

```
C:\Windows\system32\cmd.exe
C:\PYTEST>python p-search.py -h
usage: Python Search [-h] [-v] -k KEYWORDS -t SRCHTARGET

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose          enables printing of additional program messages
  -k KEYWORDS, --keywords KEYWORDS
                        specify the file containing search words
  -t SRCHTARGET, --srchTarget SRCHTARGET
                        specify the target file to search

C:\PYTEST>
```

FIGURE 4.5

p-search execution using only the -h or help option.

Examining ValidateFileRead(theFile)

This function supports the ParseCommandLine and parser.parse_args() by validating any filenames provided by the user. The function first ensures that the file exists and then verifies that the file is readable. Failing either of these checks will result in the program aborting and will raise the appropriate exception.

```
def ValidateFileRead(theFile):
    # Validate the path is a valid
    if not os.path.exists(theFile):
        raise argparse.ArgumentTypeError('File does not exist') # Validate the path is
        readable

    if os.access(theFile, os.R_OK):
        return theFile
    else:
        raise argparse.ArgumentTypeError('File is not readable')
```

Examining the SearchWords function

The core of p-search is the SearchWords function which compares the keywords provided by the user with the contents of the target file specified. We will take a deep dive line by line of SearchWords.

```
def SearchWords():
    # Create an empty set of search words
```

As I mentioned in the design considerations, I decided to use a Python set object to hold the keywords to search. To do this I simply assign searchWords as a set object.

```
searchWords = set()
```

Next, I need to load the keywords from the file specified by the user in the command line arguments. I do this using the 'try, except, finally' method to ensure we trap or capture any failures.

```
# Attempt to open and read search words
```

```
try:
```

```
fileWords = open(gl_args.keyWords)
```

Once we successfully open the keywords file I process each line stripping the words from the line and adding it to the searchWords set.

```
for line in fileWords: searchWords.add(line.strip())
```

```
except:
```

Any exceptions encountered are logged and the program is aborted

```
log.error('Keyword File Failure:'+ gl_args.keyWords) sys.exit()
```

```
finally:
```

Once all the lines have been processed successfully, the file is closed.

```
fileWords.close()
```

Create Log Entry Words to Search For Next, I write entries into the forensic log to record the words that are to be included in the search.

```
log.info('Search Words')
```

```
log.info('Input File:'+gl_args.keyWords)
```

```
log.info(searchWords)
```

Attempt to open and read the target file # and directly load the file into a bytearray Now that I have the keywords to search for, I will read the target file provided and store the data into a Python byte array object.

```
try:
```

```
targetFile = open(gl_args.srchTarget,'rb')
```

```
baTarget = bytearray(targetFile.read())
```

```
except:
```

Any exceptions will be caught, logged and the program will exit.

```

log.error('Target File Failure:'+ gl_args.srchTarget) sys.exit()
finally:
targetFile.close()
I record the size of the target file on success.
sizeofTarget¼ len(baTarget)
# Post to log I also post this information to the forensic log file.
log.info('Target of Search:'+ gl_args.srchTarget) log.info('File
Size:'+str(sizeofTarget))

```

I'm going to modify baTarget by substituting zero for all not alpha characters. In order to ensure that I can display the original contents of the target file, I make a copy.

```

baTargetCopy ¼ baTarget
# Search Loop
# step one, replace all non characters with zero's

```

The first search step is to traverse the target and replace all non alpha characters with zero to make the search of words much easier and faster. The loop below starts at the beginning of the file and performs the replacement.

```

for i in range(0, sizeofTarget):
character ¼ chr(baTarget[i])
if not character.isalpha():

```

```

baTarget[i] ¼ 0
# step # 2 extract possible words from the bytearray # and then inspect the
search word list

```

Now that the bytearray baTarget only contains valid alpha characters and zeros I can count any consecutive alpha character sequences and if the sequence meets the size characteristics defined I can check the collected sequences of characters against the smaller keyword list

```

# create an empty list of not found items

```

To be thorough, I'm creating a list of notFoundcharacter sequences that meet our criteria of sequences of characters, but do not match the keywords specified by the user.

```
notFound¼ []
```

Now I begin the actual search. I simply check each byte and if it qualifies as a character I increase the count by one. If I encounter a zero, I stop the count and see if I have encountered enough consecutive characters to qualify as a possible word. If I don't then I set the count back to zero and continue the search. If on the other hand the count meets the criteria for the minimum and maximum number of characters I have a possible word.

```
cnt¼ 0
for i in range(0, sizeofTarget):
    character¼ chr(baTarget[i])
    if character.isalpha():
        cnt +¼ 1
    else:
        if (cnt >¼ MIN_WORD and cnt <¼ MAX_WORD):
            newWord¼ ""
```

If I have a sequence that meets the criteria I need to collect the characters that I have now skipped. One of the tricks is to not bother collecting or building possible strings until you meet the criteria. Once the criteria is met I backtrack and pull the consecutive letters together and store them in the variable newWord. I use the newWord variable to search the set of keywords.

```
for z in range(i-cnt, i):
    newWord¼ newWord þ chr(baTarget[z])
```

To search the set of keywords stored in the set searchWords, I simply write the one line test. This is the power of properly choosing the correct object type for storing the keywords.

```
if (newWord in searchWords):
```

If I get a hit in the searchWord list, I call the PrintBuffer function with the details of the hit which includes the word I found, the offset in the buffer, the saved unmodified buffer, where to start printing the result and where to stop.

```
PrintBuffer(newWord, i-cnt, baTargetCopy, iPREDECESSOR_ SIZE,
WINDOW_ SIZE)
```

```

print
else:
notFound.append(newWord)
cnt += 1
else:
cnt += 1

```

After I have processed the complete bytearray, I also print out the notFound list I created, this will give the investigator additional possible words to consider that were not in the keyword list.

PrintNotFound(notFound) # End of SearchWords Function

Examining the PrintBuffer function

Most of us that investigate cybercrime for a living use Hex editors or viewers quite often. The PrintBuffer function is a very simple Hex and ASCII viewer written in Python. It is simple to ensure that it will run on almost any platform (Windows, Linux, Mac, and even mobile devices). The trick is to not get fancy, but instead just provide the facts. Let us take a look how this can be accomplished with just a few lines of Python code.

```
def PrintBuffer(word, directOffset, buff, offset, hexSize):
```

The inputs to the function are as follows:

word: The alpha string that was identified

directOffset: The offset in the target file to the beginning of the alpha string buff: The actual buffer

offset: The offset where the Hex/ ASCII Window should start hexSize: What is the size in bytes that should be displayed

I start by printing the basic facts, the word and offset where the string was found.

```
print "Found: "+ word + " At Address: ", print "%08x " % (directOffset)
```

Next, I print the heading which prints the following heading. Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0 F ASCII PrintHeading() Next, I just need to print the Hex and ASCII values found in the buffer starting that the specified offset. The outer loop continues in 16 value chunks since each line

Results Presentation 107

will hold the next 16 values. Since we are printing this out in Hex this makes the most sense.

```
for i in range(offset, offset+hexSize, 16):
    for j in range(0,17):
```

The inside loop then needs to print each of the next 16 bytes in hex. I start by printing the offset value in hex leaving the appropriate amount of same. Note also, that I use the syntax %08x which allows for an 8 digit Hex address zero filled. This should be plenty for this demonstration application.

```
if (j % 4 == 0):
    print "%08x " % i,
else:
    byteValue = buff[i+j]
    print "%02x " % byteValue,
    print " ",
```

Next we repeat this loop but this time we print the ASCII character representation instead of the Hex values.

```
for j in range (0,16):
    byteValue = buff[i+j]
```

To avoid any special characters we only print the ASCII printable set and then replace the other values with a period.

```
if (byteValue > 0x20 and byteValue < 0x7f):
    print "%c" % byteValue,
else:
    print '.',
print
# End Print Buffer
```

RESULTS PRESENTATION

Now that the walk-through has been completed and I have gone through a deep dive into the code, let us take a look at the results. [Figure 4.6](#) is a snapshot of the directory that contains all the requisite files. This includes the two Python program files p-search.py and _p-search.py. It includes the narc.txt file that contains the search words. Capture.raw is a file containing binary data that we wish to search. pSearchLog.log is the forensic log file output from p-search and finally results.txt is a file containing the standard output results that were piped to a file for easy viewing.

Figure 4.7 lists the file containing the drug-related keywords that we intend to search for.

Figure 4.8 is the direct output of the program starting with the command line entry which specified that the results be piped to the file results.txt. Also included is the results.txt file. You can see that we identified three hits in the target file, they include sugar, tornado, and moonrocks. You also see the Hex/ASCII printout generated by the program along with the unidentified words at the bottom.



```
C:\Windows\system32\cmd.exe

C:\PYTEST>dir
Volume in drive C is HP
Volume Serial Number is 5EC5-6025

Directory of C:\PYTEST

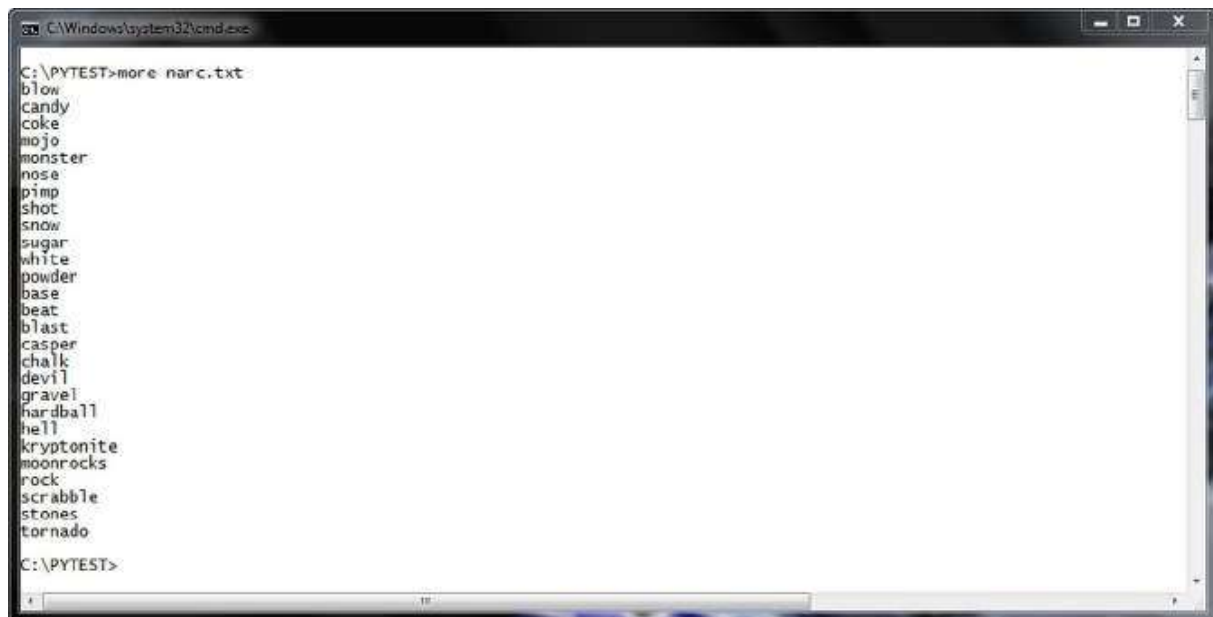
08/27/2013  09:18 PM    <DIR>          .
08/27/2013  09:18 PM    <DIR>          ..
08/27/2013  09:18 AM             77,650 capture.raw
08/26/2013  07:48 PM              208 narc.txt
08/27/2013  09:15 PM             1,365 p-search.py
08/27/2013  09:16 PM              650 pSearchLog.log
08/27/2013  09:16 PM             5,383 results.txt
08/27/2013  09:04 PM             6,966 _psearch.py
               6 File(s)          92,222 bytes
               2 Dir(s) 172,577,951,744 bytes free

C:\PYTEST>
```

FIGURE

4.6

Execution test directory for p-search.



```
C:\Windows\system32\cmd.exe
C:\PYTEST>more narc.txt
blow
candy
coke
mojo
monster
nose
pimp
shot
snow
sugar
white
powder
base
beat
blast
casper
chalk
devil
gravel
hardball
hell
kryptonite
moonrocks
rock
scrabble
stones
tornado
C:\PYTEST>
```

FIGURE 4.7
Keyword file dump.

Finally in [Figure 4.9](#), you see the contents of the log file generated by p-search that includes the start message, the search word file, and contents. Also included is the name and size of the target file capture.raw. Finally, you see the elapsed time for execution in seconds and the program termination message.

To round out this example and demonstrate the portability, I have included two additional screenshots running the unmodified p-search program on both a Linux machine ([Figure 4.10](#)) and iMac ([Figure 4.11](#)).


```

C:\Windows\system32\cmd.exe
C:\PYTEST>python p-search.py -k narc.txt -t capture.raw > results.txt

C:\PYTEST>more results.txt
Found: sugar At Address: 00000406
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII
-----
000003eb 00 77 48 00 6a 75 6c 69 6f 00 79 61 68 6f 6f 00 .wH.julio.yahoo.
000003fb 63 6f 6d 00 00 7a 00 6f 66 00 73 75 67 61 72 00 .com.z.of.sugar.
0000040b 00 00 00 00 00 00 00 00 00 66 69 00 00 00 00 00 .f.
0000041b 00 00 00 00 00 00 00 00 00 6d 66 00 00 00 00 00 .mF.
0000042b 00 00 00 00 00 00 00 6b 00 51 00 00 00 00 00 00 .k.Q.
0000043b 00 00 00 00 00 00 00 00 00 00 00 55 55 4e 00 .UUN.
0000044b 00 00 00 00 42 00 00 00 00 00 00 00 00 00 00 00 .B.
0000045b 00 00 00 00 00 00 00 00 00 00 68 73 68 6d 00 6e .hsh.m.n

Found: tornado At Address: 000007ad
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII
-----
00000794 79 00 61 6f 6c 00 61 6f 6d 00 6e 65 65 64 00 74 ny.aol.com.need.t
000007a4 6f 00 73 65 65 00 61 00 74 6f 73 6e 61 64 6f 00 to.see.a.tornado.
000007b4 00 00 00 00 00 00 00 00 00 00 52 00 00 68 68 .R.hh
000007c4 00 4e 00 51 4b 44 00 00 00 00 00 00 00 00 00 00 h.N.QKD.
000007d4 00 00 66 00 00 00 00 00 00 00 73 6b 75 00 00 48 .f.sku.H
000007e4 66 00 6f 00 76 00 00 00 64 59 6b 00 00 00 00 Hf.o.v.dYk.
000007f4 00 65 00 00 00 74 6c 00 00 00 00 58 6a 74 00 .e.tl.Xjt.
00000804 00 00 63 00 00 00 00 56 00 00 00 5a 00 00 00 .c.V.Z.

Found: moonrocks At Address: 00000e10
Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII
-----
00000df9 63 6f 6d 00 68 61 76 65 00 79 6f 75 00 73 65 65 .com.have.you.see
00000e09 6e 00 61 6e 79 00 6d 6f 6f 6e 72 6f 63 6b 73 00 en.any.moonrocks.
00000e19 00 00 00 00 00 00 00 00 00 00 4f 00 00 00 4f 00 .o.O.
00000e29 00 00 63 00 00 00 00 00 5a 00 00 00 57 00 00 00 .c.Z.W.
00000e39 00 00 00 00 00 00 00 00 00 59 00 00 00 00 00 .Y.
00000e49 00 00 00 00 00 58 00 00 00 00 00 53 6d 00 00 .X.Sm.
00000e59 00 00 4e 00 00 00 00 00 00 00 6e 00 00 00 00 .N.n.
00000e69 00 72 5a 00 00 00 45 00 00 00 00 00 00 00 00 .rZ.E.

----- Possible Words Identified, but not found -----
JFIF, Photoshop, JFIF, File, written, Adobe, Photoshop, Adobe, GwGw, AqAq, GwGw, julio, yahoo, ugar
, hshh, johnny, need, ornado
, QJdF, gmail, have, seen, oonrocks
, vINC, lonV, kmnn, XICC, PROFILE, HLino, mntrRGB, acspMSFT, sRGB, cppt, desc, lwtpt, bkpt, rXYZ, gXYZ,
rTRC, gTRC, bTRC, text, Copyright, Hewlett, Packard, Company, desc, sRGB, sRGB, desc, http, http, desc,
pace, sRGB, desc, Reference, Viewing, Condition, Reference, Viewing, Condition, view, meas, curv, File, wri
qYZOD, CIYX, ouno, anDq, uvxXCW, RQhX, FpId, wavt, zGdP, KMEDH, MGIB, CuqN, MpVP, yWLj, yHJR, LJUOf, h
dhmHg, with, GLnG, UjmQO, imZd, tchS, tidz, wIKca, lzTe, qFWX, khRP, REAZ, sEXx, iRWg, pGYk, fwqz, AJS
qRMP, YDxf, qyij, ixmCm, FhmIR, zwNFX, CZyq, lyCk, aeVG, IsrP, iEjPI, wTkt, JIRm, cgshH, NqWd, qImO, O
CMFl, aoGiJ, OzKqPd, kFKia, lRyNq, CduX, jtvC, UVRz, ARzT, iSEm, BRHW, ROREN, uiEtQ, rYIq, rvPo, nVino,
a, oJCj, RpFd, mkpXO, eUHG, yTiV, PZtp, zIyso, ipLId, SuXo, FvaD, KqVT, kuBS, UPHzkm, HMAKF, IJJA, aikz
ty, USytZ, hLeV, ZYtk, zhHJ, OgCm, xvFq, nKny, OdPfg, UgHSF, AQIR, iIPA, OuLZV, YSuR, xFvc, hRdd, mjin,

C:\PYTEST>

```

FIGURE 4.8
p-search sample execution.

```

C:\Windows\system32\cmd.exe
C:\PYTEST>more pSearchLog.log
2013-08-27 21:16:30,755 p-search started
2013-08-27 21:16:30,756 Search Words
2013-08-27 21:16:30,756 Input File: narc.txt
2013-08-27 21:16:30,756 set(['stones', 'shot', 'scrabble', 'powder', 'casper', 'hell', 'sugar', 'mojo', 'blow', 'devil',
ardball', 'moonrocks', 'monster', 'beat', 'coke', 'base', 'blast', 'pimp', 'candy', 'chalk', 'nose', 'rock'])
2013-08-27 21:16:30,756 Target of Search: capture.raw
2013-08-27 21:16:30,757 File Size: 77650
2013-08-27 21:16:30,973 Elapsed Time: 0.218999862671 seconds
2013-08-27 21:16:30,974
2013-08-27 21:16:30,974 Program Terminated Normally

C:\PYTEST>

```

FIGURE 4.9
Log file contents post execution.
INDEXING

Indexing provides the investigator with a different look at the data and potential

evidence contained within a file, disk image, memory snapshot, or network trace. Instead of determining what keywords to look for, we want p-search to provide us

```

chert@PythonForensics: ~/Desktop/Python Samples/p-search
chert@PythonForensics:~/Desktop/Python Samples/p-search$ python p-search.py -k narc.txt -t capture.raw
Found: sugar At Address: 00000406
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
000003eb    00 77 48 00 6a 75 6c 69 6f 00 79 61 68 6f 6f 00      . . w H . j u l i o . y a h o o .
000003fb    63 6f 6d 00 00 7a 00 6f 66 00 73 75 67 61 72 00      . c o m . . z . o f . s u g a r .
0000040b    00 00 00 00 00 00 00 00 66 69 00 00 00 00 00      . . . . . f i . . . .
0000041b    00 00 00 00 00 00 00 00 6d 66 00 00 00 00 00      . . . . . m f . . . .
0000042b    00 00 00 00 00 00 00 6b 00 51 00 00 00 00 00      . . . . . k . Q . . . .
0000043b    00 00 00 00 00 00 00 00 00 00 00 55 53 4e 00      . . . . . U U N .
0000044b    00 00 00 00 42 00 00 00 00 00 00 00 00 00 00      . . . . . B . . . . .
0000045b    00 00 00 00 00 00 00 00 00 00 68 73 68 6d 00 6e      . . . . . h s h m . n

Found: tornado At Address: 000007ad
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
00000794    79 00 61 6f 6c 00 63 6f 6d 00 6e 65 65 64 00 74      n y . a o l . c o m . n e e d . t
000007a4    6f 00 73 65 65 00 61 00 74 6f 72 6e 61 64 6f 00      t o . s e e . a . t o r n a d o .
000007b4    00 00 00 00 00 00 00 00 00 00 00 52 00 00 68 68      . . . . . R . . h h
000007c4    00 4e 00 51 4b 44 00 00 00 00 00 00 00 00 00      h . N . Q K D . . . .
000007d4    00 00 66 00 00 00 00 00 00 00 73 6b 75 00 00 48      . . f . . . . s k u . . H
000007e4    66 00 6f 00 70 00 00 00 00 64 59 6b 00 00 00 00      H f . o . v . . d y k . .
000007f4    00 65 00 00 00 74 6c 00 00 00 00 00 58 6a 74 00      . . e . . t l . . . . X j t .
00000804    00 00 63 00 00 00 00 56 00 00 00 00 5a 00 00      . . . c . . . . V . . . . Z .

Found: moonrocks At Address: 00000e10
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
00000df9    63 6f 6d 00 68 61 76 65 00 79 6f 75 00 73 65 65      . c o n . h a v e . y o u . s e e
00000e09    6e 00 61 6e 79 00 6d 6f 6f 6e 72 6f 63 6b 73 00      e n . a n y . m o o n r o c k s .
00000e19    00 00 00 00 00 00 00 00 00 00 4f 00 00 00 4f 00      . . . . . O . . . . O .
00000e29    00 00 63 00 00 00 00 00 5a 00 00 00 57 00 00 00      . . . c . . . . Z . . . . W .
00000e39    00 00 00 00 00 00 00 00 00 00 59 00 00 00 00 00      . . . . . Y . . . .
00000e49    00 00 00 00 00 00 58 00 00 00 00 00 53 6d 00      . . . . . X . . . . S m
00000e59    00 00 4e 00 00 00 00 00 00 00 6e 00 00 00 00 00      . . . N . . . . . n . . .
00000e69    00 72 5a 00 00 00 00 45 00 00 00 00 00 00 00 00      . . r Z . . . . E . . . .

----- Possible Words Identified, but not found -----
JFIF, Photoshop, JFIF, File, written, Adobe, Photoshop, Adobe, CWGw, AQaq, CWGw, julio, yahoo, ugar, hshm, john
ny, need, ornado, QJdF, gmall, have, seen, oonrocks, vLMC, lonV, kmnn, XICC, PROFILE, HLlno, mntrRGB, acspMSFT,
sRGB, cppt, desc, lwtpt, bkpt, rXYZ, gXYZ, bXYZ, dnnd, pdmdd, vued, view, luni, meas, tech, rTRC, gTRC, bTR
C, text, Copyright, Hewlett, Packard, Company, desc, sRGB, sRGB, desc, http, http, desc, Default, colour, space
, sRGB, Default, colour, space, sRGB, desc, Reference, Viewing, Condition, Reference, Viewing, Condition, view,
meas, curv, File, written, Adobe, Photoshop, Adobe, EelJ, hMyp, vacpj, qYZUD, CIYX, ouno, anDq, uvXXCw, RQhX,
FpId, wavt, zGdP, KMEDH, MGIB, CuqN, HpVP, yWLJ, yHJR, LJUqF, hOBX, PRTI, Mcfs, sSpaHE, vYZo, oRoI, XShx, qSN
I, dhnhg, with, GLnG, UJmQO, Imzd, tchS, tldz, wIKca, lzTe, qFWx, KhRP, REAZ, sEXx, lRWg, pGYk, fwqz, AJSI,
FORzq, xUEw, yFcqCw, UNwk, Tdtd, lRMOb, vLyn, qRMP, YDxf, qyij, lxmCm, FhnIR, zwNFX, CZyq, lyck, aeVG, IsrP,
lEjPI, wTkt, JIRm, cgsHh, NqWD, qJmD, OoZk, rChS, sKqk, qJAJ, Ambj, zCIyn, BcKu, Wmku, CMfl, aoGiJ, OzKqPd, K
fkLa, lRyHq, CduX, jtvc, UVRz, ARzT, iSEm, BRHw, sKqk, uIEtQ, rVIq, rVPO, nVINQ, ptQrm, tFes, Fnaq, Cifs, tp
aq, Olkw, zrcW, ZxuCo, oJcJ, RpfD, mkpXO, eUJG, yTiV, PZtp, zIYso, ipTLid, Suxo, FvaD, KqVT, kuBS, UPhzkm, HM
AKF, IJJA, aikz, xWNT, zIMP, ilwd, gJUB, nKGM, hAIq, LbJR, XFwtY, USytZ, hLeV, ZYtk, zhHJ, OgCm, xvFq, nKny,
OdPfg, UghSf, AQIR, tiPA, OuLZV, YSuR, xFVC, hrdd, nJlN, PCBjU, eMvRs, hSHP, ilYq,

```

FIGURE 4.10
Execution of p-search running on Ubuntu Linux 12.04 LTS.

with a list of all the words that meet the minimum definition of a string, but do not match any of the defined key words. However, the current approach is lacking two key pieces of data. The first is simple, I need to include the offset of where each string was found in the target file. The second problem is a bigger challenge, that being to only print words that are likely to be legitimate words. Finally, it would be nice if the words were provided in a sorted list.

To handle the more difficult problem of identifying strings that have a high

probability of being words, I have devised an approach that works exceptionally well and is fast. I depict the basic concept in [Figure 4.12](#). I first process known words with an algorithm that produces a numeric weight based on the internal characteristics of each word.

Once a numeric weight is calculated for the word, the weights are sorted and any duplicates are removed. The key here is to process a large list of words. For this example, I used a dictionary containing over 600,000 words. This produces a matrix

```
janet-hosmers-imac:p-search janet$ python p-search.py -k narc.txt -t capture.raw -m matrix.txt
Found: sugar At Address: 00000406
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
000003eb    00 77 48 00 6a 75 6c 69 6f 00 79 61 68 6f 6f 00      . . u H . j u l i o . y a h o o .
000003fb    63 6f 6d 00 00 7a 00 6f 66 00 73 75 67 61 72 00      . c o m . . z . o f . s u g a r .
0000040b    00 00 00 00 00 00 00 00 00 66 69 00 00 00 00 00      . . . . . . . . f i . . . .
0000041b    00 00 00 00 00 00 00 00 00 6d 66 00 00 00 00 00      . . . . . . . . m f . . . .
0000042b    00 00 00 00 00 00 00 6b 00 51 00 00 00 00 00 00      . . . . . . . . k . Q . . . .
0000043b    00 00 00 00 00 00 00 00 00 00 00 00 55 55 4e 00      . . . . . . . . . . U U N .
0000044b    00 00 00 00 42 00 00 00 00 00 00 00 00 00 00 00      . . . . . B . . . . . . . .
0000045b    00 00 00 00 00 00 00 00 00 00 66 73 68 6d 00 6e      . . . . . . . . . . h s h m . n

Found: tornado At Address: 000007ad
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
0000079f    79 00 61 6f 6c 00 63 6f 6d 00 6e 65 65 6f 00 7f      n y . a o l . c o m . n e e d . t
000007af    6f 00 73 65 65 00 61 00 7f 6f 72 6e 61 6f 6f 00      t o . s e e . a . t o r n a d o .
000007bf    00 00 00 00 00 00 00 00 00 00 00 52 00 00 68 68      . . . . . . . . . . R . . h h
000007cf    00 4e 00 51 4b 44 00 00 00 00 00 00 00 00 00 00      h . N . Q K D . . . . . . . .
000007df    00 00 66 00 00 00 00 00 00 00 00 73 6b 75 00 48      . . . f . . . . . s k u . . H
000007ef    66 00 6f 00 76 00 00 00 00 6f 59 6b 00 00 00 00      H f . o . v . . . . d v k . . .
000007ff    00 65 00 00 00 74 6c 00 00 00 00 00 58 6a 74 00      . . e . . . t l . . . . . X j t .
0000080f    00 00 63 00 00 00 00 00 56 00 00 00 00 5a 00 00      . . . c . . . . . V . . . . Z . .

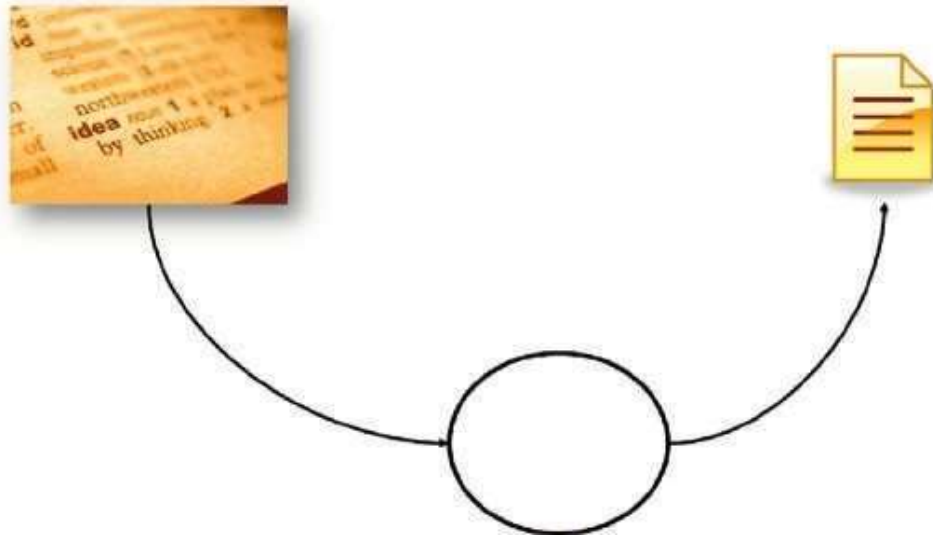
Found: moonrocks At Address: 00000e10
Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F      ASCII
-----
00000df9    63 6f 6d 00 68 61 76 65 00 79 6f 75 00 73 65 65      . c o m . h a v e . y o u . s e e
00000e09    6e 00 61 6e 79 00 6d 6f 6f 6e 72 6f 63 6b 73 00      e n . a n y . m o o n r o c k s .
00000e19    00 00 00 00 00 00 00 00 00 00 4f 00 00 00 4f 00      . . . . . . . . . . 0 . . . 0 .
00000e29    00 00 63 00 00 00 00 00 5a 00 00 00 57 00 00 00      . . . c . . . . . Z . . . H . .
00000e39    00 00 00 00 00 00 00 00 00 00 59 00 00 00 00 00      . . . . . . . . . . Y . . . .
00000e49    00 00 00 00 00 58 00 00 00 00 00 00 53 6d 00 00      . . . . . . . . X . . . . S m .
00000e59    00 00 4e 00 00 00 00 00 00 00 6e 00 00 00 00 00      . . . N . . . . . . . . n . . .
00000e69    00 72 5a 00 00 00 45 00 00 00 00 00 00 00 00 00      . . r Z . . . . . E . . . . .
```

Index of All Words

```
[ 'adobe', 470 ]
[ 'adobe', 494 ]
[ 'adobe', 10042 ]
[ 'adobe', 10066 ]
[ 'aogij', 46360 ]
[ 'colour', 7624 ]
[ 'colour', 7681 ]
[ 'company', 7257 ]
[ 'condition', 7752 ]
[ 'condition', 7887 ]
[ 'copyright', 7222 ]
[ 'default', 7612 ]
[ 'default', 7669 ]
[ 'gmail', 3572 ]
[ 'hewlett', 7241 ]
```

FIGURE 4.11

Execution of p-search running on iMac.



Unique sorted weights

Words

Word Word weights_{weighng}
algorithm

FIGURE 4.12

Diagram of word weighting approach.

r u n n i n g
c u n n i n g
g u n n i n g
d u n n i n g

Used in weighng calculaon FIGURE 4.13

Weighted characteristics illustration.

of weights that can be used to compare strings found in raw data (using the same weighting algorithm used to create the original matrix) to the matrix. If the calculated weight matches one found in the matrix, I consider this a possible word, if not the string is discarded. [Figure 4.13](#) provides an illustration of how this works. Each of the core letters contained in the word (ignoring the leading character) are used in the weighted calculation.

Since many words, at least in the English language, are derived from other words, this eliminates the need for having an all-encompassing dictionary of every possible word. In addition, if words are misspelled or newly created or derived, it is likely the weighting system vs. a direct match will provide a more liberal interpretation of a word being probable.

The importance of approximate searches and indexes can be vital. In the Casey Anthony case, for example, investigators missed probative Internet search history data because they looked for various search terms which *they* spelled correctly. She, however, had misspelled several search terms; while Google corrected the spelling during the search, it did not fix the misspelling in the Internet search history.

CODING isWordProbable

In order to create a method isWordProbable(word), I created a class named Matrix; let us take our typical deep dive into this new class.

class class_Matrix:

I start out by declaring a new set object named the weighted matrix.

weightedMatrix = set()

Coding isWordProbable 113

When the class is instantiated, the object will load the values contained in the matrix into the weighted matrix set. To handle this I simply added a new command line argument -m or --theMatrix to allow the user to specify a matrix file. I take the same precautions as you have seen before to trap any errors that might occur during the file handling operations.

```
def __init__(self):
```

```
try:
```

```
fileTheMatrix = open(gl_args.theMatrix,'rb') for line in fileTheMatrix:
```

```
value = line.strip()
```

```
self.weightedMatrix.add(int(value,16)) except:
```

```
log.error('Matrix File Error:'+ gl_args.theMatrix) sys.exit()
```

```
finally:
```

```
fileTheMatrix.close()
```

```
return
```

Next I define the method isWordProbable which will calculate the weight of the

passed string and perform a lookup in the weightedMatrix to determine if we have a match. Note I first check to see if theWord passed is of minimum size.

```
def isWordProbable(self, theWord):
```

```
    if (len(theWord) < MIN_WORD):
```

```
        return False
```

```
    else:
```

```
        BASE = 96
```

```
        wordWeight = 0
```

This is the core weight calculation and produces an unsigned long integer value name wordWeight.

```
        for i in range(4,0,-1):
```

```
            charValue = (ord(theWord[i]) - BASE) shiftValue = (i-1)*8
```

```
            charWeight = charValue << shiftValue wordWeight = (wordWeight |  
            charWeight)
```

Once I have calculated the word weight I just check to see if the weight exists in the weighted matrix.

```
        if ( wordWeight in self.weightedMatrix):
```

```
            return True
```

```
        else:
```

```
            return False
```

To properly incorporate this into the program the following code is all that is needed.

The initialization is simple, I create an object named wordCheck (which executes the initialization code of the Matrix to load in the weighed values. I also, create a list named indexOfWords where I will store the probable words that we find.

```
wordCheck = class_Matrix() indexOfWords = []
```

```
...
```

```
...
```

Next, I embed the following code within the existing search loop to evaluate each string as a probable word. If wordCheck returns True, then I add the word to the indexOfWorks list. Note that this list has two elements per entry: 1) the string newWord and 2) the offset from the beginning of the target object where it was found.

```
if wordCheck.isWordProbable(newWord):
    indexOfWorks.append([newWord, i-cnt])
```

Finally, I added a method to print the indexOfWorks list at the end of the search that includes both the word and offset. I also sorted the list prior to print in order to make it alphabetical.

```
def PrintAllWordsFound(wordList):
    print "Index of All Words"
    print "_____ "
    wordList.sort()
    for entry in nfList:
        print entry
        print "_____ "
    print
    return
```

From an execution perspective you can see the results of the combined search and index program in [Figure 4.14](#).

P-SEARCH COMPLETE CODE LISTINGS p-search.py

```
#
# p-search : Python Word Search
# Author: C. Hosmer
# August 2013
# Version 1.0
#
# Simple p-search Python program
#
```

Found: moonrocks At Address: 00000e10																	
Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00000df9	63	6f	6d	00	68	61	76	65	00	79	6f	75	00	73	65	65	. c o m . h a v e . y o u . s e e
00000e09	6e	00	61	6e	79	00	6d	6f	6f	6e	72	6f	63	6b	73	00	e n . a n y . m o o n r o c k s .
00000e19	00	00	00	00	00	00	00	00	00	00	4f	00	00	00	4f	00 0 0 .
00000e29	00	00	63	00	00	00	00	00	00	5a	00	00	00	57	00	00	. . . c Z . . . W . . .
00000e39	00	00	00	00	00	00	00	00	00	00	59	00	00	00	00	00 Y
00000e49	00	00	00	00	00	00	58	00	00	00	00	00	00	53	6d	00 X S m .
00000e59	00	00	4e	00	00	00	00	00	00	00	00	6e	00	00	00	00	. . . N n
00000e69	00	72	5a	00	00	00	00	45	00	00	00	00	00	00	00	00	. . r Z E
Index of All Words																	
['adobe', 470]																	
['adobe', 494]																	
['adobe', 10042]																	
['adobe', 10066]																	
['aogij', 46360]																	
['colour', 7624]																	
['colour', 7681]																	
['company', 7257]																	
['condition', 7752]																	
['condition', 7807]																	
['copyright', 7222]																	
['default', 7612]																	
['default', 7669]																	
['gmail', 3572]																	
['hewlett', 7241]																	
['hlino', 6881]																	
['johnny', 1936]																	
['julio', 1008]																	
['moonrocks', 3600]																	
['packard', 7249]																	
['photoshop', 24]																	
['photoshop', 476]																	
['photoshop', 10048]																	
['profile', 6868]																	
['qyzod', 11694]																	
['reference', 7734]																	
['reference', 7789]																	
['roren', 51038]																	
['space', 7631]																	
['space', 7688]																	
['sspahe', 22967]																	
['sugar', 1030]																	
['tornado', 1965]																	
['viewing', 7744]																	
['viewing', 7799]																	
['written', 459]																	
['written', 10031]																	
['yahoo', 1014]																	

FIGURE 4.14
p-search execution with indexing capability.

```
# Read in a list of search words
# Read a binary file into a bytearray
# Search the bytearray for occurrences of any specified search words # Print a
# HEX/ ASCII display localizing the matching words # Print out a list of possible
# words identified that didn't match #
# Definition of a word. a word for this example is an uninterrupted
sequence of
# 4 to 12 alpha characters
#
```

```
import logging
import time
import _psearch
```



```

if __name__ == '__main__':
PSEARCH_VERSION = '1.0'
# Turn on Logging

logging.basicConfig(filename='pSearchLog.log',level=logging.DEBUG,
format='%%(asctime)s %(message)s')
# Process the Command Line Arguments _psearch.ParseCommandLine()
log = logging.getLogger('main._psearch') log.info("p-search started")
# Record the Starting Time startTime = time.time() # Perform Keyword Search
_psearch.SearchWords()

# Record the Ending Time
endTime = time.time()
duration = endTime - startTime

logging.info('Elapsed Time:' + str(duration) + 'seconds') logging.info('')
logging.info('Program Terminated Normally')
_p-search.py

#
# psearch support functions, where all the real work gets done #
# Display Message() ParseCommandLine() # ValidateFileRead() # Matrix (class)
#
import argparse

import os
import logging ValidateFileWrite()

# Python Standard Library Parser for command-line options, arguments

# Standard Library OS functions
# Standard Library Logging
functions

log = logging.getLogger('main._psearch')

#Constants
MIN_WORD = 5
MAX_WORD = 15

```

```

PREDECESSOR_SIZE ¼ 32 WINDOW_SIZE¼ 128
# Minimum word size in bytes

# Maximum word size in bytes # Values to print before match found # Total
values to dump when match found

# Name: ParseCommand() Function #
# Desc: Process and Validate the command line arguments # use Python
Standard Library module argparse #
# Input: none
#
# Actions:
# Uses the standard library argparse to process the command

line
#
def ParseCommandLine():

parser¼ argparse.ArgumentParser('Python Search')

parser.add_argument('- v','--verbose', help¼"enables printing of additional
program messages", action¼'store_true')
parser.add_argument('- k','--keyWords', type¼ ValidateFileRead, required¼True,
help¼"specify the file containing search words") parser.add_argument('- t','--
srchTarget', type¼ ValidateFileRead, required¼True, help¼"specify the target file
to search") parser.add_argument('- m','--theMatrix', type¼ ValidateFileRead,
required¼True, help¼"specify the weighted matrix file")

global gl_args gl_args ¼ parser.parse_args() DisplayMessage("Command line
processed: Successfully") return

# End Parse Command Line

#
# Name: ValidateFileRead Function
#
# Desc: Function that will validate that a file exists and is readable #
# Input: A file name with full path
#

```

[illegible]

```

# Name SearchWords()
#
# Uses command line arguments
#
# Searches the target file for keywords #

def SearchWords():
# Create an empty set of search words searchWords = set()

# Attempt to open and read search words try:
fileWords = open(gl_args.keyWords)
for line in fileWords:
searchWords.add(line.strip())
except:
log.error('Keyword File Failure:'+ gl_args.keyWords)
sys.exit()
finally:
fileWords.close()
# Create Log Entry Words to Search For
log.info('Search Words')
log.info('Input File:'+gl_args.keyWords) log.info(searchWords)

# Attempt to open and read the target file # and directly load into a bytearray

try : targetFile = open(gl_args.srchTarget, 'r b') baTarget =
bytearray(targetFile.read())

except :
log.error('Target File Failure:'+ gl_args.srchTarget) sys.exit()

finally:
targetFile.close()

sizeOfTarget = len(baTarget) # Post to log

log.info('Target of Search:'+ gl_args.srchTarget) log.info('File
Size:'+str(sizeOfTarget))

baTargetCopy = baTarget wordCheck = class_Matrix()

```

```

# Search Loop
# step one, replace all non characters with zero's

for i in range (0, sizeofTarget): character  $\frac{1}{4}$  chr(baTarget[i]) if not
character.isalpha():

baTarget[i]  $\frac{1}{4}$  0

# step # 2 extract possible words from the bytearray # and then inspect the
search word list
# create an empty list of probable not found items

indexOfWords  $\frac{1}{4}$  []

cnt  $\frac{1}{4}$  0
for i in range(0, sizeofTarget):
character  $\frac{1}{4}$  chr(baTarget[i])
if character.isalpha():
cnt  $\frac{1}{4}$  1
else:
if (cnt  $\frac{1}{4}$  MIN_WORD and cnt  $\frac{1}{4}$  MAX_WORD): newWord  $\frac{1}{4}$  ""
for z in range(i-cnt, i):
newWord  $\frac{1}{4}$  newWord + chr(baTarget[z]) newWord  $\frac{1}{4}$  newWord.lower()
if (newWord in searchWords):
PrintBuffer(newWord, i-cnt, baTargetCopy,
i-PREDECESSOR_SIZE, WINDOW_SIZE)
indexOfWords.append([newWord, i-cnt])
cnt  $\frac{1}{4}$  0
print
else:
if wordCheck.isWordProbable(newWord): indexOfWords.append([newWord, i-
cnt])
cnt  $\frac{1}{4}$  0
else:
cnt  $\frac{1}{4}$  0

PrintAllWordsFound(indexOfWords) return
# End of SearchWords Function

```

```
#
# Print Hexidecimal / ASCII Page Heading
#
def PrintHeading():

print ("Offset 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F ASCII")
print("_____")
```

```
return
# End PrintHeading
```

```
#
# Print Buffer
#
# Prints Buffer contents for words that are discovered
# parameters
# 1) Word found
# 2) Direct Offset to beginning of the word
# 3 buff The bytearray holding the target
# 4) offset starting position in the buffer for printing
# 5) hexSize, size of hex display windows to print
#
def PrintBuffer(word, directOffset, buff, offset, hexSize):

print "Found: "+ word + " At Address: ",
print "%08x"%(directOffset)
PrintHeading()
for i in range(offset, offset+hexSize, 16):
for j in range(0,16):
if (j % 4 == 0):

print "%08x " % i,
else:
byteValue = buff[i+j]
print "%02x " % byteValue,
print " ",
for j in range (0,16):
byteValue = buff[i+j]
```

```

if (byteValue >¼ 0x20 and byteValue <¼ 0x7f):
print "%c" % byteValue,
else:
print'. ',
print
return

# End Print Buffer

#
# PrintAllWordsFound #

def PrintAllWordsFound(wordList):
print "Index of All Words" print
"_____ " wordList.sort() for entry in
wordList: print entry
print "_____ " print
return
#End PrintAllWordsFound

#
# Class Matrix
#
# init method, loads the matrix into the set # weightedMatrix
#
# isWordProbable method
# 1) Calculates the weight of the provided word # 2) Verifies the minimum
length
# 3) Calculates the weight for the word # 4) Tests the word for existence in the
matrix # 5) Returns true or false
#
class class_Matrix:

weightedMatrix ¼ set()

def __init__(self): try:
fileTheMatrix ¼ open(gl_args.theMatrix, 'r b') for line in fileTheMatrix:

value¼ line.strip()

```

```

self.weightedMatrix.add(int(value,16)) except:
log.error('Matrix File Error:'+ gl_args.theMatrix) sys.exit()
finally:
fileTheMatrix.close()
return def isWordProbable(self, theWord):

if (len(theWord) < MIN_WORD): return False
else:
BASE ¼ 96
wordWeight ¼ 0

for i in range (4,0,1):
charValue¼ (ord(theWord[i]) BASE) shiftValue ¼ (i-1)*8
charWeight ¼ charValue << shiftValue wordWeight ¼ (wordWeight |
charWeight)

if ( wordWeight in self.weightedMatrix): return True
else:
return False

## End Class Matrix
CHAPTER REVIEW

```

In this chapter, I put to use new Python language elements including sets and bytearrays and I expanded the use of lists and classes. I also provided examples of how to open and read files containing a diverse set of values such as keywords, raw binary, and hexadecimal values and work with those directly within Python bytearrays, lists, and sets. I demonstrated how leveraging the right language elements can lead to simple, readable code that accomplishes our objectives. As a result, p-search is a useable application that provides investigators with the capability to search for keywords and

Additional Resources 123

generate an index of words found in documents, disk images, memory snapshots, and even network traces.

