IBM

developerWorks Technical topics AIX and UNIX Technical library

Using Python to create UNIX command line tools

A Python command line interface manifesto

If you work in IT, as a UNIX® Sysadmin, a software developer, or even a manager, there a few skills that will set you apart from the crowd. Do you fully understand the OSI model? Are you comfortable with subnetting? Do you understand UNIX permissions? Let me add to this list the humble command line tool. By the end of this article, anyone involved in IT at any capacity should be able to create at least a simple command line tool.

Share:

Noah Gift is the co-author of "Python For Unix and Linux" by O'Reilly. He is an author, speaker, consultant, and community leader, writing for publications such as IBM developerWorks, Red Hat Magazine, O'Reilly and MacTech. His consulting company's website is www.giftcs.com, and his personal website is www.noahgift.com. Noah is also the current organizer for www.pyatl.org, which is the Python User Group for Atlanta, GA. He has a Master's degree in CIS from Cal State Los Angeles, B.S. in Nutritional Science from Cal Poly San Luis Obispo, is an Apple and LPI certified SysAdmin, and has worked at companies such as, Caltech, Disney Feature Animation, Sony Imageworks, and Luxo Digital Studios. In his free time he enjoys spending time with his wife Leah, and their son Liam, playing the piano, and exercising religiously

18 March 2008

Also available in Chinese

Introduction

Can you write a command line tool? Maybe you can, but can you write a really good command line tool? This articles covers making a robust command line tool in Python, complete with built-in help menus, error handling, and option handling. For some strange reason, it is not widely known that the standard library in Python® has all of the tools necessary to make incredible powerful *NIX command line tools.

Arguably, Python is the best language for making *NIX command line tools, period, due to its batteries-included philosophy, and its emphasis on readable code. Just a warning, though; these are dangerous ideas, when you find out how easy it is to create a command line tool in Python, you might be spoiled for life. To my knowledge, there has never been an article published in this detail on creating command line tools in Python, so I hope you enjoy it.

Setup

The optparse module in the Python Standard Library does most of the dirty work of creating a command line tool. Optparse was included in Python 2.3, so this module will be included on many *NIX Operating Systems. If for some reason, the operating system you are on does not include a module you need, it is a great comfort to know the latest versions of Python have been tested and compiles on just about any *NIX Operating Systems. Python support includes IBM® AIX®, HP-UX, Solaris, Free BSD, Red Hat Linux®, Ubuntu, OS X, IRIX, and even a Nokia phone or two.

Creating a Hello World command line tool

The first step to writing a great command line tool is to define the problem you wish to solve. This is critical to the success of your tool. It is also important to solve the problem in the simplest way possible. The KISS principle, Keep It Simple Stupid, definitely applies here. Add options and increased functionality only after the planned functionality is implemented and tested.

Let's get started by creating a Hello World command line tool. Following the advice above, let's define the problem in the simplest possible terms.

Problem Definition: I would like to create a command line tool that prints Hello World by default, but takes an option to print the name of a different person.

Given that description, this is one possible solution that comes in at just a handful of lines of code.

Hello World command line interface (CLI)

```
#!/usr/bin/env python
import optparse

def main():
  p = optparse.OptionParser()
  p.add_option('--person', '-p', default="world")
  options, arguments = p.parse_args()
  print 'Hello %s' % options.person

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

If we run this code, we get the expected output of:

```
Hello world
```

Our handful of code does quite a bit more than that, though. We get a help menu for free:

```
python hello_cli.py --help
   Usage: hello_cli.py [options]

   Options:
   -h, --help            show this help message and exit
   -p PERSON, --person=PERSON
```

We can see from the help menu that we have two ways to change the output of Hello World:

```
python hello_cli.py -p guido
Hello guido
```

We also get error handling for free:

```
python hello_cli.py --name matz
Usage: hello_cli.py [options]

hello_cli.py: error: no such option: --name
```

If you have not used the optparse module in Python, you might be jumping out of your seat right now, thinking about all of the incredible tools you can write in Python. If you are new to Python, it might surprise you know that everything is this easy in Python. There is a funny comic about this very subject of "how easy Python is," by "xkcd" that I have included in the Resources.

Creating a useful command line tool

Now that we have the basics out of the way, we can move onto creating a tool to solve a specific problem. For this example, we will be using a Python network library and interactive tool called Scapy. Scapy works on most *NIX systems, can send packets on layer 2 and layer 3, and allows you to create incredibly sophisticated tools in just a few lines of Python. If you would like to follow along from home, make sure you have the proper prerequisites installed.

Let's define a new problem to solve.

Problem: I would like to create a command line tool takes an IP address or a subnet as an argument, and return a MAC address or a list of MAC addresses with their respective IP address to standard out.

Now that we have a clearly defined problem, let's try to separate the problem in the simplest possible pieces, and solve those pieces individually. For this problem I see two separate pieces. The first piece is

to write a function that takes an IP address or a subnet range, and returns a MAC address or MAC addresses. We can worry about integrating this into a command line tool after we solve this problem.

Solution Part 1: Creating a Python function to determine MAC addresses from IP addresses

arping

I borrowed and simplified some code from the Scapy website to build a simple script that pings an IP address to find a MAC address. You will need to download the latest copy of scapy here: scapy and keep it in the same directory as your script if you are following along from home.

