'll see why we have to do this later in this section. The configuration file is loaded and we're ready to run SNMP queries against the configured devices.

Using the PySNMP Library

In this project we are going to use the PySNMP library, which is implemented in pure Python and doesn't depend on any precompiled libraries. The `pysnmp` package is available for most Linux distributions and can be installed using the standard distribution package manager. In addition to `pysnmp` you will also need the ASN.1 library, which is used by `pysnmp` and is also available as part of the Linux distribution package selection. For example, on a Fedora system you can install the `pysnmp` module with the following commands:

```
$ sudo yum install pysnmp
$ sudo yum install python-pyasn1
```

Alternatively, you can use the Python Package manager (PiP) to install this library for you:

```
$ sudo pip install pysnmp
$ sudo pip install pyasn1
```

If you don't have the `pip` command available, you can download and install this tool from http://pypi.python.org/pypi/pip. We will use it in later chapters as well.

The PySNMP library hides all the complexity of SNMP processing behind a single class with a simple API. All you have to do is create an instance of the `CommandGenerator` class. This class is available from the `pysnmp.entity.rfc3413.oneliner.cmdgen` module and implements most of the standard SNMP protocol commands: `getCmd()`, `setCmd()`, and `nextCmd()`. Let's look at each of these in more detail.

The SNMP GET Command

All the commands we are going to discuss follow the same invocation pattern: import the module, create an instance of the CommandGenerator class, create three required parameters (an authentication object, a transport target object, and a list of arguments), and finally invoke the appropriate method. The method returns a tuple containing the error indicators (if there was an error) and the result object.

In Listing 1-4, we query a remote Linux machine using the standard SNMP OID (1.3.6.1.2.1.1.1.0).

Listing 1-4. An Example of the SNMP GET Command

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> cg = cmdgen.CommandGenerator()
>>> comm_data = cmdgen.CommunityData('my-manager', 'public')
>>> transport = cmdgen.UdpTransportTarget(('192.168.1.68', 161))
>>> variables = (1, 3, 6, 1, 2, 1, 1, 1, 0)
>>> errIndication, errStatus, errIndex, result = cg.getCmd(comm_data, transport, variables)
>>> print errIndication
None
>>> print errStatus
0
>>> print errIndex
0
>>> print result
[(ObjectName('1.3.6.1.2.1.1.1.0'), OctetString('Linux fedolin.example.com↩
 2.6.32.11-99.fc12.i686 #1 SMP Mon Apr 5 16:32:08 EDT 2010 i686'))]
>>>
```

Let's look at some steps more closely. When we initiate the community data object, we have provided two strings—the community string (the second argument) and the agent or manager security name string; in most cases this can be any string. An optional parameter specifies the SNMP version to be used (it defaults to SNMP v2c). If you must query version 1 devices, use the following command:

```
>>> comm_data = cmdgen.CommunityData('my-manager', 'public', mpModel=0)
```

The transport object is initiated with the tuple containing either the fully qualified domain name or the IP address string and the integer port number.

The last argument is the OID expressed as a tuple of all node IDs that make up the OID we are querying. Therefore, we had to convert the dot-separated string into a tuple earlier when we were reading the configuration items.

Finally, we call the API command `getCmd()`, which implements the SNMP GET command, and pass these three objects as its arguments. The command returns a tuple, each element of which is described in Table 1-3.

Table 1-3. *CommandGenerator Return Objects*

Tuple Element	Description
`errIndication`	If this string is not empty, it indicates the SNMP engine error.
`errStatus`	If this element evaluates to True, it indicates an error in the SNMP communication; the object that generated the error is indicated by the `errIndex` element.
`errIndex`	If the `errStatus` indicates that an error has occurred, this field can be used to find the SNMP object that caused the error. The object position in the `result` array is `errIndex-1`.
`result`	This element contains a list of all returned SNMP object elements. Each element is a tuple that contains the name of the object and the object value.

The SNMP SET Command

The SNMP SET commandis mapped in PySNMP to the `setCmd()` method call. All parameters are the same; the only difference is that the variables section now contains a tuple: the OID and the new value. Let's try to use this command to change a read-only object; Listing 1-5 shows the command-line sequence.

Listing 1-5. An Example of the SNMP SET Command

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> from pysnmp.proto import rfc1902
>>> cg = cmdgen.CommandGenerator()
>>> comm_data = cmdgen.CommunityData('my-manager', 'public')
>>> transport = cmdgen.UdpTransportTarget(('192.168.1.68', 161))
>>> variables = ((1, 3, 6, 1, 2, 1, 1, 1, 0), rfc1902.OctetString('new system description'))
>>> errIndication, errStatus, errIndex, result = cg.setCmd(comm_data, transport,↵
variables)
>>> print errIndication
None
>>> print errStatus
6
>>> print errIndex
1
```

```
>>> print errStatus.prettyPrint()
noAccess(6)
>>> print result
[(ObjectName('1.3.6.1.2.1.1.1.0'), OctetString('new system description'))]
>>>
```