In addition, I once again demonstrated the interoperability of Python code by running the sample unaltered code on Windows, Linux, and Mac producing the same results.

SUMMARY QUESTIONS

1. What are the advantages and disadvantages of the built-in language types (lists, sets, and dictionary).
2. Explain the advantages of using a set to hold a list of keywords.
3. How does Python's ability to deal with a broad set of data (binary, text, and integers) differ from other languages?
4. Many modern files, disk images, memory snapshots, and network traces are going to contain Unicode data not only simple ASCII. Modify p-search to provide the ability to search and index data that contains a mixture of simple ASCII and Unicode.
5. Expand the matrix to include a broader set of weighted values that include foreign language, proper names, places, and vocabulary from other domains such as medical terminology.

Additional Resources

The Complete Sherlock Holmes: All 4 Novels and 56 Short Stories. Deluxe edition. Bantam Classics; October 1, 1986.

Python Programming Language—Official Website, Python.org.

<http://www.python.org>.

The Python Standard Library. <http://docs.python.org/2/library/>.

CHAPTER

Forensic Evidence Extraction (JPEG and TIFF) 5

CHAPTER CONTENTS

Introduction	125
The Python Image Library	126
Before Diving Straight In	128
PIL Test-Before Code	131
Determining the Available EXIF TAGS	131
Determining the Available EXIF GPSTAGS	133
p-ImageEvidenceExtractor Fundamental Requirements	137
Design Considerations	137
Code Walk-Through	

137	
Main Program	141
Class Logging	141
cvsHandler	141
Command Line Parser	141
EXIF and GPS Handler	141
Examining the Code	141
Main Program	141
EXIF and GPS Processing	144
Logging Class	148
Command Line Parser	149
Comma Separated Value (CSV) Writer Class	150
Full Code Listings	151
Program Execution	158
Chapter Review	
159	
Summary Questions	
159	
Additional Resources	
163	

INTRODUCTION

Simple and complex digital data structures offer significant opportunities to collect and extract valuable evidence. Files have simple metadata that define modified, access, and created times along with file ownership, file size, and other attributes that label the file as read-only, system, or archive. This information is easy to retrieve and examine with common desktop tools. Memory snapshots contain both structured and unstructured data that can be carved and reconstructed to reveal running processes and threads, recent user activity, network data, and even user typed passwords and cryptographic keying material.

Python Forensics 125© 2014 Elsevier Inc. All rights reserved.

Other complex files structures, such as the JPEG files that include EXIF data (The Exchangeable Image File Format), contain a plethora of potentially valuable information about the image itself. This can include the camera type used to take the photograph, the time and date the photo was taken, and literally

hundreds of individual data elements associated with the photograph. With the advent of Facebook, Instagram, Twitter, Flickr, and other social media services, the sharing of digital photographs has been weaved into the fabric of our culture. Today, over a billion people carry smart phones and tablets that have built-in cameras capable of taking high-quality photographs and movies that can be shared instantly across the globe using the aforementioned social network applications, text messaging, or good old-fashioned e-mail.

In addition, everyone from the Weather Channel to law enforcement investigators use digital photography to provide insight and understanding of physical events. The Federal Communications Commission (FCC) has adopted new rules that include mandatory improvements to the accuracy of the location information transmitted when cell phone-based 911 calls are made. Specifically, E911 rules require wireless service providers to deliver the latitude and longitude of the caller within 50-300 m [FCC]. These requirements have driven smart phone manufactures to include Global Positioning System (GPS) chips in their handsets along with high-quality cameras that can automatically embed location and time data directly into digital photographs. Consequently these innovations have delivered potentially valuable evidence including “what: the content of the photo,” “when: the photo was snapped,” and “where: the photo was taken.”

Based on this, I wanted to delve deeper into the extraction of evidence from digital photographs, specifically those photos taken with smart phones that include EXIF data. I will focus most of my attention on the EXIF data associated with JPEG and TIFF images, mainly because the preponderance of photos generated today from smart phones support EXIF embedding.

Many of you may have already examined the contents of digital photographs and the EXIF information using Hex editors and other primitive tools, and as a result require an upgraded prescription from the optometrist afterword. To help us in peeling back the onion-like layers of digital photographs, I will be adding a new library that will do most of the heavy lifting for us—namely the Python Image Library(PIL). The PIL provides a great deal of capabilities to not only extract information from images but also provides capabilities to manipulate them. I am going to focus only on the data extraction of EXIF data using the PIL, but once you learn how to properly use the library you can use the PIL for other image processing activities [Python Image Library].

The Python Image Library

The first step in using the PIL is to install the library. If you are running Python on Windows platform, you are in luck as a simple installer is available as shown in Figure 5.1.

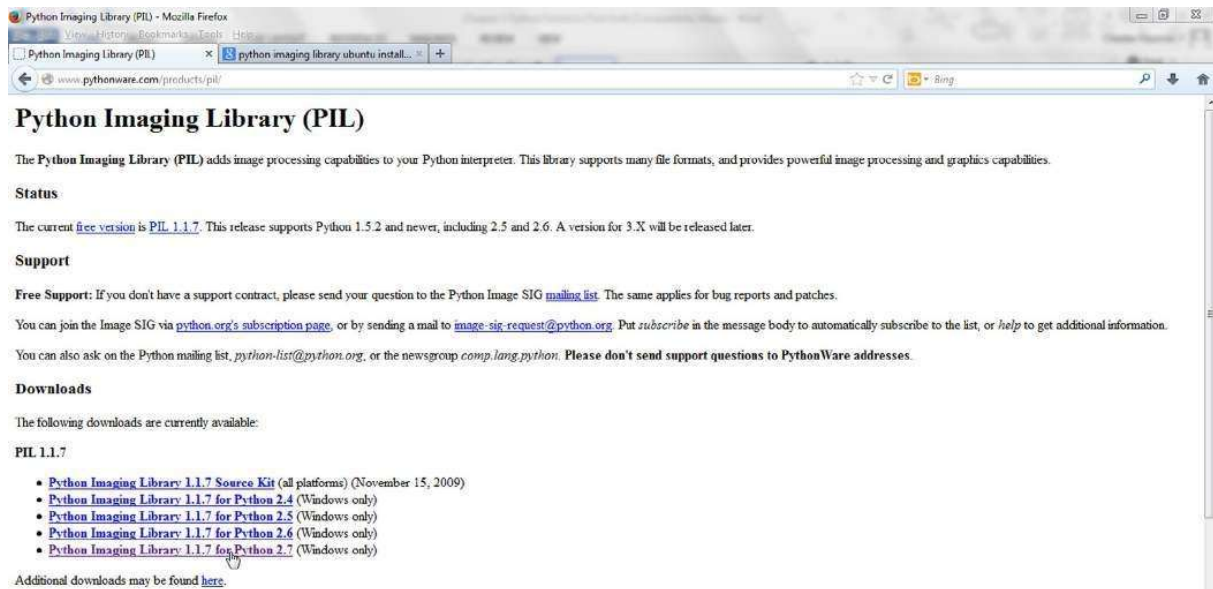


FIGURE 5.1
Downloading Python Image Library for Windows.



FIGURE 5.2
Windows installation Wizard for the Python Image Library.

Once you have successfully downloaded the proper version for your Python installation (I am using PIL version 1.1.7 for Python 2.7), you simply execute the downloaded file and accept the defaults provided by the Wizard as shown in [Figure 5.2](#). (This is a great example of why I choose to utilize Python 2.x for the book, as you can see the PIL library has not yet been converted to work with Python 3.x. Once this is done, the programs in this chapter can be easily updated to work with Python 3.x.)

If you are working with Linux or Mac, the steps for library installation are readily available on the Web. Here is one excellent example for Ubuntu 12.04 LTS: [https:// gist.github.com/dwayne/3353083](https://gist.github.com/dwayne/3353083) ([Figure 5.3](#)).

Before diving straight in
Before I dive right into the PIL, I need to introduce the built-in Python Dictionary type. Up to this point we have used the lists and sets structures within Python in several examples. In some cases, structures with a bit more flexibility

and capability are required. More specifically, since the PIL uses dictionaries, we need to better understand their operation.

As you can see in [Code Script 5.1](#), each entry in the dictionary has two parts, first the key which is followed by the value. For example, the key “fast” has an associated value of “Porsche.”

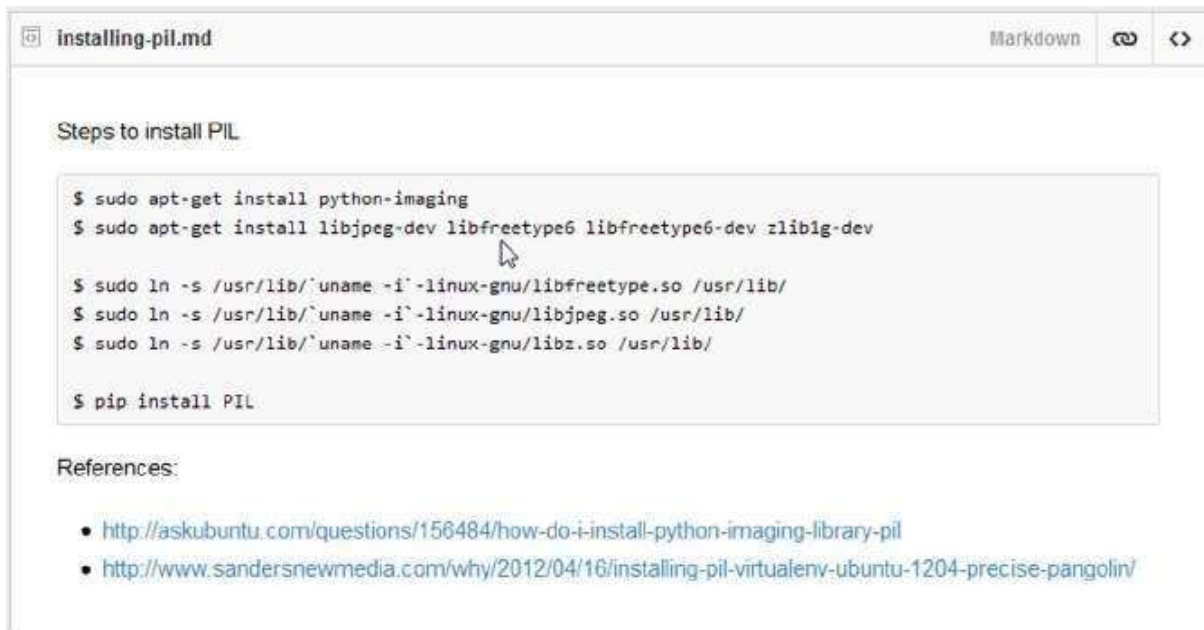


FIGURE 5.3

Installing Python Image Library on Ubuntu 12.04 LTS.

CODE SCRIPT 5.1 CREATING A DICTIONARY

```
>>> dCars = {"fast": "Porsche", "clean": "Prius", "expensive": "Rolls"}
>>> print dCars
{'expensive': 'Rolls', 'clean': 'Prius', 'fast': 'Porsche'}
```

You might have noticed that when I printed the contents of the dictionary, the order has changed. This is due to the fact that the key is converted to a hash value when stored in order to both save space and to improve performance when searching or indexing the dictionary. If you want the dictionary printed in a specific order you can use the following code, which extracts the dictionary items and then sorts the items by the key.

CODE SCRIPT 5.2 PRINTING A SORTED DICTIONARY

```
>>> dCarsItems = sorted(dCars.items())
```

```
>>> dCarsItems.sort()
>>> print dCarsItems
[('clean','Prius'), ('expensive','Rolls'), ('fast','Porsche')]
```

What if the dictionary is large and you wish to find all the keys? Or once you find the keys you wish to extract the value of a specific key?

CODE SCRIPT 5.3 SIMPLE KEY AND VALUE EXTRACTION

```
>>> # what keys are available?
>>> dCars.keys()
['expensive','clean','fast']
>>> # what is the value associated with the key "clean"? >>> dCars.get("clean")
'Prius'
```

Finally, you might want to iterate over a dictionary and extract and display all the key, value pairs.

CODE SCRIPT 5.4 ITERATING OVER A DICTIONARY

```
>>> for key, value in dCars.items():
...     print key, value
expensive Rolls
clean Prius
fast Porsche
```

If you are not familiar with Python but have familiarity with other programming environments, [Script 5.4](#) may seem a bit odd for several reasons.

1. We did not need to declare the variables' key and value, as Python takes care of that automatically. In this simple case both key and value are strings, but they are not required to be.
2. Since we know that dCars is a dictionary, the method items automatically returns both key and values. The keys are “clean,” “expensive,” and “fast” in this example and the values are the string names of the cars “Porsche,” “Prius,” and “Rolls.”

Since Python will automatically handle the data typing for you, let us take a look at something a bit more complicated.

CODE SCRIPT 5.5 A MORE COMPLEX DICTIONARY EXAMPLE

```
>>> dTemp¼ { "2013:12:31": [22.0, "F", "High"],
"2013:11:28": [24.5, "C", "Low"], "2013:12:27": [32.7, "F", "High"]} >>>
dTempItems ¼ dTemp.items()
>>> dTempItems.sort()
>>> for key, value in dTempItems:
...     print key, value .. .
2013:11:28 [24.5, 'C', 'Low'] 2013:12:27 [32.7, 'F', 'High'] 2013:12:31
[22.0, 'F', 'High']
```

As you can see in Code Script 5.5, the keys are strings that represent a date while value is actually a Python list that contains the time, the time reference, the temperature, and the units of the temperature.

Now that you have a good general understanding of the dictionaries, go create some of your own and experiment with the available methods. From this example, I hope you see the power of the Dictionary type and it has you thinking about other ways to apply this structure.

PIL test-before code

As with other modules and libraries, the best way to become comfortable with the library is to experiment with it. Remember our adage test-then code, this is especially true for PIL, mainly because the PIL uses a new Python language structure, the Dictionary data structure. In addition, the EXIF module of the PIL requires a solid understanding before you can put it to work for you. I will use the Python Shell and short scripts to demonstrate.

Determining the available EXIF TAGS

The PIL has two important sets of TAGS that provide keys that are used to access dictionary elements. They are EXIFTAGS and GPSTAGS. I developed a script to determine which EXIF TAGS are available by using the following method:

- (1) Import the EXIFTAGS from the PIL library.
- (2) Extract the dictionary items (the key, value pairs) from TAGS.
- (3) Sort them to make it easier later to identify them.
- (4) Print the sorted dictionary of key, value pairs.

CODE SCRIPT 5.6 EXTRACTING EXIF TAGS


```
>>> from PIL.EXIFTags import TAGS >>> EXIFTAGS ¼ TAGS.items()
>>> EXIFTAGS.sort()
>>> print EXIFTags
```

This produces the following printed list of available tags. Note this is not the data, just the global list of EXIF TAGS:

```
[(256,'ImageWidth'), (257,'ImageLength'), (258,'BitsPerSample'), (259,
'Compression'), (262,'PhotometricInterpretation'), (270,
'ImageDescription'), (271,'Make'), (272,'Model'), (273,'StripOffsets'), (274,
'Orientation'), (277,'SamplesPerPixel'), (278,'RowsPerStrip'), (279,
'StripByteConunts'), (282,'XResolution'), (283,'YResolution'), (284,
'PlanarConfiguration'), (296,'ResolutionUnit'), (301,'TransferFunction'),
(305,'Software'), (306,'DateTime'), (315,'Artist'), (318,'WhitePoint'), (319,
'PrimaryChromaticities'), (513,'JpegIFOffset'), (514,'JpegIFByteCount'),
(529,'YCbCrCoefficients'), (530,'YCbCrSubSampling'), (531,
'YCbCrPositioning'), (532,'ReferenceBlackWhite'), (4096,
'RelatedImageFileFormat'), (4097,'RelatedImageWidth'), (33421,
'CFARRepeatPatternDim'), (33422,'CFAPattern'), (33423,'BatteryLevel'),
(33432,'Copyright'), (33434,'ExposureTime'), (33437,'FNumber'), (34665,
'EXIFOffset'), (34675,'InterColorProfile'), (34850,'ExposureProgram'),
(34852,'SpectralSensitivity'), (34853,'GPSInfo'), (34855,
'ISOSpeedRatings'), (34856,'OECF'), (34857,'Interlace'), (34858,
'TimeZoneOffset'), (34859,'SelfTimerMode'), (36864,'EXIFVersion'), (36867,
'DateTimeOriginal'), (36868,'DateTimeDigitized'), (37121,
'ComponentsConfiguration'), (37122,'CompressedBitsPerPixel'), (37377,
'ShutterSpeedValue'), (37378,'ApertureValue'), (37379,'BrightnessValue'),
(37380,'ExposureBiasValue'), (37381,'MaxApertureValue'), (37382,
'SubjectDistance'), (37383,'MeteringMode'), (37384,'LightSource'), (37385,
'Flash'), (37386,'FocalLength'), (37387,'FlashEnergy'), (37388,
'SpatialFrequencyResponse'), (37389,'Noise'), (37393,'ImageNumber'),
(37394,'SecurityClassification'), (37395,'ImageHistory'), (37396,
'SubjectLocation'), (37397,'ExposureIndex'), (37398,'TIFF/EPStandardID'),
(37500,'MakerNote'), (37510,'UserComment'), (37520,'SubsecTime'), (37521,
'SubsecTimeOriginal'), (37522,'SubsecTimeDigitized'), (40960,
'FlashPixVersion'), (40961,'ColorSpace'), (40962,'EXIFImageWidth'), (40963,
'EXIFImageHeight'), (40964,'RelatedSoundFile'), (40965,
'EXIFInteroperabilityOffset'), (41483,'FlashEnergy'), (41484,
'SpatialFrequencyResponse'), (41486,'FocalPlaneXResolution'), (41487,
'FocalPlaneYResolution'), (41488,'FocalPlaneResolutionUnit'), (41492,
```

'SubjectLocation'), (41493,'ExposureIndex'), (41495,'SensingMethod'), (41728,'FileSource'), (41729,'SceneType'), (41730,'CFAPattern')]

Time is a concept that allows society to function in an orderly fashion where all parties are able to easily understand the representation of time and agree on the sequence of events. Since 1972, the International standard for time is Coordinated Universal Time (UTC). UTC forms the basis for the official time source in most nations around the globe; it is governed by a diplomatic treaty adopted by the world community that designates National Metrology Institutes (NMIs) as UTC time sources. The National Institute of Standards and Technology in the United States and the National Physical Laboratory (NPL) in England are two examples of NMIs. There are about 50 similar metrology centers that are responsible for official time throughout the world. In addition, time zones can play a key role when examining digital evidence; they provide localization of events and actions based upon the computer's time zone setting. For example, if the computer's time zone is set to New York, the local time is 5 hours from UTC (or 5 hours behind). Finally, many U.S. States and foreign countries observe day light savings time, most modern operating systems automatically adjust for this observance which causes additional differences and calculations between UTC or when attempting to synchronize events across time zones.

Within the printed list, I have highlighted TAGS that are of immediate interest including GPSInfo, TimeZoneOffset, and DateTimeOriginal.

It is important to note that it is not guaranteed that all of these TAGS will be available for every image; rather this is the broad set of key value pairs that are potentially within an image. Each manufacturer independently determines which values will be included. Thus you must process the key value pairs for each target image to determine which TAGS are in fact available. I will show you how this is done shortly.

As you scan the list, you may find other TAGS that are of interest in your investigative activities and you can utilize the same approach defined here to experiment with those values.

Determining the available EXIF GPSTAGS

Next, I want to drill a bit deeper into the GPSInfo TAG, as our objective in this chapter is to extract GPS-based location data of specific images. I take a similar

approach to determine what GPSTAGS are available at the next level by writing the following lines of code.

CODE SCRIPT 5.7 EXTRACTING GPS TAGS

```
>>> from PIL.EXIFTags import GPSTAGS >>> gps = GPSTAGS.items()
>>> gps.sort()
>>> print gps
```

```
[(0,'GPSVersionID'), (1,'GPSLatitudeRef'), (2,'GPSLatitude'),
(3,'GPSLongitudeRef'), (4,'GPSLongitude'), (5,'GPSAltitudeRef'),
(6,'GPSAltitude'),(7,'GPSTimeStamp'), (8,'GPSSatellites'), (9,'GPSStatus'),
(10,'GPSMeasureMode'), (11,'GPSDOP'), (12,'GPSSpeedRef'), (13,'GPSSpeed'),
(14,'GPSTrackRef'), (15,'GPSTrack'), (16,'GPSImgDirectionRef'), (17,
'GPSImgDirection'), (18,'GPSMapDatum'), (19,'GPSDestLatitudeRef'), (20,
'GPSDestLatitude'), (21,'GPSDestLongitudeRef'), (22,'GPSDestLongitude'),
(23,'GPSDestBearingRef'), (24,'GPSDestBearing'), (25,
'GPSDestDistanceRef'), (26,'GPSDestDistance')]
```

I have highlighted a few GPSTAGS that we need in order to pinpoint the location and time of the photograph. Again pointing out that there is no guarantee that any or all of these tags will be available within the subject image.

Longitude and latitude locations provide the ability to pinpoint a specific location on the earth. The Global Position System and GPS receivers built into smart phones and cameras can deliver very accurate geolocation information in the form of longitude and latitude coordinates. If so configured, this geolocation information can be directly embedded in photographs, movies, and other digital objects when certain events occur. Extracting this geolocation information from these digital objects and then entering that data into online services such as Mapquest, Google Maps, Google earth, etc., can provide evidence of where and when that digital object was created.

If we plan to utilize the TAGS to pinpoint the location, we need a better definition and understanding of the TAGS of interest, their meaning, and usage. **GPSLatitudeRef:** Latitude is specified by either the North or South position in degrees from the equator. Those points that are north of the equator range from 0 to 90 degrees. Those points that are south of the equator range from 0 to À90 degrees. Thus this reference value is used to define where the latitude is in

relationship to the equator. Since the value latitude is specified in EXIF as a positive integer, if the Latitude Reference is “S” or South, the integer value must be converted to a negative value.

GPSLatitude: Indicates the latitude measured in degrees. The EXIF data specify these as whole degrees, followed by minutes and then followed by seconds.

GPSTimeStamp: Longitude is specified by either the east or west position in degrees. The convention is based on the Prime Meridian which runs through the Royal Observatory in Greenwich, England, which is defined as longitude zero degrees. Those points that are west of the Prime Meridian range from 0 to 180 degrees. Those points that are east of the Prime Meridian range from 0 to 180 degrees. Thus this reference value is used to define where the longitude value is in relationship to Greenwich. Since the value of longitude is specified in EXIF as a positive integer, if the Longitude Reference is west or “W,” the integer value must be converted to a negative value.

GPSTimeStamp: Indicates the longitude measured in degrees. The EXIF data specify these as whole degrees, followed by minutes and then followed by seconds.

GPSTimeStamp: Specifies the time as Universal Coordinated Time that the photo was taken.

Since the data contained within the EXIF structure are digital values, they are subject to manipulation and forgery. Thus, securing the images or obtaining them directly from the Smart Mobile device makes fraudulent modification less likely and much more difficult to accomplish.

Now that we have a basic understanding of what TAGS are potentially available within a photograph, let us examine an actual photograph. [Figure 5.4](#) depicts an image downloaded from the Internet and will be the subject of this experiment.



FIGURE 5.4

Internet Photo Cat.jpg.

CODE SCRIPT 5.8 EXTRACT EXIF TAGS OF A SAMPLE IMAGE

```
>>> from PIL import Image
>>> from PIL.EXIFTags import TAGS, GPSTAGS
>>>
>>> pilImage = Image.open("c:\\pictures\\cat.jpg")
... EXIFData = pilImage._getEXIF()
...
>>> catEXIF = EXIFData.items()
>>> catEXIF.sort()
>>> print catEXIF
```

This script imports the PIL Image module that allows me to open the specific image for processing and to create the object `pilImage`. I can then use the object `pilImage` to acquire the EXIF data by using the `._getEXIF()` method. As we did when extracting the EXIFTAGS and GPSTAGS, in this example I am extracting the dictionary items and then sorting them. Note, in this example I am extracting the actual EXIF data (or potential evidence) not the TAG references.

The output from the [Code Script 5.3](#) is shown here.

```
[(271, 'Canon'), (272, 'Canon EOS 400D DIGITAL'), (282, (300, 1)), (283, (300, 1)), (296, 2), (306, '2008:08:05 23:48:04'), (315, 'unknown'), (33432,
```

'Eirik Solheim www.eirikso.com'), (33434, (1, 100)), (33437, (22, 10)), (34665, 240), (34850, 2), (34853, {0: (2, 2, 0, 0), 1: 'N', 2: ((59, 1), (55, 1), (37417, 1285)), 3: 'E', 4: ((10, 1), (41, 1), (55324, 1253)), 5: 0, 6: (81, 1)}), (34855, 400), (36864, '0221'), (36867, '2008:08:05 20:59:32'), (36868, '2008:08:05 20:59:32'), (37378, (2275007, 1000000)), (37381, (96875, 100000)), (37383, 1), (37385, 16), (37386, (50, 1)), (41486, (3888000, 877)), (41487, (2592000, 582)), (41488, 2), (41985, 0), (41986, 0), (41987, 0), (41990, 0)]

I have highlighted the GPSInfo data associated with this photograph. I was able to identify the GPSInfo TAG based on the output of the previous Code Script 5.1, whereby the output showed that the GPSInfo TAG had a key value of 34853. Therefore, this camera, the Canon EOS 400D DIGITAL, provides the GPS data highlighted between the curly braces. Closer examination of the GPS data reveals the specific key value pairs that are available. As you can see, the key value pairs embedded in the GPSInfo data dictionary are 0, 1, 2, 3, 4, 5, and 6. These TAGS refer to the standard GPS TAGS. In Code Script 5.2, I printed out all the possible GPS Tags, by referring to that output we discover the following GPS TAGS that were provided by the Cannon EOS 400D. They include:

- 0 GPSVersionID
- 1 GPSLatitudeRef
- 2 GPSLatitude
- 3 GPSLongitudeRef
- 4 GPSLongitude
- 5 GPSAltitudeRef
- 6 GPSAltitude

With this information I can calculate the longitude and latitude provided by the EXIF data within the photograph using the following process. But, first, I need to extract the latitude from the key value pair GPSTag 2:

((59, 1), (55, 1), (37417, 1285))

Each latitude and longitude value contains three sets of values. They are:

Degrees: (59, 1)

Minutes: (55, 1)

Seconds: (37,417, 1285)

You might be wondering why they are represented like this. The answer is the EXIF data does not hold floating point numbers (only whole integers). In order to provide more precise latitude and longitude data, each pair represents a ratio.

Next, I need to perform the ratio calculation to obtain a precise fractional value, yielding:

Degrees: $59\frac{1}{4}59.0$

Minutes: $55\frac{1}{4}55.0$

Seconds: $37,417/1285\frac{1}{4}29.1182879$

Or more properly stated 59 degrees, 55 minutes, 29.1182879 seconds.

Note : GPSTag 1:“N” the Latitude Reference key value pair is a north reference from the equator and does not require us to convert this to a negative value, if it were “S” we would have make this additional conversion.

Next, most online mapping programs (i.e., Google Maps) require data to be provided as a fractional value. This is accomplished by using the following formula.

Degrees $\frac{1}{4}(\text{minutes}/60.0)\frac{1}{4}(\text{seconds}/3600.0)$

Note: the denominators of 60.0 and 3600.0 represent the number minutes (60.0) and seconds (3600.0) in each hour, respectively.

Thus this latitude value represented as floating point number would be:

Latitude $\frac{1}{4}39.0\frac{1}{4}(55/60.0)\frac{1}{4}(29.1182879/3600.0)$

Latitude $\frac{1}{4}39.0\frac{1}{4}.91666667\frac{1}{4}.00808841\frac{1}{4} 59.9247551$

Performing the same calculation on the longitude value extracted from GPSTag 4:

$((10, 1), (41, 1), (55324, 1253))$

With a GPSTag 3: “E”

Yields a longitude value of 10.6955981201

Giving us a latitude/longitude value of 59.924755, 10.6955981201

Using Google Maps, I can now plot the 59.924755, 10.6955981201 yielding a point just north and west of Oslo, Norway as shown in [Figure 5.5](#). Noting that Oslo, Norway is in fact East of the Prime Meridian in Greenwich, England.

One final note, this camera did not include other GPSTAGS such as the

GPSTimestamp, which could have given us the time as reported by GPS.

p-ImageEvidenceExtractor fundamental requirements

Now that I have experimented with the PIL, discovered how to extract the EXIF

TAGS, the GPS tags, and the EXIF data from a target image, I want to define our first full extraction application.

Design considerations

Now that I have defined the basic approach, [Table 5.1](#) and [Table 5.2](#) define the program requirements and design considerations for the p-ImageEvidenceExtractor.

Next, I want to define the overall design of the p-gpsExtractor as depicted in [Figure 5.6](#). I will again be using the command line parser to convey selections from the user. I will be directing output to standard out, a CSV file, and the forensic log. The user will specify a folder containing images to process. I have also created a class to handle forensic logging operations that I can reuse or expand in the future. Finally, I will be reusing a slightly modified version of the CSVWriter class I created in [Chapter 3](#) for the pFish application.

CODE WALK-THROUGH

In order to get a feel for the program take a look at [Figure 5.7](#), which is a screenshot of the WingIDE project. You notice on the far right this program contains five Python source files:

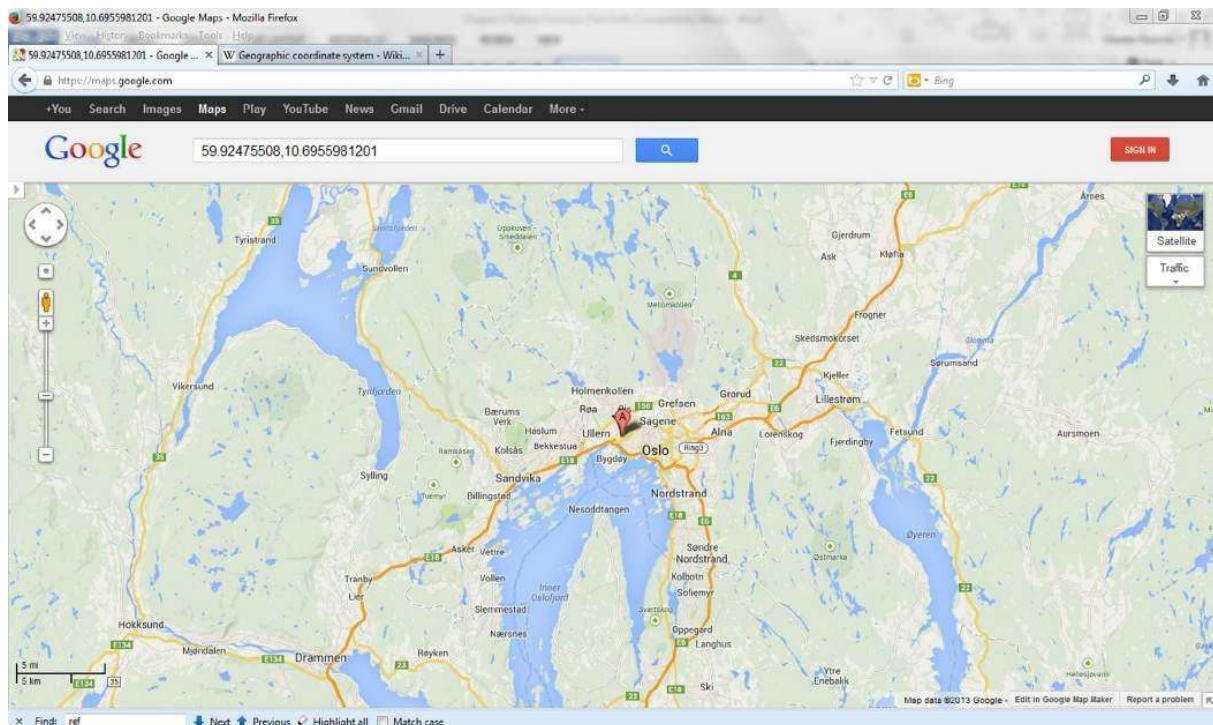


FIGURE 5.5

Map of GPS coordinates extracted From Cat.jpg.

Table 5.1 Basic Requirements for GPS/EXIF Extraction Requirement
Requirement number name

GPS-001 Command Line Arguments

GPS-002 Log GPS-003 Output

GPS-004 Error Handling Short description

Allow the user to specify a directory that contains sample images to attempt to extract GPS information from

The program must produce a forensic audit log The program should produce a list of latitude and longitude values from the GPS information found. The format must be simple lat, lon values in order to be pasted into online mapping applications

The application must support error handling and logging of all operations performed. This will include a textual description and a timestamp

Table 5.2 Design Consideration and Library Mapping Requirement Design
considerations Library selection

User Input (001) User input for keyword and target file

Output (003) The extracted GPS location will be printed out using simple built-in print commands

(Additional)

(002 and 004) Logging and Error Handling Extract additional EXIF and GPS values and record the data for each file encountered in a CSV file Errors may occur during the processing of the files and EXIF data. Therefore, handling these potential errors must be rigorous

I will be using argparse from the Standard Library module to obtain input from the user

I will be using the Python Image Library and the standard Python language to extract and format the output in order to cut and paste directly into the Web page: <http://www.darrinward.com/lat-long/>

Use the Python Image

Library and Python CVS Library

The Python Standard

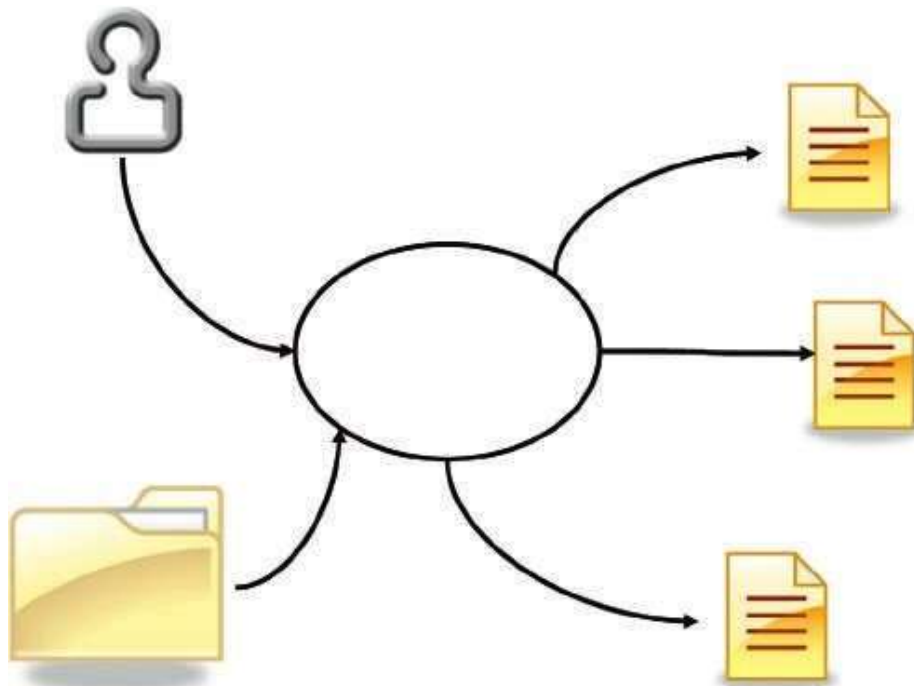
Library includes a logging facility which I can leverage to report any events or errors that occur during processing. Note for this application I will create a reuseable logging class

user Standard out

Program Arguments

p-gpsExtractor CSV
results

Folder with



file

images

Event and error log

p-gpsExtractor context diagram FIGURE 5.6

p-gpsExtractor context diagram.

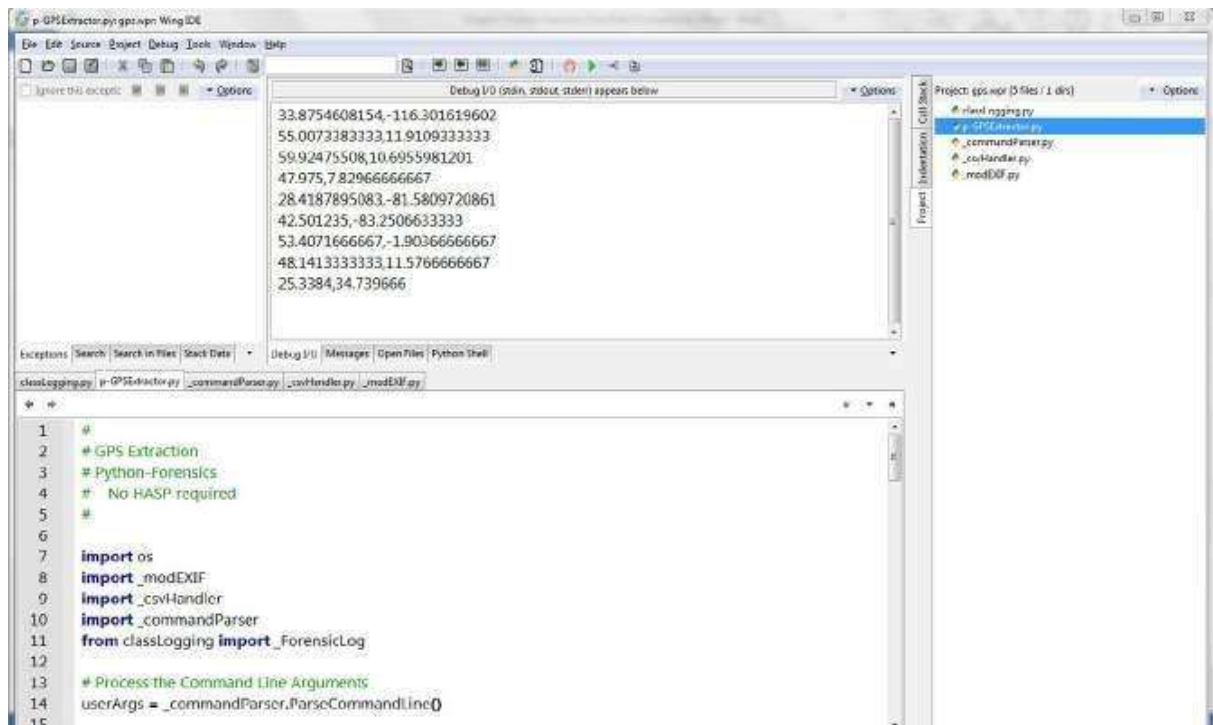


FIGURE 5.7
WingIDE Project Overview.