```
from scapy import srp,Ether,ARP,conf

conf.verb=0
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="10.0.1.1"),
timeout=2)

for snd, rcv in ans:
print rcv.sprintf(r"%Ether.src% %ARP.psrc%")
```

The output of the command is:

```
        sudo python arping.py
        00:00:00:00:00:01 10.0.1.1
```

Note that doing operations with scapy requires escalated privileges, so we must use sudo. I also change the real output to include a fake MAC address for the purpose of this article. Now that we have proven we can find out a MAC address from an IP address. We need clean up this code to take an IP address or subnet and return a MAC address and IP address pair.

arping function

```
        #!/usr/bin/env python

        from scapy import srp,Ether,ARP,conf

        def arping(iprange="10.0.1.0/24"):
          conf.verb=0
          ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
                        timeout=2)

          collection = []
          for snd, rcv in ans:
            result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
            collection.append(result)
          return collection

        #Print results
        values = arping()

        for ip,mac in values:
        print ip,mac
```

As you can see, we have a function that takes an IP address or network and returns a nested list of IP/MAC addresses. We are now ready for part two, which is creating a command line interface for our tool.

Solution Part 2: Creating a command line tool from our arping function

In this example, we combine ideas from the previous portions of the article to make a complete command line tool that solves our original specification.

arping CLI

```
   #!/usr/bin/env python

   import optparse
   from scapy import srp,Ether,ARP,conf

   def arping(iprange="10.0.1.0/24"):
     """Arping function takes IP Address or Network, returns nested mac/ip list"""

     conf.verb=0
     ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
```

```
            timeout=2)

    collection = []
    for snd, rcv in ans:
        result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
        collection.append(result)
    return collection

  def main():
    """Runs program and handles command line options"""

    p = optparse.OptionParser(description=' Finds MAC Address of IP address(es)',
                                  prog='pyarping',
                                  version='pyarping 0.1',
                                  usage='%prog [10.0.1.1 or 10.0.1.0/24]')

  options, arguments = p.parse_args()
  if len(arguments) == 1:
    values = arping(iprange=arguments)
    for ip, mac in values:
      print ip, mac
  else:
    p.print_help()

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

A few points of clarification on the script above might help our understanding of howoptparse works.

First, an instance of optparse.OptionParser() must be created and it takes the optional parameters shown above:

```
description, prog, version, and usage
```

These should be self explanatory for the most part, but I want to make sure that it is understood that while optparse is incredible, it is not magic. It has a clearly defined interface that enables rapid creation of command line tools.

Second, in the line:

```
options, arguments = p.parse_args()
```

this is where values for options and arguments get divided into distinct bits. In the above code, there will be exactly one argument that we expect, so I have specified that there must be only one argument value, and that value gets passed into our arping function.

```
          if len(arguments) == 1:
            values = arping(iprange=arguments)
```

To clarify even further, let's run the command and see how this works:

```
          sudo python arping.py 10.0.1.1

          10.0.1.1 00:00:00:00:00:01
```

In the above example, the argument is 10.0.1.1 and because there is only one argument,as I specified in the conditional statement, it is passed into the arping function. If we had options, they would be handed to options in the options, arguments = p.parse_args() idiom. Let's look at what happens when we break the intended use case of our command line tool and assign it two arguments:

```
          sudo python arping.py 10.0.1.1 10.0.1.3
          Usage: pyarping [10.0.1.1 or 10.0.1.0/24]

          Finds MAC Address or IP address(es)

          Options:
          --version   show program's version number and exit
          -h, --help  show this help message and exit
```

Because of how I structured the conditional statement for arguments, it will automatically bring up a help menu if the number of arguments is below one or greater than one:

```
          if len(arguments) == 1:
            values = arping(iprange=arguments)
            for ip, mac in values:
```

```
            print ip, mac
        else:
          p.print_help()
```