What happened here is that we tried to write to a read-only object, and that resulted in an error. What's interesting in this example is how we format the parameters. You have to convert strings to SNMP object types; otherwise; they won't pass as valid arguments. Therefore the string had to be encapsulated in an instance of the `OctetString` class. You can use other methods of the `rfc1902` module if you need to convert to other SNMP types; the methods include `Bits()`, `Counter32()`, `Counter64()`, `Gauge32()`, `Integer()`, `Integer32()`, `IpAddress()`, `OctetString()`, `Opaque()`, `TimeTicks()`, and `Unsigned32()`. These are all class names that you can use if you need to convert a string to an object of a specific type.

The SNMP GETNEXT Command

The SNMP GETNEXT command is implemented as the `nextCmd()` method. The syntax and usage are identical to `getCmd()`; the only difference is that the result is a list of objects that are immediate subnodes of the specified OID node.

Let's use this command to query all objects that are immediate child nodes of the SNMP system OID (1.3.6.1.2.1.1); Listing 1-6 shows the `nextCmd()` method in action.

Listing 1-6. An Example of the SNMP GETNEXT Command

```
>>> from pysnmp.entity.rfc3413.oneliner import cmdgen
>>> cg = cmdgen.CommandGenerator()
>>> comm_data = cmdgen.CommunityData('my-manager', 'public')
>>> transport = cmdgen.UdpTransportTarget(('192.168.1.68', 161))
>>> variables = (1, 3, 6, 1, 2, 1, 1)
>>> errIndication, errStatus, errIndex, result = cg.nextCmd(comm_data, transport, variables)
>>> print errIndication
requestTimedOut
>>> errIndication, errStatus, errIndex, result = cg.nextCmd(comm_data, transport, variables)
>>> print errIndication
None
>>> print errStatus
0
>>> print errIndex
0
>>> for object in result:
...  print object
...
[(ObjectName('1.3.6.1.2.1.1.1.0'), OctetString('Linux fedolin.example.com↩
 2.6.32.11-99.fc12.i686 #1 SMP Mon Apr 5 16:32:08 EDT 2010 i686'))]
[(ObjectName('1.3.6.1.2.1.1.2.0'), ObjectIdentifier('1.3.6.1.4.1.8072.3.2.10'))]
[(ObjectName('1.3.6.1.2.1.1.3.0'), TimeTicks('340496'))]
[(ObjectName('1.3.6.1.2.1.1.4.0'), OctetString('Administrator (admin@example.com)'))]
[(ObjectName('1.3.6.1.2.1.1.5.0'), OctetString('fedolin.example.com'))]
[(ObjectName('1.3.6.1.2.1.1.6.0'), OctetString('MyLocation, MyOrganization,↩
MyStreet, MyCity, MyCountry'))]
```

```
[(ObjectName('1.3.6.1.2.1.1.8.0'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.1'), ObjectIdentifier('1.3.6.1.6.3.10.3.1.1'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.2'), ObjectIdentifier('1.3.6.1.6.3.11.3.1.1'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.3'), ObjectIdentifier('1.3.6.1.6.3.15.2.1.1'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.4'), ObjectIdentifier('1.3.6.1.6.3.1'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.5'), ObjectIdentifier('1.3.6.1.2.1.49'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.6'), ObjectIdentifier('1.3.6.1.2.1.4'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.7'), ObjectIdentifier('1.3.6.1.2.1.50'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.2.8'), ObjectIdentifier('1.3.6.1.6.3.16.2.2.1'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.1'), OctetString('The SNMP Management↩
Architecture MIB.'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.2'), OctetString('The MIB for Message Processing↩
and Dispatching.'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.3'), OctetString('The management information↩
 definitions for the SNMP User-based Security Model.'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.4'), OctetString('The MIB module for SNMPv2↩
entities'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.5'), OctetString('The MIB module for managing TCP↩
 implementations'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.6'), OctetString('The MIB module for managing IP↩
 and ICMP implementations'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.7'), OctetString('The MIB module for managing UDP↩
 implementations'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.3.8'), OctetString('View-based Access Control Model↩
for SNMP.'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.1'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.2'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.3'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.4'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.5'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.6'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.7'), TimeTicks('3'))]
[(ObjectName('1.3.6.1.2.1.1.9.1.4.8'), TimeTicks('3'))]
>>>
```

As you can see, the result is identical to that produced by the command-line tool `snmpwalk`, which uses the same technique to retrieve the SNMP OID subtree.