Main Program

p-gpsExtractor.py: This is the Main Program for this application handling the overall program flow and the processing of each potential image file contained in a folder specified by the user via the command line. Main writes data to standard out in order to make it easy to cut and paste that data into a Web page for map viewing. Main also handles output to the CSV file that provides the results for each of the files processed, as well as any forensic log entries.

Class Logging

classLogging.py: The new class logger abstracts the logging function in a Python class for handling Forensic Logging functions. This will allow for log object passing for future applications to any module or function eliminating any need for global variables.

cvsHandler

csvHandler.py: This modified class for handling CSV File Creation and Writing provides a single point interface to writing output files in CVS format, once again to abstract the functions and make interface simpler. More work will be done on this class in the future to abstract this further.

Command line parser

commandParser.py: I separated out the commandParser.py into its own file for greater portability for this application and for future applications. This allows command line parser to be handled, modified, or enhanced and is more loosely coupled from the Main Program. Like the Logging functions this will eventually be turned into a Class-based module to allow cross-module access and interface.

EXIF and GPS Handler

_modEXIF.py: This module directly interfaces with the PIL to perform the needed operations to extract both EXIF and GPS data from image files that contain EXIF data.

Examining the code Main Program

```
# GPS Extraction # Python-Forensics # No HASP required #
import os
import _modEXIF
import _csvHandler
import _commandParser
from classLogging import ForensicLog
```

For the main program we need to import os to handle folder processing, the command line processor, the cvsHandler and the new ForensicLog Class. # Process the Command Line Arguments
userArgs ¼ _commandParser.ParseCommandLine()

As I typically do, I process the command line arguments to obtain the user specified values. If command parsing is successful we create a new logging object to handle any necessary forensic log events and write the first log event.

```
# create a log object
logPath ¼ userArgs.logPath+"ForensicLog.txt"
oLog ¼ _ForensicLog(logPath)
```

oLog.writeLog("INFO", "Scan Started)" Next I create a CSV File Object that I will use to write data to the CSV results file that will contain the files processed and the resulting GPS data extracted csvPath ¼
userArgs.csvPath+"imageResults.csv" oCSV ¼
_csvHandler._CSVWriter(csvPath)

Needing to obtain the user specified target directory for our GPS extraction, I attempt to open and then process each of the files in the target directory. If any errors occur they are captured using the try except model and appropriate errors are written to the forensic log file.

```
# define a directory to scan
scanDir = userArgs.scanPath
try:
    pics = os.listdir(scanDir)

except:
    oLog.writeLog("ERROR", "Invalid Directory "+ scanDir) exit(0)

for aFile in pics: targetFile = scanDir+aFile if os.path.isfile(targetFile):
```

Once the preliminaries are out of the way, I can now utilize the `_modEXIF` module to process each target file, with the result being a `gpsDictionary` object if successful. This object will contain the EXIF data that is available for this file as embedded in the EXIF record. Before using the data, I check to ensure that the `gpsDictionary` object contains data.

```
gpsDictionary = _modEXIF.ExtractGPSDictionary(targetFile) if
(gpsDictionary):

# Obtain the Lat Lon values from the gpsDictionary # converted to degrees
# the return value is a dictionary key value pairs

dCoord = _modEXIF.ExtractLatLon(gpsDictionary)
```

Next, I call another `_modEXIF` function to process further the `gpsDictionary`. This function as you will see, processes the GPS coordinates, and if successful returns the raw coordinate data. Then using the Dictionary methods you learned about earlier in this chapter, I extract the latitude and longitude data from the dictionary by using the `get` method.

```
lat = dCoord.get("Lat")
latRef = dCoord.get("LatRef")
lon = dCoord.get("Lon")
lonRef = dCoord.get("LonRef")
```

if (lat and lon and latRef and lonRef):

Upon successful extraction of the latitude and longitude values I print the results to standard out, write the data to the CSV file and update the forensic log accordingly. Or if unsuccessful, I report the appropriate data to the forensic log file.

```
print str(lat)+' '+str(lon)
```

```
# write one row to the output csv file
```

```
oCSV.writeCSVRow(targetFile, EXIFList[TS], EXIFList [MAKE],  
EXIFList[MODEL],latRef, lat, lonRef,  
lon)
```

```
oLog.writeLog("INFO", "GPS Data Calculated for:" + targetFile)
```

```
else:
```

```
oLog.writeLog("WARNING", "No GPS EXIF Data for"+ targetFile)
```

```
else:
```

```
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)
```

```
else:
```

```
oLog.writeLog("WARNING", targetFile + " not a valid file)" Finally before  
program exit I delete the forensic log and CSV objects which in turn closes the  
log and the associated CSV file.
```

```
# Clean up and Close Log and CSV File
```

```
del oLog
```

```
del oCSV
```

EXIF and GPS processing

The core data extraction elements of the p-gpsExtractor are contained within the `_modEXIF.py` module.

```
#
```

```
# Data Extraction - Python-Forensics
```

```
# Extract GPS Data from EXIF supported Images (jpg, tiff) # Support Module
```

```
#
```

I start by importing the libraries and modules needed by `_modEXIF.py`.

Important module of note is the Python Imaging Library and associated capabilities. `import os` # Standard Library OS functions

`from classLogging import _ForensicLog` # Logging Class From the PIL I import

the Image Library and the EXIF and GPS Tags in order to index into the dictionary structures, much like I did in the script examples 5-6 thru 5-8.

```
# import the Python Image Library # along with TAGS and GPS related TAGS
from PIL import Image
from PIL.ExifTags import TAGS, GPSTAGS
```

Now let's take a look and breakdown the individual functions. I will start with the extraction of the GPS Dictionary. The function take the full path name of the target file that the extraction is to occur with.

```
#
# Extract EXIF Data
#
# Input: Full Pathname of the target image #
# Return: gpsDictionary and extracted EXIFData list #
def ExtractGPSDictionary(fileName):
```

Using the familiar try / except module I attempt to open the filename provided and if successful I then extract the EXIF data by using the PIL getEXIF() function.

```
try: pilImage = Image.open(fileName) EXIFData = pilImage._getEXIF()
```

```
except Exception:
# If exception occurs from PIL processing # Report the
return None, None
```

```
# Iterate through the EXIFData # Searching for GPS Tags
```

Next, I'm going to iterate through the EXIFData TAGS and collect some basic EXIF data and then obtain the GPSTags if they are included within the EXIF data of the specific image.

```
imageTimeStamp = "NA" CameraModel = "NA" CameraMake = "NA"
```

```
if EXIFData: for tag, theValue in EXIFData.items(): # obtain the tag
tagValue = TAGS.get(tag, tag) # Collect basic image data if available
```

As I iterate through the tags that are present within this image, I check for the presence of the DateTimeOriginal along with the camera make and model in


```

if tagValue ¼¼¼'DateTimeOriginal':
imageTimeStamp ¼ EXIFData.get(tag)
if tagValue ¼¼¼ "Make":
cameraMake ¼ EXIFData.get(tag)
if tagValue ¼¼¼'Model':
cameraModel ¼ EXIFData.get(tag) I also check for the GPSInfo TAG, and if
present I iterate through GPSInfo and extract the GPS Dictionary using the
GPSTAGS.get()functions. # check the tag for GPS if tagValue ¼¼¼ "GPSInfo": #
Found it !
# Now create a Dictionary to hold the GPS Data

gpsDictionary ¼ {}
# Loop through the GPS Information

for curTag in theValue:
gpsTag¼ GPSTAGS.get(curTag, curTag) gpsDictionary[gpsTag] ¼
theValue[curTag]

```

```
basicEXIFData ¼ [imageTimeStamp, cameraMake, cameraModel]
return gpsDictionary, basicEXIFData else:
```

`return None, None`

End ExtractGPSDictionary

The next support function is `ExtractLatLon`. As I covered in the narrative of this

#

```
# Extract the Latitude and Longitude Values
# from the gpsDictionary
#
```

The ExtractLatLon function takes as input a gps Dictionary structure. def ExtractLatLon(gps):
to perform the calculation we need at least
lat, lon, latRef and lonRef

Before I attempt the conversion, I need to validate the gps Dictionary contains the proper key value pairs. This includes the latitude, longitude, latitude reference and longitude reference.

```
if (gps.has_key("GPSLatitude") and
gps.has_key("GPSLongitude")
and gps.has_key("GPSLatitudeRef")
and gps.has_key("GPSLongitudeRef")):
```

Once we have the minimum inputs I extract the individual values.

```
latitude¼ gps["GPSLatitude"]
latitudeRef ¼ gps["GPSLatitudeRef"]
longitude ¼ gps["GPSLongitude"]
longitudeRef ¼ gps["GPSLongitudeRef"]
```

I then call the conversion function for both latitude and longitude values. lat ¼ ConvertToDegrees(latitude)

lon ¼ ConvertToDegrees(longitude)

Next I need to account for the latitude and longitude reference values and set the proper negative values if necessary.

```
# Check Latitude Reference
```

```
# If South of the Equator then lat value is negative if latitudeRef ¼¼ "S":
```

```
lat ¼ 0 - lat
```

```
# Check Longitude Reference
```

```
# If West of the Prime Meridian in
```

```
# Greenwich then the Longitude value is negative
```

```
if longitudeRef ¼¼ "W": lon ¼ 0 - lon
```

Once this has been accounted for, I create a new dictionary to hold the final values and return that to the caller. In this case the main program.

```
gpsCoor = {"Lat": lat, "LatRef": latitudeRef,  
"Lon": lon, "LonRef": longitudeRef}  
return gpsCoor else:
```

Once again, if the minimum values are not present, I return the built in Python constant None to indicate an empty return.

```
return None
```

The final support function is the ConvertToDegrees function that converts the GPS data into a floating point value. This code is very straightforward and follows the formula presented in the narrative of this chapter. The only important aspect to point out is the try/except model used whenever dividing two numbers, as a divide by zero would cause a failure to occur and the program to abort. As with any data coming from the wild, anything is possible and we need to account for possible zero division. I have seen this with EXIF data, when the value for seconds is zero the ratio is reported as 0:0. Since this is a legal value, I simply set the degrees, minutes, or seconds to zero when the exception occurs.

```
#  
# Convert GPSCoordinates to Degrees  
#  
# Input gpsCoordinates value from in EXIF Format  
#  
  
def ConvertToDegrees(gpsCoordinate):  
    d0 = gpsCoordinate[0][0]  
    d1 = gpsCoordinate[0][1]  
  
    try:  
        degrees = float(d0) / float(d1)  
    except:  
        degrees = 0.0  
  
    m0 = gpsCoordinate[1][0] m1 = gpsCoordinate[1][1] try:  
  
    minutes = float(m0) / float(m1) except:  
    minutes = 0.0  
  
    s0 = gpsCoordinate[2][0] s1 = gpsCoordinate[2][1] try:
```

```
seconds  $\frac{1}{4}$  float(s0) / float(s1) except:
seconds  $\frac{1}{4}$  0.0
```

```
floatCoordinate  $\frac{1}{4}$  float (degrees + (minutes / 60.0) + (seconds / 3600.0))
return floatCoordinate
```

Logging Class

For this application I have include a new class logging that abstracts and simplifies the handling of forensic logging events across modules.

```
import logging
#
# Class: _ForensicLog
#
# Desc: Handles Forensic Logging Operations
#
# Methods
# constructor: # writeLog:
# destructor: #
```

```
class _ForensicLog: Initializes the Logger
Writes a record to the log Writes an information message
```

and shuts down the logger

init is the constructor method that is called whenever a new ForensicLog object is created. The method initializes a new Python logging object using the Python Standard Library. The filename of the log is passed in as part of the object instantiation. If an exception is encountered a message is sent to standard output and the program is aborted.

```
def __init__(self, logName):
try:
# Turn on Logging
logging.basicConfig(filename= $\frac{1}{4}$ logName,
```

```
level= $\frac{1}{4}$ logging.DEBUG,format= $\frac{1}{4}$ '%(asctime) s %(message)s') except:
print "Forensic Log Initialization
Failure ... Aborting"
```

```
exit(0)
```

The next method is writeLog which takes two parameters (along with self), the first is the type or level of the log event (INFO, ERROR or WARNING) and then a string representing the message that will be written to the log.

```
def writeLog(self, logType, logMessage): if logType == "INFO":
logging.info(logMessage)

elif logType == "ERROR":
logging.error(logMessage)
elif logType == "WARNING": logging.warning(logMessage)
else:
logging.error(logMessage)
return
```

Finally the del method is used to shutdown logging and to close the log file. The message Logging Shutdown is sent to the log as an information message just prior to shutdown of the log.

```
def __del__(self):
logging.info("Logging Shutdown")
logging.shutdown()
```

Command line parser

Command line parser handles the user input, provides validation for the arguments passed, and reports error along with help text to assist the user.

```
import argparse # Python Standard Library - Parser for command-line options,
arguments
import os # Standard Library OS functions
# Name: ParseCommand() Function
#
# Desc: Process and Validate the command line arguments
# use Python Standard Library module argparse
#
# Input: none
#
# Actions:
```

```

#
#
Uses the standard library argparse
to process the command line

#
def ParseCommandLine():

    gpsExtractor requires three inputs 1) the directory path for the forensic log, 2)
    the directory path to be scanned and 3) the directory path for the CSV result file.
    All these directory paths must exist and be writable in order for parsing to
    succeed.

    parser = argparse.ArgumentParser('Python gpsExtractor') parser.add_argument('-
    v', '--verbose', help="enables printing of additional program
    messages", action='store_true') parser.add_argument('-l', '--logPath',
    type=ValidateDirectory, required=True,
    help="specify the directory
    for forensic log output file")

    parser.add_argument('-c', '--csvPath',
    type=ValidateDirectory, required=True, help="specify the output directory for
    the csv file")

    parser.add_argument('-d', '--scanPath',
    type=ValidateDirectory, required=True, help="specify the directory to scan")

    theArgs = parser.parse_args() return theArgs
# End Parse Command Line

```

The ValidateDirectory function verifies that value passed is a bona fide directory and the path is in fact writable. I use the os.path.isdir() method to verify the existence and the os.access() method to verify the writing.

```

def ValidateDirectory(theDir):
# Validate the path is a directory
if not os.path.isdir(theDir):
raise argparse.ArgumentTypeError('Directory does not exist') # Validate the path
is writable

```

Comma separated value (CSV) Writer class
The `_CSVWriter` class provides object-based abstraction for writing headings and data to a standard CSV file.

```
#
# Class: _CSVWriter
#
# Desc: Handles all methods related to
# comma separated value operations
#
# Methods
# constructor: # writeCSVRow: # writerClose:
```

The constructor opens the filename provided as a csvFile and then writes the header to the csv File.

```
# create a writer object and then write the header row self.csvFile14
open(fileName,'wb')
```

```
self.writer.writerow( ('Image Path','TimeStamp', 'Camera Make',
'Camera Model',
```

```
'Lat Ref','Latitude', 'Lon Ref','Longitude') )
```

```
except:
```

```
log.error('CSV File Failure')
```

The actual writeCSVRow method takes as input the individual columns defined and writes them to the file. In addition, the function converts the floating point latitude and longitude values to properly formatted strings.

```
def writeCSVRow(self, fileName, timeStamp, CameraMake,
CameraModel,latRef, latValue, lonRef, lonValue):
```

```
latStr¼ '%.8f'% latValue
```

```
lonStr¼ '%.8f'% lonValue
```

```
self.writer.writerow( (fileName, timeStamp, CameraMake, CameraModel,
latRef, latStr, lonRef, lonStr))
```

Finally the del method closes the csvFile for proper deconstruction of the object.

```
def __del__(self):
```

```
self.csvFile.close()
```

Full code listings

```
#
```

```
# GPS Extraction
```

```
# Python-Forensics # No HASP required #
```

```
import os
```

```
import _modEXIF
```

```
import _csvHandler
```

```
import _commandParser
```

```
from classLogging import _ForensicLog
```

```
# Offsets into the return EXIFData for # TimeStamp, Camera Make and Model
```

```
TS¼ 0
```

```
MAKE¼ 1
```

```
MODEL¼ 2
```



```

# Process the Command Line Arguments
userArgs ¼ _commandParser.ParseCommandLine()

# create a log object
logPath ¼ userArgs.logPath+"ForensicLog.txt" oLog ¼ _ForensicLog(logPath)

oLog.writeLog("INFO", "Scan Started") csvPath ¼
userArgs.csvPath+"imageResults.csv" oCSV ¼
_csvHandler._CSVWriter(csvPath)

# define a directory to scan scanDir ¼ userArgs.scanPath try:

picts ¼ os.listdir(scanDir)
except:
oLog.writeLog("ERROR", "Invalid Directory "+ scanDir)
exit(0)

print "Program Start" print

for aFile in picts : targetFile ¼ scanDir+aFile if os.path.isfile(targetFile):

gpsDictionary, EXIFList ¼ _modEXIF.ExtractGPSDictionary (targetFile)
if (gpsDictionary):

# Obtain the Lat Lon values from the gpsDictionary # Converted to degrees
# The return value is a dictionary key value pairs

dCoor¼ _modEXIF.ExtractLatLon(gpsDictionary) lat ¼ dCoor.get("Lat") latRef
¼ dCoor.get("LatRef") lon ¼ dCoor.get("Lon")
lonRef ¼ dCoor.get("LonRef")
if ( lat and lon and latRef and lonRef): print str(lat)+' '+str(lon)

# write one row to the output file
oCSV.writeCSVRow(targetFile, EXIFList[TS], EXIFList [MAKE],
EXIFList[MODEL],latRef, lat, lonRef, lon) oLog.writeLog("INFO", "GPS Data
Calculated for :" + targetFile)

else :
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)

```

```

else :
oLog.writeLog("WARNING", "No GPS EXIF Data for "+ targetFile)

else:
oLog.writeLog("WARNING", targetFile + " not a valid file")

# Clean up and Close Log and CSV File del oLog
del oCSV
import argparse
for command-line options, arguments import os
# Python Standard Library - Parser

# Standard Library OS functions

# Name: ParseCommand() Function #
# Desc: Process and Validate the command line arguments
# use Python Standard Library module argparse
#
# Input: none
#
# Actions:
# Uses the standard library argparse to process the command

line
#
def ParseCommandLine():

parser¼ argparse.ArgumentParser('Python gpsExtractor')

parser .add_argument('- v','--verbose', help¼"enables printing of additional
program messages", action¼'store_true')
parser.add_argument('- l','--logPath', type¼ ValidateDirectory, required¼True,
help¼"specify the directory for forensic log output file")
parser.add_argument('- c ','--csvPath', type¼ ValidateDirectory, required¼True,
help¼"specify the output directory for the csv file")
parser.add_argument('- d','--scanPath', type¼ ValidateDirectory, required¼True,
help¼"specify the directory to scan")

theArgs ¼ parser.parse_args() return theArgs

```

```
# End Parse Command Line def
ValidateDirectory(theDir):
# Validate the path is a directory
if not os.path.isdir(theDir):
raise argparse.ArgumentTypeError('Directory does not exist') # Validate the path
is writable
if os.access(theDir, os.W_OK):
return theDir
else:
raise argparse.ArgumentTypeError('Directory is not writable')
#End ValidateDirectory

import logging
#
# Class: _ForensicLog
#
# Desc: Handles Forensic Logging Operations
#
# Methods constructor: Initializes the Logger
# writeLog: # destructor: Writes a record to the log
Writes an information message and shuts down the logger

class _ForensicLog: def __init__(self, logName): try:
# Turn on Logging
logging.basicConfig(filename=logName, level=logging.DEBUG, format='%
(asctime)s %(message)s')
except:
print "Forensic Log Initialization Failure ... Aborting" exit(0)
def writeLog(self, logType, logMessage): if logType == "INFO":
logging.info(logMessage)

elif logType == "ERROR":
logging.error(logMessage)
elif logType == "WARNING":
logging.warning(logMessage)
else:
logging.error(logMessage)
return
```

```

def __del__(self):
    logging.info("Logging Shutdown")
    logging.shutdown()

#
# Data Extraction - Python-Forensics
# Extract GPS Data from EXIF supported Images (jpg, tiff)
# Support Module
#
import os # Standard Library OS functions from classLogging import
_ForensicLog # Abstracted Forensic Logging

Class
# import the Python Image Library # along with TAGS and GPS related TAGS
from PIL import Image
from PIL.EXIFTags import TAGS, GPSTAGS

#
# Extract EXIF Data
#
# Input: Full Pathname of the target image #
# Return: gps Dictionary and selected EXIFData list #

def ExtractGPSDictionary(fileName):

    try :
        pilImage = Image.open(fileName) EXIFData = pilImage._getEXIF()

    except Exception :
        # If exception occurs from PIL processing # Report the
        return None, None

    # Iterate through the EXIFData
    # Searching for GPS Tags

    imageTimeStamp = "NA"
    CameraModel = "NA"
    CameraMake = "NA"

    if EXIFData: for tag, theValue in EXIFData.items(): # obtain the tag

```


minutes_{1/4} 0.0

```
s0 ¼ gpsCoordinate[2][0] s1 ¼ gpsCoordinate[2][1] try:
```

```
seconds ¼ float(s0)/ float(s1) except:  
seconds ¼ 0.0
```

```
floatCoordinate ¼ float (degrees+(minutes / 60.0)+(seconds / 3600.0))  
return floatCoordinate
```

```
import csv #Python Standard Library - reader and writer for csv files #
```

```
# Class: _CSVWriter
```

```
#
```

```
# Desc: Handles all methods related to comma separated value operations #
```

```
# Methods constructor: Initializes the CSV File
```

```
# writeCSVRow: # writerClose: Writes a single row to the csv file
```

```
Closes the CSV File
```

```
class _CSVWriter: def __init__(self, fileName):
```

```
try :
```

```
# create a writer object and then write the header row self.csvFile ¼
```

```
open(fileName, 'w b')
```

```
self.writer¼ csv.writer(self.csvFile, delimiter¼',', quoting¼csv.QUOTE_ALL)
```

```
self.writer.writerow(('Image Path','M a k e','M o d e l','UTC Time',
```

```
'L a t R e f','Latitude','L o n R e f','Longitude','A l t R e f','Altitude')) except:  
log.error('CSV File Failure')
```

```
def writeCSVRow(self, fileName, cameraMake, cameraModel, utc, latRef,
```

```
latValue, lonRef, lonValue, altRef, altValue): latStr¼'%.8f'% latValue
```

```
lonStr¼'%.8f'% lonValue
```

```
altStr¼'%.8f'% altValue
```

```
self.writer.writerow(fileName, cameraMake, cameraModel, utc,
```

```
latRef, latStr, lonRef, lonStr, altRef, AltStr)
```

```
def __del__(self):
```

```
self.csvFile.close()
```

Program execution

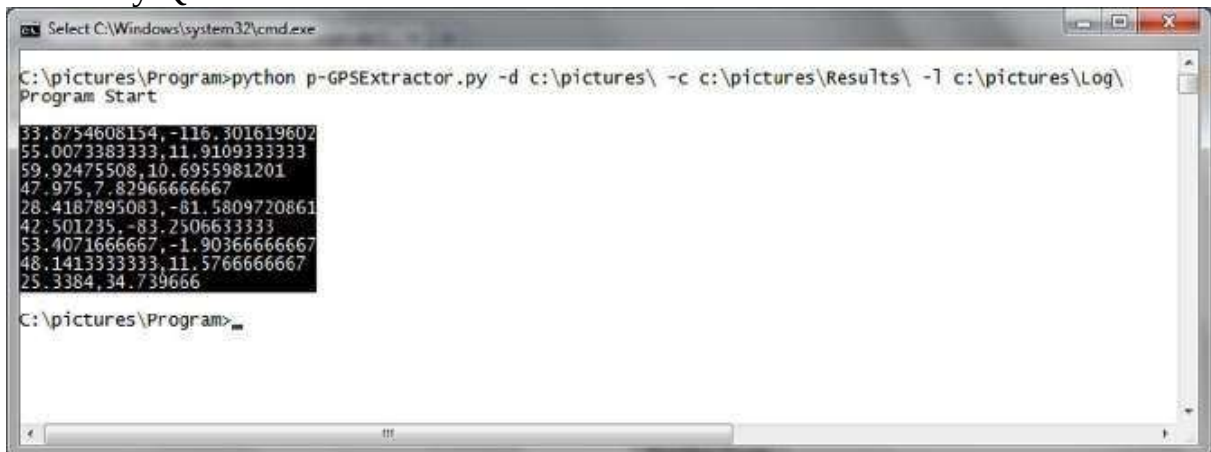
The p-gpsExtractor produces three separate results: (1) The standard output rendering of the latitude and longitude values extracted from the target files. (2)

The CSV file that contains the details of each EXIF and GPS extraction. (3) The forensic log file that contains the log of program execution. Execution of the program is accomplished by using the following command and line options.

```
C:\pictures\Program>Python p-GPSExtractor.py -d c:\pictures\ -c  
c:\pictures\Results\ -l c:\pictures\Log\
```

Figure 5.8 depicts the results of program execution. I have selected the output results and copied them into the paste buffer in order to plot all the points within an online mapping program.

Figure 5.8 contains a screenshot of a Windows execution of p-gpsExtractor. Summary Questions 159



```
Select C:\Windows\system32\cmd.exe  
C:\pictures\Program>python p-GPSExtractor.py -d c:\pictures\ -c c:\pictures\Results\ -l c:\pictures\Log\  
Program Start  
33.8754608154, -116.301619602  
55.0073383333, 11.9109333333  
59.92475508, 10.6955981201  
47.975, 7.82966666667  
28.4187895083, -81.5809720861  
42.501235, -83.2506633333  
53.4071666667, -1.90366666667  
48.1413333333, 11.5766666667  
25.3384, 34.739666  
C:\pictures\Program>
```

FIGURE 5.8
p-gpsExtractor.py execution.

In Figure 5.9, I paste the selected coordinates from the program execution into the online Web site, <http://www.darrinward.com/lat-long/> [Ward] and submit the values. The page plots each of the points on the map as a push pin. Figures 5.10 and 5.11 illustrate the zoom into the locations in Germany.

In addition to the mapped data, I also generated a .csv result file for the scan. Figure 5.12 shows the resulting .csv file.

Finally, I also create a Forensic Log file associated with program execution. This file contains the forensic log results shown in Figure 5.13.

CHAPTER REVIEW

In this chapter, I put to use new Python language elements including the

dictionary in order to handle more complex data types and to interface with the PIL. I also utilized the PIL to systematically extract EXIF data from photographs including GPS data when available with photographs. I demonstrated not only how to extract the GPS raw data but also how to calculate the lon and lat positions and then later integrated the output of the program into an online mapping program to plot the locations of the photographs. In addition, I demonstrated how to extract data from the EXIF data structure and included some of those values in a resulting .csv file that contains information about the file, camera, timestamp, and location information.

SUMMARY QUESTIONS

1. Based on the experimentation with the Python built-in data type dictionary, what limitations for key and value do you see?
2. Choose five additional fields from the EXIF structure, develop the additional code to extract them, locate several photographs that contain this additional data, and add them to the .csv output.

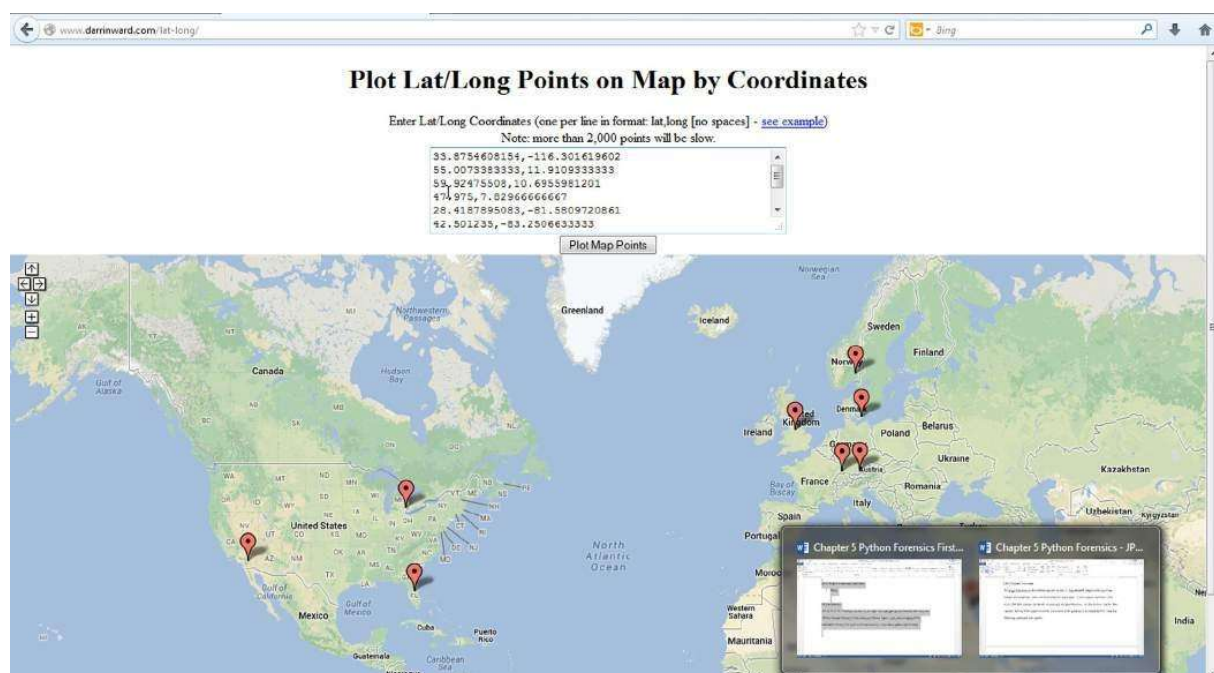


FIGURE 5.9
Mapping the coordinates extracted from photos.

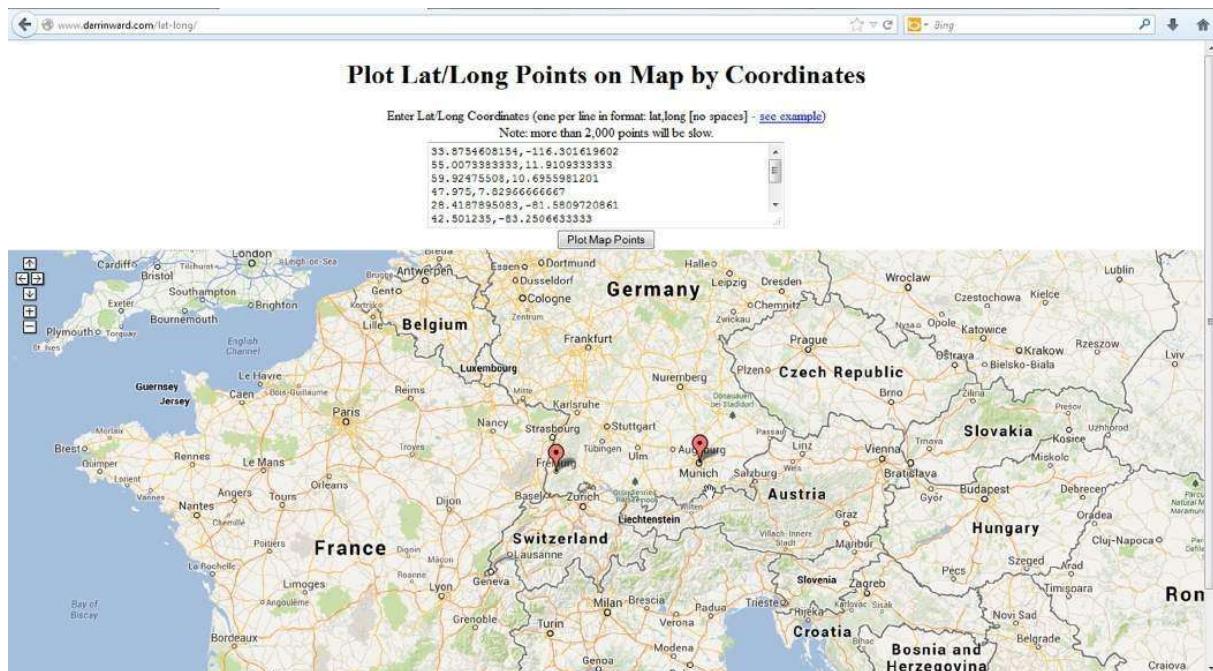


FIGURE 5.10

Map zoom into Western Europe.



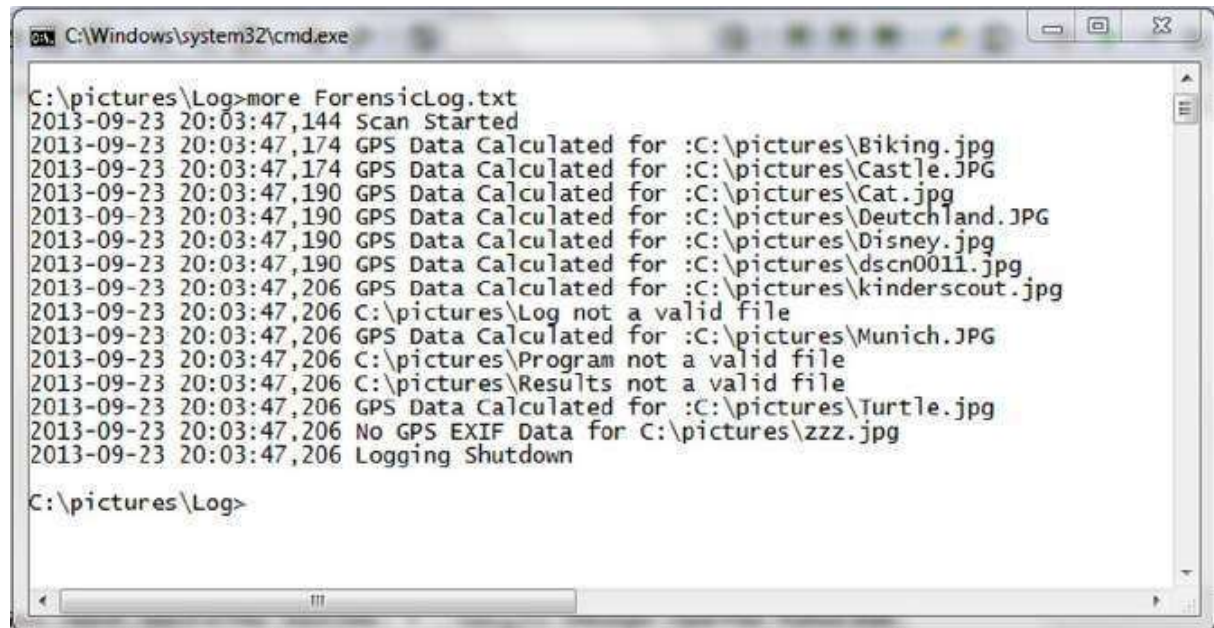
FIGURE 5.11
Map zoom to street level in Germany.

imageResults - Excel

	A	B	C	D	E	F	G	H	I
	Image Path	TimeStamp	Camera Make	Camera Model	Lat Ref	Latitude	Lon Ref	Longitude	
1	c:\pictures\Biking.jpg	2006:02:11 11:06:37	Canon	Canon PowerShot A80	N	33.87546002	W	-116.3016196	
2	c:\pictures\Castle.JPG	2012:06:09 12:42:24	PENTAX	PENTAX K-5		55.00733833		11.91093333	
3	c:\pictures\Cat.jpg	2008:08:05 20:59:32	Canon	Canon EOS 400D DIGITAL	N	59.92475508	E	10.69559812	
4	c:\pictures\Deutschland.JPG	2010:06:25 15:32:25	Apple	iPhone 3G	N	47.975	E	7.82966667	
5	c:\pictures\Disney.jpg	2010:08:18 11:38:37	Canon	Canon EOS 1000D	N	28.41878951	W	-81.58097209	
6	c:\pictures\dscn0011.jpg	2009:03:14 13:46:34	NIKON	COOLPIX P6000	N	42.501235	W	-83.25066333	
7	c:\pictures\kinderscout.jpg	2008:01:12 12:06:52	NIKON CORPORATION	NIKON D50	N	53.40716667	W	-1.90366667	
8	c:\pictures\Munich.JPG	2010:06:21 16:00:57	Apple	iPhone 3G	N	48.14133333	E	11.57666667	
9	c:\pictures\Turtle.jpg	2008:05:08 16:55:58	Canon	Canon EOS SD	N	25.3384	E	34.739666	
10									
11									
12									
13									
14									

FIGURE 5.12

Snapshot of Results.csv file.
Additional Resources 163



```
C:\Windows\system32\cmd.exe
C:\pictures\Log>more ForensicLog.txt
2013-09-23 20:03:47,144 Scan Started
2013-09-23 20:03:47,174 GPS Data Calculated for :C:\pictures\Biking.jpg
2013-09-23 20:03:47,174 GPS Data Calculated for :C:\pictures\Castle.JPG
2013-09-23 20:03:47,190 GPS Data Calculated for :C:\pictures\Cat.jpg
2013-09-23 20:03:47,190 GPS Data Calculated for :C:\pictures\Deutschland.JPG
2013-09-23 20:03:47,190 GPS Data Calculated for :C:\pictures\Disney.jpg
2013-09-23 20:03:47,190 GPS Data Calculated for :C:\pictures\dscn0011.jpg
2013-09-23 20:03:47,206 GPS Data Calculated for :C:\pictures\kinderscout.jpg
2013-09-23 20:03:47,206 C:\pictures\Log not a valid file
2013-09-23 20:03:47,206 GPS Data Calculated for :C:\pictures\Munich.JPG
2013-09-23 20:03:47,206 C:\pictures\Program not a valid file
2013-09-23 20:03:47,206 C:\pictures\Results not a valid file
2013-09-23 20:03:47,206 GPS Data Calculated for :C:\pictures\Turtle.jpg
2013-09-23 20:03:47,206 No GPS EXIF Data for C:\pictures\zzz.jpg
2013-09-23 20:03:47,206 Logging Shutdown
C:\pictures\Log>
```

FIGURE 5.13

Snapshot of the Forensic Log file.

3. Locate other online mapping resources (for example, ones that would allow you to tag the photo, for example, with a filename) and then use the mapping function to render the additional data.
4. Using the PIL, develop a program that will maliciously modify the latitude, longitude, and/or timestamp values of an existing photograph.
5. Expand the extraction method to include altitude and then find a mapping program that would allow you to map latitude, longitude, and altitude. Then locate photos with varying altitudes, for example, photos taken from hot air balloons.

Additional Resources

FCC 911 Wireless Service Guide. <http://www.fcc.gov/guides/wireless-911-services>. Python Imaging Library. <https://pypi.python.org/pypi/PIL>. The Python Standard Library. <http://docs.python.org/2/library/>. Web Based multiple coordinate mapping website. <http://www.darrinward.com/lat-long/>.

CHAPTER

Forensic Time 6

CHAPTER CONTENTS

Introduction	165
Adding Time to the Equation	167
The Time Module	169
The Network Time Protocol	173
Obtaining and Installing the NTP Library ntplib	174
World NTP Servers	177
NTP Client Setup Script	179
Chapter Review	181
Summary Questions	181
Additional Resources	181

INTRODUCTION

Before we can even consider time from an evidentiary point of view, we need to better understand what time is. I will start with an excerpt from one of my favorite books “Longitude” by Dava Sobel:

“Anyone alive in the eighteenth century would have known that the longitude problem was the thorniest scientific dilemma of the day and had been for centuries. Lacking the ability to measure their longitude, sailors throughout the great ages of the exploration had been literally lost at sea as soon as they lost sight of land.

The quest for a solution had occupied scientists and their patrons for the better part of two centuries when, in 1714, England’s Parliament upped the ante by

offering a king's ransom (£20,000, approximately \$12 million in today's currency) to anyone whose method or device proved successful. The scientific establishment throughout Europe from Galileo to Sir Isaac Newton had mapped the heavens in the both hemispheres in its certain pursuit of a celestial answer. In stark contrast, one man, John Harrison, dared to imagine a mechanical solution – a clock that would keep precise time at sea, something no clock had every been able to do on land. Mr. Harrison's invention resulted in a solution to the greatest scientific problem of this time.” [Sobel, 1995]

The basis of Harrison's clock and solution still guide navigation today as the use of time within GPS and other navigation systems remains the key underlying

Python Forensics 165 © 2014 Elsevier Inc. All rights reserved.

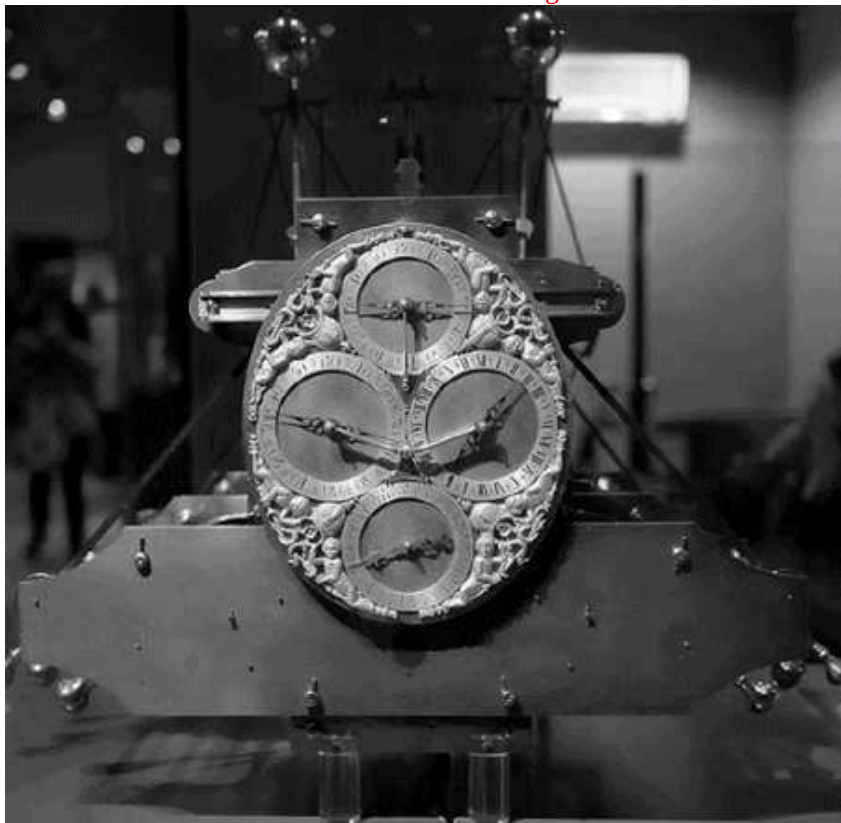


FIGURE 6.1

John Harrison H1 clock

component that allows us to calculate where we are. Figure 6.1 depicts the first Harrison sea clock H1.