This is an important way to control how your tool works as you can use the number of arguments,or name of a specific option as the mechanism to control the flow of your command line tool. Since we have covered creating options in the very first Hello World example, let's add a couple of options to our command line tool by changing the main function slightly:

arping CLI main function

```
def main():
  """Runs program and handles command line options"""

  p = optparse.OptionParser(description='Finds MAC Address of IP address(es)',
                            prog='pyarping',
                            version='pyarping 0.1',
                            usage='%prog [10.0.1.1 or 10.0.1.0/24]')
  p.add_option('-m', '--mac', action ='store_true', help='returns only mac address')
  p.add_option('-v', '--verbose', action ='store_true', help='returns verbose output')

  options, arguments = p.parse_args()
  if len(arguments) == 1:
    values = arping(iprange=arguments)
    if options.mac:
      for ip, mac in values:
        print mac
    elif options.verbose:
      for ip, mac in values:
        print "IP: %s MAC: %s " % (ip, mac)
    else:
      for ip, mac in values:
        print ip, mac

  else:
    p.print_help()
```

The main thing to take away from the changes is that we have created conditional statements based on whether an option has been specified or not. Notice, unlike the Hello World command line tool, we are only using the options as a true/false signal to our tool. In the case of the --MAC option, if it is specified, our conditional elif statement only prints the MAC address.

Here is the output of the new options:

arping output

```
        sudo python arping2.py
        Password:
        Usage: pyarping [10.0.1.1 or 10.0.1.0/24]

        Finds MAC Address of IP address(es)

        Options:
        --version      show program's version number and exit
        -h, --help     show this help message and exit
        -m, --mac      returns only mac address
        -v, --verbose  returns verbose output
        [ngift@M-6][H:11184][J:0]> sudo python arping2.py 10.0.1.1
        10.0.1.1 00:00:00:00:00:01
        [ngift@M-6][H:11185][J:0]> sudo python arping2.py -m 10.0.1.1
        00:00:00:00:00:01
        [ngift@M-6][H:11186][J:0]> sudo python arping2.py -v 10.0.1.1
        IP: 10.0.1.1 MAC: 00:00:00:00:00:01
```

Advanced studies in creating command line tools

Here are a few more ideas for further study. They will be covered in great detail in the book I am co-authoring on Python for *NIX Systems Administration, which is due out in mid-2008.

Using subprocess module in a command line tool

The subprocess module is included with Python 2.4 or greater and is a unified interface for dealing with system calls and processes. You could easily replace the arping function above to use an arping tool available to your specific *NIX operating system. Here is a rough example of what that could look like:

Subprocess arping

```
  import subprocess
  import re
```

```
def arping(ipaddress="10.0.1.1"):
  """Arping function takes IP Address or Network, returns nested mac/ip list"""

  #Assuming use of arping on Red Hat Linux
  p = subprocess.Popen("/usr/sbin/arping -c 2 %s" % ipaddress, shell=True,
                       stdout=subprocess.PIPE)
  out = p.stdout.read()
  result = out.split()
  pattern = re.compile(":")
  for item in result:
    if re.search(pattern, item):
      print item
arping()
```

Here is the output of this function run standalone: `[root@localhost]~# python pyarp.py [00:16:CB:C3:B4:10]`

Notice the use of subprocess to grab the output of the arping command, and the use of a compiled regular expression match to the MAC address. Note, if you are on Python 2.3, you could use the popen module instead of subprocess, which became available in Python 2.4 or greater.

Using an Object Relational Mapper, such as SQLAlchemy or Storm with SQLite, in a command line tool

Another potential option for a command line tool is to use an ORM, or Object Relational Mapper, to store records for data generated by a command line tool. There are quite a few ORM's available for Python, but SQLAlchemy and Storm happen to be two of the most popular. I flipped a coin and decided to use Storm for this example:

Storm ORM arping

```
#!/usr/bin/env python
import optparse
from storm.locals import *
from scapy import srp,Ether,ARP,conf

class NetworkRecord(object):
  __storm_table__ = "networkrecord"
  id = Int(primary=True)
  ip = RawStr()
  mac = RawStr()
  hostname = RawStr()

def arping(iprange="10.0.1.0/24"):
  """Arping function takes IP Address or Network,
  returns nested mac/ip list"""

  conf.verb=0
  ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=iprange),
                timeout=2)
  collection = []
  for snd, rcv in ans:
    result = rcv.sprintf(r"%ARP.psrc% %Ether.src%").split()
    collection.append(result)
  return collection

def main():
  """Runs program and handles command line options"""

  p = optparse.OptionParser()
  p = optparse.OptionParser(description='Finds MACAddr of IP address(es)',
                            prog='pyarping',
                            version='pyarping 0.1',
                            usage= '%prog [10.0.1.1 or 10.0.1.0/24]')

  options, arguments = p.parse_args()
  if len(arguments) == 1:
    database = create_database("sqlite:")
    store = Store(database)
    store.execute("CREATE TABLE networkrecord "
          "(id INTEGER PRIMARY KEY, ip VARCHAR,\
          mac VARCHAR, hostname VARCHAR)")
    values = arping(iprange=arguments)
    machine = NetworkRecord()
    store.add(machine)
    #Creates Records
    for ip, mac in values:
      machine.mac = mac
      machine.ip = ip
    #Flushes to database
    store.flush()
    #Prints Record
    print "Record Number: %r" % machine.id
    print "MAC Address: %r" % machine.mac
    print "IP Address: %r" % machine.ip
  else:
```

```
        p.print_help()

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

The main thing to look at in this example is the creation of a class called NetworkRecord that maps to an "in memory" SQLite database. In the main function, I changed the output of our arping function to map to our record objects, flushed them to the database, then pulled them back out again to print the results. This is obviously not a production tool, but an instructive example of the steps involved in using an ORM with our tool.

Integrating config files into the CLI

Python INI config syntax