Implementing the SNMP Read Functionality

Let's implement the read functionality in our application. The workflow will be as follows: we need to iterate through all systems in the list, and for each system we iterate through all defined checks. For each check we are going to perform the SNMP GET command and store the result in the same data structure.

For debugging and testing purposes we will add some print statements to verify that the application is working as expected. Later we'll replace those print statements with the RRDTool database store commands. I'm going to call this method `query_all_systems()`. Listing 1-7 shows the code, which you would want to add to the `snmp-manager.py` file you created earlier.

Listing 1-7. Querying All Defined SNMP Objects

```
def query_all_systems(self):
    cg = cmdgen.CommandGenerator()
    for system in self.systems.values():
        comm_data = cmdgen.CommunityData('my-manager', system['communityro'])
        transport = cmdgen.UdpTransportTarget((system['address'], system['port']))
        for check in system['checks'].values():
            oid = check['oid']
            errInd, errStatus, errIdx, result = cg.getCmd(comm_data, transport, oid)
            if not errInd and not errStatus:
                print "%s/%s -> %s" % (system['description'],
                                       check['description'],
                                       str(result[0][1]))
```

If you run the tool you'll get results similar to these (assuming you correctly pointed your configuration to the working devices that respond to the SNMP queries):

```
$ ./snmp-manager.py
My Laptop/WLAN outgoing traffic -> 1060698
My Laptop/WLAN incoming traffic -> 14305766
```

Now we're ready to write all this data to the RRDTool database.

Storing Data with RRDTool

RRDTool is an application developed by Tobias Oetiker, which has become a de facto standard for graphing monitoring data. The graphs produced by RRDTool are used in many different monitoring tools, such as Nagios, Cacti, and so on. In this section we'll look at the structure of the RRTool database and the application itself. We'll discuss the specifics of the round robin database, how to add new data to it, and how to retrieve it later on. We will also look at the data-plotting commands and techniques. And finally we'll integrate the RRDTool database with our application.

Introduction to RRDTool

As I have noted, RRDTool provides three distinct functions. First, it serves as a database management system by allowing you to store and retrieve data from its own database format. It also performs complex data-manipulation tasks, such as data resampling and rate calculations. And finally, it allows you to create sophisticated graphs incorporating data from various source databases.

Let's start by looking at the round robin database structure I apologize for the number of acronyms that you'll come across in this section, but it is important to mention them here, as they all are used in the configuration of RRDTool, so it is vital to become familiar with them.

The first property that makes an RRD different from conventional databases is that the database has a limited size. This means that the database size is known at the time it is initialized, and the size never changes. New records overwrite old data, and that process is repeated over and over again. Figure 1-3 shows a simplified version of the RRD to help you to visualize the structure.

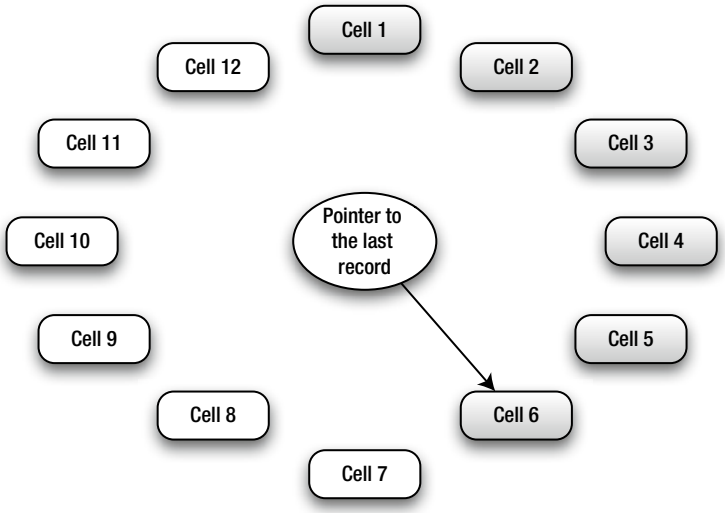

Figure 1-3. *The RRD structure*

Let's assume that we have initialized a database that is capable of holding 12 records, each in its own cell. When the database is empty, we start by writing data to cell number 1. We also update the pointer with the ID of the last cell we've written the data to. Figure 1-3 shows that 6 records have already been written to the database (as represented by the shaded boxes). The pointer is on cell 6, and so when the next write instruction is received, the database will write it to the next cell (cell 7) and update the pointer accordingly. Once the last cell (cell 12) is reached, the process starts again, from cell number 1.