Mr. Harrison's lifetime work was the impetus for the formation of Greenwich Mean Time or GMT. In today's high-speed global economy and communication infrastructure, precise, accurate, reliable, nonforgeable, and nonreputable time is

almost as elusive as it was in the eighteenth century. Of course, atomic, rubidium, and cesium clocks are guaranteed to deliver accurate time—and without missing a beat—for over a 1000 years. But, delivering that accuracy to digital evidence is elusive. My favorite saying is, “anyone can tell you what time it is, proving what time it was is still the challenge” [[Hosmer 2002](#)].

Time is a concept that allows society to function in an orderly fashion where all parties are able to easily understand the representation of time and agree on a sequence of events. Since 1972, the international standard for time is Coordinated Universal Time or UTC. UTC forms the basis for the official time source in most nations around the globe; it is governed by a diplomatic treaty adopted by the world community that designates National Measurement (or Metrology) Institutes (NMIs) as UTC time sources. The National Institute of Standards and Technology (NIST) in the United States, the Communications Research Laboratory (CRL) in Japan, and the National Physical Laboratory (NPL) in the United Kingdom are examples of NMIs. There are about 50 similar metrology centers that are responsible for official time throughout the world.

Time has been a critical element in business for hundreds of years. The time—via a mechanical record—established and provided evidence as to the “when” of Adding Time to the Equation 167

business transactions and events. (Usually recorded on paper, these time records provided an evidentiary trail due to a number of unique attributes—the distinctiveness of the ink embedded in the paper, the style and method of type, whether mechanical or hand written, the paper and its characteristics, and the detectability of modifications, insertions, or deletions). In the electronic world where documents and timestamps are simply bits and bytes, authoritative proof of the time of an event has been hard to nail down. Since the security of time is absolutely linked to its origination from an official source, the reliance on easily manipulated time sources (e.g., typical computer system clocks) is problematic.

Precision and reliability have long been the major factors driving the design of electronic timing technology. The next technology wave adds the dimensions of authenticity, security, and auditability to digital timestamps. These three attributes, when applied to documents, transactions, or any other digital entity, provide the following advantages over traditional computer clock-based timestamps:

Assurance that the time came from an official source Assurance that time has not been manipulated An evidentiary trail for auditing and nonrepudiation

Authentication (knowledge of the originator's identity) and integrity (protection against content modification) are essential elements in trusted transactions. But if one cannot trust the time, can the transaction really be trusted? Secure timestamps add the missing link in the trusted transaction equation: "when" did it happen? However, for the timestamp to provide a reliable answer to that question, the time upon which it is based must be derived from a trusted source.

ADDING TIME TO THE EQUATION

Using the best practices afforded to us, today digital signatures are able to successfully bind "who" (the signer) with the "what" (the digital data). However, digital signatures have shortcomings that leave two critical questions unanswered:

1. When did the signing of the digital evidence occur?
2. How long can we prove the integrity of the digital evidence that we signed?

For both of these questions, time becomes a critical factor in proving the integrity of digital evidence. Through the work of the Internet Engineering Task Force (IETF) and private companies, timestamping has advanced to a realistic deployment stage.

To understand this, we must first understand a little about time itself and what is necessary if we choose to utilize time as a digital evidence trust mechanism. From ancient societies to the present day, time has been interpreted in many ways. Time is essentially an agreement that allows society to function in an orderly fashion—where all parties are easily able to understand the representation. Some examples of time measurement are included in [Figure 6.2](#).

1967 1972 2009



1. Early calendars based on moon phases

2. Egyptians discover solar year
3. Atomic clocks utilized
4. UTC time
named official time
5. NIST and USNO official US time sources

FIGURE 6.2 A very brief history of time.

As shown in [Figure 6.2](#), the first calendars known to man were based upon the moon because everyone could easily agree on this as a universal measure of time. Moving forward, the Egyptians were the first to understand the solar year and they were able to develop a calendar based on the rotation of the earth around the sun. In 1967, an international agreement defined the unit of time as the second, measured by the decay of Cesium using precision instruments known as atomic clocks. And in 1972, the Treaty of the Meter (also known as the Convention of the Metre, established in 1875) was expanded to include the current time reference known as UTC, which replaced GMT as the world's official time.

Today, more than 50 national laboratories operate over 300 atomic clocks in order to provide a consistent and accurate UTC. Thus, in order to create a trusted source of time, you must first reference an official time source. In the United States, we have two official sources of time; the National Institute of Standards and Technology (NIST) in Boulder, CO for commercial time and the United States Naval Observatory (USNO) for military time.

Despite the broad accessibility of accurate time (watches, computer clocks, time servers, etc.), the incorporation of trusted time within a system requires a secure, auditable digital date/timestamp. Unless you have direct connection to the NMIs, most trusted sources today utilize GPS as a source of time. When used properly to tune Rubidium timeservers, the accuracy provided by GPS signals provides the exactness and precision that are required for most transactions and record keeping. This is true if and only if these requirements, standards, and processes are followed:

Accuracy : The time presented is from an authoritative source and is accurate to the precision required by the transaction, whether day, hour, or millisecond.

Authentication: The source of time is authenticated to a trusted source of time, such as an NMI timing lab, GPS signal, or the NIST long-wave standard time

signal (WWVB in the United States).

Integrity: The time should be secure and not be subject to corruption during normal “handling.” If it is corrupted, either inadvertently or deliberately, the corruption should be apparent to a third party.

Nonrepudiation: An event or document should be bound to its time so that the association between event or document and the time cannot be later denied.

Accountability: The process of acquiring time, adding authentication and integrity, and binding it to the subject event should be accountable so that a third party can determine that due process was applied, and that no corruption transpired.

Most people reading this book will quickly recognize that from a practical perspective the timestamps that we collect during digital investigations rarely meet the standards set forth above. However, the knowledge of the standards we are striving for is vital. This should cause you to question everything—especially anything that is related to time. To get started, let us take a deep dive into the Python Standard Library examining the time, datetime, and calendar modules.

THE TIME MODULE

The time module has many attributes and methods that assist in processing timerelated information. To get started, I will use the proven technique of “test before code” to experiment with the time module.

The first issue we encounter is the definition of “the epoch.” An epoch is defined as a notable event that marks the beginning of a period in time. Modern digital computers reference specific points in history, for example, most Unix-based systems select midnight on January 1, 1970, whereas Windows uses January 1, 1601; and Macintosh systems typically reference January 1, 1904.

The original Unix Epoch was actually defined on January 1, 1971 and was later revised to January 1, 1970. This epoch allows for a 32-bit unsigned integer to represent approximately 136 years, if each increment since the epoch is equivalent to 1 second. Thus by specifying a 32-bit unsigned integer as the number of seconds since January 1, 1970, you can mark any moment in time until February 7, 2106. If you are limited to signed 32-bit numbers, the maximum date is January 19, 2038.

As stated most system epochs start on January 1, 1970. The question is, how

would you know? The simple answer is to ask the Python time module to tell us using the gmtime() method, gmtime converts the number of seconds (provided as a parameter to the gmtime() method) into the equivalent GMT. To illustrate, let us pass 0 (zero) as the parameter and see what epoch date the method returns.

```
>> > import time
>>> print time.gmtime(0)
time.struct_time(tm_year¼1970, tm_mon¼1, tm_mday¼1, tm_hour¼0,
tm_min¼0, tm_sec¼0, tm_wday¼3, tm_yday¼1, tm_isdst¼0)
```

Similarly, if I wish to know the maximum time value (based upon a maximum limit of an unsigned 32-bit number), I would execute the following code.

```
>> > import time
>>> time.gmtime(0xffffffff)
time.struct_time(tm_year¼2106, tm_mon¼2, tm_mday¼7, tm_hour¼6,
tm_min¼28, tm_sec¼15, tm_wday¼6, tm_yday¼38, tm_isdst¼0)
```

Additionally, you might be wondering, how many seconds have passed since the epoch right now? To determine this, I use the time() method provided by the time module. This method calculates the number of seconds since the epoch.

```
>>> time.time() 1381601236.189
```

As you can see, when executing this command under the Windows version of the time() method, the time() method returns a floating point number not a simple integer, providing subsecond precision.

```
>> > secondsSinceEpoch¼time.time() >>> secondsSinceEpoch
1381601264.237
```

I can then use this time (converted to an integer first) as input to the gmtime() method to determine the date and time in Greenwich.

```
>> > time.gmtime(int(secondsSinceEpoch))
time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼12, tm_hour¼18,
tm_min¼7, tm_sec¼44, tm_wday¼5, tm_yday¼285, tm_isdst¼0)
```

The result is October 12, 2013 at 18 hours, 7 minutes, 44 seconds; the weekday is 5 (assuming that the week begins with 0¼Monday, this value would indicate

Saturday); this is the 285th day of 2013; and we are not currently observing daylight savings time. It should be further noted for complete clarity that this time represents UTC/GMT time not local time. Also, it is important to note that the time module is using the computer's system time and timezone settings to calculate the number of seconds since the epoch, not some magical time god. Therefore, if I wish to change what the now is I could do that by simply changing my system clock and then I rerun the previous script.

```
>> >secondsSinceEpoch¼time.time()
>>>secondsSinceEpoch
1381580629.793
>>>time.gmtime(int(secondsSinceEpoch))
time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼12, tm_hour¼12,
tm_min¼23, tm_sec¼49, tm_wday¼5, tm_yday¼285, tm_isdst¼0)
```

This results in our first successful forgery of time.
Now, if I compare GMT/UTC with local time I utilize both the gmtime() and localtime() methods.

```
>> >import time >>>now¼time.time() >>>now 1381670992.539
>>>time.gmtime(int(now))

time.struct_time(tm_year¼2013, tm_mon¼10, tm_mday¼13, tm_hour¼13,
tm_min¼29, tm_sec¼52, tm_wday¼6, tm_yday¼286, tm_isdst¼0)
>>>time.localtime(int(now)) time.struct_time(tm_year¼2013, tm_mon¼10,
tm_mday¼13, tm_hour¼9, tm_min¼29, tm_sec¼52, tm_wday¼6, tm_yday¼286,
tm_isdst¼1) As you can see this yields:
Local Time: Sunday October 13, 2013 09:29:52
GMT Time: Sunday October 13, 2013 13:29:52
```

Both the localtime() and gmtime() take into consideration many factors to determine the date and time, one important consideration is the timezone that my system is currently configured to:

I can use Python to provide me with the current timezone setting of my system, by using the following script. Note time.timezone is an attribute rather than a method so it can be read directly.

```
>> >import time >>>time.timezone 18000
```

Ok, what does 18,000 mean? We tend to think of timezones being a specific area or zone, which they are. We also tend to think each timezone is exactly 1 hour apart. This is not always true.

For example, some locations around the globe utilize 30-minute offsets from UTC/GMT.

Afghanistan

Australia (both Northern and Southern Territories)

India

Iran

Burma

Sri Lanka

Other locations use 15-minute offsets values related to UTC/GMT. Examples include:

Nepal

Chatham Island in New Zealand

Parts of Western Australia

The 18,000 represents the number of seconds west of UTC and points that are east of UTC are recorded in negative seconds. Since 18,000 represents the number of seconds west of UTC, I can easily calculate the number of hours west of UTC by dividing 18,000 seconds by 60 (seconds in a minute) and then again by 60 (minutes in an hour):

$18,000 \div 60 = 300$ minutes
 $300 \div 60 = 5$ hours

This means that my local time is 5 hours west of UTC/GMT. Going back to our comparison then, there should be 5 hours difference between localtime and GMT correct?

Local Time: Sunday October 13, 2013 09:29:52

GMT Time: Sunday October 13, 2013 13:29:52

Not quite! The difference between localtime() and gmtime() in this example only depicts 4 hours difference. It turns out that Sunday October 13, 2013 falls within daylight savings time here in the Eastern U.S. timezone as noted by `tm_isdst`, 1 found in the local time example, therefore the time difference on this particular date is 4 hours from GMT/UTC correctly reported by the two methods.

Next, you might be wondering how one determines the name of the current time zone set for my system. The time module provides an additional attribute specifically time.tzname that provides a tuple. The first value is the standard time zone designation while the second value is local daylight savings time zone designation.

Here is the example using my local time.

```
>> >import time
>>>time.tzname
('Eastern Standard Time','Eastern Daylight Time')
```

Note, this information is based upon the local operating system internal representation. For example, on a Mac, this returns (“EST,” “EDT”). Therefore, if we wish to print the current time zone designation of a local system, the following code will provide that for you.

```
import time
# Get the current local time
now=time.localtime()

# if we are observing daylight savings time # tm_isdst is an attribute that can be
examined, # if the value is 0 the current local time is in # standard time
observations and if tm_isdst is 1 # then daylight savings time is being observed

if now.tm_isdst:
# print the daylight savings time string
print time.tzname[1]
The Network Time Protocol 173
else:
# Otherwise print the standard time string print time.tzname[0]
```

Eastern Daylight Time

Another very useful method within the time module is the strftime(). This method provides great flexibility in generating custom strings from the base time structure provided. This allows us to format output without having to extract individual time attributes manually.

For example:

```
import time
print time.strftime("%a, %d %b %Y %H:%M:%S %p",time.localtime())
time.sleep(5)
print time.strftime("%a, %d %b %Y %H:%M:%S %p",time.localtime())
```

This short script prints out the local time value using the `strftime` method and uses the `sleep()` method to delay (in this example 5 seconds) and then prints another time string. The result produces the following output:

Sun, 13 Oct 2013 10:38:44 AM Sun, 13 Oct 2013 10:38:49 AM

The `strftime` method has great flexibility in the available options, which are shown in [Table 6.1](#).

As you can see the time module has a plethora of methods and attributes and is straightforward to use. For more information on the time module, you can check out the Python Standard Library [PYLIB].

THE NETWORK TIME PROTOCOL

Today, the most widely accepted practice for synchronizing time is to employ the Network Time Protocol (NTP). NTP utilizes the User Datagram Protocol (UDP) to communicate minimalistic timing packets communicated between a server (containing a highly accurate source of time) and a client wishing to synchronize with that time source. (The default server port for the NTP protocol is 123.) The NTP model includes not one, but many easily accessible time servers synchronized to national laboratories. The point of NTP is to convey or distribute time from these servers to clients via the Internet. The NTP protocol standard is governed by the IETF and the Proposed Standard is RFC 5905, titled “Network Time Protocol Version 4: Protocol and Algorithms Specification” [NTP RFC]. Many programs, operating systems, and applications have been developed to utilize this protocol to synchronize time. As you have probably guessed we are going to develop a simple time synchronization Python program and use this to synchronize forensic operations. Instead of implementing the protocol from scratch I am going to leverage a third-party Python Library `ntplib` to handle the heavy lifting and then compare the results to my local system clock.

Table 6.1 `strftime` Output Specification

Parameter Definition

%a Abbreviation of weekday string

%A Complete weekday string

%b Abbreviation of month string

%B Complete month name

%c Appropriate date and time representation based on locale
%d Day of the month as a decimal number 01-31

%H Hour (24-hour clock) as a decimal number 00-24

%I Hour (12-hour clock) as a decimal number 00-12

%j Day of the year as a decimal number 001-356

%m Month as a decimal number 01-12

%M Minute as a decimal number 00-59

%p AM or PM representation based on locale

%S Second as a decimal number 00-59

%U Week number of the year 00-53 (assumes week begins on Sunday)
%w Weekday 0-6 Sunday is 0

%W Week number of the year 00-53 (assumes week begins on Monday)

%x Date representation based on locale

%X Time representation based on locale

%y Year without century as a decimal number 00-99

%Y Year with century as a decimal number

%Z Time zone name

OBTAINING AND INSTALLING THE NTP LIBRARY ntplib

The ntplib is available for download at <https://pypi.python.org/pypi/ntplib/> as shown in Figure 6.3 and as of this writing it is currently version 0.3.1. The library provides a simple interface to NTP servers along with methods that can translate NTP protocol fields to text allowing easy access to other key values such as leap seconds and special indicators.



FIGURE 6.3

Python ntplib download page.

Obtaining and Installing the NTP Library ntplib 175

Installing the current library is a manual process, but still straightforward and because ntplib is written completely in native Python the library is compatible across platforms (Windows, Linux, and Mac OS X).

The first installation step is to download the installation package in this case a tar. gz file as shown in [Figure 6.4](#).

Once download you must decompress the tar.gz into a local directory. I have unzipped the archive into c:\Python27\Lib\ntplib-0.3.1 ([Figure 6.5](#)). Since I am utilizing Python 2.75, decompressing the download in this directory organizes the library with the other installed libs for easy access and update in the future. The installation process for manually installed libraries like ntplib is accomplished by executing the included setup program setup.py that is included in the directory. To perform the installation you simply open a command window as shown in [Figure 6.6](#) and enter the command:

Python setup.py install

The setup.py program performs all the necessary installation steps. Next, I always like to verify the installation of the new library by opening a Python Shell and importing the library and then by simply typing the name of the library. Python provides basic information regarding the library confirming it is available for use (see [Figure 6.7](#)).

As with any module you can always enter the dir(objName) built-in function to obtain details about the properties, classes, and methods that are available for the

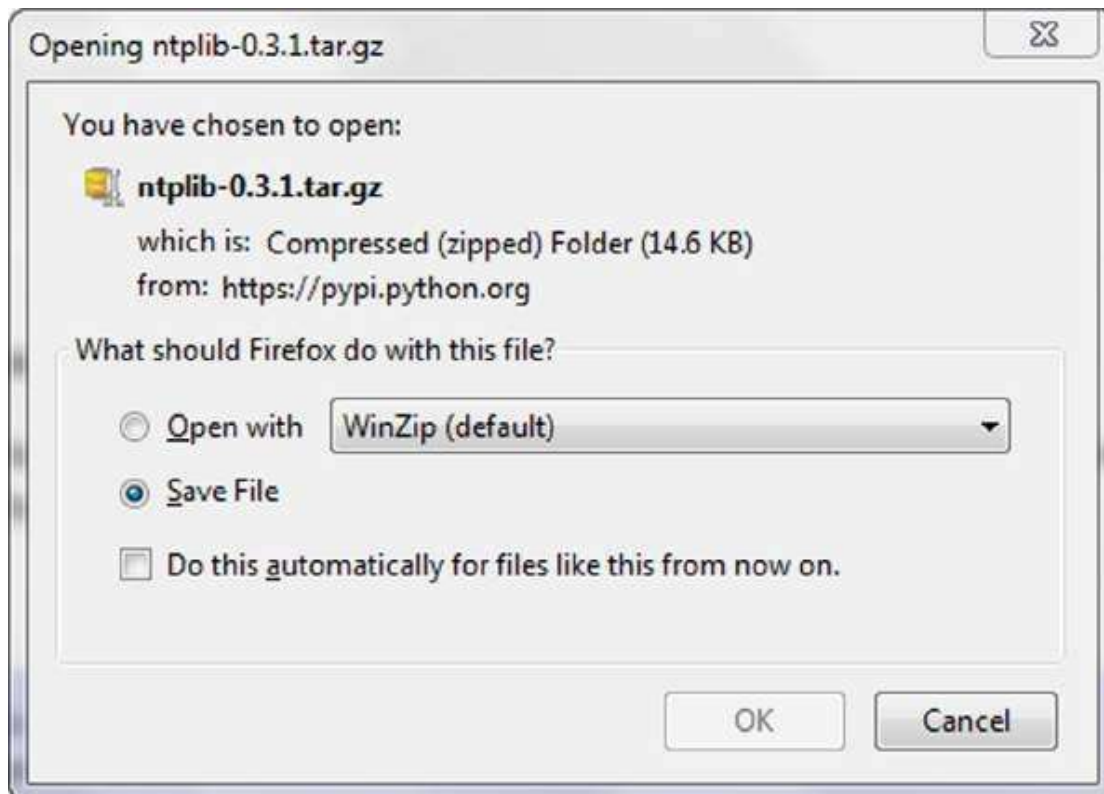


FIGURE 6.4
Download of ntplib-0.3.1.tar.gz.

```
C:\Windows\system32\cmd.exe

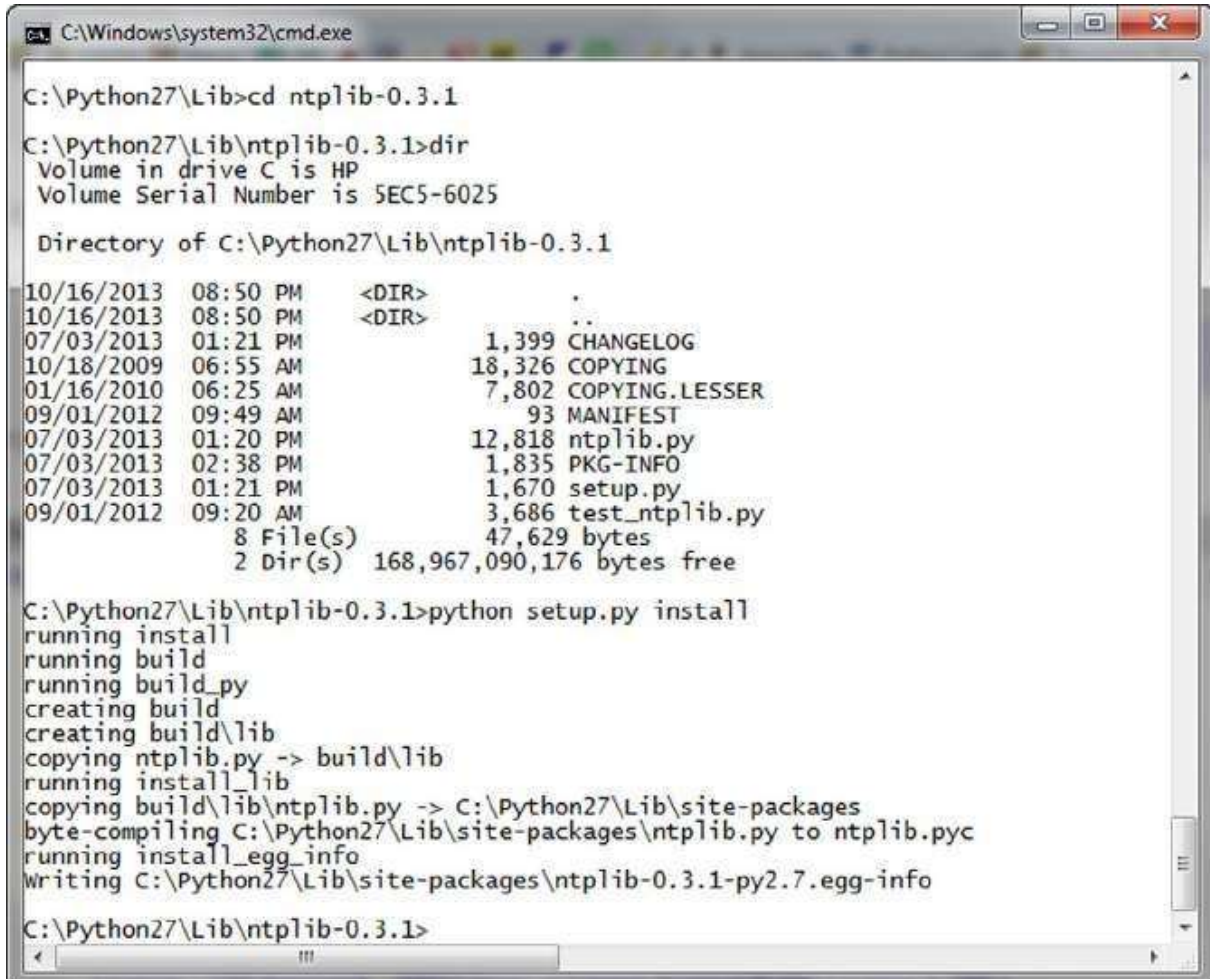
C:\Python27\Lib>cd ntplib-0.3.1
C:\Python27\Lib\ntplib-0.3.1>dir
Volume in drive C is HP
Volume Serial Number is 5EC5-6025

Directory of C:\Python27\Lib\ntplib-0.3.1

10/16/2013  08:50 PM    <DIR>          .
10/16/2013  08:50 PM    <DIR>          ..
07/03/2013  01:21 PM             1,399 CHANGELOG
10/18/2009  06:55 AM            18,326 COPYING
01/16/2010  06:25 AM             7,802 COPYING.LESSER
09/01/2012  09:49 AM                93 MANIFEST
07/03/2013  01:20 PM            12,818 ntplib.py
07/03/2013  02:38 PM             1,835 PKG-INFO
07/03/2013  01:21 PM             1,670 setup.py
09/01/2012  09:20 AM             3,686 test_ntplib.py
               8 File(s)          47,629 bytes
               2 Dir(s)  168,967,090,176 bytes free

C:\Python27\Lib\ntplib-0.3.1>
```

FIGURE 6.5
Decompressed ntplib-0.3.1.



```
C:\Windows\system32\cmd.exe

C:\Python27\Lib>cd ntplib-0.3.1
C:\Python27\Lib\ntplib-0.3.1>dir
Volume in drive C is HP
Volume Serial Number is 5EC5-6025

Directory of C:\Python27\Lib\ntplib-0.3.1

10/16/2013  08:50 PM    <DIR>          .
10/16/2013  08:50 PM    <DIR>          ..
07/03/2013  01:21 PM             1,399 CHANGELOG
10/18/2009  06:55 AM            18,326 COPYING
01/16/2010  06:25 AM             7,802 COPYING.LESSER
09/01/2012  09:49 AM              93 MANIFEST
07/03/2013  01:20 PM            12,818 ntplib.py
07/03/2013  02:38 PM             1,835 PKG-INFO
07/03/2013  01:21 PM             1,670 setup.py
09/01/2012  09:20 AM             3,686 test_ntplib.py
               8 File(s)          47,629 bytes
               2 Dir(s)  168,967,090,176 bytes free

C:\Python27\Lib\ntplib-0.3.1>python setup.py install
running install
running build
running build_py
creating build
creating build\lib
copying ntplib.py -> build\lib
running install_lib
copying build\lib\ntplib.py -> C:\Python27\Lib\site-packages
byte-compiling C:\Python27\Lib\site-packages\ntplib.py to ntplib.pyc
running install_egg_info
Writing C:\Python27\Lib\site-packages\ntplib-0.3.1-py2.7.egg-info

C:\Python27\Lib\ntplib-0.3.1>
```

FIGURE 6.6
Install ntplib.
World NTP Servers 177

```
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC
v.1500 32 bit (Intel)]
Type "help", "copyright", "credits" or "license" for
more information.
>>> import ntplib
>>> ntplib
<module 'ntplib' from
'C:\Python27\lib\site-packages\ntplib.pyc'>
>>> |
```

FIGURE 6.7

Verifying the installation.

```
>>> dir(ntplib)
['NTP', 'NTPClient', 'NTPException', 'NTPPacket',
'NTPStats', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '_to_frac', '_to_int',
'_to_time', 'datetime', 'leap_to_text', 'mode_to_text',
'ntp_to_system_time', 'ref_id_to_text', 'socket',
'stratum_to_text', 'struct', 'system_to_ntp_time', 'time']
>>>
>>>
>>> |
```

FIGURE 6.8

`dir(ntplib)` results.

object (see Figure 6.8). For even more information, you can use the `help(objName)` built-in function.

WORLD NTP SERVERS

Examining the methods and properties may seem a bit confusing at first; however, using the library module is in fact quite simple. Simply put we want to create an ntp client capable of accessing a specified ntp server to obtain a third-party source of time. In the United States, NIST manages a list of time servers that can be accessed to obtain “root” time (Figure 6.9).

Updated lists can be found at: <http://tf.nist.gov/tf-cgi/servers.cgi>. In Europe, you can find an active list of NTP servers at the NTP Pool Project. Figure 6.10 shows a screenshot from their home page at <http://www.pool.ntp.org/zone/europe>.

Name	IP Address	Location	Status
nist1-ny.ustiming.org	64.90.182.55	New York City, NY	
nist1-nj.ustiming.org	96.17.67.105	Bridgewater, NJ	All services available
nist1-nj2.ustiming.org	165.193.126.229	Westlawton, NJ	All services available
nist1-ny2.ustiming.org	216.171.112.36	New York City, NY	All services available
nist1-pa.ustiming.org	206.246.122.250	Hatfield, PA	All services available
time-a.nist.gov	129.6.15.28	NIST, Gaithersburg, Maryland	All services busy, not recommended
time-b.nist.gov	129.6.15.29	NIST, Gaithersburg, Maryland	All services busy, not recommended
time-c.nist.gov	129.6.15.30	NIST, Gaithersburg, Maryland	All services available
time-d.nist.gov	2610:20:6F15:15::27	NIST, Gaithersburg, Maryland	All services via IPv6
nist1-acl-va.symmetricom.com	64.236.96.53	Reston, Virginia	All services available
nist1-macon.macon.gsfc.us	98.175.203.200	Macon, Georgia	All services available
nist1-atl.ustiming.org	64.250.177.145	Athens, Georgia	All services available
uolinktime.com	207.232.123.18	Birmingham, Alabama	All services available
nist1-chi.ustiming.org	216.171.120.36	Chicago, Illinois	All services available

FIGURE 6.9
Partial list of NIST time servers.



News

How do I use pool.ntp.org?

How do I join pool.ntp.org?

Information for vendors

The mailing lists

Additional links

Translations

- Deutsch
- English
- Español
- Suomi
- Français
- 日本語
- 한국어
- Nederlands
- Português
- русский
- Svenska

 JOIN THE POOL
 USE THE POOL
 MANAGE SERVERS

Europe — europe.pool.ntp.org

To use this pool zone, add the following to your ntp.conf file:

```
server 0.europe.pool.ntp.org
server 1.europe.pool.ntp.org
server 2.europe.pool.ntp.org
server 3.europe.pool.ntp.org
```

IPv4

There are 2084 active servers in this zone.

2092 (-8) active 1 day ago

2089 (-5) active 7 days ago

2101 (-17) active 14 days ago

2080 (+4) active 60 days ago

1992 (+92) active 1 year ago

1342 (+742) active 3 years ago

829 (+1255) active 6 years ago

See all zones in [Global](#).

Austria — at.pool.ntp.org (78)

Switzerland — ch.pool.ntp.org (119)

Germany — de.pool.ntp.org (702)

Denmark — dk.pool.ntp.org (52)

Spain — es.pool.ntp.org (22)

France — fr.pool.ntp.org (297)

Italy — it.pool.ntp.org (40)

Luxembourg — lu.pool.ntp.org (25)

Netherlands — nl.pool.ntp.org (225)

IPv6

There are 617 active servers in this zone.

619 (-2) active 1 day ago

613 (+4) active 7 days ago

637 (-20) active 14 days ago

606 (+11) active 60 days ago

468 (+149) active 1 year ago

FIGURE 6.10
European NTP Pool Project.

For those of you wishing to obtain your time from the U.S. Naval Observatory or

USNO, you might be interested to know that for many years, the USNO has provided access to NTP servers' aptly named tick and tock. For more information on the U.S. Naval Observatory you can visit:

<http://www.usno.navy.mil/USNO>.

NTP Client Setup Script 179

NTP CLIENT SETUP SCRIPT

To setup a Python ntp client using the ntplib is a simple process: import ntplib # import the ntplib

import time # import the Python time module

url of the closest NIST certified NTP server

NIST¼'nist1-macon.macon.ga.us'

Create NTP client object using the ntplib

ntp¼ntplib.NTPClient()

initiate an NTP client request for time

ntpResponse¼ntp.request(NIST)

Check that we received a response

if ntpResponse:

obtain the seconds since the epoch from response

nistUTC¼ntpResponse.tx_time

print'NIST reported seconds since the Epoch :', print nistUTC

else:

print'NTP Request Failed'

Program Output

NIST reported seconds since the Epoch : 1382132161.96

Now that we can obtain the seconds since the epoch, we can use the Python Standard Library time module to display the time in either local or GMT/UTC time, and can even compare the current NTP time to our own local system clock as shown below.

CODE LISTING 6.1

import ntplib import time

NIST¼'nist1-macon.macon.ga.us'

ntp¼ntplib.NTPClient()

ntpResponse¼ntp.request(NIST)

```

if (ntpResponse):
    now¼time.time()
    diff¼now-ntpResponse.tx_time print'Difference :',

Continued print diff,
print'seconds'

print'Network Delay:', print ntpResponse.delay
print'UTC: NIST :¼ptime.strftime("%a, %d %b %Y %H:%M:%S +0000",
time.gmtime(int(ntpResponse.tx_time)))
print 'UTC: SYSTEM : ¼ptime.strftime("%a, %d %b %Y %H:%M:%S +0000",
time.gmtime(int(now)))
else:
print'No Response from Time Service'

```

Program Output

```

Difference : 3.09969758987 seconds Network Delay : 0.0309739112854
UTC NIST : Fri, 18 Oct 2013 21:48:48 +0000 UTC SYSTEM : Fri, 18 Oct 2013
21:48:51 +0000

```

Notable observations:

(1) It is important that you obtain the local system time immediately after obtaining the time from the time server (in my case from NIST):

```

ntpResponse¼ntp.request(NIST) if (ntpResponse):
    now¼time.time()

```

This ensures that our comparison with the local time maintained by our system clock is influenced by the smallest amount of processing.

(2) It is also important to consider the network delay reported by the NTP client. In this example, the delay was a little over 30 milliseconds (specifically 0.0309739112854 seconds), however, in certain situations the delay can be much longer based on the time of day, your network connection speed, and other network latency issues.

(3) In this example, I calculated a difference of 3.09969758987 seconds. This means that my local system clock is running 3½seconds faster than that of the NIST server.

```

diff¼now-ntpResponse.tx_time

```

If the resulting value was negative instead of positive, I can conclude that my system clock is running behind that reported by

NIST.

(4) Finally, the ntplib client operations are synchronous. In other words, once I make the request:

`ntpResponse¼ntp.request(NIST)` the code does not continue processing until the request is fulfilled or fails just like any method or function call.

Additional Resources 181

CHAPTER REVIEW

In this chapter, I explained the basics of time and computing. This includes a better understanding of the epoch, the origins of modern time keeping along with a bit of history on time. I then took a deep dive into the Standard Library module `time` and I explained and demonstrated many methods and properties associated with the module. I developed several short Python scripts to give you a feel for how to apply and interpret the results of time processing.

I then provided an overview of the NTP that provides us with the ability to synchronize time with National Measurement sources. Next, I walked you through the installation and setup of a Python NTP Library `ntplib`. I then experimented with the module to setup a NTP client that interfaces with a network-based time source. I wrote a script that would compare and calculate the difference between my local system clock and that of a NIST time server.

SUMMARY QUESTIONS

1. Back in [Chapter 3](#), I developed a program that walked the files system, hashed the files, and recorded the modified, access, and created (MAC) times of each file. Modify that program using the `time` module to convert the MAC times from local time to GMT/UTC time values.

2. Expanding on Code Listing 6.1, choose five additional time servers throughout the world and devise a method to compare and contrast the time reported by each in relationship to your local system clock. Also, record the network delay values from each. Note, you should execute multiple runs and average the results.

3. What other time sources should be carefully examined and/or normalized during a forensic investigation?

Additional Resources

Proving the integrity of digital evidence with time. *Int J Digital Evid* 2002;1(1)

1–7.

The Python Standard Library. <http://docs.python.org/2/library/time.html?highlight¼time#time>.

The Network Time Protocol Version 4: Protocol and Algorithms Specification. <http://tools.ietf.org/html/rfc5905>.

Longitude. Longitude: the true story of a Lone Genius who solved the greatest scientific problem of his time. New York, NY: Walker & Co.; 1995.

CHAPTER

Using Natural Language Tools in Forensics 7

CHAPTER CONTENTS

What is Natural Language Processing?	183
Dialog-Based Systems	184
Corpus	184
Installing the Natural Language Toolkit and Associated Libraries	185
Working with a Corpus	185
Experimenting with NLTK	186
Creating a Corpus from the Internet	193
NLTKQuery Application	194
NLTKQuery.py	195
_classNLTKQuery.py	196
_NLTKQuery.py	198
NLTKQuery Example Execution	199
NLTK Execution Trace	199
Chapter Review	202
Summary Questions	202
Additional Resources	

WHAT IS NATURAL LANGUAGE PROCESSING?

Before I begin, Natural Language needs to be defined. Back in 1987, I read the book *Introduction to Natural Language Processing* as part of a research project I was working on with Dr. James Cook at the Illinois Institute of Technology Research Institute (IITRI). My wife Janet was also working on Natural Language projects at IITRI with Dr. Larry Jones. Her work employed Natural Language as part of a decision engine that could diagnose and better understand the behaviors of cattle. The book we referenced had the best definitions that I have seen for Natural Language, therefore I will share that excerpt with you here.

“Natural Language is any language that humans learn from their environment and use to communicate with each other. Whatever the form of the communication, Natural Languages are used to express our knowledge and emotions and to convey our responses to other people and to our surroundings.” [Harris]

Python Forensics 183 © 2014 Elsevier Inc. All rights reserved.

Obviously, like all good definitions they stand the test of time. Since the Internet really did not materialize into a recognizable form until the late 1980s, the concept of expressing emotion and knowledge digitally did not include Facebook, Twitter, Emoticons, Blogging, Texting, Sexting, BitStrips, or even Internet-based e-mail! However, the definition certainly provides coverage for these latest forms of Natural Language expression. Extracting meaning (semantics) from Natural Language is quite a bit more difficult than it looks.

Judging the effectiveness of Natural Language systems that attempt to interpret, understand, and respond accordingly falls to the Turing Test, named after Alan Turing based on his 1950 paper “Computing Machinery and Intelligence.” In his paper, Turing asks the proactive question, “Can machines think?” [Turing]

Dialog-based systems

The goal of dialog-based systems is to respond to a user input naturally, meaning that we cannot distinguish between human or machine-based responses. Some may be familiar with systems that are moving toward this capability. For example, here is a dialog between me and the Sync system in my car:

Me: I'm hungry.

Sync: Ok, Where would you like to search nearby, along your route or in a city?

Me: Nearby.

Sync: How would you like to search—by name or by category? Me: By category.

Sync: Ok, what category?

Me: Italian.

Sync: Ok, please select from the list on your screen, for example say Line 1. Me: Line 4.

Sync: Ok, Rossi's Restaurant. Would you like to make a call or set as destination? Me: Make a call.

Sync: Ok: Calling Rossi's Restaurant.....

Although useful, it certainly does not meet the Turing standard because I know that I am talking to a computer not a human. However, Sync does translate the spoken word into text and then responds in a timely fashion and as a result the system usually gets me where I want to go or provides me the information I am seeking.

Corpus

One of the key elements needed to perform Natural Language Processing (NLP) is a corpus. A corpus is a large volume of linguistic samples that is relevant to a specific genre. For example, the Guttenberg Project [Gutenberg] that is available at [http:// www.gutenberg.org/](http://www.gutenberg.org/) contains over 40,000 books in electronic form.

One of the challenges in creating meaningful examples in this chapter was establishing a corpus. Then I needed to demonstrate the use of the corpus to create these

Working with a Corpus 185

examples that will serve as a catalyst for the development of new applications of NLP within the field of forensic science. Bear in mind, depending on the application you intend for NLP, more than one genre may be necessary.

INSTALLING THE NATURAL LANGUAGE TOOLKIT AND ASSOCIATED LIBRARIES

As with other third-party libraries and modules, installation is available for Windows, Linux, and Mac. The Natural Language Toolkit (NLTK) library is free

and can be easily obtained online, from nltk.org. Installation of NLTK does require the installation of other dependency libraries including Numpy and PyYAML. Figure 7.1 depicts the nltk.org installation page with easy to follow instructions. Once you have installed everything, you can verify the installation through your favorite Python Shell by typing:

```
Import NLTK
```

WORKING WITH A CORPUS

The first step in working with a corpus is to either load an existing corpus that is included with NLTK or create your own from local files or the Internet. Once this is done, there are a variety of powerful operations you can perform on the corpus,



FIGURE 7.1

nltk.org Installation url.

Table 7.1 Introductory List of NLTK Operations

Method Description `raw()`

`word_tokenize()`

`collocations()`

concordance()
findall(search)
index(word)
similar(word)

vocab() Extracts the raw contents of the corpus. The method returns the type: Python “str”

Removes white space and punctuation and then creates a list of the individual tokens (basically words). The method returns the type: Python “list”

Words often appear consecutively within a text and these provide useful information to investigators. In addition, once detected they can be used to find associations between word occurrences. This is accomplished by calculating the frequencies of words and their connections with other words. The collocations method performs all of this in a single method

This method provides the ability to generate every occurrence of a specific word along with the context (how the word was used in a specific sentence, for example)

This method can be used for simple and even regular expression searches of the corpus text

This method provides an index to the first occurrence of the word provided

Note this is not synonyms, rather the method provides distribution similarities, stated simply it identifies other words which appear in the same “context” as the specified word

Produces a comprehensive vocabulary list of the text submitted

operations that would be both complicated and time-consuming without the aid of a toolset like NLTK. [Table 7.1](#) depicts some of the key operations that can be performed on a corpus, I will be demonstrating these in code later in this chapter.

EXPERIMENTING WITH NLTK

In this section, I am going to invoke the methods in [Table 7.1](#) and create a simple program to first create a new text corpus, and then examine that corpus with these methods. First, a little background on the corpus I am going to create.