```
[AIX]
MAC: 00:00:00:00:02
IP: 10.0.1.2
Hostname: aix.example.com
[HPUX]
MAC: 00:00:00:00:03
IP: 10.0.1.3
Hostname: hpux.example.com
[SOLARIS]
MAC: 00:00:00:00:04
IP: 10.0.1.4
Hostname: solaris.example.com
[REDHAT]
MAC: 00:00:00:00:05
IP: 10.0.1.5
Hostname: redhat.example.com
[UBUNTU]
MAC: 00:00:00:00:06
IP: 10.0.1.6
Hostname: ubuntu.example.com
[OSX]
MAC: 00:00:00:00:07
IP: 10.0.1.7
Hostname: osx.example.com
```

Next we need to parse this using the ConfigParser module:

ConfigParser function

```
#!/usr/bin/env python
import ConfigParser

def readConfig(file="config.ini"):
  Config = ConfigParser.ConfigParser()
  Config.read(file)
  sections = Config.sections()
  for machine in sections:
    #uncomment line below to see how this config file is parsed
    #print Config.items(machine)
    macAddr = Config.items(machine)[0][1]
    print machine, macAddr
readConfig()
```

The output of this function is:

```
OSX 00:00:00:00:07
SOLARIS 00:00:00:00:04
AIX 00:00:00:00:02
REDHAT 00:00:00:00:05
UBUNTU 00:00:00:00:06
HPUX 00:00:00:00:03
```

I will leave the rest of the problem solving as an exercise to the reader. What I would do next is to integrate this config file into my script such that I could compare an inventory of machines kept in my config file against the actual inventory of MAC addresses that appear in the ARP cache. An IP address or a hostname is only so useful in tracking down a machine, but our potential tool could be a very useful way to track the hardware address of a machine on your network and determine if it was on the network.

Conclusion

We started with a few lines of code to create a very basic but powerful Hello World command line tool. We then moved on to create a sophisticated network tool using a Python Networking Library. Finally, we

proceeded to look at more advanced areas of study for the motivated reader. In the advanced research section, we looked at integrating the subprocess module, Object Relational Mappers,and finally configuration files.

Although it is apparently a secret, anyone with an IT background can create a command line tool with Python with a little effort. I hope this article has inspired you to go out on your own and create the next revolutionary command line tool.

Download

Description	Name	Size
Sample command line tools, scripts, and resources	python_command_line_tools_code.zip	411KB

Resources

Learn

Read Noah Gift's PyCon 2008 talk on command line tools.

See David Beazly's PyCon 2008 tutorial on using generators for system's programming.

Visit the official Scapy website to find product documentation.

The Scapy portability page provides installation information for various operating system

Scapy Presentation Article in Science and Engineering

The Style Guide for Python Code gives coding conventions for the Python code comprising the standard library in the main Python distribution.

Python For Bash Scripters shows you how to write a function in Python that you would normally write in Bash.

Visit The ConfigParser module for more configuration documentation.

See Config Parsing Examples on using ConfigParser.

The Subprocess Module lets you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

Python 2.3 Popen Module

Using Net-SNMP and IPython (Noah Gift, developerW orks, February 2008) shows you how to use Net-SNMP, Python, and the IPython shell to interactively explore and manage a network.

Unittests supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

Doctest Module searches for pieces of text that look like interactive Python sessions.

Python Is Easy Comic by XKCD.COM

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper.

Dig deeper into AIX and Unix on developerW orks

Overview

Technical library (tutorials and more)

Forums

Community

Downloads and products

Open source projects

Events

Storm is an object-relational mapper (ORM) for Python.

New to AIX and UNIX: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.

The developerWorks AIX and UNIX zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.

Stay current with developerWorks technical events and webcasts.

Technology bookstore Browse this site for books and other technical topics.

Discuss

Participate in the AIX and UNIX forums:

- AIX 5L -- technical forum
- AIX for Developers Forum
- Cluster Systems Management
- IBM Support Assistant
- Performance Tools -- technical
- Virtualization -- technical
- More AIX and UNIX forums