The RRD data store's only purpose is to store performance data, and therefore it does not require maintaining complex relations between different data tables. In fact, there are no tables in the RRD, only the individual data sources (DSs).

The last important property of the RRD is that the database engine is designed to store the time series data, and therefore each record needs to be marked with a timestamp. Furthermore, when you create a new database you are required to specify the sampling rate, the rate at which entries are being written to the database. The default value is 300 seconds, or 5 minutes, but this can be overridden if required.

The data that is stored in the RDD is called a Round Robin Archive (RRA). The RRA is what makes the RRD so useful. It allows you to consolidate the data gathered from the DS by applying an available consolidation function (CF). You can specify one of the four CFs (`average`, `min`, `max`, and `last`) that will be applied to a number of the actual data records. The result is stored in a round robin "table." You can store multiple RRAs in your database with different granularity. For example, one RRA stores average values of the last 10 records and the other one stores an average of the last 100 records.

This will all come together when we look at the usage scenarios in the next sections.

Using RRDTool from a Python Program

Before we start creating the RRDTool databases, let's look at the Python module that provides the API to RRDTool. The module we are going to use in this chapter is called the Python RRDTool, and it is available to download at http://sourceforge.net/projects/py-rrdtool/.

However, most Linux distributions have this prepackaged and available to install using the standard package management tool. For example, on a Fedora system you would run the following command to install the Python RRDTool module:

```
$ sudo yum install rrdtool-python
```

On Debian-based systems the install command is:

```
$ sudo apt-get install python-rrd
```

Once the package is installed, you can validate that the installation was successful:

```
$ python
Python 2.6.2 (r262:71600, Jan 25 2010, 18:46:45)
[GCC 4.4.2 20091222 (Red Hat 4.4.2-20)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rrdtool
>>> rrdtool.__version__
'$Revision: 1.14 $'
>>>
```

Creating a Round Robin Database

Let's start by creating a simple database. The database we are going to create will have one data source, which is a simple increasing counter: the counter value increases over time. A classical example of such a counter is bytes transmitted over the interface. The readings are performed every 5 minutes.

We also are going to define two RRAs. One is to average over a single reading, which effectively instructs RRDTool to store the actual values, and the other will average over six measurements. Following is an example of the command-line tool syntax for creating this database:

```
$ rrdtool create interface.rrd \
> DS:packets:COUNTER:600:U:U \
> RRA:AVERAGE:0.5:1:288 \
> RRA:AVERAGE:0.5:6:336
```

Similarly, you can use the Python module to create the same database:

```
>>> import rrdtool
>>> rrdtool.create('interface.rrd',
...                'DS:packets:COUNTER:600:U:U',
...                'RRA:AVERAGE:0.5:1:288',
...                'RRA:AVERAGE:0.5:6:336')
>>>
```

The structure of the DS (data source) definition line is:

```
DS:<name>:<DS type>:<heartbeat>:<lower limit>:<upper limit>
```

The *name* field is what you name this particular data source. Since RRD allows you to store the data from multiple data sources, you must provide a unique name for each so that you can access it later. If you need to define more than one data source, simply add another DS line.

The DS type (or data source type) field indicates what type of data will be supplied to this data source. There are four types available: COUNTER, GAUGE, DERIVE, and ABSOLUTE:

- The COUNTER type means that the measurement value is increasing over time. To calculate a rate, RRDTool subtracts the last value from the current measurement and divides by the measurement step (or sampling rate) to obtain the rate figure. If the result is a negative number, it needs to compensate for the counter rollover. A typical use is monitoring ever-increasing counters, such as total number of bytes transmitted through the interface.
- The DERIVE type is similar to COUNTER, but it also allows for a negative rate. You can use this type to check the rate of incoming HTTP requests to your site. If the graph is above the zero line, this means you are getting more and more requests. If it drops below the zero line, it means your website is becoming less popular.
- The ABSOLUTE type indicates that the counter is reset every time you read the measurement. Whereas with the COUNTER and DERIVE types, RRDTool subtracted the last measurement from the current one before dividing by the time period, ABSOLUTE tells it not to perform the subtraction operation. You use this on counters that are reset at the same rate that you do the measurements. For example, you could measure the system average load (over the last 15 minutes) reading every 15 minutes. This would represent the rate of change of the average system load.
- The GAUGE type means that the measurement is the rate value, and no calculations need to be performed. For example, current CPU usage and temperature sensor readings are good candidates for the GAUGE type.