Jack Walraven maintains the web site <http://simpson.walraven.org/>. At that site, Jack has organized the transcripts of the O.J. Simpson trial. For this experiment, I downloaded the Trial Transcripts for January 1995 (you can of course download as many months and excerpts as you like). There are nine Trial

Transcript text files stored there for the January 1995 proceedings, and they represent transcripts for the dates of January 11-13, 23-26, 30, and 31. I created a directory on my Windows system, c:\simpson\ , to hold each of the separate files. My goal is to use these files to create a new text corpus that includes all of these files. The best way to illustrate how this works is to walk you through some sample code that deals with each aspect. I have annotated the code to make sure you understand each step of the process.

```
# NLTK Experimentation This first section imports the modules necessary for
the experiment. The most important is the newly installed NLTK module
from __future__ import division
import nltk
```

Next, I'm going to import the PlaintextCorpusReader from the NLTK module. This method will allow me to read in and ultimately create a text corpus that I can work with.

```
from nltk.corpus import PlaintextCorpusReader
```

The first thing I need to specify is the location of the files that will be included in the corpus. I have placed those files in c:\simpson\. Each of the 9 files are stored there and each is a text document.

```
rootOfCorpus = "c:\\simpson\\"
```

Now I'm going to use the PlaintextCorpusReader method to collect all the files in the directory I specified in the variable rootOfCorpus. I specify all the files by the second parameter '*'. If you have directories that had multiple file types and you only wanted to include the text documents you could have specified '.txt'. The result of this call will be an NLTK Corpus Object named newCorpus.

```
newCorpus = PlaintextCorpusReader(rootOfCorpus, '*')
print type(newCorpus)
```

The print type(newCorpus) produces the following output providing me with the NLTK type of the newCorpus.

```
<class'nltk.corpus.reader.plaintext.PlaintextCorpusReader'> I can also print the
file identification of each file that makes up the new corpus by using the fileids()
method.
print newCorpus.fileids()
```

```
[ 'Trial-January-11.txt','Trial-January-12.txt','Trial-January-13.txt', 'Trial-January-23.txt','Trial-January-24.txt','Trial-January-25.txt', 'Trial-January-26.txt','Trial-January-30.txt','Trial-January-31.txt']
```

I can also, determine the absolute paths of the individual documents contained in the new corpus, by using the `abspaths()` method.

```
print newCorpus.abspaths()
```

```
[FileSystemPathPointer('c:\\simpson\\Trial-January-11.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-12.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-13.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-23.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-24.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-25.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-26.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-30.txt'),  
FileSystemPathPointer('c:\\simpson\\Trial-January-31.txt')]
```

Now I can leverage the `newCorpus` object to extract the raw text which represents the complete collection of files contained in the `rootOfCorpus` directory specified. I can then determine the length or size of the combination of the 9 trial transcript files.

```
rawText = newCorpus.raw()  
print len(rawText)  
This produces the output:  
2008024
```

I can now apply the `nlk.Text` module methods in order to better understand and interpret the content of the corpus. I start this process by tokenizing the `rawText` of the corpus. This produces tokens (mainly words, but also includes numbers and recognized special character sequences).

```
tokens = nltk.word_tokenize(rawText)
```

Tokenization creates a standard Python list that I can now work with using both standard Python language operations and of course any methods associated with list objects. For example I can use `len(tokens)` to determine the number of tokens that were extracted from the text.

```
print len(tokens)
```

Which produces the result for this corpus of 401 ,032 tokens.

```
401032
```

Since this is now a simple list , I can display some or a portion of the contents. In this example I print out the first 100 list elements representing the first 100 tokens returned from the tokenization process.

```
print tokens[0:100]
```

```
[
'*LOS','ANGELES','CALIFORNIA','WEDNESDAY','JANUARY','11','','1
'9:05','A.M.*','DEPARTMENT','NO.','103','HON.','LANCE','A.','ITO','','JUDGE',
'APPEARANCES',:','('APPEARANCES','AS','HERETOFORE','NOTED','','DE
'DISTRICT','ATTORNEY','HANK','ALSO','PRESENT','ON','BEHALF','OF','S
','','MS.','PAMELA','W.','WITHEY','','ATTORNEY-AT-LAW.'),'('JANET','M.',
'MOXHAM','','CSR','NO.','4855','','OFFICIAL','REPORTER.'),'('CHRISTINE
'M.','OLSON','','CSR','NO.','2378','','OFFICIAL','REPORTER.'),'('THE',
'FOLLOWING','PROCEEDINGS','WERE','HELD','IN','OPEN','COURT','','OUT
'PRESENCE','OF','THE','JURY',:',')*THE','COURT',:','*','ALL','RIGHT.','THE'
'SIMPSON']
```

Next , I want to create a text that I can apply the NLTK text methods to. I do this by creating NLTK Text Object textSimpson using the Text method to the tokens extracted by the tokenization process.

```
textSimpson = nltk.Text(tokens)
print type(textSimpson)
```

As expected the object type is nltk .text.Text <class'nltk.text.Text'>

I would like to know the size of the vocabulary used across the corpus .I know the number of tokens, but there are of course duplicates. Therefore, I use the Python set method to obtain the unique set of vocabulary present in the text in the trial transcript.

```
vocabularyUsed = set(textSimpson)
print len(vocabularyUsed)
12604
```


Which produces 12,604 unique tokens from the total of 401,032 a much more manageable number. From this I would like to see all the unique tokens, and I can accomplish that by sorting the set.

```
print sorted(set(textSimpson))
```

```
'ABERRANT','ABIDE','ABIDING','ABILITIES','ABILITIES.','ABILITY','ABIL  
'ABLAZE','ABLE','ABOUT','ABOUT.','ABOVE','ABRAHAM','ABROGATE','A  
'ABRUPT','ABRUPTLY','ABSENCE','ABSENCE.','ABSENT','ABSOLUTE','AF  
'ABSOLUTELY.','ABSORB','ABSTRACT','ABSTRACT.','ABSURD.','ABSURD  
'ABUDRAHM.','ABUNDANCE','ABUNDANT','ABUNDANTLY','ABUSE','AB  
'BATTERED','ABUSED','ABUSER','ABUSER.','ABUSES','ABUSES.','ABUSIN  
'ABUSIVE.'
```

```
...
```

```
... skipping about 10,000 tokens
```

```
...
```

```
'WRITTEN','WRITTEN.','WRONG','WRONG.','WRONGFULLY','WRONGLY',  
'YAMAUCHI','YAMAUCHI.','YARD','YARD.','YARDS','YEAGEN','YEAH','YE  
'YEAR.','YEARS','YEARS.','YELL','YELLED','YELLING','YELLING.','YELL  
'YES','YES.','YESTERDAY','YESTERDAY.','YET','YET.','YIELD','YORK','YO  
"YOU'LL", "YOU'RE",  
"YOU'VE",'YOU.','YOUNG','YOUNGER','YOUNGSTERS','YOUR',  
'YOURS.','YOURSELF','YOURSELF.','YOURSELVES','YOUTH','YUDOWITZ  
'Z','ZACK','ZACK.','ZEIGLER','ZERO','ZLOMSOWITCH',  
"ZLOMSOWITCH'S", 'ZLOMSOWITCH.','ZOOM'
```

Next, I can determine how many times a particular word appears in the Trial Transcript. I accomplish using the count() method of the nltk.Text object type.

```
myWord = "KILL"
```

```
textSimpson.count(myWord)
```

```
84
```

This produces a count of the 84 times that the word KILL appeared in the Trial Transcription for the 9 days included in this corpus, during the month of January 1995. Up to this point all of this is pretty straight-forward. Let's take a look at a couple of the more advanced methods included in the nltk.Text Module. I'm going to start with the collocations method. This will provide me with a list of

words that occur together statistically often in the text.

```
print textSimpson.collocations()
```

Building collocations list

*THE COURT; *MS. CLARK; *MR. COCHRAN; MR. SIMPSON; NICOLE BROWN; *MR.DARDEN; OPENING STATEMENT; LOS ANGELES; MR. COCHRAN; DETECTIVE FUHRMAN; DISCUSSION HELD; WOULD LIKE; *MR. DOUGLAS; BROWN SIMPSON;THANK YOU.; MR. DARDEN; DEPUTY DISTRICT; FOLLOWING PROCEEDINGS; DISTRICT ATTORNEYS.; MISS CLARK

As you can see , the collocation list generated makes a great deal of sense. These pairings would naturally occur more often than others during the proceedings. Next, I want to generate a concordance from the transcript for specific words that might be of interest. NLTK will produce each occurrence of the word along with context (in other words the sentence surrounding the concordance word). A second optional parameter indicates the size of the window of surrounding words you would like to see. I will demonstrate a couple examples.

I will start with the word “KILL”, this produces 84 matches and I have included the first 6.

```
myWord = "KILL"
```

```
print textSimpson.concordance(myWord)
```

Displaying 6 of 84 matches:

WAS IN EMINENT DANGER AND NEEDED TO KILL IN SELF-DEFENSE.
BUT THE ARIZONA COURT

R OCCURRED. "I KNOW HE'S GOING TO KILL ME. I WISH HE WOULD
HURRY UP AND GET FLICKED HARM WAY BEYOND NECESSARY TO
KILL THE VICTIM. AND THIS IS A QUOTE FROM

'M GOING TO HURT YOU , I'M GOING TO KILL YOU , I'M GOING TO
BEAT YOU."THO

HAVE HER AND NO ONE ELSE WILL IS TO KILL HER. THAT IS CLEAR
IN THE RESEARCH.

WAS A FIXED PURPOSE , FULL INTENT TO KILL , WHERE THE
DEFENDANT LITERALLY WENT

If I use the word “GLOVE”, the code produces 92 matches; I have included the

first 6:

```
myWord¼ "GLOVE"
```

```
print textSimpson.concordance(myWord)
```

Displaying 6 of 92 matches:

CE DEPARTMENT PLANTED EVIDENCE , THE GLOVE AT
ROCKINGHAM. NOW , THAT OF COURSE
HAT A POLICE DETECTIVE WOULD PLANT A GLOVE , AND IT MADE
HOT NEWS AND THE DEFEN
R THIS POLICE DETECTIVE PLANTED THIS GLOVE AT ROCKINGHAM.
NOW , YOUR HONOR , BE
DETECTIVE FUHRMAN'S RECOVERY OF THE GLOVE AND AS –
INSOFAR AS IT RELATES TO T
HEY SAW THE LEFT-HANDED BULKY MAN'S GLOVE THERE AT THE
FEET OF RONALD GOLDMAN.
E BUNDY CRIME SCENE. THEY SAW A LONE GLOVE , A SINGLE
GLOVE AT THE FEET OF RONA

NLTK also provides more sophisticated methods that can operate on text corpus . For example the ability to identify similarities of word usage. The method identifies words that are used within similar contexts.

```
myWord¼ "intent"
```

```
print textSimpson.similar(myWord)
```

Building word-context index ..

time cochran court evidence and house it blood defense motive that jury other
people the this witnesses case defendant discovery

if I change the word to “victim” the results are as follows:

Building word-context index ..

court defendant jury defense prosecution case evidence and people record police
time relationship house question statement tape way crime glove

I can also utilize NLTK to produce a comprehensive vocabulary list and frequency distribution that covers all the tokens in the corpus. By using this method across documents, one can derive tendencies of the writer or creator of a document. The vocab method returns an object based on the class `nltk.probability.FreqDist` as shown here.

```
simpsonVocab¼ textSimpson.vocab()
type(simpsonVocab)
<class'nltk.probability.FreqDist'>
```

Thus the `simpsonVocab` object can now be utilized to examine the frequency distribution of any and all tokens within the text. By using just one of the methods, for example, `simpsonVocab.items()`, I can obtain a sorted usage list (most used first) of each vocabulary item.

```
simpsonVocab.items()
<bound method FreqDist.items of <FreqDist with 12604 samples and 401032
outcomes>>
```

```
[('THE', 19386), ('', 18677), ('TO', 11634), ('THAT', 10777), ('AND', 8938),
(':', 8369), ('OF', 8198), ('*', 7850), ('IS', 6322), ('I', 6244), ('A', 5590), ('IN',
5456), ('YOU', 4879), ('WE', 4385), ('THIS', 4264), ('IT', 3815), ('COURT',
3763), ('WAS', 3255), ('HAVE', 2816), ('-', 2797), ('?', 2738), ('HE', 2734),
('S', 2677), ('NOT', 2417), ('ON', 2414), ('THEY', 2287), ('*THE', 2275),
('BE', 2240), ('ARE', 2207), ('YOUR', 2200), ('WHAT', 2122), ('AT', 2112),
('WITH', 2110),
```

```
...
... Skipping to the bottom of the output for brevity
...
```

```
('WRENCHING', 1), ('WRESTLING.', 1), ('WRISTS.', 1), ('WRITERS', 1),
('WRONGLY', 1), ('X-RAYS', 1), ('XANAX', 1), ('XEROXED.', 1),
('XEROXING.', 1), ('YAMAUCHI.', 1), ('YARD.', 1), ('YARDS', 1),
('YEAGEN', 1), ('YELL', 1), ('YELLING.', 1), ('YEP.', 1), ('YIELD', 1),
('YOUNGSTERS', 1), ('YOURS.', 1), ('YUDOWITZ', 1), ('YUDOWITZ.', 1),
('Z', 1), ('ZEIGLER', 1), ('ZERO', 1), ("ZLOMSOWITCH'S", 1)]
```

Note: A nice by-product of the `vocab` method is when different vocabulary items have the same number of occurrences, they are sorted alphabetically.

Creating a Corpus from the Internet 193
 CREATING A CORPUS FROM THE INTERNET

In many cases, the text that we wish to examine is found on the Internet. By combining Python, the Python Standard Library `urllib`, and NLTK we can

perform the same types of analysis of online documents. I will start with a simple text document downloaded from the Guttenberg Project to illustrate.

I start by importing the necessary modules in this case NLTK and the urlopen module from the urllib. Note: As you can see, I can selectively bring in specific modules from a library instead of loading the entire library.

```
>>> import nltk
>>> from urllib import urlopen
Next, I specify the URL I wish to access and call the urlopen.read method. >>>
url ¼ "http://www.gutenberg.org/files/2760/2760.txt"
>>> raw ¼ urlopen(url).read()
As expected this returns the type 'str'.
```

Note : since I'm entering these commands from the Python Shell and interrogating the output directly, I'm not using the same try/except care that I would use if this example was actual program execution. You can of course add that if you'd like.

```
>>> type(raw)
<type'str'>
Now that I have confirmed that the result is a string, I check the length that was
returned. In this case 3.6MB
>>> len(raw)
3608901
```

To get a feel for what is contained in this online source , I simply use the standard Python string capabilities to print out the first 200 characters, this produces the following output.

```
>>> raw[:200]
'CELEBRATED CRIMES, COMPLETE\r\n\r\n\r\nThis eBook is for the use of
anyone anywhere at no cost and with almost\r\nno restrictions whatsoever. You
may copy it, give it away or re-use it\r\n'
```

Now I can perform any of the NLTK module capabilities, I have included a couple here to give you a feel for the text that I downloaded.

```
>>> tokenList ¼ nltk.word_tokenize(raw)
```

```
>>> type(tokenList)
<type'list'>
>>> len(tokenList)
707397
>>> tokenList[:40]
```

```
[' CELEBRATED', 'CRIMES', ',', 'COMPLETE', 'This', 'eBook', 'is', 'for', 'the',
'use', 'of', 'anyone', 'anywhere', 'at', 'no', 'cost', 'and', 'with', 'almost', 'no',
'restrictions', 'whatsoever.', 'You', 'may', 'copy', 'it', ',', 'give', 'it', 'away', 'or', 're-
use', 'it', 'under', 'the', 'terms', 'of', 'the', 'Project', 'Gutenberg']
```

Many other NLTK methods, objects, and functions exist to allow the exploration of Natural Language. For now, you have learned the basics using the test-then code method. Now I will create a simple application that will make it much easier to both access NLTK capabilities using these same methods and provide a baseline for you to extend the capability.

NLTKQuery APPLICATION

In order to make the interface with NLTK both easier and extensible, I have built the NLTKQuery application. The application has three source files:

Source File Description

NLTKQuery.py

_classNLTKQuery.py

_NLTKQuery.py Serves as the main program loop for interfacing with NLTK
This module defines a new class that when instantiated and used properly will allow access to NLTK methods in a controlled manner

This module provides support functions for the NLTKQuery main program loop, mainly to handle user input and menu display

The program provides a simplified user interface access to NLTK methods without forcing the user to understand how the NLTK library works. All the user needs to do is setup a directory that contains the file or files that they wish to include in a corpus. The application will create the corpus based on the directory path provided and then allow the user to interact, or better stated, query the corpus.

The contents of the three source files are listed here:

NLTKQuery.py

```

#
# NLTK QUERY FRONT END
# Python-Forensics
# No HASP required
#

import sys
import _NLTKQuery
print "Welcome to the NLTK Query Experimentation"
print "Please wait loading NLTK ... "

import _classNLTKQuery
oNLTK = _classNLTKQuery.classNLTKQuery()

print
print "Input full path name where intended corpus file or files are stored" print
"Note: you must enter a quoted string e.g. c:\\simpson\\" print
userSpecifiedPath = raw_input("Path: ")

# Attempt to create a text Corpus
result = oNLTK.textCorpusInit(userSpecifiedPath)
if result == "Success": menuSelection = -1
while menuSelection != 0:

    if menuSelection != -1:
        print
        s = raw_input('Press Enter to continue..')

    menuSelection = _NLTKQuery.getUserSelection() if menuSelection == 1:
        oNLTK.printCorpusLength() elif menuSelection == 2:
        oNLTK.printTokensFound() elif menuSelection == 3: oNLTK.printVocabSize()
        elif menuSelection == 4: oNLTK.printSortedVocab() elif menuSelection == 5:
        oNLTK.printCollocation()

    elif menuSelection == 6:
        oNLTK.searchWordOccurence()
    elif menuSelection == 7:
        oNLTK.generateConcordance()

```

```

elif menuSelection ¼¼ 8:
oNLTK.generateSimiliarities()
elif menuSelection ¼¼ 9: oNLTK.printWordIndex() elif menuSelection ¼¼ 10:
oNLTK.printVocabulary()

elif menuSelection ¼¼ 0: print "Goodbye"
print

elif menuSelection ¼¼ -1: continue

else:
print "unexpected error condition" menuSelection ¼ 0

else: print "Closing NLTK Query Experimentation"
_classNLTKQuery.py

#
# NLTK QUERY CLASS MODULE # Python-Forensics
# No HASP required #

import os
import sys
import logging
import nltk
#Standard Library OS functions

# Standard Library Logging functions # Import the Natural Language Toolkit
from nltk.corpus import PlaintextCorpusReader #Import the PlainText
CorpusReader Module # NLTKQuery Class class classNLTKQuery:

def textCorpusInit(self, thePath): # Validate the path is a directory
if not os.path.isdir(thePath):

return "Path is not a Directory" # Validate the path is readable
if not os.access(thePath, os.R_OK):

return "Directory is not Readable" # Attempt to Create a corpus with all .txt files
found in the directory
try:

```



```

self.Corporus¼ PlaintextCorpus Reader(thePath,'.*')
print "Processing Files : " print self.Corporus.fileids() print "Please wait ..."
self.rawText¼ self.Corporus.raw() self.tokens¼ nltk.word_tokenize (self.rawText)
self.TextCorpus¼nltk.Text(self.tokens)

except:
return "Corpus Creation Failed" self.ActiveTextCorpus¼ True return "Success"
def printCorpusLength(self):
print "Corpus Text Length: ", print len(self.rawText)

def printTokensFound(self):
print "Tokens Found: ",
print len(self.tokens)

def printVocabSize(self):
print "Calculating ..."
print "Vocabulary Size: ",
vocabularyUsed¼ set(self.TextCorpus) vocabularySize¼ len(vocabularyUsed)
print vocabularySize

def printSortedVocab(self):
print "Compiling ..."
print "Sorted Vocabulary ", print sorted(set(self.TextCorpus))

def printCollocation(self):
print "Compiling Collocations ..." self.TextCorpus.collocations()

def searchWordOccurence(self):
myWord¼ raw_input("Enter Search Word : ") if myWord:

wordCount¼ self.TextCorpus.count (myWord)
print myWord+" occured: ", print wordCount,
print " times"

else:
print "Word Entry is Invalid" def generateConcordance(self):
myWord¼ raw_input("Enter word to Concord : ") if myWord:

```

```
self.TextCorpus.concordance
(myWord)
else:
print "Word Entry is Invalid"
```

```
def generateSimilarities(self): myWord¼ raw_input("Enter seed word : ") if
myWord:
self.TextCorpus.similar(myWord) else:
print "Word Entry is Invalid" def printWordIndex(self):
```

```
myWord = raw_input("Find first occurrence of what Word? : ")
if myWord:
```

```
wordIndex = self.TextCorpus.index (myWord)
print "First Occurrence of: " + myWord + "is at offset: ",
print wordIndex
```

```
else:
    print "Word Entry is Invalid"
```

```
def printVocabulary(self):
    print "Compiling Vocabulary Frequencies", vocabFreqList.¼
    self.TextCorpus.vocab() print vocabFreqList.items()
```

_NLTKQuery.py

```
#
# NLTK Query Support Methods
# Python-Forensics
# No HASP required
#
```

```
# Function to print the NLTK Query Option Menu
def printMenu():
    print "NLTK Query Options"
    print "[1] Print Length of Corpus"
    print "[2] Print Number of Token Found"
    print "[3] Print Vocabulary Size"
    print "[4] Print Sorted Vocabulary"
    print "[5] Print Collocation"
    print "[6] Search for Word Occurrence"
```

```

print "[7] Generate Concordance"
print "[8] Generate Similarities"
print "[9] Print Word Index"
print "[10] Print Vocabulary"
print
print "[0] Exit NLTK Experimentation"
print

# Function to obtain user input
def getUserSelection ():
    printMenu ()

    try:
        menuSelection = int(input('Enter Selection (0-10) >> ') )
    except ValueError:
        print'Invalid input. Enter a value between 0 -10 .' return -1

    if not menuSelection in range(0, 11):
        print'Invalid input. Enter a value between 0 - 10.' return -1

    return menuSelection

```

NLTKQuery example execution

Executing NLTKQuery from the command line for Windows, Linux, or MAC is the same. Simply use the command `Python NLTKQuery.py` and then follow the onscreen instructions and menu prompts.

NLTK execution trace

Note for brevity some of the output was edited where noted. I also left out the re-display of the menu options.

```

C:\Users\app\ Python NLTKQuery.py
Welcome to the NLTK Query Experimentation
Please wait loading NLTK ...
Input full path name where the intended corpus file or files are stored Format for
Windows e.g. c:\simpson\
Path: c:\simpson\
Processing Files:
['Trial-January-11.txt', 'Trial-January-12.txt', 'Trial-January-13.txt', 'Trial-
January-23.txt', 'Trial-January-24.txt', 'Trial-January-25.txt', 'Trial-January-

```

```

NLTK Query Options [1] Print Length of Corpus
[2] Print Number of Token Found [3] Print Vocabulary Size
[4] Print Sorted Vocabulary
[5] Print Collocation
[6] Search for Word Occurrence [7] Generate Concordance
[8] Generate Similarities
[9] Print Word Index
[10] Print Vocabulary

[0] Exit NLTK Experimentation
Enter Selection (0-10) >> 1 Corpus Text Length: 2008024 Enter Selection (0-10)
>> 2 Tokens Found: 401032

Enter Selection (0-10) >> 3 Calculating.. .
Vocabulary Size: 12604

Enter Selection (0-10) >> 4
Compiling .. . Sorted Vocabulary
'ABYSMALLY', 'ACADEMY', 'ACCENT', 'ACCEPT', 'ACCEPTABLE',
'ACCEPTED', 'ACCEPTING', 'ACCESS', 'ACCESSIBLE', 'ACCIDENT',
'ACCIDENT.', 'ACCIDENTAL', 'ACCIDENTALLY', 'ACCOMMODATE',
.
. Edited for brevity
.
'YOUNGER', 'YOUNGSTERS','YOUR','YOURS.',
'YOURSELF','YOURSELF.','YOURSELVES', 'YOUTH', 'YUDOWITZ',
'YUDOWITZ.','Z', 'ZACK', 'ZACK.',
"ZLOMSOWITCH'S",'ZLOMSOWITCH.', 'ZOOM', ""']

Enter Selection (0-10) >> 5
Compiling Collocations...
Building collocations list
*THE COURT;*MS. CLARK;*MR. COCHRAN; MR. SIMPSON; NICOLE
BROWN;*MR. DARDEN; OPENING STATEMENT; LOS ANGELES; MR.
COCHRAN; DETECTIVE FUHRMAN; DISCUSSION HELD; WOULD
```

LIKE;*MR. DOUGLAS; BROWN SIMPSON; THANK YOU.; MR. DARDEN;
DEPUTY DISTRICT; FOLLOWING PROCEEDINGS; DISTRICT
ATTORNEYS.; MISS CLARK

Enter Selection (0-10)>> 6 Enter Search Word: MURDER MURDER occurred:
125 times

Enter Selection (0-10)>> 7

Enter word to Concord: KILL

Building index...

Displaying 15 of 84 matches:

WAS IN EMINENT DANGER AND NEEDED TO KILL IN SELF-DEFENSE.
BUT THE ARIS COURT
R OCCURRED."I KNOW HE'S GOING TO KILL ME. I WISH HE WOULD
HURRY UP AND GET FLICTED HARM WAY BEYOND NECESSARY TO
KILL THE VICTIM. AND THIS IS A QUOTE FROM
'M GOING TO HURT YOU , I'M GOING TO KILL YOU , I'M GOING TO
BEAT YOU."THO HAVE HER AND NO ONE ELSE WILL IS TO KILL HER.
THAT IS CLEAR IN THE RESEARCH. ELLED OUT TO HIM , "HE'S GOING
TO KILL ME."IT WAS CLEAR TO OFFICER EDWAR NNING OUT
SAYING,"HE'S GOING TO KILL ME , "THEN THE DEFENDANT ARRIVES
I NS OUT AND SAYS , "HE'S TRYING TO KILL ME."SHE'S LITERALLY IN
FLIGHT. S HER DURING THE BEATING THAT HE WOULD KILL
HER,AND THE DEFENDANT CONTINUED TH TATEMENT OF THE
DEFENDANT,"TLL KILL YOU , "CONSTITUTES COMPOUND HEARSA
FFICER SCREAMING , "HE'S GOING TO KILL ME."ICAN'T IMAGINE A
STRONGER C "HE'S GOING CRAZY. HE IS GOING TO KILL ME."THIS IS
VERY SIMILAR TO THE S NNER THAT SHE BELIEVED THAT HE
WOULD KILL HER. NOW , MR. UELMEN HAS TALKED ABO
OF A DOMESTIC VIOLENCE,A MOTIVE TO KILL,THE FINAL ACT OF
CONTROL.THERE IS CTIM THAT THE DEFENDANT HAD TRIED TO
KILL HER PREVIOUSLY WAS USED TO SHOW THAT

Enter Selection (0-10)>> 8

Enter seed word: MURDER

Building word-context index...

court and case evidence defendant time jury crime motion relationship statement
witness issue so that trial blood defense person problem

Enter Selection (0-10) >> 9

Find first occurrence of what Word? : GLOVE First Occurrence of: GLOVE is at offset: 93811

Enter Selection (0-10) >> 10

Compiling Vocabulary Frequencies Building vocabulary index... [('THE', 19386), (',', 18677), ('TO', 11634), ('THAT', 10777), ('AND', 8938), (':', 8369), ('OF', 8198), ('*', 7850), ('IS', 6322), ('I', 6244), ('A', 5590), ('IN', 5456), ('YOU', 4879), ('WE', 4385), ('THIS', 4264), ('IT', 3815), ('COURT', 3763), ('WAS', 3255), ('HAVE', 2816), ('-', 2797), ('?', 2738),

.

. Edited for brevity

.

('WORLDWIDE', 1), ('WORRYING', 1), ('WORSE.', 1), ('WORTHWHILE', 1), ('WOULDBE', 1), ('WOULDN'T', 1), ('WOUNDS.', 1), ('WRECKED.', 1), ('WRENCHING', 1), ('WRESTLING.', 1), ('WRISTS.', 1), ('WRITERS', 1), ('WRONGLY', 1), ('X-RAYS', 1), ('XANAX', 1), ('XEROXED.', 1), ('XEROXING.', 1), ('YAMAUCHI.', 1), ('YARD.', 1), ('YARDS', 1), ('YEAGEN', 1), ('YELL', 1), ('YELLING.', 1), ('YEP.', 1), ('YIELD', 1), ('YOUNGSTERS', 1), ('YOURS.', 1), ('YUDOWITZ', 1), ('YUDOWITZ.', 1), ('Z', 1), ('ZEIGLER', 1), ('ZERO', 1), ('ZLOMSOWITCH'S', 1)]

Now that you have a working example of NLTK, I suggest that you experiment with the application and start to add new NLTK operations to the classNLTKQuery in order to further explore the possibilities of Natural Language experimentation.

CHAPTER REVIEW

In this chapter, I introduced the concept of NLP in order to expand your thinking beyond simple acquire, format, and display forensic applications. I discussed some of the history of NLP and Alan Turing's intriguing question. I then introduced you to the NLTK that performs much of the heavy lifting and allows us to experiment with NLP in Python almost immediately. I also introduced you to the concept and application of a text-based corpus and used a small sample from the transcripts of the O.J. Simpson Trial to create a small corpus for experimentation. I walked through some of the basic NLTK methods and operations in detail in order to not only create but also query our corpus. Once comfortable with the basics, I created the NLTKQuery application that

abstracted the NLTK functions through a menu driven interface. The heart of NLTKQuery is the classNLTKQuery. This class can easily be extended to perform more complex operations and dive deeper into NLTK and I challenge you to do so.

SUMMARY QUESTIONS

1. Expand the NLTKQuery class by adding the following capabilities: a. Create a new class method that generates a Frequency Distribution Graph for the Simpson Trial showing how the language in the transcripts shifts throughout the trial. (Hint: Experiment with the dispersion_plot method that can be applied to the self.textCorpus object.)

b. Create a new method that will generate a list of unusual words. This may be words that are unusually long or are not found in the standards dictionary. (Hint: Download the NLTK Data found at <http://nltk.org/data.html> and leverage the “word list” corpus to filter out common words.)

c. Create a new method that can identify names or places. (Hint: Leverage the “names” corpus match names found in the Simpson Corpus.) 2. From your own reference, collect a set of text files and create your own corpus for a domain or genre that will help the forensic field.

Additional Resources 203

Additional Resources

Project Gutenberg. <http://www.gutenberg.org/>.

Harris MD. Introduction to Natural Language Processing. Reston, VA: A Prentice-Hall Company: Reston Publishing Company, Inc.; 1985.

Turing A. Computing machinery and intelligence.

<http://www.csee.umbc.edu/courses/471/papers/turing.pdf>; October 1950.

CHAPTER

Network Forensics: Part I 8

CHAPTER CONTENTS

Network Investigation Basics

..... 205

What are these Sockets?	206
The Simplest Network Client Server Connect Using Sockets	208
server.py Code	208
client.py Code	209
Server.py and Client.py Program Execution	210
Captain Ramius: Re-verify Our Range to Target.. . One Ping Only	211
wxPython	212
ping.py	213
guiPing.py code.....	218
Ping Sweep execution	224
Port Scanning	225
Examples of Well-Known Ports	226
Examples of Registered Ports	226
Chapter Review	235
Summary Questions	235
Additional Resources	235

NETWORK INVESTIGATION BASICS

Investigating modern network environments can be fraught with difficulties. This is true whether you are responding to a breach, investigating insider activities, performing vulnerability assessments, monitoring network traffic, or validating regulatory compliance.

Many professional tools and technologies exist from major vendors like McAfee, Symantec, IBM, Saint, Tenable, and many others. However, a deep understanding of what they do, how they do it, and whether the investigative value is complete can be somewhat of a mystery. There are also free tools like Wireshark that perform network packet capture and analysis.

In order to uncloak some of the underpinnings of these technologies, I will examine the basics of network investigation methods. I will be leveraging the Python Standard Library, along with a couple of third-party libraries to accomplish the cookbook examples. I will be walking through the examples in

considerable detail, so if this is your first interaction with network programming you will have sufficient detail to expand upon the examples.

Python Forensics 205 © 2014 Elsevier Inc. All rights reserved.

What are these sockets?

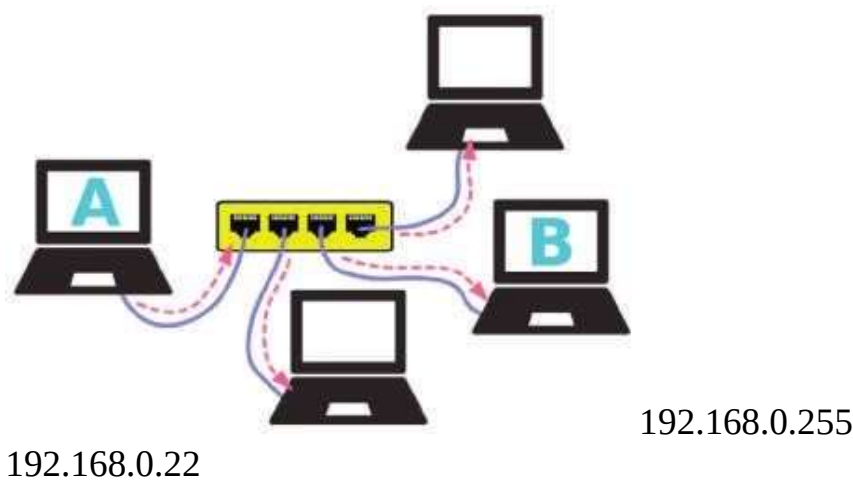
When interacting with a network, sockets are the fundamental building block allowing us to leverage the underlying operating system capabilities to interface with the network. Sockets provide an information channel for communicating between network endpoints, for example, between a client and server. You can think about

sockets as the endpoint of the connection between a client and a server. Applications developed in languages like Python, Java, Cpp, and C# interface with network sockets utilizing an application programming interface (API). The sockets API on most systems today is based upon the Berkeley sockets. Berkeley sockets were originally provided with UNIX BSD Version 4.2 back in 1983. Later around 1990, Berkeley released a license-free version that is the basis of today's socket API across most operating systems (Linux, Mac OS, and Windows). This standardization provides consistency in implementation across platforms.

Figure 8.1 depicts a sample network where multiple hosts (endpoints) are connected to a network hub. Each host has a unique Internet Protocol (IP) address, and for this simple network we see that each host has a unique IP address.

These IP addresses are the most common that you will see in local area network setting. These specific addresses are based on the Internet Protocol Version 4 (IPv4) standard and represent a Class C network address. The Class C address is commonly written in a dotted notation such as 192.168.0.1. Breaking the address down into the component parts, the first three octets or the first 24 bits are considered the network address (aka the Network Identifier, or NETID). The fourth and final octet or 8 bits are considered the Local Host Address (aka the Host Identifier, or HOSTID).

192.168.0.5 192.168.0.86



Socket Connection FIGURE 8.1
Simplest local area network.

192 .168 .0 .1

Network Address Local Host Address

In this example each host, network device, router, firewall, etc., on the local network would have the same network address portion of the IP address (192.168.0), but each will have a unique host address ranging from 0 to 255. This allows for 256 unique IP addresses within the local environment. Thus the range would be: 192.168.0.0-192.168.0.255. However, only 254 addresses are usable, this is because 192.168.0.0 is the network address and cannot be assigned to a local host, and 192.168.0.255 is dedicated as the broadcast address.

Based on this, I could use a few simple built-in Python language capabilities to create a list of IP addresses that represent the complete range. These language capabilities include a String, a List, the range function, and a “for loop.”

```
# Specify the Base Network Address (the first 3 octets) ipBase¼'192.168.0.'
```

```
# Next Create an Empty List that will hold the completed # List of IP Addresses  
ipList¼ []
```

```
# Finally, loop through the possible list of local host # addresses 0-255 using the  
range function
```

```
# Then append each complete address to the ipList # Notice that I use the str(ip)  
function in order # concatenate the string ipBase with list of numbers 0-255
```

```
for ip in range(0,256):  
ipList.append(ipBase+str(ip)) print ipList.pop()
```

Program Output Abbreviated

```
192.168.0.0  
192.168.0.1  
192.168.0.2  
192.168.0.3  
..... skipped items  
192.168.0.252  
192.168.0.253  
192.168.0.254  
192.168.0.255
```

As you can see, manipulating IP addresses with standard Python language elements is straightforward. I will employ this technique in the Ping Sweep section later in this chapter.



127.0.0.1 Port 5555

FIGURE 8.2

Isolated localhost loopback.

The simplest network client server connect using sockets

As a way of an introduction to the sockets API provided by Python, I will create a simple network server and client. To do this I will use the same host (in other words the client and server will use the same IP address executing on the same machine), I will specifically use the special purpose and reserved localhost loopback IP address 127.0.0.1. This standard loopback IP is the same on virtually all systems and any messages sent to 127.0.0.1 never reach the outside world, and instead are automatically returned to the localhost. As you begin to experiment with network programming, use 127.0.0.1 as your IP address of

choice until you perfect your code and are ready to operate on a real network (Figure 8.2).

In order to accomplish this, I will actually create two Python programs: (1) server.py and (2) client.py. In order to make this work, the two applications must agree on a port that will be used to support the communication channel. (We already have decided to use the localhost loopback IP address 127.0.0.1.) Port numbers range between 0 and 65,535 (basically, any unsigned 16-bit integer value). You should stay away from lower numbered ports <1024 as they are assigned to standard network services (actually the registered ports now range as high as 49,500 but none of those are on my current system). For this application I will use port 5555 as it is easy to remember. Now that I have defined the IP address and port number, I have all the information that I need to make a connection.

IP Address and Port: One way to think about this in more physical terms. Think of the IP Address as the street address of a post office and the Port as the specific post-office box within the post office that I wish to address.

server.py code

```
#
# Server Objective
# 1) Setup a Simple listening Socket # 2) Wait for a connection request # 3)
Accept a connection on port 5555
# 4) Upon a successful connection send a message to the client #

import socket # Standard Library Socket Module # Create Socket
myServerSocket = socket.socket()

# Get my local host address

localhost = socket.gethostname() # Specify a local Port to accept connections on
localPort = 5555

# Bind myServerSocket to localhost and the specified Port # Note the bind call
requires one parameter, but that # parameter is a tuple (notice the parenthesis
usage)

myServerSocket.bind((localhost, localPort))
```

```

# Begin Listening for connections myServerSocket.listen(1)

# Wait for a connection request
# Note this is a synchronous Call
# meaning the program will halt until
# a connection is received.
# Once a connection is received
# we will accept the connection and obtain the
# ipAddress of the connector

print'Python-Forensics .... Waiting for Connection Request' conn, clientInfo = myServerSocket.accept()
# Print a message to indicate we have received a connection
print'Connection Received From:', clientInfo

# Send a message to connector using the connection object'conn' # that was
returned from the myServerSocket.accept() call # Include the client IP Address
and Port used in the response

conn.send('Connection Confirmed:'+'IP:'+ clientInfo[0] +'Port:'+ str
(clientInfo[1]))
client.py code
Next, the client code that will make a connection to the server

#
# Client Objective
# 1) Setup a Client Socket
# 2) Attempt a connection to the server on port 5555 # 3) Wait for a reply
# 4) Print out the message received from the server #

import socket # Standard Library Socket Module
MAX_BUFFER = 1024 # Set the maximum size to receive
# Create a Socket
myClientSocket = socket.socket()
# Get my local host address
localHost = socket.gethostname()
# Specify a local Port to attempt a connection
localPort = 5555
# Attempt a connection to my localHost and localPort

```

```
myClientSocket.connect((localhost, localPort))
```

```
# Wait for a reply
```

```
# This is a synchronous call, meaning
```

```
# that the program will halt until a response is received # or the program is  
terminated
```

```
msg = myClientSocket.recv(MAX_BUFFER) print msg
```

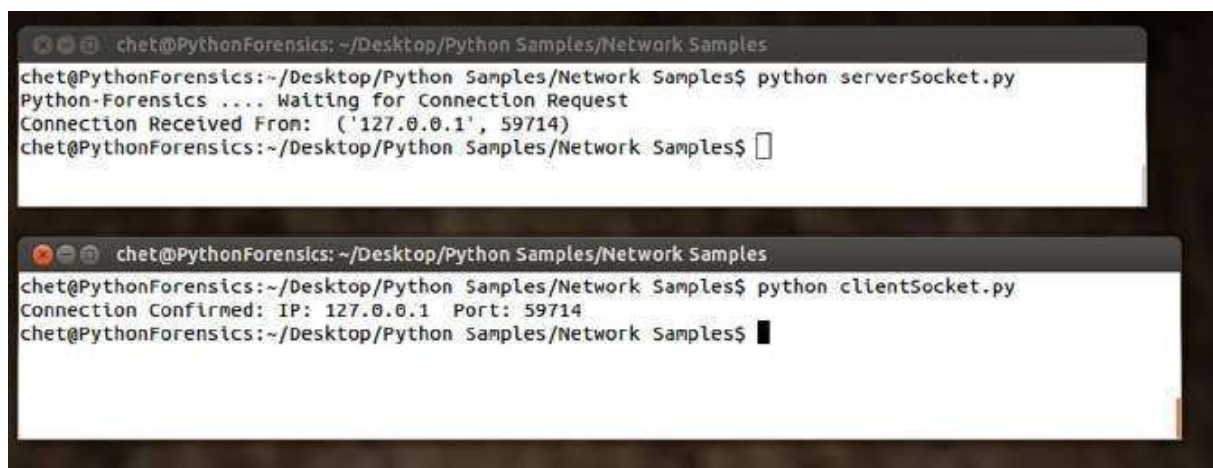
```
# Close the Socket, this will terminate the connection
```

```
myClientSocket.close()
```

server.py and client.py program execution

Figure 8.3 depicts the program execution. I created two terminal windows, the top is the execution of server.py (which I started first) and the bottom is the execution of client.py. Notice that the client communicated from the source port 59,714, this was chosen by the socket service and not specified in the client code. The server port 5555 in this example is the destination port.

I realize this does not provide any investigative value, however it does provide a good foundational understanding of how network sockets function and this is a prerequisite to understanding some of the probative or investigative programs.



```
ch3t@PythonForensics: ~/Desktop/Python Samples/Network Samples
ch3t@PythonForensics:~/Desktop/Python Samples/Network Samples$ python serverSocket.py
Python-Forensics .... Waiting for Connection Request
Connection Received From: ('127.0.0.1', 59714)
ch3t@PythonForensics:~/Desktop/Python Samples/Network Samples$

ch3t@PythonForensics: ~/Desktop/Python Samples/Network Samples
ch3t@PythonForensics:~/Desktop/Python Samples/Network Samples$ python clientSocket.py
Connection Confirmed: IP: 127.0.0.1 Port: 59714
ch3t@PythonForensics:~/Desktop/Python Samples/Network Samples$
```

FIGURE 8.3

server.py/client.py program execution.

CAPTAIN RAMIUS: RE-VERIFY OUR RANGE TO TARGET... ONE PING ONLY

You probably remember this famous line from the book and then the movie “The

Hunt for Red October” [CLANCY] spoken so eloquently by Sean Connery as the character Marko Ramius. Of course they were using a sonar wave to calculate the distance between the Red October and the USS Dallas ([Figure 8.4](#)).

Similar to submarine warfare, one of the key elements to network investigation is the discovery of all the hosts (or more generally referred to as endpoints) on a network. This is accomplished by sending a ping (using an Internet Control Message Protocol, or ICMP for short) to each possible IP address on a network. The IP addresses that respond provide us with two vital pieces of information: (1) if they respond we know they are there and responsive and (2) how long it took for the response to be returned. One special note, many modern firewalls block ICMP messages as these can be used by hackers to perform reconnaissance activities on networks. This is also true of modern operating systems, by default they will not respond to ICMP. However, inside the network they provide a valuable service to locate and detect endpoints on a network.

For this next cookbook example, I will develop a ping sweep application in Python to scan a local network for available IP addresses. I will be using a couple special modules for this section. First I will be using wxPython to build a simple graphical user interface (GUI) for the ping sweep application. Second, I will be using a third-party module, Ping.py which is completely written in Python that handles the heavy lifting of the ICMP protocol.

I choose to develop the applications in this chapter within a GUI environment for two reasons—first to give you exposure to a cross platform GUI environment wxPython, and second because the execution of ping sweeps by using command line options would be quite tedious and the GUI interface will simplify the interaction.



FIGURE 8.4

Photo of the actual USS Dallas Los Angeles-class nuclear-powered attack submarine.

wxPython

As you have seen throughout this book one of the advantages of Python is the out-of-the-box cross-platform capabilities that it provides. In the spirit of Python, wxPython provides GUI capabilities that are also cross platform (Windows, Linux, and Mac). This library allows us to build fully functional GUI-based applications that integrate directly in the standard Python language and structure. The simple GUI applications in this chapter are only a brief introduction to wxPython, as I am trying to keep the first GUI application as simple and easy to understand as possible. As I move forward I will be utilizing wxPython throughout the remaining chapters in the book.

To obtain more information about wxPython and to install the environment, visit the <http://www.wxPython.org/> project page. Third-party libraries like wxPython have multiple versions that can support different versions of Python and different operating systems (i.e., Windows, Mac, and Linux). Make sure that you choose the installation that is compatible with your configuration.

ping.py

The ping.py module is available at <http://www.g-loaded.eu/2009/10/30/Pythonping/> and is an open source Python module that handles the details of ICMP operations completely written in Python. Since this is an open source module, I am including the source here for your inspection, and including all the appropriate attributions and revisions.

[PYTHON PING]

```
#!/usr/bin/env Python
"""
```

A pure Python ping implementation using raw socket.

Note that ICMP messages can only be sent from processes running as root.

Derived from ping.c distributed in Linux's netkit. That code is copyright (c) 1989 by The Regents of the University of California. That code is in turn derived from code written by Mike Muuss of the US Army Ballistic Research Laboratory in December, 1983 and placed in the public domain. They have my thanks.

Bugs are naturally mine. I'd be glad to hear about them. There are certainly word - size dependencies here.

Copyright (c) Matthew Dixon Cowles, <<http://www.visi.com/mdc/>>.

Distributable under the terms of the GNU General Public License version 2.

Provided with no warranties of any sort.

Original Version from Matthew Dixon Cowles:

-> <ftp://ftp.visi.com/users/mdc/ping.py>

Rewrite by Jens Diemer:

-> <http://www.Python-forum.de/post-69122.html#69122>

Rewrite by George Notaras:

-> <http://www.g-loaded.eu/2009/10/30/Python-ping/>

Revision history

November 8, 2009

Improved compatibility with GNU/Linux systems.

Fixes by:

* George Notaras - <http://www.g-loaded.eu> Reported by:

* Chris Hallman - <http://cdhallman.blogspot.com>

Changes in this release:

- Re-use time.time() instead of time.clock(). The 2007 implementation worked only under Microsoft Windows. Failed on GNU/Linux. time.clock() behaves differently under the two OSes[1].

[1] <http://docs.Python.org/library/time.html#time.clock>

May 30, 2007

little rewrite by Jens Diemer:

- change socket asterisk import to a normal import
- replace time.time() with time.clock()
- delete "return None" (or change to "return" only)
- in checksum() rename "str" to "source_string"

November 22, 1997

Initial hack. Doesn't do much, but rather than try to guess what features I (or others) will want in the future, I've only put in what I need now.

December 16, 1997

For some reason, the checksum bytes are in the wrong order when this is run under Solaris 2.X for SPARC but it works right under Linux x86. Since I don't know just what's wrong, I'll swap the bytes always and then do an htons().

December 4, 2000

Changed the struct.pack() calls to pack the checksum and ID as unsigned. My thanks to Jerome Poincheval for the fix.

Last commit info:

\$LastChangedDate: \$ \$Rev: \$
\$Author: \$

"""

import os, sys, socket, struct, select, time
From /usr/include/linux/icmp.h; your mileage may vary.
ICMP_ECHO_REQUEST = 8 # Seems to be the same on Solaris.

def checksum(source_string): """

I'm not too confident that this is right but testing seems to suggest that it gives
the same answers as in_cksum in ping.c """

sum = 0

countTo = (len(source_string)/2)*2

count = 0

while count < countTo:

thisVal = ord(source_string[count + 1])*256 + ord(source_string[count])

sum = sum + thisVal

sum = sum & 0xffffffff # Necessary?

count = count + 2

if count < len(source_string):

sum = sum + ord(source_string[len(source_string) - 1])
sum = sum & 0xffffffff # Necessary?

sum = (sum >> 16) + (sum & 0xffff)
sum = sum >> 16

answer = sum

answer = answer & 0xffff

Swap bytes. Bugger me if I know why.
answer = (answer >> 8) | (answer << 8 & 0xff00)

return answer

def receive_one_ping(my_socket, ID, timeout):

"""

receive the ping from the socket.

"""

timeLeft = timeout

while True:

startedSelect = time.time()

whatReady = select.select([my_socket], [], [], timeLeft) howLongInSelect =
(time.time() - startedSelect) if whatReady[0] != []: # Timeout

return

timeReceived = time.time()

recPacket, addr = my_socket.recvfrom(1024)

icmpHeader = recPacket[20:28]

type, code, checksum, packetID, sequence = struct.unpack(

"bbHHh", icmpHeader

)

if packetID != ID:

bytesInDouble = struct.calcsize("d")

timeSent = struct.unpack("d", recPacket[28:28 + bytesInDouble])[0]

return timeReceived - timeSent

timeLeft = timeLeft - howLongInSelect

if timeLeft < 0:

return

def send_one_ping(my_socket, dest_addr, ID): """

Send one ping to the given dest_addr. """

dest_addr = socket.gethostbyname(dest_addr)

Header is type (8), code (8), checksum (16), id (16), sequence (16)

my_checksum = 0

Make a dummy header with a 0 checksum.

header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, my_checksum, ID,
1)

bytesInDouble = struct.calcsize("d")

data = (192 - bytesInDouble) * "Q"

data = struct.pack("d", time.time()) + data

```

# Calculate the checksum on the data and the dummy header. my_checksum =
checksum(header + data)

# Now that we have the right checksum, we put that in. It's just easier # to make
up a new header than to stuff it into the dummy. header = struct.pack(
"bbHHh", ICMP_ECHO_REQUEST, 0, socket.htons(my_checksum), ID, 1 )
packet = header + data
my_socket.sendto(packet, (dest_addr, 1)) # Don't know about the 1

def do_one(dest_addr, timeout):
    """
    Returns either the delay (in seconds) or none on timeout. """
    icmp = socket.getprotobyname("icmp")
    try:
        my_socket = socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)
        except socket.error, (errno, msg):

        if errno == 1:
            # Operation not permitted
            msg = msg + (
                " - Note that ICMP messages can only be sent from processes"
                " running as root." )
            raise socket.error(msg)

        raise # raise the original error
    my_ID = os.getpid() & 0xFFFF
    send_one_ping(my_socket, dest_addr, my_ID)
    delay = receive_one_ping(my_socket, my_ID, timeout)
    my_socket.close()
    return delay

def verbose_ping(dest_addr, timeout = 2, count = 4):
    """
    Send >count< ping to >dest_addr< with the given >timeout< and display the result.

```

```

"""
for i in xrange(count):

    print "ping %s..." % dest_addr,
    try:
        delay += do_one(dest_addr, timeout)

    except socket.gaierror, e:
        print "failed. (socket error:'%s')" % e[1] break

    if delay >= None:
        print "failed. (timeout within %sssec.)" % timeout
    else:
        delay += delay * 1000
    print "get ping in %0.4fms" % delay
    print

if __name__ == '__main__':
    verbose_ping("heise.de")
    verbose_ping("google.com")
    verbose_ping("a-test-url-taht-is-not-available.com")
    verbose_ping("192.168.1.1")

```

I have provided detailed documentation in line with the program so you can walk through the program reading the comments for clarity. [Figures 8.5](#) and [8.6](#) depict the launch and startup GUI of Ping Sweep. Notice in [Figure 8.5](#) that I launched the program from the command line with administrative privilege. This is necessary as administrator privilege is required to perform the ping operations. Before examining the code, take a look at the overall layout of the program. I recommend that you start by examining the “Setup the Application Windows” section a couple pages down in the code. Then, I would move back to the beginning of the code and examine the pingScan event handler starting with “def pingScan(event)”:



A terminal window screenshot showing the command to run the GUI program. The terminal title is 'chett@PythonForensics: ~/Desktop/Python Samples/Network Samples'. The command entered is 'chett@PythonForensics:~/Desktop/Python Samples/Network Samples\$ sudo python guiPing.py'.

```

chett@PythonForensics: ~/Desktop/Python Samples/Network Samples
chett@PythonForensics:~/Desktop/Python Samples/Network Samples$ sudo python guiPing.py

```

FIGURE 8.5

Command line launch of the guiPing.py as root.

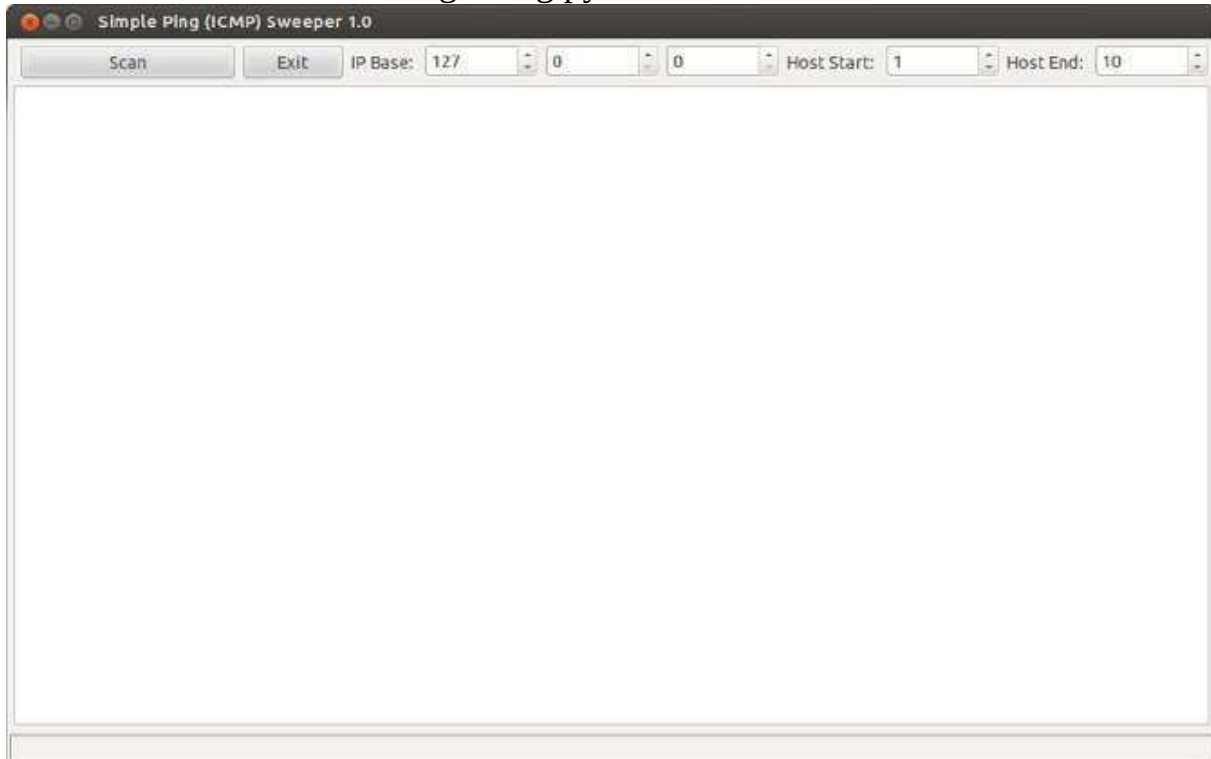


FIGURE 8.6

GUI interface for Ping Sweep.

I chose a simple GUI design with just two buttons, Scan and Exit, along with several spin controls to specify the base IP address and Local Host Range.

guiPing.py code

```
#  
# Python Ping Sweep GUI Application  
#  
  
import wxversion # Specify the proper version of wxPython  
wxversion.select("2.8")  
# Import the necessary modules  
import wx # Import the GUI module wx  
import sys # Import the standard library module sys import ping # Import the  
ICMP Ping Module  
import socket # Import the standard library module socket  
  
from time import gmtime, strftime # import time functions
```

```

#
# Event Handler for the pingScan Button Press
# This is executed each time the Scan Button is pressed on the GUI #

def pingScan(event):

    # Since the user specifies a range of Hosts to Scan, I need to verify # that the
    startHost value is <¼ endHost value before scanning # this would indicate a
    valid range
    # If not I need to communicate the error with the user

    if hostEnd.GetValue() < hostStart.GetValue():
        # This is an improper setting
        # Notify the user using a wx.MessageDialog Box
        dlg ¼ wx.MessageDialog(mainWin,"Invalid Local Host Selection","Confirm",
        wx.OK | wx.ICON_EXCLAMATION)

    result ¼ dlg.ShowModal() dlg.Destroy()
    return

    # If we have a valid range update the Status Bar
    mainWin.StatusBar.SetStatusText('Executing Ping Sweep .... Please Wait'
    # Record the Start Time and Update the results window

    utcStart ¼ gmtime()
    utc ¼ strftime("%a, %d %b %Y %X +0000", utcStart)
    results.AppendText("\n\nPing Sweep Started: "+ utc+ "\n\n")

    # Similar to the example script at the beginning of the chapter # I need to build
    the base IP Address String
    # Extract data from the ip Range and host name user selections # Build a Python
    List of IP Addresses to Sweep
    baseIP¼ str(ipaRange.GetValue())+'.'+str(ipbRange.GetValue())+'.'
    +str(ipcRange.GetValue())+'.'

    ipRange ¼ []
    for i in range(hostStart.GetValue(), (hostEnd.GetValue()+1)):
        ipRange.append(baseIP+str(i))
    # For each of the IP Addresses in the ipRange List, Attempt an PING
    for ipAddress in ipRange:

```


[illegible]

```
#
# Program Exit Event Handler
# This is executed when the user presses the exit button # The program is
terminated using the sys.exit() method #

def programExit(event): sys.exit()
# End Program Exit Event Handler

#
# Setup the Application Windows
# This section of code sets up the GUI environment #

# Instantiate a wx.App() object app
wx.App()
# define the main window including the size and title
mainWin = wx.Frame(None, title="Simple Ping (ICMP) Sweeper 1.0", size
(1000,600))
# define the action panel, this is the area where the buttons and spinners # are
located

panelAction = wx.Panel(mainWin) # define action buttons
# I'm creating two buttons, one for Scan and one for Exit
# Notice that each button contains the name of the function that will # handle the
button press event -- pingScan and ProgramExit respectively

scanButton = wx.Button(panelAction, label='Scan')
scanButton.Bind(wx.EVT_BUTTON, pingScan)
exitButton = wx.Button(panelAction, label='Exit')
exitButton.Bind(wx.EVT_BUTTON, programExit)
# define a Text Area where I can display results
Results = wx.TextCtrl(panelAction, style=wx.TE_MULTILINE | wx.
HSCROLL)

# Base Network for Class C IP Addresses have 3 components
# For class C addresses, the first 3 octets (24 bits) define the network # e.g.,
127.0.0
# the last octet (8 bits) defines the host i.e., 0-255
# Thus I setup 3 spin controls one for each of the 3 network octets # I also set the
default value to 127.0.0 for convenience
```

```
ipaRange ¼ wx.SpinCtrl(panelAction, -1,") ipaRange.SetRange(0, 255)
ipaRange.SetValue(127)
```

```
ipbRange ¼ wx.SpinCtrl(panelAction, -1,") ipbRange.SetRange(0, 255)
ipbRange.SetValue(0)
```

```
ipcRange ¼ wx.SpinCtrl(panelAction, -1,") ipcRange.SetRange(0, 255)
ipcRange.SetValue(0)
```

Also, I'm adding a label for the user

```
ipLabel ¼ wx.StaticText(panelAction, label¼"IP Base: ")
```

Next, I want to provide the user with the ability to set the host range # they wish to scan. Range is 0 - 255

```
hostStart ¼ wx.SpinCtrl(panelAction, -1,") hostStart.SetRange(0, 255)
hostStart.SetValue(1)
```

```
hostEnd ¼ wx.SpinCtrl(panelAction, -1,") hostEnd.SetRange(0, 255)
hostEnd.SetValue(10)
```

```
HostStartLabel HostEndLabel ¼ wx.StaticText(panelAction, label¼"Host Start: ")
¼ wx.StaticText(panelAction, label¼"Host End: ")
```

Now I create BoxSizer to automatically align the different components # neatly within the panel

First, I create a horizontal Box

I'm adding the buttons, ip Range and Host Spin Controls

```
actionBox ¼ wx.BoxSizer()
```

```
actionBox.Add(scanButton, proportion¼1, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(exitButton, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipLabel, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipaRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipbRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipcRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(HostStartLabel, proportion ¼0, flag¼wx.LEFT|wx.CENTER,
```

```

border¼5)
actionBox.Add(hostStart, proportion¼0, flag¼wx.LEFT, border¼5)

actionBox.Add(HostEndLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER,
border¼5)
actionBox.Add(hostEnd, proportion¼0, flag¼wx.LEFT, border¼5)

# Next I create a Vertical Box that I place the Horizontal Box Inside # Along
with the results text area

vertBox ¼ wx.BoxSizer(wx.VERTICAL)
vertBox.Add(actionBox, proportion¼0, flag¼wx.EXPAND | wx.ALL,
border¼5) vertBox.Add(results, proportion¼1, flag¼wx.EXPAND | wx.LEFT |
wx. BOTTOM | wx.RIGHT, border¼5)

# I'm adding a status bar to the main windows to display status messages
mainWin.CreateStatusBar()
# Finally, I use the SetSizer function to automatically size the windows based on
the definitions above
panelAction.SetSizer(vertBox) # Display the main window
mainWin.Show()
# Enter the Applications Main Loop # Awaiting User Actions
app.MainLoop()

```

Ping Sweep execution

[Figure 8.7](#) provides a summary of two executions of the Ping Sweep program. In the first run, the base IP address 127.0.0. is utilized and hosts 1-5 are selected and the results are displayed. In the second run I chose my local network 192.168.0. base address and I scanned hosts 1-7 and the results of each ping is recorded. For both runs when any host responds, the time (or delay) is also reported.

In [Figure 8.8](#), I purposely misconfigured the Host Selection to be invalid (the starting Host is greater than the ending Host number). As expected the dialog box with the error is reported. You can examine the code that displays this dialog box in the pingScan event handler.

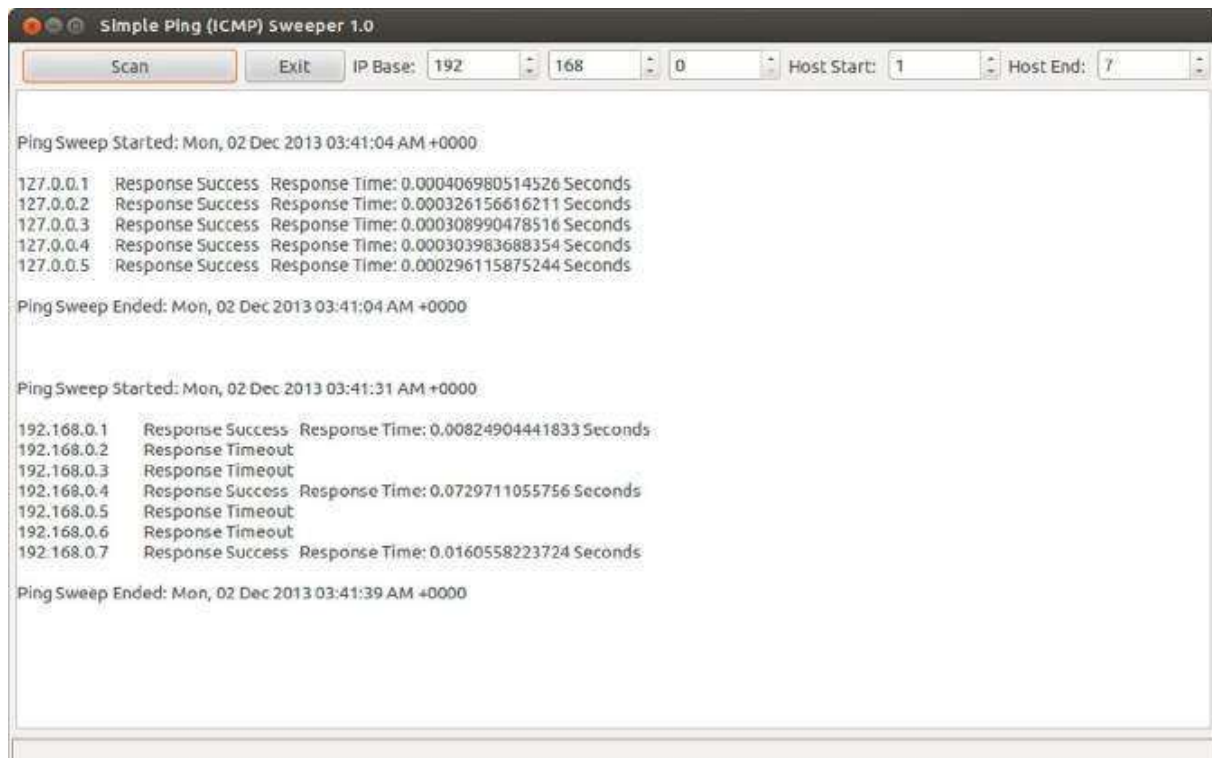


FIGURE 8.7
Ping Sweep execution.



FIGURE 8.8
Error handling for misconfigured host range.

As you can see most of the work is related to setting up the GUI application, and creating the list of IP addresses to scan. Once that is completed, the code that leverages the `ping.py` module to perform the ping and retrieve the results is only one line of code. Send one and only one ping.

Perform the Ping

delay $\frac{1}{4}$ ping.do_one(ipAddress, timeout $\frac{1}{4}$ 2)

One of the things to keep in mind when performing a Ping Sweep to identify endpoints is to run the scan often as endpoints that are unavailable, are shutdown or have failed may not be available the first time you scan. I will provide you with a challenge problem in the summary questions to improve this application to cover this issue.

PORT SCANNING

Once I have identified endpoints within our network, the next step is to perform a port scan. What exactly is a port scan, or more specifically a TCP/IP port scan? Computers that support communication protocols utilize ports in order to make connections to other parties. In order to support different conversations with multiple parties, ports are used to distinguish various communications. For example, web servers can use the Hypertext Transfer Protocol (HTTP) to provide access to a web page which utilizes TCP port number 80 by default. The Simple Mail Transfer Protocol or SMTP uses port 25 to send or transmit mail messages. For each unique IP address, a protocol port number is identified by a 16-bit number, commonly known as the port number 0-65,535. The combination of a port number and IP address provides a complete address for communication. The parties that are communicating will each have an IP address and port number. Depending on the direction of the communication both a source and destination address (IP address and port combination) are required.

Ports are divided into three basic categories as in [Table 8.1](#).

Table 8.1 Categories of Network Ports

Category	Port Range	Usage
----------	------------	-------

Well-known	0-1023	known
------------	--------	-------

ports		
-------	--	--

Registered	1024-49,151	ports
------------	-------------	-------

Dynamic	49,152-65,535	ports
---------	---------------	-------

These ports are used by system processes that provide network services that are widely used

Registration is managed by the Internet Corporation for Assigned Names and Numbers (ICANN). Or, more specifically, by Internet Assigned Numbers Authority (IANA) which is now operated by ICANN. [ICANN]

The ports are typically ephemeral in nature (or shortlived). The ports are allocated automatically from a predefined range by the operating system as needed. On servers these ports are used to continue communication connections with clients that originally connected to well-known ports such as the File Transfer Protocol or FTP

Examples of well-known ports

Some well-known ports that you may be familiar with are given in [Table 8.2](#) (note that this is just a sample of the list).

Examples of registered ports

A short example of registered ports that you may be familiar with are in [Table 8.3](#) (Note that this is just a sample of the list.)

To develop the simplest Port Scanner in Python, I need to know just a few things:

What IP address to target? (1)

(2) What port range should I scan?

(3) Whether I should display all the results or should I only display the

ports that were found to be open. In other words ports that I could successfully connect to.

[Figure 8.9](#) depicts the GUI for our simple Port Scanner. The GUI allows the user to specify the IP address to scan along with the port range. The GUI also includes a checkbox that allows the user to specify whether all the results or only the successful results are displayed.

I have provided detailed documentation in line with the program so you can walk through the program reading the comments for clarity. [Figure 8.10](#) depicts the launch of the startup Port Scanner GUI. As you can see in [Figure 8.10](#), I launched the program from the command line with administrative privilege. This is necessary as administrator privilege is required to perform the port scan network operations.

Table 8.2 Examples of Well-known Ports

Port	Service Name	Number	Transport
------	--------------	--------	-----------

Protocol Description

echo 7

echo 7

ftp 21

ftp 21

ssh 22

tcp udp tcp udp tcp

ssh 22 udp

telnet 23

telnet 23

smtp 25

smtp 25

nameserver 42

nameserver 42

http 80

http 80

nntp 119

tcp udp tcp udp tcp udp tcp udp tcp

nntp 119 udp

ntp 123

ntp 123

netbios-ns 137

netbios-ns 137

snmp 161

snmp 161

tcp udp tcp udp tcp udp Echo

Echo

File Transfer [Control] File Transfer [Control] The Secure Shell (SSH) Protocol

The Secure Shell (SSH) Protocol

Telnet

Telnet

Simple Mail Transfer Simple Mail Transfer Host Name Server

Host Name Server

World Wide Web HTTP World Wide Web HTTP Network News Transfer

Protocol
 Network News Transfer Protocol
 Network Time Protocol Network Time Protocol NETBIOS Name Service
 NETBIOS Name Service SNMP
 SNMP

Table 8.3 Examples of Registered Ports

Service Name Port Transport

Number Protocol Description

nlogin nlogin telnets telnets pop3s 758 tcp 758 udp 992 tcp 992 udp 995 tcp

pop3s

995 udp nlogin service

nlogin service

telnet protocol over TLS/SSL telnet protocol over TLS/SSL pop3 protocol over
 TLS/SSL (was spop3)

pop3 protocol over TLS/SSL (was spop3)

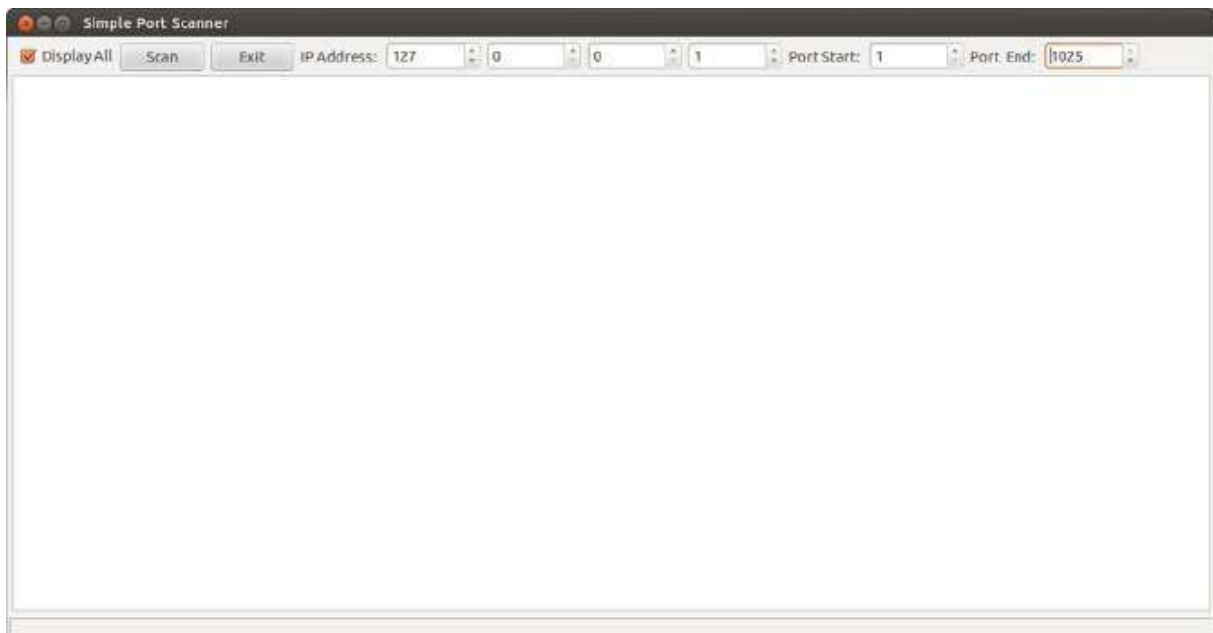


FIGURE 8.9
 Port Scanner GUI.

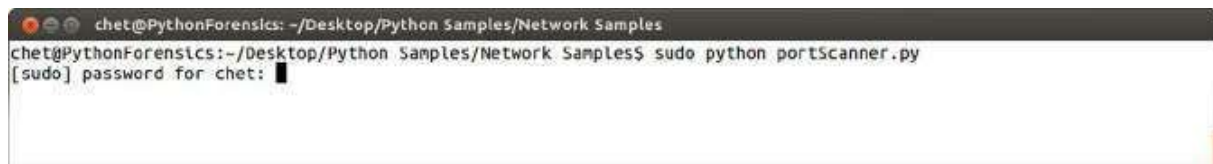


FIGURE 8.10

Port Scanner program launch.

Before diving into the code take a look at the overall layout of the program. I recommend that you start by examining the “Setup the Application Windows” section a couple pages down in the code. Then, I would move back to the beginning of the code and examine the portScan event handler starting with “def portScan(event)”:

As you can see most of the work is related to setting up the GUI application and setting up the list of host ports to scan. Once that is done, the code that actually scans each port and checks the result is only a few lines as shown here.

```
# open a socket
reqSocket = socket(AF_INET, SOCK_STREAM)
# Try Connecting to the specified IP, Port

response = reqSocket.connect_ex((baseIP, port)) #
# Python Port Scanner #

import wxversion
wxversion.select("2.8")

import wx # Import the GUI module wx
import sys # Import the standard library module sys import ping # Import the
ICMP Ping Module
from socket import * # Import the standard library module socket

from time import gmtime, strftime # import time functions

#
# Event Handler for the portScan Button Press #

def portScan(event):
# First, I need to check that the starting port is<ending port value
if portEnd.GetValue() < portStart.GetValue():
```

```

# This is an improper setting # Notify the user and return
dlg ¼ wx.MessageDialog(mainWin,"Invalid Host Port Selection", "Confirm",
wx.OK | wx.ICON_EXCLAMATION)

result¼ dlg.ShowModal() dlg.Destroy()
return

# Update the Status Bar
mainWin.StatusBar.SetStatusText('Executing Port Scan .... Please Wait')
# Record the Start Time

utcStart ¼ gmtime()
utc ¼ strftime("%a, %d %b %Y %X +0000", utcStart)
results.AppendText("\n\nPort Scan Started: "+ utc+ "\n\n")

# Build the base IP Address String
# Extract data from the ip Range and host name user selections # Build a Python
List of IP Addresses to Sweep
baseIP¼ str(ipaRange.GetValue())+

'.'+str(ipbRange.GetValue())+ '.'+str(ipcRange.GetValue())+
'.'+str(ipdRange.GetValue())

# For the IP Addresses Specified, Scan the Ports Specified
for port in range(portStart.GetValue(), portEnd.GetValue()+1):
try:

# Report the IP Address to the Window Status Bar
mainWin.StatusBar.SetStatusText('Scanning:'+ baseIP+' Port:'+str(port))

# open a socket
reqSocket ¼ socket(AF_INET, SOCK_STREAM)
# Try Connecting to the specified IP, Port
response ¼ reqSocket.connect_ex((baseIP, port))
# if we receive a proper response from the port # then display the results
received

if(response ¼¼ 0) :
# Display the ipAddress and Port

```

```
results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Open')
results.AppendText("\n")
```

```
else:
```

```
# if the result failed, only display the result # when the user has selected the
"Display All" check box if displayAll.GetValue() == True:
```

```
results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Closed')
results.AppendText("\n")
```

```
# Close the socket reqSocket.close()
```

```
except socket.error, e:
```

```
# for socket Errors Report the offending IP
```

```
results.AppendText(baseIP+'\t'+str(port)+'\t') results.AppendText('Failed:')
```

```
results.AppendText(e.message)
```

```
results.AppendText("\n")
```

```
# Record and display the ending time of the sweep utcEnd == gmtime()
```

```
utc == strftime("%a, %d %b %Y %X +0000", utcEnd)
```

```
results.AppendText("\nPort Scan Ended: " + utc + "\n\n")
```

```
# Clear the Status Bar
```

```
mainWin.StatusBar.SetStatusText("")
```

```
# End Scan Event Handler
```

```
#
```

```
# Program Exit Event Handler #
```

```
def programExit(event): sys.exit()
```

```
# End Program Exit Event Handler
```

```
#
```

```
# Setup the Application Windows
```

```
app == wx.App()
```

```
# define window
```

```
mainWin == wx.Frame(None, title="Simple Port Scanner", size=(1200,600))
```

```

#define the action panel
panelAction = wx.Panel(mainWin)

#define action buttons
# I'm creating two buttons, one for Scan and one for Exit
# Notice that each button contains the name of the function that will # handle the
button press event. Port Scan and ProgramExit respectively

displayAll = wx.CheckBox(panelAction, -1, 'Display All', (10, 10))
displayAll.SetValue(True)
scanButton = wx.Button(panelAction, label='Scan')
scanButton.Bind(wx.EVT_BUTTON, portScan)
exitButton = wx.Button(panelAction, label='Exit')
exitButton.Bind(wx.EVT_BUTTON, programExit) # define a Text Area where I
can display results
results = wx.TextCtrl(panelAction, style=wx.TE_MULTILINE | wx.
HSCROLL)

# Base Network for Class C IP Addresses has 3 components
# For class C addresses, the first 3 octets define the network i.e 127.0.0 # the last
8 bits define the host i.e. 0-255

# Thus I setup 3 spin controls one for each of the 4 network octets # I also, set
the default value to 127.0.0.0 for convenience

ipaRange = wx.SpinCtrl(panelAction, -1, "") ipaRange.SetRange(0, 255)
ipaRange.SetValue(127)

ipbRange = wx.SpinCtrl(panelAction, -1, "") ipbRange.SetRange(0, 255)
ipbRange.SetValue(0)

ipcRange = wx.SpinCtrl(panelAction, -1, "") ipcRange.SetRange(0, 255)
ipcRange.SetValue(0)

ipdRange = wx.SpinCtrl(panelAction, -1, "") ipdRange.SetRange(0, 255)
ipdRange.SetValue(1)

# Add a label for clarity
ipLabel = wx.StaticText(panelAction, label="IP Address: ")

```

Next, I want to provide the user with the ability to set the port range # they wish to scan. Maximum is 20 - 1025

```
portStart ¼ wx.SpinCtrl(panelAction, -1,") portStart.SetRange(1, 1025)
portStart.SetValue(1)
```

```
portEnd ¼ wx.SpinCtrl(panelAction, -1,") portEnd.SetRange(1, 1025)
portEnd.SetValue(5)
```

```
PortStartLabel ¼ wx.StaticText(panelAction, label¼"Port Start: ") PortEndLabel
¼ wx.StaticText(panelAction, label¼"Port End: ")
```

Now I create BoxSizer to automatically align the different components neatly
First, I create a horizontal Box
I'm adding the buttons, ip Range and Host Spin Controls

```
actionBox ¼ wx.BoxSizer()
```

```
actionBox.Add(displayAll, proportion¼0, flag¼wx.LEFT|wx.CENTER,
border¼5)
```

```
actionBox.Add(scanButton, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(exitButton, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER, border¼5)
```

```
actionBox.Add(ipaRange, proportion ¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipbRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipcRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(ipdRange, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(PortStartLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER,
border¼5)
```

```
actionBox.Add(portStart, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
actionBox.Add(PortEndLabel, proportion¼0, flag¼wx.LEFT|wx.CENTER,
border¼5)
```

```
actionBox.Add(portEnd, proportion¼0, flag¼wx.LEFT, border¼5)
```

```
# Next I create a Vertical Box that I place the Horizontal Box components #  
inside along with the results text area
```

```
vertBox ¼ wx.BoxSizer(wx.VERTICAL)  
vertBox.Add(actionBox, proportion¼0, flag¼wx.EXPAND | wx.ALL,  
border¼5) vertBox.Add(results, proportion¼1, flag¼wx.EXPAND | wx.LEFT |  
wx. BOTTOM | wx.RIGHT, border¼5)
```

```
# I'm adding a menu and status bar to the main window
```

```
mainWin.CreateStatusBar()
```

```
# Finally, I use the SetSizer function to automatically size the windows # based  
on the definitions above
```

```
panelAction.SetSizer(vertBox)
```

```
# Display the main window
```

```
mainWin.Show() # Enter the Applications Main Loop # Awaiting User Actions
```

```
app.MainLoop
```

Now that you have reviewed the code, [Figures 8.11](#) and [8.12](#) depict program execution. The only difference between the two figures is the setting of the Display All checkbox.

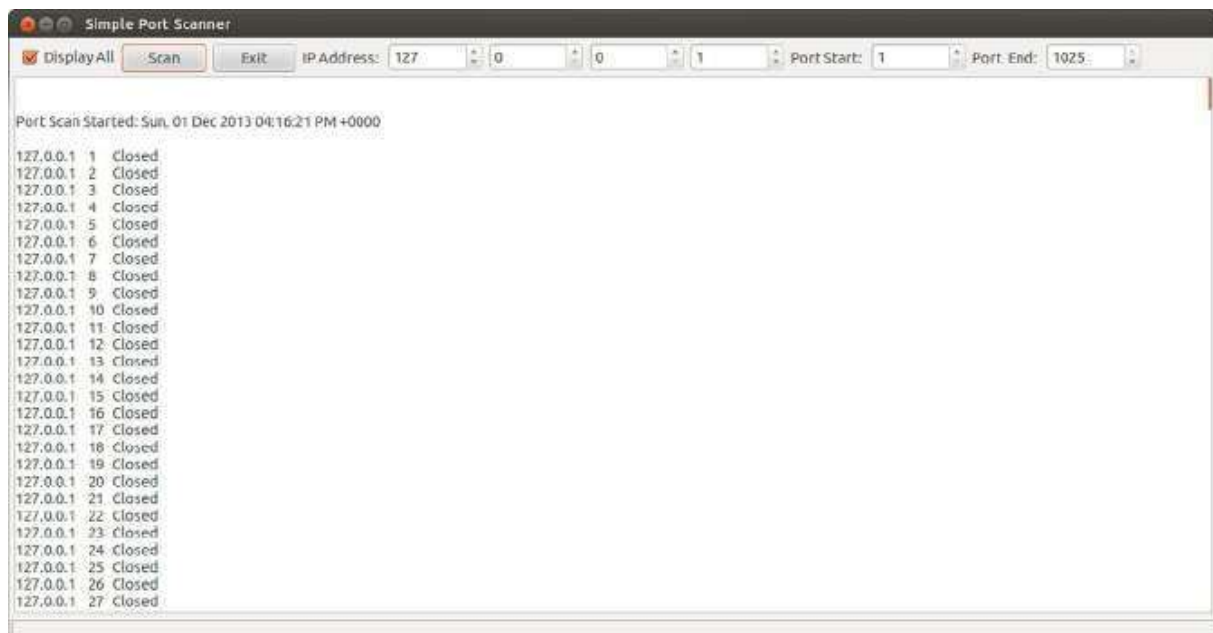


FIGURE 8.11

Port Scanner execution with Display All selected.