The *heartbeat* value indicates how much time to allow for the reading to come in before resetting it to the unknown state. RRDTool allows for data misses, but it does not make any assumptions and it uses the special value unknown if the data is not received. In our example we have the heartbeat set to 600, which means that the database waits for two readings (remember, the step is 300) before it declares the next measurement to be unknown.

The last two fields indicate the minimum and maximum values that can be received from the data source. If you specify those, anything falling outside that range will be automatically marked as unknown.

The RRA definition structure is:

```
RRA:<consolidation function>:<XFiles factor>:<dataset>:<samples>
```

The *consolidation function* defines what mathematical function will be applied to the *dataset* values. The *dataset* parameter is the last dataset measurements received from the data source. In our example we have two RRAs, one with just a single reading in the dataset and the other with six measurements in the dataset. The available consolidation functions are AVERAGE, MIN, MAX, and LAST:

- AVERAGE instructs RRDTool to calculate the average value of the dataset and store it.
- MIN and MAX selects either the minimum or maximum value from the dataset and stores it.
- LAST indicates to use the last entry from the dataset.

The *XFiles factor* value shows what percentage of the dataset can have unknown values and the consolidation function calculation will still be performed. For example, if the setting is 0.5 (50%), then three out of six measurements can be unknown and the average value for the dataset will still be calculated. If four readings are missed, the calculation is not performed and the unknown value is stored in the RRA. Set this to 0 (0% miss allowance) and the calculation will be performed only if all data points in the dataset are available. It seems to be a common practice to keep this setting at 0.5.

As already discussed, the *dataset* parameter indicates how many records are going to participate in the consolidation function calculation.

And finally, *samples* tells RRDTool how many CF results should be kept. So, going back to our example, the number 288 tells RRDTool to keep 288 records. Because we're measuring every 5 minutes, this is 24 hours of data (288/(60/5)). Similarly, the number 336 means that we are storing 7 days' worth of data (336/(60/30)/24) at the 30-minute sampling rate. As you can see, the data in the second RRA is resampled; we've changed the sampling rate from 5 minutes to 30 minutes by consolidating data of every six (5-minute) samples.

Writing and Reading Data from the Round Robin Database

Writing data to the RRD data file is very simple. You just call the update command and, assuming you have defined multiple data sources, supply it a list of data source readings in the same order as you specified when you created the database file. Each entry must be preceded by the current (or desired) timestamp, expressed in seconds since the epoch (1970-01-01). Alternatively, instead of using the actual number to express the timestamp, you can use the character *N*, which means the current time. It is possible to supply multiple readings in one command:

```
$ date +"%s"
1273008486
$ rrdtool update interface.rrd 1273008486:10
$ rrdtool update interface.rrd 1273008786:15
$ rrdtool update interface.rrd 1273009086:25
$ rrdtool update interface.rrd 1273009386:40 1273009686:60 1273009986:66
$ rrdtool update interface.rrd 1273010286:100 1273010586:160 1273010886:166
```

The Python alternative looks very similar. In the following code, we will insert another 20 records, specifying regular intervals (of 300 seconds) and supplying generated measurements:

```
>>> import rrdtool
>>> for i in range(20):
...  rrdtool.update('interface.rrd',
...                 '%d:%d' % (1273010886 + (1+i)*300, i*10+200))
...
>>>
```