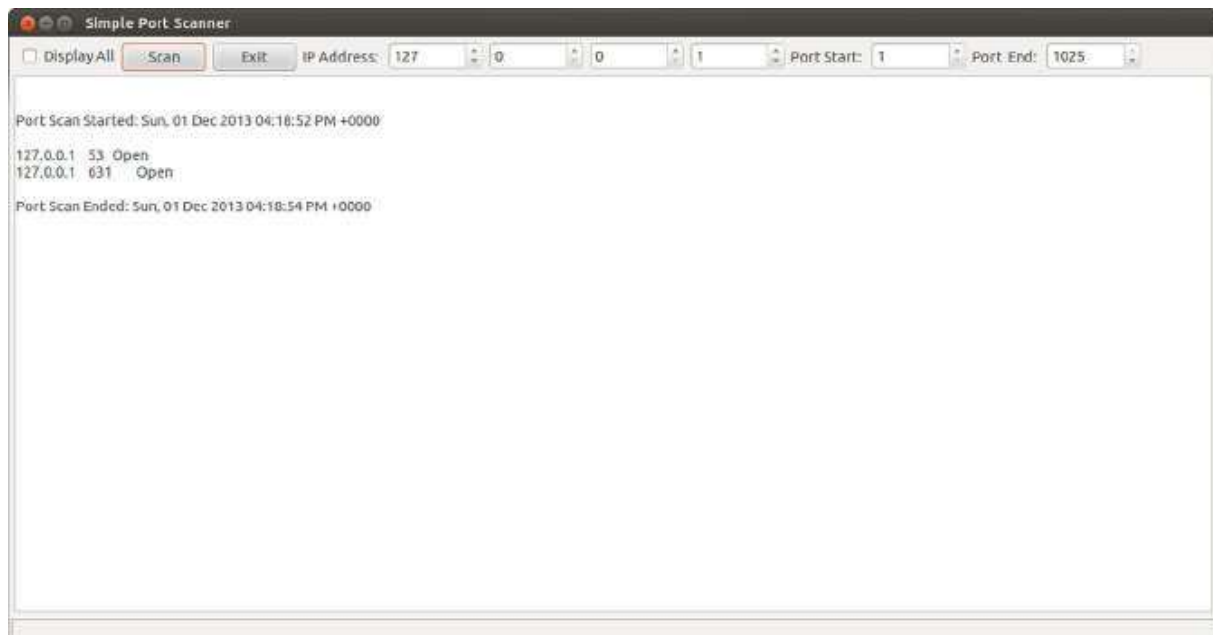


FIGURE 8.12

Port Scanner execution with Display NOT selected.

Additional Resources 235

CHAPTER REVIEW

In this chapter, I first introduced the concept of digital investigation of a network. Then I walked through the fundamentals required to perform basic synchronous network socket operations within Python. I created three programs: (1) Server and (2) Client in order to demonstrate how connections are made. Next, I discussed the concept of a ping sweep and the value it can bring to network-based investigations. I then created a basic ping sweeper application. This application employed two third-party modules: wxPython and ping.py. wxPython was used to create a simple GUI application to control ping sweep operations. Finally, I developed a port scanning application again using a GUI, and created the application entirely in Python.

In [Chapter 9](#) “Network Investigation Part II,” I will expand the Port Scan application to provide examples of OS fingerprinting and provide an example of passive monitoring to identify host and port usage.

SUMMARY QUESTIONS

1. What are the investigative benefits that can be obtained when performing a ping sweep?

2. What are the investigative benefits that can be obtained when performing a port scan?
3. Modify the ping sweep application to save the results of the sweep in a Python list.
4. Modify the ping sweep application and add additional scanning options. For example:
 - a. Program the scan to automatically and repeatedly run at predefined intervals to identify a broader range of endpoints. Keep track of the IPs identified in the list (or you might choose a set for this operation, why would a Python set be beneficial?).
 - b. Create a stealth mode that randomly pings the hosts within the specified address range over a multiday time period.
5. Modify the port scan application to expand the range of allowed ports.
6. Integrate the port scan and ping sweep applications in such a way that port scans will be executed automatically against hosts responding to a ping and ignore nonresponding hosts.

Additional Resources

Internet Corporation for Assigned Numbers and Names, <http://www.icann.org/>.
 The Hunt for Red October, http://www.tomclancy.com/book_display.php?isbn13¼

9780425240335 .

Python Ping Module, <https://pypi.python.org/pypi/ping>.
 wxPython GUI environment, www.wxPython.org.

CHAPTER

Network Forensics: Part II 9

CHAPTER CONTENTS

Introduction	237
Packet Sniffing	238
Raw Sockets in Python	240
What is Promiscuous Mode or Monitor Mode?	240

Setting Promiscuous Mode Ubuntu 12.04 LTS Example	240
Raw Sockets in Python Under Linux	241
Unpacking Buffers	242
Python Silent Network Mapping Tool	247
PSNMT Source Code	249
psnmt.py Source Code	249
decoder.py Source Code	253
commandParser.py	256
classLogging.py Source Code	257
csvHandler.py Source Code	258
Program Execution and Output	259
Forensic Log	260
TCP Capture Example	260
UDP Capture Example	261
CSV File Output Example	261
Chapter Review	262
Summary Question/Challenge	262
Additional Resources	263

INTRODUCTION

As we discovered in [Chapter 8](#), Python has a rich set of Standard Library capabilities to perform network interface, discovery, and analysis. The Ping Sweep and Port Scan applications are quite straightforward once you become familiar with the underlying libraries and modules. However, interactive scanning and probing has several important limitations:

1. In order for the sweeps and scans to be effective, the targets (hosts, routers,

Python Forensics 237 © 2014 Elsevier Inc. All rights reserved.

switches printers, servers) need to be powered on and functional in order to interact with them.

2. The environment that you are working in must be tolerant to these types of

“noisy” scanning activity. Actually, most intrusion prevention systems (IPS) are precisely looking for this type of activity and will categorize them as attacks and respond accordingly, unless they are configured to ignore them. This is more difficult to accomplish than you might think, and most cyber-security folks are not inclined to make configuration changes to their IPS.

3. Port scans are only effective when services owning these ports respond to the probes, as we would expect. Malicious services that we may be searching for do not necessarily play nice and would not respond to these novice inquiries.

4. Many of the malicious services utilize the User Datagram Protocol (UDP) when communicating with their handlers and can operate silently when being probed by vulnerability assessment technologies.

5. Finally, the operators of critical infrastructure environments are unlikely to allow active scanning and probing of those networks, because in many cases the probing activity may disrupt operations and crash systems. If this were to happen in a supervisory control and data acquisition environment, these disruptions or system crashes could shut the lights off for thousands of customers or worse.

PACKET SNIFFING

Network (or packet) sniffing is another method that can be used, and when used properly can provide insight. Network sniffing, again if done well, can provide three critical benefits over port scanning:

1. The sniffer is completely silent and will not place a single packet on the network ensuring zero impact on network operations.
2. The sniffer is an observer that can run for hours, days, weeks, months, or even continuously collecting information that will more completely describe activities of local hosts, servers, network devices, or even rogue devices that were previously undetected.
3. Finally, it can capture activities that are stealthy and only occur periodically or sporadically.

In the primitive form, a packet sniffer (also referred to a network sniffer) captures all of the packets of data that pass through a given network interface. In order to capture these packets, your network interface must be in Promiscuous Mode and the interface needs to be connected to a port that has visibility to all of the packets. For example, if you are interested in a specific subnet, you would

connect the sniffer to a switch or hub on that subnet. Most modern switches today support port mirroring via a Switched Port ANalyzer (SPAN) or Remote Switched Port ANalyzer as shown in [Figures 9.1](#) and [9.2](#). These ports are typically connected to IPS, network monitoring devices or performance measurement devices that can detect network loading.

Packet Sniffing 239

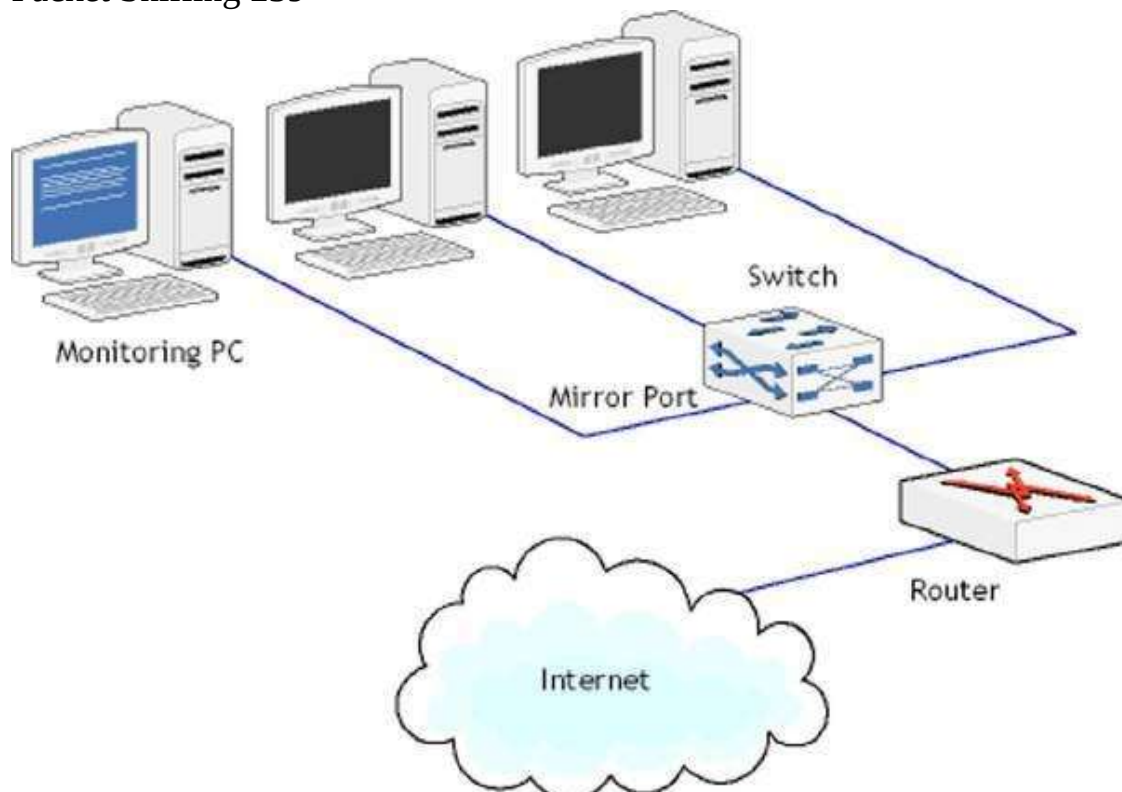


FIGURE 9.1
SPAN port diagram.

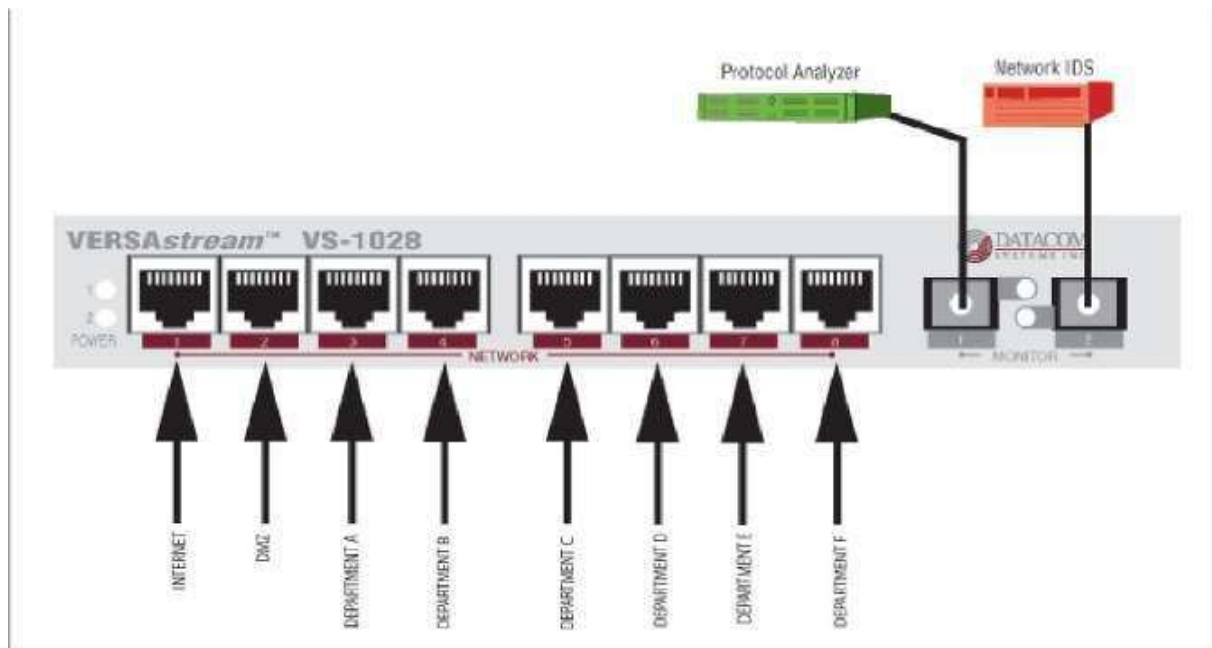


FIGURE 9.2
SPAN port connections.

SPAN ports are most commonly attributed to Cisco (where they were originally referred to as port mirroring). Modern switches can be configured to mirror specific network ports to a common interface used for network monitoring and interface with a variety of security appliances.

RAW SOCKETS IN PYTHON

In order to perform packet sniffing in Python, we need the following:

1. We must be using a network interface card (NIC) that has the ability to operate in Promiscuous Mode.
2. On most modern operating systems—i.e., Windows, Linux, and Mac OS X—you must also have administrator privilege.
3. Once we have accomplished 1&2, we can create a raw socket.

What is Promiscuous Mode or Monitor Mode?

When a capable NIC is placed in Promiscuous Mode, it allows the NIC to intercept and read each arriving network packet in its entirety. If the NIC is not in Promiscuous

Mode, it will only receive packets that are specifically addressed to the NIC. Promiscuous Mode must be supported by the NIC and by the operating system

and any associated driver. Not all NICs support Promiscuous Mode, however it is pretty easy to determine if you have a NIC and OS capable of Promiscuous Mode.

Setting Promiscuous Mode Ubuntu 12.04 LTS Example

On Linux you can place your NIC into Promiscuous Mode by using ifconfig command. (Note administrative rights are required.)

Command to enable Promiscuous Mode:

```
chet@PythonForensics:$ sudo ifconfig eth0 promisc
```

Next I validate the result:

(Notice the message UP BROADCAST RUNNING PROMISC MULTICAST message.)

```
chet@PythonForensics:$ sudo ifconfig
```

```
eth0 Link encap:Ethernet HWaddr 00:1e:8c:b7:6d:64
inet addr:192.168.0.25 Bcast:192.168.0.255 Mask:255.255.255.0 inet6 addr:
fe80::21e:8cff:feb7:6d64/64 Scope:Link
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
RX packets:43284 errors:0 dropped:0 overruns:0 frame:0
TX packets:11338 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:17659022 (17.6 MB) TX bytes:1824060 (1.8 MB)
```

Next I have turned off Promiscuous Mode and validate the result: You notice that the message now states: UP BROADCAST RUNNING MULTICAST without the PROMISC

```
chet@PythonForensics:$ sudo ifconfig eth0 -promisc
```

```
chet@PythonForensics:$ sudo ifconfig eth0
```

```
eth0 Link encap:Ethernet HWaddr 00:1e:8c:b7:6d:64
```

```
inet addr:192.168.0.25 Bcast:192.168.0.255 Mask:255.255.255.0 inet6 addr:
fe80::21e:8cff:feb7:6d64/64 Scope:Link UP BROADCAST RUNNING
MULTICAST MTU:1500 Metric:1 RX packets:43381 errors:0 dropped:0
overruns:0 frame:0 TX packets:11350 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:17668285 (17.6 MB) TX bytes:1827000 (1.8 MB)
```

Once you have determined that you have a NIC capable of entering Promiscuous Mode, we are ready to work with raw sockets in Python.

Raw sockets in Python under Linux

Special Note: I will be using a Linux environment throughout the rest of this chapter. Since the code for handling raw sockets differs between operating systems, the code that interfaces with the raw socket will need to be modified to support Windows.

Creating a raw socket is quite straightforward in Python, as the script below demonstrates. The script accomplishes the following:

- (1) Enable Promiscuous Mode on the NIC
- (2) Create a raw socket
- (3) Capture the next TCP packet passing by the NIC
- (4) Print the contents of the packet
- (5) Disable Promiscuous Mode on the NIC
- (6) Close the raw socket

Note: Script must be run with admin privledge

```
# import the socket and os libraries
import socket
import os
```

```
# issue the command to place the adapter in promiscious mode
ret=os.system("ifconfig eth0 promisc")
# if the command was successful continue if ret == 0:
```

```
# Create a Raw Socket in Linux
# AF_INET specifies ipv4 packets
# SOCK_RAW specifies a raw protocol at the network layer # IPPROTO_TCP
specifies the protocol to capture
```

```
mySocket=socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket IPPROTO_TCP)
```

```
# Receive the next packet up to 255 bytes
# Note this is a synchronous call and will wait until # a packet is received
```

```
recvBuffer, addr=mySocket.recvfrom(255)
```

```
# Print out the contents of the buffer
print recvBuffer
ret=os.system("ifconfig eth0 -promisc)")

else:
# if the system command fails print out a message print'Promiscuous Mode not
Set'
```

The script will create output as shown in [Figure 9.3](#). As you can see the output of the packet contents appears quite cryptic.

Unpacking buffers

Extracting information from a buffer like this one may seem quite tedious, as we must parse the pertinent information from the buffer. In order to deal with buffered data like this one with a defined structure, Python provides the function `unpack()`. Many examples and applications of `unpack()` are available on the web that parse various well-known data structures, however the explanation of how the functions work are typically left to the reader's imagination, or at least requires further research.

For illustration, you will find examples on the web like this one to extract information from an IPv4 header.

```
ipHeader=packet[0:20]
buffer=unpack('!BBHHHBBH4 s4 s', ipHeader)
```

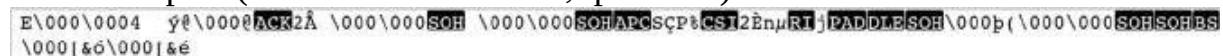


FIGURE 9.3

Raw TCP/IP packet contents.

As with all chapters in this book, I want to make sure that you deeply understand how functions work, so that you can apply them to the challenge at hand, as well as to other problems in the future. The `unpack()` function takes two parameters, the first is a string that defines the format of the data held in the buffer and second is the buffer that needs to be parsed. The function returns a tuple which can be processed just like a list.

In order to figure this out, we must first study the structure of an IPv4 packet header as shown in [Figure 9.4](#) and compare that with the format string “!BBHHHBBH4 s4 s.”

Each of the characters of the format string have a specific meaning which governs the processing of the `unpack()` function. If you get the format string wrong, then you will get garbage back, or if the buffer you pass as the second variable does not conform to the exact format you specify, the results will be incorrect.

Let us examine the meaning of each of the characters in the format string “!BBHHHBBH4 s4 s.” Note: other format specifications exist and are documented within the Standard Library documentation [UNPACK]:

Format Python Type Bytes

! Big Endian

B Integer 1

H Integer 2

s String n

IPv4 Header Format

Offsets Bytes 0 1 2 3

Octet

Bit

0 1 2 3 4 5 6 7 8 9

1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

E

0 0 Version IHL DSCP C Total Length

N

4 32 Identification Flags Fragment Offset

8 64 Time To Live **Protocol** Header Checksum

12 96 **Source IP Address**

16 128 **Destination IP Address**

20 160 Options (if IHL > 5)

FIGURE 9.4

Typical IPv4 packet header.

The first character in the format string represents the byte order of the data, for network packets this is in big endian format. This is represented as the exclamation point that is the first character in the format specification. You could also use the “>” greater than sign which also means big endian, but the explanation point is there for those of us who can never remember the typical

byte order of network data. Actually, it is good to use the exclamation point because this immediately identifies the format string as relating to a network packet.

The table below provides a mapping of each of the format characters related to the IPv4 header.

"!BBHHHBBH4 s4 s"

Size Mapping to Format (Bytes) IPv4

Definition

B 1 Version and IHL

B 1 DSCP and ECN

H 2 Total length

H 2 Identification

H 2 Flags and fragment offset

B 1 Time to live (TTL)

B 1 Protocol

H 2 Header Checksum value

4 s 4 Source IP address

4 s 4 Destination IP address 4-bit version field (this will be 4, for IPv4) 4-bit Internet Header Length representing the number of 32 bit words contained in the header

7-bit Differentiated Services Code Point 1-bit Congestion Notification

16 bits defines the entire packet size 16 bits identifier for a group of IP fragments

3-bit fragmentation flag

13-bit fragment offset value

8-bit TTL value to prevent packet looping

8-bit value identifying the protocol used in the data portion of the packet

16-bit checksum value for error detection

4-byte source IP address

4-byte destination IP address

Now that you understand the meaning of the format string and the basic unpack() function. The following code will unpack an IPv4 header and then extract each of the fields into variables for processing. I also included the code

here to convert the source and destination IP addresses into the human readable forms using the built-in socket method.

```
# unpack an IPv4 packet
# note the packet variable is a buffer returned from # a socket.recvfrom() method
that was illustrated in Figure 9.3.
ipHeaderTuple¼unpack('!BBHHHBBH4s4s', packet)
# Field Contents
```

```
verLen ¼ipHeaderTuple[0] dscpECN ¼ipHeaderTuple[1]
packetLength¼ipHeaderTuple[2] packetID
flagFrag
timeToLive
protocol
checksum
sourceIP
destIP
¼ipHeaderTuple[3] ¼ipHeaderTuple[4] ¼ipHeaderTuple[5] ¼ipHeaderTuple[6]
¼ipHeaderTuple[7] ¼ipHeaderTuple[8] ¼ipHeaderTuple[9] # Field 0: Version
and Length # Field 1: DSCP and ECN
# Field 2: Packet Length # Field 3: Identification # Field 4: Flags/Frag Offset #
Field 5: Time to Live (TTL) # Field 6: Protocol Number # Field 7: Header
Checksum # Field 8: Source IP
# Field 9: Destination IP
```

```
# Convert the sourceIP and destIP into a # standard dotted-quad string
representation # for example '192.168.0.5'
```

```
sourceAddress¼socket.inet_ntoa(sourceIP); destAddress
¼socket.inet_ntoa(destIP);
# Extract the version and header size, this will give # us the offset to the data
portion of the packet
```

```
version
length
ipHdrLength fragOffset fragment
¼verLen>>4 # get upper nibble version ¼verLen & 0x0F # get lower nibble
header length ¼length * 4 # calculate the hdr size in bytes ¼flagFrag & 0x1FFF
# get lower 13 bits... ¼fragOffset*8# calculate start of fragment
```

Next, we would use the same process to extract fields from the data portion of the packet, which in this case is the TCP header. You can determine the type of the data portion of the packet by examining the protocol field. [Figure 9.5](#) depicts a typical TCP header and the format string “!HLLBBHHH” along with the `unpack()` function, this can be used to extract the individual fields of the TCP header.

```
# By using the results of the IPv4 header unpacking, we can # strip the TCP
Header from the original packet
```

```
# Note the ipHdrLength is the offset from the beginning of # the buffer. The
standard length of a TCP packet is 20 # bytes. For our purposes these 20 bytes
contain the # the pertinent information we are looking for
```

```
stripTCPHeader¼packet[ipHdrLength:ipHdrLength+ 20]
```

TCP Header

```
Offset Bytes 0 1 2 3
Octet BITS 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
00 Source port Destination port
4 32 Sequence number
8 64 Acknowledgment number (if ACK set)
12
96
Data offset
Reserved
N
C E U A P R S F
W C R C S S Y I Window Size 0 0 S R E G K H T N N
16 128 Checksum Urgent pointer (if URG set)
20 160 Options (if data offset > 5. Padded at the end with "0" bytes if necessary.) ... ..
```

FIGURE 9.5

Typical TCP packet header.

```
# unpack returns a tuple, for illustration I will extract # each individual values
using the unpack() function
```

```
tcpHeaderBuffer¼unpack('!HLLBBHHH', stripTCPHeader)
```

```
sourcePort
destinationPort
sequenceNumber
acknowledgement
dataOffsetandReserve tcpHeaderLength
flags
```

```

FIN
SYN
RST
PSH
ACK
URG
ECE
CWR
windowSize
tcpChecksum
urgentPointer
¼tcpHeaderBuffer[0]
¼tcpHeaderBuffer[1]
¼tcpHeaderBuffer[2]
¼tcpHeaderBuffer[3]
¼tcpHeaderBuffer[4]
¼(dataOffsetandReserve >>4) * 4 ¼tcpHeaderBuffer[5]
¼flags & 0x01
¼(flags>>1) & 0x01
¼(flags>>2) & 0x01
¼(flags>>3) & 0x01
¼(flags>>4) & 0x01
¼(flags>>5) & 0x01
¼(flags>>6) & 0x01
¼(flags>>7) & 0x01
¼tcpHeaderBuffer[6]
¼tcpHeaderBuffer[7]
¼tcpHeaderBuffer[8]

```

Now that we have the basics under our belt, let us look at:

- How to place our NIC into Promiscuous Mode 1.
2. How to create a raw socket in Linux
3. How to unpack the packet to obtain the individual fields

Python Silent Network Mapping Tool 247

We are ready to build an application that can capture network packets and extract the information that will allow us to monitor traffic.

PYTHON SILENT NETWORK MAPPING TOOL (PSNMT)

Now that we have the basics for sniffing a network packet, I need to parse the data and extract the information I need. For this example, I am not interested in collecting packets and simply printing the results, rather I want to achieve the following objectives:

- (1) Collect IP addresses that are active on the network I am monitoring. (I plan to leave the monitor in place for a long period of time to capture network devices that only turn on periodically or sporadically.)
- (2) Collect IP addresses of remote computers that are interacting with my local network. These could be web, mail, or a plethora of Cloud services.
- (3) Collect service ports being used by local and/or remote computers. Specifically, I am interested in “Well Defined Ports”: 0-1023 or “Registered Ports”: 1024-49151.
- (4) Next I wish to report only unique entries. In other words, if the local host 192.168.0.5 is discovered and is found to be using host port 80, I only want to see that unique entry once, not each time it is discovered.
- (5) Finally, to limit the scope of the program, I want to collect only TCP or UDP packets within an IPv4 environment. The program can be easily expanded to handle other protocols and IPv6 in the future.

In order to meet the requirements stated above I only need to extract the following fields from the headers:

- (1) Protocol
- (2) Source IP address
- (3) Destination IP address
- (4) Source port
- (5) Destination port

Examining [Figures 9.4](#) and [9.5](#), the Protocol field, along with the Source and Destination IP addresses exist in the IPv4 header, while the Source and Destination ports are in the TCP header. This means I will have to parse out both headers to obtain the needed information. I have also included [Figure 9.6](#) which depicts the UDP header, which I also use to handle UDP packet extraction.

There are several technical issues that need to be addressed along with the high level requirements:

(1) What type of data element should I use to store the information collected? a. I am going to use a simple list to hold the data collected from the packets and append data to the lists for each packet received.

```
ipObservations¼[]
```

UDP Header

Offsets Bytes 0 1 2 3

Octet BITS 0 1 2 3 4 5 6 7 8 9 1011 12131415161718 19202122232425 262728293031

00 **Source port Destination port**

4 32 Length Checksum

FIGURE 9.6

Typical UDP packet header.

(2) Since the `socket.recvfrom()` method is synchronous, how will I signal when to stop collection, and how will I limit the time of the collection activities? a. I am going to use the Python Standard Library `signal` module and integrate

this into the collection loop. I set this up by first creating a class `myTimeout` that will be raised by a handler when a specified time has expired. I then integrate the `myTimeout` exception handler into the `try/except` handler of the receive packet loop.

```
class myTimeout(Exception): pass
```

```
def handler(signum, frame):
```

```
    print'timeout received', signum
    raise myTimeout()
```

```
# Set the signal handler
```

```
signal.signal(signal.SIGALRM, handler)
```

```
# set the signal to expire in n seconds
signal.alarm(n)
```

```
...
```

```
...
```

```
try:
```

```
    while True:
```

```
        recvBuffer, addr¼mySocket.recvfrom(65535)
```

```
        src,dst ¼decoder.PacketExtractor(recvBuffer,\ False)
```

```
        sourceIPObservations.append(src)
```

```
        destinationIPObservations.append(dst)
```

except myTimeout: pass

(3) How will I create only unique entries?

a. The code above will record every pair of source IP/Port and destination IP/Port, with a result being an unsorted list and will contain duplicate entries. To solve this problem, once the collection is complete, I will use a little knowledge of Python data types to help here. Once collection is completed (for the entire time frame), I first convert the list into a set, this will immediately collapse any duplicates (as this is a fundamental property of sets). Then I will convert the set back to a list and then sort the list.

```
uniqueSrc = set(map(tuple, ipObservations))
finalList = list(uniqueSrc)
finalList.sort()
```

(4) How should I output the results?

a. In order to provide a workable list, the program will generate a commaseparated value (CSV) file that can then be further processed or examined in a worksheet.

PSNMT SOURCE CODE

The source code is broken into the following five source files. Each of the files contains detailed comments describing all aspects of the program. [Figure 9.7](#) depicts the WingIDE environment for this application.

Source Purpose

psnmt.py

decoder.py

_commandparser.py _csvHandler.py

_classLogging.py Main program setup and loop

Decoder for raw packets

Parser for the user command line Handler for creating/writing csv file output

Class for handling forensic logging

psnmt.py source code

#

Python Passive Network Monitor and Mapping Tool


```
# Import Standard Library Modules
import socket
import signal
import os
import sys

# network interface library used for raw sockets
# generation of interrupt signals i.e. timeout
# operating system functions i.e. file I/o
# system level functions i.e. exit()
```

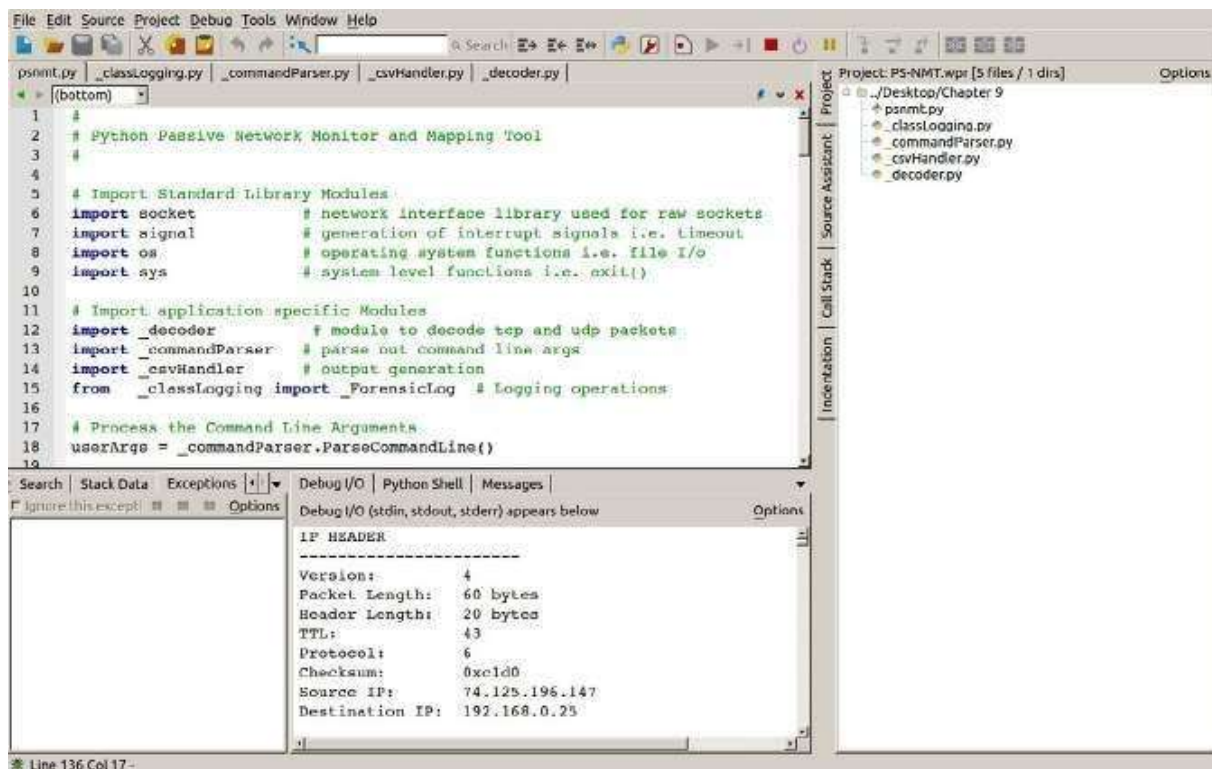


FIGURE 9.7

WingIDE environment for the PSNMT application.

```
# Import application specific Modules
import decoder
import _commandParser
import _csvHandler # module to decode tcp and udp packets
# parse out command line args
# output generation

from _classLogging import _ForensicLog # Logging operations
# Process the Command Line Arguments
userArgs¼_commandParser.ParseCommandLine()

# create a log object
logPath¼os.path.join(userArgs.outPath,"ForensicLog.txt")
```

```

oLog¼_ForensicLog(logPath)

oLog.writeLog("INFO", "PS-NMT Started")
csvPath¼os.path.join(userArgs.outPath,"ps-nmtResults.csv")
oCSV¼_csvHandler._CSVWriter(csvPath)
# Setup the protocol to capture
if userArgs.TCP:

PROTOCOL ¼socket.IPPROTO_TCP elif userArgs.UDP:
PROTOCOL¼socket.IPPROTO_UDP else:
print'Capture protocol not selected'
sys.exit()

# Setup whether output should be verbose
if userArgs.verbose:

VERBOSE ¼True else:
VERBOSE¼False

# Calculate capture duration
captureDuration¼userArgs.minutes * 60
# Create timeout class to handle capture duration
class myTimeout(Exception): pass
# Create a signal handler that raises a timeout event # when the capture duration
is reached

def handler(signum, frame):
print'timeout received', signum raise myTimeout()

# Enable Promiscuous Mode on the NIC
ret¼os.system("ifconfig eth0 promisc")
if ret ¼¼ 0:

oLog .writeLog("INFO",'Promiscuous Mode Enabled') # create an INET, raw
socket
# AF_INET specifies # SOCK_RAW specifies # IPPROTO_TCP or UDP ipv4
a raw protocol at the network layer Specifies the protocol to capture

try : mySocket¼socket.socket(socket.AF_INET, socket.SOCK_RAW,
PROTOCOL)

```

```

oLog.writeLog("INFO",'Raw Socket Open')

except :
# if socket open fails
oLog.writeLog("ERROR",'Raw Socket Open Failed') del oLog
if VERBOSE:

print'Error Opening Raw Socket'
sys.exit()
# Set the signal handler to the duration specified by the user
signal.signal(signal.SIGALRM, handler) signal.alarm(captureDuration)

# create a list to hold the results from the packet capture # I'm only interested in
Protocol Source IP, Source Port, Destination IP, Destination Port

ipObservations¼[]
# Begin receiving packets until duration is received # the inner while loop will
execute until the timeout
try:
while True:
# attempt receive (this call will wait) recvBuffer, addr¼mySocket.recvfrom(255)
# decode the received packet
content¼decoder.PacketExtractor(recvBuffer, VERBOSE)
# append the results to our list ipObservations.append(content)

# write details to the forensic log file oLog.writeLog('INFO', \
'RECV:'þcontent[0]þ\
'SRC :'þcontent[1]þ\
'DST :'þcontent[3])

except myTimeout: pass

# Once time has expired disable Promiscuous Mode ret¼os.system("ifconfig eth0
-promisc") oLog.writeLog("INFO",'Promiscious Mode Disabled')

# Close the Raw Socket
mySocket.close()
oLog.writeLog("INFO",'Raw Socket Closed')

# Create unique sorted list

```

```

uniqueSrc = set(map(tuple, ipObservations))
finalList = list(uniqueSrc)
finalList.sort()
# Write each unique sorted packet to the csv file for packet in finalList:

oCSV.writeCSVRow(packet)
oLog.writeLog('INFO', 'Program End')

# Close the Log and CSV objects del oLog
del oCSV

else:
print 'Promiscuous Mode not Set'
decoder.py source code
# Packet Extractor / Decoder Module #
import socket, sys from struct import *

# Constants
PROTOCOL_TCP = 6
PROTOCOL_UDP = 17

# PacketExtractor
#
# Purpose: Extracts fields from the IP, TCP and UDP Header #
# Input: packet: buffer from socket.recvfrom() method # displaySwitch: True:
Display the details, False omits # Output: result list containing
# protocol, srcIP, srcPort, dstIP, dstPort #
def PacketExtractor(packet, displaySwitch):

#Strip off the first 20 characters for the ip header stripPacket = packet[20:]
#now unpack them
ipHeaderTuple = unpack('!BBHHHBBH4s4s', stripPacket)
# unpack returns a tuple, for illustration I will extract # each individual values

verLen = ipHeaderTuple[0] dscpECN = ipHeaderTuple[1] # Field Contents
# Field 0: Version and Length # Field 1: DSCP and ECN
packetLength = ipHeaderTuple[2] packetID
flagFrag
= ipHeaderTuple[3] = ipHeaderTuple[4]

timeToLive protocol checksum sourceIP destIP
= ipHeaderTuple[5] = ipHeaderTuple[6] = ipHeaderTuple[7] = ipHeaderTuple[8]

```

```
print'Destination IP:' + str(destinationAddress)
# _____
if protocol == IPPROTO_TCP:
    stripTCPHeader = packet[ipHdrLength:ipHdrLength + 20]
    # unpack returns a tuple, for illustration I will extract # each individual values
```

```

tcpHeaderBuffer¼unpack('!HLLBBHHH', stripTCPHeader) sourcePort
destinationPort sequenceNumber acknowledgement
dataOffsetandReserve¼tcpHeaderBuffer[4]
tcpHeaderLength
tcpChecksum
¼(dataOffsetandReserve >4)* 4 ¼tcpHeaderBuffer[7]
¼tcpHeaderBuffer[0] ¼tcpHeaderBuffer[1] ¼tcpHeaderBuffer[2]
¼tcpHeaderBuffer[3]

if displaySwitch : print
print'TCP Header' print'_____ '

print'Source Port: 'þstr(sourcePort) print'Destination Port :'þstr(destinationPort)
print'Sequence Number : 'þstr(sequenceNumber) print'Acknowledgement :
'þstr(acknowledgement) print'TCP Header Length: 'þstr(tcpHeaderLength)þ
'bytes '
print'TCP Checksum:'þhex(tcpChecksum)
print

return(['TCP', sourceAddress, sourcePort, destinationAddress, destinationPort])
elif protocol ¼¼ PROTOCOL_UDP:
stripUDPHeader¼packet[ipHdrLength:ipHdrLength + 8]
# unpack returns a tuple, for illustration I will extract # each individual values
using the unpack() function
udpHeaderBuffer¼unpack('!HHHH', stripUDPHeader)

sourcePort ¼udpHeaderBuffer[0] destinationPort¼udpHeaderBuffer[1]
udpLength ¼udpHeaderBuffer[2] udpChecksum ¼udpHeaderBuffer[3]

if displaySwitch :
print
print'UDP Header'
print'_____ '

print'Source Port: 'þstr(sourcePort) print'Destination Port :'þstr(destinationPort)
print'UDP Length: 'þstr(udpLength)þ'bytes' print'UDP
Checksum:'þhex(udpChecksum) print

```

```

return(['UDP', sourceAddress, sourcePort, destinationAddress, destinationPort])
else:
# For expansion protocol support
if displaySwitch:
print'Found Protocol :'\bstr(protocol)
return(['Unsupported',sourceAddress,0,\ destinationAddress,0])
commandParser.py

#
# PSNMT Argument Parser #

import argparse

import os # Python Standard Library - Parser for command-line options,
arguments
# Standard Library OS functions

# Name: ParseCommand() Function
#
# Desc: Process and Validate the command line arguments
# use Python Standard Library module argparse
#
# Input: none
#
# Actions:
# Uses the Standard Library argparse to process the command line #
def ParseCommandLine():

parser¼argparse.ArgumentParser('PS-NMT')
parser.add_argument('- v', '--verbose', help¼'Display packet details',
action¼'store_true')

# setup a group where the selection is mutually exclusive and required.
group¼parser.add_mutually_exclusive_group(required¼True)
group.add_argument('--TCP', help¼'TCP Packet Capture', action¼'store_true')
group.add_argument('--UDP', help¼'UDP Packet Capture', action¼'store_true')
parser.add_argument('- m', '--minutes',help¼'Capture Duration in
minutes',type¼int)

```

```
parser.add_argument('- p', '--outPath', type=ValidateDirectory, required=True,
help="Output Directory")
```

```
theArgs=parser.parse_args()
```

```
return theArgs
```

```
# End Parse Command Line
```

```
def ValidateDirectory(theDir):
```

```
# Validate the path is a directory
```

```
if not os.path.isdir(theDir):
```

```
raise argparse.ArgumentTypeError('Directory does not exist') # Validate the path
is writable
```

```
if os.access(theDir, os.W_OK):
```

```
return theDir
```

```
else:
```

```
raise argparse.ArgumentTypeError('Directory is not writable') #End
```

```
ValidateDirectory
```

```
classLogging.py source code
```

```
import logging
```

```
#
```

```
# Class: _ForensicLog
```

```
#
```

```
# Desc: Handles Forensic Logging Operations
```

```
#
```

```
# Methods constructor: Initializes the Logger
```

```
# writeLog: Writes a record to the log
```

```
# destructor: Writes message and shutdown the logger
```

```
class _ForensicLog:
```

```
def __init__(self, logName): try:
```

```
# Turn on Logging
```

```
logging.basicConfig(filename=logName,level=logging.DEBUG,format'%(
asctime)s %(message)s')
```

```
except :
```

```
print "Forensic Log Initialization Failure ... Aborting" exit(0)
```



```
def writeLog(self, logType, logMessage): if logType == "INFO":
logging.info(logMessage)
```

```
elif logType == "ERROR":
logging.error(logMessage)
elif logType == "WARNING":
logging.warning(logMessage)
else:
logging.error(logMessage)
return
```

```
def __del__(self):
logging.info("Logging Shutdown") logging.shutdown()
csvHandler.py source code
import csv #Python Standard Library - for csv files
```

```
#
# Class: _CSVWriter
#
# Desc: Handles all methods related to comma separated value operations #
# Methods constructor: Initializes the CSV File
# writeCSVRow: Writes a single row to the csv file # writerClose: Closes the
CSV File
```

```
class _CSVWriter:
```

```
def __init__(self, fileName): try:
# create a writer object and then write the header row
self.csvFile=open(fileName, 'w b')
self.writer=csv.writer(self.csvFile, delimiter=',', quoting=csv.QUOTE_ALL)
self.writer.writerow(('Protocol','Source IP','Source Port', 'Destination
IP','Destination Port'))
```

```
except:
log.error('CSV File Failure')
def writeCSVRow(self, row):
self.writer.writerow((row[0], row[1], str(row[2]), row [3], str(row[4]) ) )
def __del__(self):
self.csvFile.close()
```

PROGRAM EXECUTION AND OUTPUT

As you can see, the psnmt Python application is constructed as a command line application. This makes sense for this type of application because it is likely to be executed as a cron job (scheduled to run at certain times for example).