Now let's fetch the data back from the RRDTool database:

```
$ rrdtool fetch interface.rrd AVERAGE
                        packets

1272983100: -nan
[...]
1273008600: -nan
1273008900: 2.3000000000e-02
1273009200: 3.9666666667e-02
1273009500: 5.6333333333e-02
```

```
1273009800: 4.8933333333e-02
1273010100: 5.5466666667e-02
1273010400: 1.4626666667e-01
1273010700: 1.3160000000e-01
1273011000: 5.5466666667e-02
1273011300: 8.2933333333e-02
1273011600: 3.3333333333e-02
1273011900: 3.3333333333e-02
1273012200: 3.3333333333e-02
1273012500: 3.3333333333e-02
1273012800: 3.3333333333e-02
1273013100: 3.3333333333e-02
1273013400: 3.3333333333e-02
1273013700: 3.3333333333e-02
1273014000: 3.3333333333e-02
1273014300: 3.3333333333e-02
1273014600: 3.3333333333e-02
1273014900: 3.3333333333e-02
1273015200: 3.3333333333e-02
1273015500: 3.3333333333e-02
1273015800: 3.3333333333e-02
1273016100: 3.3333333333e-02
1273016400: 3.3333333333e-02
1273016700: 3.3333333333e-02
1273017000: -nan
[...]
1273069500: -nan
```

If you count the number of entries, you'll see that it matches the number of updates we've performed on the database. This means that we are seeing results at the maximum resolution— in our case, a sample per record. Showing results at the maximum resolution is the default behavior, but you can select another resolution (provided that it has a matching RRA) by specifying the `resolution` flag. Bear in mind that the resolution must be expressed in the number of seconds and not in the number of samples in the RRA definition. Therefore, in our example the next available resolution is 6 (samples) * 300 (seconds/sample) = 1800 (seconds):

```
$ rrdtool fetch interface.rrd AVERAGE -r 1800
                        packets

[...]
1273010400: 6.1611111111e-02
1273012200: 6.1666666667e-02
1273014000: 3.3333333333e-02
1273015800: 3.3333333333e-02
1273017600: 3.3333333333e-02
[...]
```

Now, you may have noticed that the records inserted by our Python application result in the same number stored in the database. Why is that? Is the counter definitely increasing? Remember, RRDTool always stores the *rate* and not the actual values. So the figures you see in the result dataset show how fast the values are *changing*. And because the Python application generates new measurements at a steady rate (the difference between values is always the same), the rate figure is always the same.

What does this number exactly mean? We know that generated values are increasing by 10 every time we insert a new record, but the value printed by the `fetch` command is 3.3333333333e-02. (For many people this may look slightly confusing, but it's just another notation for the value 0.0333(3).) Where did that come from? In discussing the different data source types, I mentioned that RRDTool takes the difference between two data point values and divides that by the number of seconds in the sampling interval. The default sampling interval is 300 seconds, so the rate has been calculated as 10/300 = 0.0333(3), which is what is written to the RRDTool database. In other words, this means that our counter on average increases by 0.0333(3) every second. Remember that all rate measurements are stored as a change per second. We'll look at converting this value to something more readable later in the section.

Here's is how you retrieve the data using the Python module method call:

```
>>> for i in rrdtool.fetch('interface.rrd', 'AVERAGE'): print i
...
(1272984300, 1273071000, 300)
('packets',)
[(None,), [...], (None,), (0.023,), (0.03966666666666667,), (0.056333333333333339,),➥
 (0.048933333333333336,), (0.055466666666666671,), (0.14626666666666666,),
 (0.13160000000000002,),  (0.055466666666666671,), (0.082933333333333331,),
 (0.033333333333333333,), (0.033333333333333333,), (0.033333333333333333,),
 (0.033333333333333333,), (0.033333333333333333,), (0.033333333333333333,),
 (0.033333333333333333,), (0.033333333333333333,), (0.033333333333333333,),
 (0.033333333333333333,), (0.033333333333333333,), (0.033333333333333333,),
 (0.033333333333333333,), (0.033333333333333333,), (0.033333333333333333,),
 (None,), [...] (None,)]
>>>
```

The result is a tuple of three elements: *`dataset information`*, *`list of datasources`*, and *`result array`*:

- *`Dataset information`* is another tuple that has three values: start and end timestamps and the sampling rate.
- *`List of datasources`* simply lists all variables that were stored in the RRDTool database and that were returned by your query.
- *`Result array`* contains the actual values that are stored in the RRD. Each entry is a tuple, containing values for every variable that was queried. In our example database we had only one variable; therefore the tuple contains only one element. If the value could not be calculated (is `unknown`), Python's None object is returned.

You can also change the sampling rate if you need to:

```
>>> rrdtool.fetch('interface.rrd', 'AVERAGE', '-r', '1800')
((1272983400, 1273071600, 1800), ('packets',), [(None,), [...] (None,),
 (0.06161111111111111,), (0.061666666666666668,), (0.033333333333333333,),
 (0.033333333333333333,), (0.033333333333333333,), (None,), [...] (None,)])
```