The command line has the following parameters:

Parameter Purpose and Usage

- v Verbose: Writes intermediate results to standard output when specified
- m Minutes: Duration in minutes to perform collection activities
- TCP | -UDP Protocol: Defines the application protocol to capture
- p Produces: Defines the output directory for the forensic log and csv file

Example command line:

```
sudo Python psnmt -v -TCP -m 60 -p /home/chet/Desktop
```

Note: sudo (is used to force the execution of the command with administrative rights, which is required). This command will capture TCP packets for 60 min, produce verbose output and create a log and csv file on the user's desktop.

[Figures 9.8](#) and [9.9](#) show sample executions for both TCP and UDP captures. These runs created both csv and forensic log files and entries.

```

ch0t@PythonForensics: ~/Desktop/Chapter 9
ch0t@PythonForensics:~/Desktop/Chapter 9$ sudo python ps-nmt.py -v -m 2 --TCP -p /hone/chet/Desktop/Chapter\ 9
IP HEADER
-----
Version:      4
Packet Length: 60 bytes
Header Length: 20 bytes
TTL:          43
Protocol:     6
Checksum:     0x5227
Source IP:    74.125.196.147
Destination IP: 192.168.0.25

TCP Header
-----
Source Port:      443
Destination Port : 56652
Sequence Number : 621577023
Acknowledgement : 2310411757
TCP Header Length: 40 bytes
TCP Checksum:     0x6d8c

IP HEADER
-----
Version:      4
Packet Length: 60 bytes
Header Length: 20 bytes
TTL:          43
Protocol:     6
Checksum:     0xf9f3
Source IP:    74.125.196.147
Destination IP: 192.168.0.25

TCP Header
-----
Source Port:      443
Destination Port : 56653
Sequence Number : 1824497575
Acknowledgement : 1049873182
TCP Header Length: 40 bytes
TCP Checksum:     0x6d67

```

FIGURE 9.8
psnmt TCP sample run.

```
ch3t@PythonForensics: ~/Desktop/Chapter 9
ch3t@PythonForensics:~/Desktop/Chapter 9$ sudo python psnnt.py -v -n 2 --UDP -p '/home/ch3t/Desktop/Chapter 9'
IP HEADER
-----
Version:      4
Packet Length: 61 bytes
Header Length: 20 bytes
TTL:          64
Protocol:     17
Checksum:     0x98fe
Source IP:    127.0.0.1
Destination IP: 127.0.0.1

UDP Header
-----
Source Port:   52309
Destination Port : 53
UDP Length:    41 bytes
UDP Checksum:  0xfe3c

IP HEADER
-----
Version:      4
Packet Length: 121 bytes
Header Length: 20 bytes
TTL:          58
Protocol:     17
Checksum:     0x86f7
Source IP:    66.153.128.98
Destination IP: 192.168.0.25

UDP Header
-----
Source Port:   53
Destination Port : 23992
UDP Length:    101 bytes
UDP Checksum:  0x474f

IP HEADER
```

FIGURE 9.9
psnmt UDP sample run.

Forensic log

TCP capture example

```
2014-01-19 11:29:51,050 PS-NMT Started
2014-01-19 11:29:51,057 Promiscuous Mode Enabled
2014-01-19 11:29:51,057 Raw Socket Open
2014-01-19 11:29:55,525 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:29:55,526 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:29:56,236 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,270 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,270 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
```

2014-01-19 11:29:56,271 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,527 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,543 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,544 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:29:56,546 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:30:37,437 RECV: TCP SRC : 66.153.250.240 DST :
192.168.0.25
2014-01-19 11:30:37,449 RECV: TCP SRC : 66.153.250.240 DST :
192.168.0.25
2014-01-19 11:30:54,546 RECV: TCP SRC : 173.194.45.79 DST : 192.168.0.25
2014-01-19 11:30:55,454 RECV: TCP SRC : 74.125.196.147 DST :
192.168.0.25
2014-01-19 11:31:35,487 RECV: TCP SRC : 66.153.250.240 DST :
192.168.0.25
2014-01-19 11:31:51,063 Promiscuous Mode Disabled
2014-01-19 11:31:51,064 Raw Socket Closed 2014-01-19 11:31:51,064 Program
End 2014-01-19 11:31:51,064 Logging Shutdown

UDP capture example

2014-01-19 13:27:09,366 PS-NMT Started
2014-01-19 13:27:09,371 Promiscuous Mode Enabled
2014-01-19 13:27:09,372 Raw Socket Open
2014-01-19 13:27:09,528 Logging Shutdown
2014-01-19 13:36:33,472 PS-NMT Started
2014-01-19 13:36:33,477 Promiscuous Mode Enabled
2014-01-19 13:36:33,477 Raw Socket Open
2014-01-19 13:36:45,234 Logging Shutdown
2014-01-19 13:37:51,748 PS-NMT Started
2014-01-19 13:37:51,754 Promiscuous Mode Enabled
2014-01-19 13:37:51,754 Raw Socket Open
2014-01-19 13:37:59,534 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
2014-01-19 13:37:59,546 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
2014-01-19 13:37:59,546 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1

2014-01-19 13:37:59,549 RECV: UDP SRC : 66.153.162.98 DST : 192.168.0.25
 2014-01-19 13:38:09,724 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
 2014-01-19 13:38:09,879 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
 2014-01-19 13:38:09,880 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
 2014-01-19 13:38:10,387 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
 2014-01-19 13:38:10,551 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
 2014-01-19 13:38:10,551 RECV: UDP SRC : 127.0.0.1 DST : 127.0.0.1
 2014-01-19 13:38:45,114 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
 2014-01-19 13:38:46,112 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
 2014-01-19 13:39:45,410 RECV: UDP SRC : 66.153.128.98 DST : 192.168.0.25
 2014-01-19 13:39:51,760 Promiscuous Mode Disabled
 2014-01-19 13:39:51,761 Raw Socket Closed
 2014-01-19 13:39:51,761 Program End
 2014-01-19 13:39:51,761 Logging Shutdown

CSV file output example

Figures 9.10 and 9.11 depict a sample output of the CSV file created by PSNMT.

	A	B	C	D	E
1	Protocol	Source IP	Source Port	Destination IP	Destination Port
2	TCP	127.0.0.1	36480	127.0.0.1	54792
3	TCP	127.0.0.1	54792	127.0.0.1	36480
4	TCP	66.153.250	443	192.168.0.25	35027
5	TCP	74.125.190	443	192.168.0.25	56580
6	TCP	74.125.190	443	192.168.0.25	56581

FIGURE 9.10

Sample TCP output file shown in Excel.

	A	B	C	D	E
1	Protocol	Source IP	Source Port	Destination IP	Destination Port
2	UDP	127.0.0.1	53	127.0.0.1	35633
3	UDP	127.0.0.1	53	127.0.0.1	51420
4	UDP	127.0.0.1	53	127.0.0.1	52309
5	UDP	127.0.0.1	35633	127.0.0.1	53
6	UDP	127.0.0.1	51420	127.0.0.1	53
7	UDP	127.0.0.1	52309	127.0.0.1	53
8	UDP	66.153.128	53	192.168.0.25	11303
9	UDP	66.153.128	53	192.168.0.25	23992
10	UDP	66.153.128	53	192.168.0.25	35021
11	UDP	66.153.128	53	192.168.0.25	43421
12	UDP	66.153.128	53	192.168.0.25	56857
13	UDP	66.153.128	53	192.168.0.25	58487
14	UDP	66.153.16	53	192.168.0.25	23992

FIGURE 9.11

Sample UDP output file shown in Excel.

CHAPTER REVIEW

In this chapter, I introduced the raw sockets and how they can be utilized to capture network packets. In order to do so, I explained the Promiscuous Mode of modern network adapters and demonstrated how to configure a NIC for this operation. I also discussed the importance of network sniffing in order to more thoroughly map and monitor network activities. I used the `unpack()` function to extract each field from the IPv4, TCP and UDP packets and described in detail, how to apply this to buffered data. Finally, I created an application that collects and records either TCP or UDP traffic isolating unique Source IP, Source Port, Destination IP, and Destination Port combinations. This Python application can be controlled through command line arguments in order to easily integrate with cron jobs or other scheduling mechanisms. As part of the command line, I used a signaling mechanism to stop packet collection after a specified time period has expired.

SUMMARY QUESTION/CHALLENGE

I decided to pose only one question in this chapter, more of a challenge problem actually that will take the PSNMT application to the next level:

In order to make this current application more useful, the translation of port 1. numbers into well-known services is necessary in order to determine the likely use or uses of each IP address that we extracted from the captured packets. Using the reserved and well-known port usage maps provided and defined by the Internet Engineering Task Force [IETF], (or by reading /etc/services) expand the

Additional Resources 263

PSNMT application to isolate each local IP address and list the services that they are likely running on each. When possible, attempt to broadly classify the operating system behind each IP address through the examination of port usage (at least classify each IP as Windows, Linux, or other).

Additional Resources

<http://docs.Python.org/2/library/struct.html?highlight¼unpack#struct.unpack>.

<http://www.ietf.org/assignments/service-names-port-numbers/service-names-port-numbers.txt>.

CHAPTER

Multiprocessing for Forensics 10

CHAPTER CONTENTS

Introduction	265
What is Multiprocessing?	266
Python Multiprocessing Support	266
Simplest Multiprocessing Example	269
Single Core File Search Solution	270
Multiprocessing File Search Solution	270
Multiprocessing File Hash	272
Single Core Solution	272
Multi-core Solution A	274
Multi-core Solution B	277
Multiprocessing Hash Table Generation	

.....	279
Single Core Password Generator Code	280
Multi-core Password Generator	283
Multi-core Password Generator Code	283
Chapter Review	288
Summary Question/Challenge	288
Additional Resources	288

INTRODUCTION

As the footprint of digital evidence and digital crime widens, our ability to perform examinations in a timely fashion is clearly strained. According to DFI News “The backlog of caseloads from law enforcement agencies has grown from weeks to months worldwide. Digital forensic specialists cannot be trained fast enough and the number of specialists required to analyze the mountains of digital evidence in common crimes is simply beyond budget constraints” (DFI, 2013). Many digital investigation tools today are single threaded, meaning that the software can only execute one command at a time. These tools were invariably originally developed before multi-core processors were commonplace. In this chapter, I introduce the multiprocessing capabilities of Python that relate to some common forensic challenges. In [Chapter 11](#), I will transfer these applications to the cloud and demonstrate how to increase performance by expanding access to additional cores and allowing forensic operations to be performed on a cloud platform.

Python Forensics 265 © 2014 Elsevier Inc. All rights reserved.

WHAT IS MULTIPROCESSING?

Simply stated multiprocessing is the simultaneous execution of a program on two or more central processing units (CPUs) or cores. In order to provide significant performance improvements, the developer of forensic applications must define areas of the code that have the following characteristics:

The code is processor-intensive.(1)

(2) It is possible to break the code apart into separate processing threads that can

execute in parallel.

(3) The processing between threads can be load-balanced. In other words, the goal is to distribute processing such that each of the threads complete at approximately the same time.

As you may already surmise, the problem with general purpose forensic tools is this: if they were not engineered to meet the objectives above from the start, adapting them to modern multi-core architectures may prove to be difficult. In addition, licensing of such technologies to run simultaneously on multi-cores or in the cloud may be cost-prohibitive. Finally, many of the most commonly used forensic tools operate within the Windows environment which is not ideally suited to run simultaneously on thousands of cores.

PYTHON MULTIPROCESSING SUPPORT

The Python Standard Library includes the package “multiprocessing” (Python multiprocessing module). Using the Python Standard Library for multiprocessing is a great place to begin multiprocessing and will ensure compatibility across a wide range of computing platforms, including the cloud. We utilize the multiprocessing package in typical fashion starting by importing the package multiprocessing. After import of the package, I execute the `help(multiprocessing)` function to reveal the details. I have eliminated some of the extraneous data produced by `help`, and I have highlighted the functions and classes that I will be utilizing to develop a couple of multiprocessing examples.

```
import multiprocessing
help(multiprocessing)
# NOTE this is only a partial excerpt from the output # of the Help command
Help on package multiprocessing:
NAME multiprocessing
Python Multiprocessing Support 267
```

```
PACKAGE CONTENTS connection
dummy (package) forking
heap
managers
pool
process
queues
```

reduction

sharedctypes synchronize util

CLASSES

class Process(__builtin__.object)

| Process objects represent activity that is run in a separate process

| The class is analagous to'threading.Thread'

| Methods defined here:

| __init__(self, group¼None, target¼None, name¼None, args¼(), kwargs¼{})

| __repr__(self)

| is_alive(self)

| Return whether process is alive

| join(self, timeout¼None)

| Wait until child process terminates

| run(self)

| Method to be run in sub-process; can be overridden in sub-class

| start(self)

| Start child process

| terminate(self)

| Terminate process; sends SIGTERM signal or uses TerminateProcess()

FUNCTIONS

Array(typecode_or_type, size_or_initializer, **kwds) Returns a synchronized shared array

BoundedSemaphore(value¼1)

Returns a bounded semaphore object

Condition(lock¼None)

Returns a condition object

Event()

Returns an event object

JoinableQueue(maxsize¼0) Returns a queue object

Lock()

Returns a non-recursive lock object

Manager()

Returns a manager associated with a running server process

The managers methods such as 'Lock()', 'Condition()' and 'Queue()' can be used to create shared objects.

Pipe(duplex¼True)

Returns two connection object connected by a pipe

Pool(processes¼None, initializer¼None, initargs¼(), maxtasksperchild¼None)

Returns a process pool object

Queue(maxsize¼0)

Returns a queue object

RLock()

Returns a recursive lock object

RawArray(typecode_or_type, size_or_initializer) Returns a shared array

RawValue(typecode_or_type, *args) Returns a shared object

Semaphore(value¼1)

Returns a semaphore object

Value(typecode_or_type, *args, **kwds) Returns a synchronized shared object

active_children()

Return list of process objects corresponding to live child processes

allow_connection_pickling()

Install support for sending connections and sockets between processes

cpu_count()

Returns the number of CPUs in the system

current_process()

Return process object representing the current process

freeze_support()

Check whether this is a fake forked process in a frozen executable. If so then run code specified by commandline and exit.

VERSION

0.70a1

AUTHOR

R. Oudkerk (r.m.oudkerk@gmail.com)

One of the first functions within the multiprocessing package you will be interested in is `cpu_count()`. Before you can distribute processing among multiple cores, you need to know how many cores you have access to.

```
import multiprocessing
multiprocessing.cpu_count()
4
```

As you can see on my Windows laptop, I have four CPU cores to work with; it would be a shame if my Python code only ran on one of those cores. Without applying the multiprocessing package and designing multi-core processing methods into my algorithm, my Python code would run on exactly one core.

Let us move on to our first example of applying multiprocessing within a Python program that has some forensic implication and relevance.

SIMPLEST MULTIPROCESSING EXAMPLE

In this first example I chose to develop the simplest multiprocessing example I could think of that would maximize the use of the four cores I have on my laptop. The program has a single function named `SearchFile()`, the function takes two parameters: (1) a filename and (2) the string that I wish to search for in that file. I am using a simple text file as my search target that contains a dictionary of words. The file is 170 MB in size which even on modern systems should create some I/O latency. I provide two examples, first a program that does not employ multiprocessing and simply makes four successive calls to the `SearchFile` function. The second example creates four processes and distributes the processing evenly between the four cores.

Single core file search solution

```
# Simple Files Search Single Core Processing
import time
```

```
def SearchFile(theFile, theString): try:
fp ¼ open(theFile,'r')
buffer ¼ fp.read()
```

```

fp.close()
if theString in buffer:

print'File:', theFile,'String:',\ theString,'\t','Found' else:
print'File:', theFile,'String:', \ theString,'\t','Not Found' except:
print'File processing error'
startTime ¼ time.time()

SearchFile('c:\\TESTDIR\\Dictionary.txt','thought')
SearchFile('c:\\TESTDIR\\Dictionary.txt','exile')
SearchFile('c:\\TESTDIR\\Dictionary.txt','xavier')
SearchFile('c:\\TESTDIR\\Dictionary.txt','$lllb!')

elapsedTime ¼ time.time() - startTime print'Duration:', elapsedTime
# Program Output

```

```

File: c:\TESTDIR\Dictionary.txt String: thought Found File:
c:\TESTDIR\Dictionary.txt String: exile Found File: c:\TESTDIR\Dictionary.txt
String: xavier Found File: c:\TESTDIR\Dictionary.txt String: $lllb! Not Found
Duration: 4.3140001297 Seconds

```

Multiprocessing file search solution

The multiprocessing solution is depicted below and as you can see the performance is substantially better even considering the file I/O aspects of opening, reading, and closing the associated file each time.

```

# Simple Files Search MultiProcessing
from multiprocessing import Process import time

```

```

def SearchFile(theFile, theString): try:
fp ¼ open(theFile,'r')
buffer¼ fp.read()
fp.close()
if theString in buffer:

print'File:', theFile,'String:', theString,

'\t','Found'
else:

```

```

print 'File: ', theFile, ' String: ', theString,
\t, ' Not Found'
except:
print'File processing error'

#
# Create Main Function #

if __name__ == '__main__':
startime = time.time()

p1 = Process(target=searchFile,
\args=('c:\\TESTDIR\\Dictionary.txt','thought') )
p1.start()

p2 = Process(target=searchFile, \
args=('c:\\TESTDIR\\Dictionary.txt','exile') )
p2.start()

p3 = Process(target=searchFile, \
args=('c:\\TESTDIR\\Dictionary.txt','xavier') )
p3.start()

p4 = Process(target=searchFile, \
args=('c:\\TESTDIR\\Dictionary.txt','$l1lb') )
p4.start()

```

Next we use the join to wait for all processes to complete

```
p1.join() p2.join() p3.join() p4.join()
```

```

elapsedTime = time.time() - startime
print'Duration:', elapsedTime
# Program Output

```

```

File: c:\TESTDIR\Dictionary.txt String: thought Found File:
c:\TESTDIR\Dictionary.txt String: exile Found File: c:\TESTDIR\Dictionary.txt
String: xavier Found File: c:\TESTDIR\Dictionary.txt String: $l1lb Not Found
Duration: 1.80399990082

```

MULTIPROCESSING FILE HASH

Certainly one of the most frequently used forensic tools is the one-way cryptographic hash. As you already know Python includes a hashing library as part of the Standard Library. As an experiment, I am going to perform a SHA512 Hash of four separate instances of the same file using a nonmultiprocessing method. I will set up a timer as well to calculate the elapsed time to perform the single-threaded approach.

Single core solution

Single Threaded File Hasher

```
import hashlib import os
import sys
import time
```

Create a constant for the local directory

```
HASHDIR = 'c:\\\\HASHTEST\\'
```

Create an empty list to hold the resulting hash results results = []

```
try: # Obtain the list of files in the HASHDIR listOfFiles = os.listdir(HASHDIR)
```

Mark the starting time of the main loop startTime = time.time()

for eachFile in listOfFiles:

Attempt File Open

```
fp = open(HASHDIR+eachFile,'rb') # Then Read the contents into a buffer
fileContents = fp.read()
```

Close the File fp.close()

Create a hasher object of type sha256 hasher = hashlib.sha256()

Hash the contents of the buffer hasher.update(fileContents)

Store the results in the results list

```
results.append([eachFile, hasher.hexdigest()])
```

delete the hasher object del hasher

Once all the files have been hashed calculate # the elapsed time

```
elapsedTime = time.time() - startTime
```

except:

If any exceptions occur notify the user and exit


```
print('File Processing Error') sys.exit(0)
# Print out the results
# Elapsed Time in Seconds and the Filename / Hash Results
print('Elapsed Time:', elapsedTime) for eachItem in results:
print eachItem

# Program Output
# Note: Each File Processed is identical # With a Size: 249 MB
```

```
Elapsed Time: 27.8510000705719 Seconds
['image0.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image1.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image2.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
['image3.raw',
'41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b']
```

Multi-core solution A

Much of the art associated with multiprocessing is decomposing a problem into slices that can run simultaneously across multiple cores. Once each of the individual slices have completed, the results from each slice are merged into a final result. The biggest mistake that is typically made is the lack of careful consideration of these two points. Purchasing systems with multi-cores and then just running any solution on that multi

core system does not necessarily buy you much over a single core solution.

Using our single solution as a baseline, I will create a multi-core solution that leverages the four cores available on my laptop. For this first example, I will instantiate the object Process, one for each core. When creating each Process object, I only need to provide two parameters: (1) the target¹ which in this case is the name of the function that will be called when the object.start() method is called. (2) the args¹ which is the argument tuple to be passed to the function. In this example, only one argument is passed, namely, the name of the file to hash. Once I have instantiated an object, I execute the object.start() method to kick off each process. Finally, I utilize the object.join() method for each object; this causes execution to be halted in the main() process until the processes have

completed. You can include a parameter with `object.join()` which is a timeout value for the process. For example `object.join(20)` would allow the process 20 s to complete.

Multiprocessing File Hasher A

```
import hashlib
```

```
import os
```

```
import sys
```

```
import time
```

```
import multiprocessing
```

```
# Create a constant for the local directory HASHDIR ¼'c:\\HASHTEST\\'
```

```
#
```

```
# hashFile Function, designed for multiprocessing #
```

```
# Input: Full Pathname of the file to hash #
```

```
#
```

```
def hashFile(fileName):
```

```
try:
```

```
    fp ¼ open(fileName,'rb')
```

```
# Then Read the contents into a buffer fileContents ¼ fp.read()
```

```
# Close the File fp.close()
```

```
# Create a hasher object of type sha256 hasher¼ hashlib.sha256()
```

```
# Hash the contents of the buffer hasher.update(fileContents)
```

```
print(fileName, hasher.hexdigest())
```

```
# delete the hasher object del hasher
```

```
except:
```

```
# If any exceptions occur notify the user and exit print('File Processing Error')
```

```
sys.exit(0)
```

```
return True
```

```
#
```

```
# Create Main Function #
```

```

if __name__ == '__main__':
# Obtain the list of files in the HASHDIR listOfFiles ¼ os.listdir(HASHDIR)
# Mark the starting time of the main loop startTime ¼ time.time()
# create 4 sub-processes to do the work # one of each core in this test

# Each Process contains:
# Target function hashFile() it this example # Filename: picked from the list
generated # by os.listdir()
# once again an instance of the # 249 MB file is used
#
# Next we start each of the processes

```

```

coreOne ¼ multiprocessing.Process(target¼hashFile,
args¼(HASHDIR+listOfFiles[0],) ) coreOne.start()

```

```

coreTwo ¼ multiprocessing.Process(target¼hashFile,
args¼(HASHDIR+listOfFiles[1],) )
coreTwo.start()

```

```

coreThree ¼ multiprocessing.Process(target¼hashFile,
args¼(HASHDIR+listOfFiles[2],) )
coreThree.start()

```

```

coreFour ¼ multiprocessing.Process(target¼hashFile,
args¼(HASHDIR+listOfFiles[3],) )
coreFour.start()

```

In this example, I was leveraging the knowledge of the hardware I was running the application on in order to maximize the distribution of processing. The Python multiprocessing library will automatically handle the distribution of the processes as cores become available. Also, since I could determine the number of cores available by using (multiprocessing.cpu_count()). I could have used that information to manually distribute the processing as well.

```

# Next we use join to wait for all processes to complete

```

```

coreOne.join() coreTwo.join() coreThree.join() coreFour.join()

```

```

# Once all the processes have completed and files have been # hashed and results

```

```

printed
# I calculate the elapsed time

elapsedTime = time.time() - startTime
print('Elapsed Time:', elapsedTime)

# Program Output
# Note: Each File Processed is identical # With a Size: 249 MB

c:\\HASHTEST\\image2.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image1.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image3.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image0.raw
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
Elapsed Time: 8.40999984741211 Seconds

```

As you can see by distributing the processing across the four cores produced the desired results by significantly improving the performance of the hashing operations.

Multi-core solution B

Here is another option for distributing processing among multiple cores that yields a small performance improvement over the multi-core solution A, and additionally provides a much cleaner implementation when you are calling the same functions with different parameters. This is a likely scenario if you consider that in your design for forensic applications. Additionally, this solution only requires the use of a single class: pool that can handle the entire multi-core processing operations in a single line of code. I have highlighted the simplified implementation below.

Multiprocessing File Hasher B

```

import hashlib
import os
import sys
import time

```

```

import multiprocessing

# Create a constant for the local directory HASHDIR ¼'c:\\HASHTEST\\'

#
# hashFile Function, designed for multiprocessing #
# Input: Full Pathname of the file to hash #
#

def hashFile(fileName):
try:
fp ¼ open(fileName,'rb')
# Then Read the contents into a buffer fileContents ¼ fp.read()
# Close the File fp.close()
# Create a hasher object of type sha256 hasher¼ hashlib.sha256()
# Hash the contents of the buffer hasher.update(fileContents) print(fileName,
hasher.hexdigest()) # delete the hasher object del hasher

except:
# If any exceptions occur notify the user and exit print('File Processing Error')
sys.exit(0)

return True

#
# Create Main Function #

if __name__ ¼¼'__main__':
# Obtain the list of files in the HASHDIR listOfFiles ¼ os.listdir(HASHDIR)
# Mark the starting time of the main loop startTime ¼ time.time()
# Create a process Pool with 4 processes mapping to # the 4 cores on my laptop
corePool ¼ multiprocessing.Pool(processes¼4)

# Map the corePool to the hashFile function
results ¼ corePool.map(hashFile, (HASHDIR+listOfFiles[0],\
HASHDIR+listOfFiles[1],\
HASHDIR+listOfFiles[2],\
HASHDIR+listOfFiles[3],))

```

Once all the files have been hashed and results printed # I calculate the elapsed time

elapsedTime = time.time() - startTime

print('Elapsed Time:', elapsedTime, 'Seconds')

Program Output

Note: Each File Processed is identical # With a Size: 249 MB

Elapsed Time: , 8.138000085830688, Seconds c:\\HASHTEST\\image0.raw,
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image2.raw,
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image1.raw,
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b
c:\\HASHTEST\\image3.raw,
41ad70ff71407eae7466eef403cb20100771ca7499cbf1504f8ed67e6d869e5b

Reviewing the results of the three implementations we find the following:
Processing

Implementation time MB (s) Notes

Single core solution 27.851 35.76

Multi-core solution A 8.409 118.44

Multi-core solution B 8.138 122.39 Typical implementation today processing each file in sequence Use of Process Class along with start and join methods Use of Pool Class simplifying implementation for common function processing yielding slightly better performance

MULTIPROCESSING HASH TABLE GENERATION

Rainbow Tables have been around for quite some time, providing a way to convert known hash values into possible password equivalents. In other words, they provide a way to lookup a hash value and associate that hash with a string that would generate that hash. Since hash algorithms are one-way, for all practical purposes two strings of characters will not generate the same hash value. They are highly collision-resistant. Rainbow Tables are utilized to crack passwords to operating systems, protected documents, and network user logins.

In recent years, the process of salting hashes has made Rainbow Tables less

effective as new tables would have to be generated as salt values are modified. These tables work because each password is hashed using the same process. In other words, two identical passwords would have the exact same hash value.

Today, this is prevented by randomizing hashed passwords, such that if two users use the same password, the hash values would be different. This is typically accomplished by appending a random string called a salt to a password prior to hashing.

Password ¼“letMeIn”

Salt¼‘&*6542Jk’

Combined Data to be Hashed: “&*6542JkletMeIn”

The salt does not need to be kept confidential as the purpose is to thwart the application of a rainbow or lookup table.

This means we need a fast way to generate more dynamic lookup tables. This is where multiprocessing can assist. Python actually has unique language mechanisms that support the generation of permutations and combinations built directly into the language that assists the developer. I will be using the itertools Standard Library (Python itertools module) to create the set of brute force passwords that I am looking for. The itertools module implements a number of iteration-like building blocks using a high-performance yet memory-optimized toolkit. Instead of having to develop permutation, combinations or product-based algebras, the library provides this for you.

First, I will create a simple Rainbow Table generator using a single core solution.

Single core password generator code

```
# Single Core Password Table Generator
```

```
# import standard libraries
```

```
import hashlib # Hashing the results
```

```
import time # Timing the operation import itertools # Creating controlled combinations
```

```
#
```

```
# Create a list of lower case, upper case, numbers # and special characters to include in the password table #
```

```
lowerCase ¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] upperCase ¼ ['G', 'H', 'I', 'J', 'K', 'L']
```

```
numbers ¼ ['0', '1', '2', '3']
```

```
special ¼ ['!', '@', '#', '$']
```

```
# combine to create a final list
```

```
allCharacters ¼ []
```

```
allCharacters ¼ lowerCase + upperCase + numbers + special
```

```
# Define Directory Path for the password file
```

```
DIR ¼ 'C:\\PW\\'
```

```
# Define a hypothetical SALT value SALT ¼ '&45Bvx9'
```

```
# Define the allowable range of password length PW_LOW ¼ 2
```

```
PW_HIGH ¼ 6
```

```
# Mark the start time startTime ¼ time.time() # Create an empty list to hold the  
final passwords pwList ¼ []
```

```
# create a loop to include all passwords # within the allowable range
```

```
for r in range(PW_LOW, PW_HIGH):
```

```
#Apply the standard library iterator
```

```
#The product iterator will generate the cartesian product # for allCharacters  
repeating for the range of
```

```
# PW_LOW to PW_HIGH
```

```
for s in itertools.product(allCharacters, repeat¼r):
```

```
# append each generated password to the
```

```
# final list
```

```
pwList.append("".join(s))
```

```
# For each password in the list generate # generate a file containing the # hash,  
password pairs
```

```
# one per line
```

```
try: # Open the output file fp ¼ open(DIR+'all','w')
```

```
# process each generated password
```

```
for pw in pwList:
```

```
# Perform hashing of the password md5Hash ¼ hashlib.md5()
```



```

md5Hash.update(SALT+pw)
md5Digest¼ md5Hash.hexdigest() # Write the hash, password pair to the file
fp.write(md5Digest +"" pw +'\n') del md5Hash

except:
print'File Processing Error'
fp.close()

# Now create a dictionary to hold the # Hash, password pairs for easy lookup
pwDict¼ {}

try: # Open each of the output file fp ¼ open(DIR+'all','r')
# Process each line in the file which # contains key, value pairs
for line in fp:

# extract the key value pairs # and update the dictionary
pairs¼ line.split()
pwDict.update({pairs[0] : pairs[1]})

fp.close()
except:
print'File Handling Error'
fp.close()

# When complete calculate the elapsed time

elapsedTime¼ time.time() - startTime print'Elapsed Time:', elapsedTime
print'Passwords Generated:', len(pwDict) print

# print out a few of the dictionary entries # as an example

cnt¼ 0
for key,value in (pwDict.items()): print key, value
cnt +¼ 1
if cnt > 10:
break;

print

# Demonstrate the use of the Dictionary to Lookup a password using a known

```

hash

Lookup a Hash Value

```
pw¼ pwDict.get('c6f1d6b1d33bcc787c2385c19c29c208') print'Hash Value  
Tested¼ c6f1d6b1d33bcc787c2385c19c29c208' print'Associated Password¼ '+ p  
w
```

Program Output

Elapsed Time: 89.117000103 Passwords Generated: 5399020

```
3e47ac3f51daffdbe46ab671c76b44fb K23IH  
a5a3614f49da18486c900bd04675d7bc $@fL1  
372da5744b1ab1f99376f8d726cd2b7c hGfdd  
aa0865a47331df5de01296bbaaf4996a 21ILG  
c6f1d6b1d33bcc787c2385c19c29c208 #I#$$  
c3c4246114ee80e9c454603645c9a416 #b$b$g  
6ca0e4d8f183c6c8b0a032b09861c95a L1H21  
fd86ec2191f415cdb6c305da5e59eb7a HJg@h  
335ef773e663807eb21d100e06b8c53e a$HbH  
d1bae7cd5ae09903886d413884e22628 ba21H  
a2a53248ed641bbd22af9bf36b422321 GHcda
```

```
Hash Value Tested¼ 2bca9b23eb8419728fdeca3345b344fc Associated  
Password¼ #I#$$
```

As you can see the code is quite succinct and straightforward, it leverages the `itertools` and `hashing` library to generate a brute force list. The result is a reference-able list of over 5.3 million salted passwords in under 90 seconds.

Multi-core password generator

Now that I have created a successful model for generating password combinations and created a dictionary with the resulting key/value pairs, I want to once again apply multiprocessing in order to generate a scalable solution. As you can see from the single core solution, it required just under 90 seconds to generate a little more than 5.3 million pairs. As we expand the number of characters to include in our generation and expand the size of allowable passwords beyond 5, the number of combinations grows exponentially. As I mentioned earlier, setting up an approach that lends itself to multiprocessing is the key. Now let us examine how well this actually scales by reviewing the

multi-core solution.

Multi-core password generator code

Multi-Core Password Table Generator

import standard libraries

import hashlib

import time

import itertools

import multiprocessing # Hashing the results

Timing the operation

Creating controlled combinations # Multiprocessing Library

Create a list of lower case, upper case, numbers # and special characters to include in the password table

lowerCase¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'] upperCase¼ ['G', 'H', 'I', 'J', 'K', 'L']

numbers¼ ['0', '1', '2', '3']

special¼ ['!', '@', '#', '\$']

combine to create a final list

allCharacters¼ []

allCharacters¼ lowerCase + upperCase + numbers + special

Define Directory Path for the password files DIR¼ 'C:\\PW\\'

Define a hypothetical SALT value SALT¼ '&45Bvx9'

Define the allowable range of password length

PW_LOW¼ 2 PW_HIGH¼ 6

def pwGenerator(size):

pwList¼ []

create a loop to include all passwords # with a length of 3-5 characters

for r in range(size, size+1):

#Apply the standard library iterator

for s in itertools.product(allCharacters, repeat¼r):

append each generated password to the

final list

pwList.append("".join(s))

```
# For each password in the list generate # an associated md5 hash and utilize the  
# hash as the key
```

```
try: # Open the output file  
fp = open(DIR+str(size),'w')
```

```
# process each generated password
```

```
for pw in pwList:  
# Perform hashing of the password md5Hash = hashlib.md5()  
md5Hash.update(SALT+pw)  
md5Digest = md5Hash.hexdigest()  
# Write the hash, password pair to the file fp.write(md5Digest + " " + pw + "\n") del  
md5Hash
```

```
except:  
print'File Processing Error'  
finally:  
fp.close()
```

```
#  
# Create Main Function #
```

```
if __name__ == '__main__':  
# Mark the starting time of the main loop startTime = time.time()  
#create a process Pool with 4 processes corePool =  
multiprocessing.Pool(processes=4)  
#map corePool to the Pool processes  
results = corePool.map(pwGenerator, (2, 3, 4, 5))  
# Create a dictionary for easy lookups pwDict = {}  
# For each file
```

```
for i in range(PW_LOW, PW_HIGH): try:  
# Open each of the output files  
fp = open(DIR+str(i),'r')  
# Process each line in the file which # contains key, value pairs  
for line in fp:
```

```
# extract the key value pairs
```

```

# and update the dictionary
pairs¼ line.split()
pwDict.update({pairs[0] : pairs[1]})

fp.close()
except:
print'File Handling Error'
fp.close()

# Once all the files have been hashed # I calculate the elapsed time

elapsedTime ¼ time.time() - startTime print'Elapsed Time:',
elapsedTime,'Seconds' # print out a few of the dictionary entries # as an example
print'Passwords Generated:', len(pwDict) print
cnt ¼ 0
for key,value in (pwDict.items()):

print key, value
cnt +=¼ 1
if cnt > 10:

break;
print
# Demonstrate the use of the Dictionary to Lookup # a password using a known
hash value
pw ¼ pwDict.get('c6f1d6b1d33bcc787c2385c19c29c208') print'Hash Value
Tested ¼ \ 2bca9b23eb8419728fdeca3345b344fc'
print'Associated Password'¼ '+' p w
# Program Output
Elapsed Time: 50.504999876 Seconds Passwords Generated: 5399020

3e47ac3f51daffdbe46ab671c76b44fb K23IH
a5a3614f49da18486c900bd04675d7bc $@fL1
372da5744b1ab1f99376f8d726cd2b7c hGfdd
aa0865a47331df5de01296bbaaf4996a 21ILG
c6f1d6b1d33bcc787c2385c19c29c208 #I#$$
c3c4246114ee80e9c454603645c9a416 #b$b$g
6ca0e4d8f183c6c8b0a032b09861c95a L1H21
fd86ec2191f415cdb6c305da5e59eb7a HJg@h

```

335ef773e663807eb21d100e06b8c53e a\$HbH
d1bae7cd5ae09903886d413884e22628 ba21H
a2a53248ed641bbd22af9bf36b422321 GHcda

Hash Value Tested_{1/4} 2bca9b23eb8419728fdeca3345b344fc Associated
Password_{1/4} #I#\$

As expected, the multi-core approach improves the performance and the solution is still straightforward and easily readable.
Reviewing the results of the single and multi-core implementations of the Rainbow Tables we find the following.

Implementation

Rainbow Processing generator time (s) Passwords

per second Notes

Single core 89.11

solution

Multi-core solution 50.50

60 K

106 K Typical implementation today processing using a single core Use of Pool
Class simplifying implementation

In addition to the performance results, the output of the generated key value pairs (MD5 hash and associated password are written to a file) and can be used later as part of a larger repository. Since the outputs are simple text files as shown in [Figure 10.1](#) they are easily readable.

SALTED Password MD5 HASH Associated Password

17ae80e34251ad4e2a61bc81d28b5a09 aa
9e6e21b8664f1590323ecaae3447ebae ab
e708b3b343cfbc6b0104c60597fff773 ac
dcea12b90e71b523db8929cf6d86e39d ad
70a3c3c78c79437dfc626ff12605ccb3 ae
41afc8d925b0b94996035b4ef25346ec af
14b45c3a22caf6c2d5ed2e3daae152ee ag
3c4493267b90a86303c1d30784b8f33b ah

```
5cbba376bbb10688ece346cbfa9b8c28 aG
ffad7a79a9b2c646ef440d1447e18ebe aH
...
...
eba8905561738d699d47bd6f62e3341c eeg2
13605b276258b214fdafccac0835fc67 eeg3
0e3a4e8aded5858c94997d9f593c4fa3 eeg!
a6ab3da0c0cc7dc22d2a4cfe629ba0f2 eeg@
8f005cf0ea4b2bf064da85e1b2405c00 eeg#
60c797433e1f1ff9175a93373cf1a6ff eeg$
e847612d8cab3184caa4d120c212afec eeha
75cc13015bd40e2a6da4609458bfbe01 ee hb
```

FIGURE 10.1

Plaintext Rainbow Table output abridged.

CHAPTER REVIEW

In this chapter, I began the process of addressing multi-core processing using Python by leveraging the cores already available on your laptop or desktop system. I introduce a couple of different approaches to multiprocessing and provide an overview of the Python Standard Library multiprocessing. I also examined two common digital investigation mainstay functions that can benefit from multiprocessing: (1) File Hashing and (2) Rainbow Table generation.

In [Chapter 11](#), I will demonstrate how you can take this approach into the cloud and leverage cloud services that could offer 10, 50, 100, or even a 1000 cores to expand the horizon of multiprocessing from the desktop to the cloud.

SUMMARY QUESTION/CHALLENGE

1. What other digital investigation or forensic applications do you believe could benefit from multiprocessing?
2. What are some of the key elements when designing multiprocessing solutions?
3. What is the best defense today against Rainbow Table-based password attacks?
4. The Rainbow Table example is limited by resources and will fail when the program runs out of memory. How would you modify the program such that it could continue to generate password/hash combinations in the face of memory

limitations?

Additional Resources

<http://www.net-security.org/article.php?id¼1932&p¼1>.

<http://docs.python.org/2/library/itertools.html#module-itertools>.

<http://docs.python.org/3.4/library/multiprocessing.html#module-multiprocessing>.

CHAPTER

Rainbow in the Cloud 11

CHAPTER CONTENTS

Introduction	289
Putting the Cloud to Work	289
Cloud Options	290
Creating Rainbows in the Cloud	292
Single Core Rainbow	295
Multi-Core Rainbow	296
Password Generation Calculations	300
Chapter Review	302
Summary Question/Challenge	303
Additional Resources	303

INTRODUCTION

One of the significant advantages of building applications in Python is the ability to deploy applications on virtually any platform, including of course the cloud (see [Figure 11.1](#)). This not only means that you can execute Python on cloud servers, but you can launch them from any device you have handy, desktop,