# Data Munging with Python, Part 3 – Writing to Files

Whil Hentzen

I know this will come as a shock to you, but once in a while, you'll run into a data-oriented situation where Visual FoxPro isn't going to cut it. In a previous article, I discussed the situation where an incoming data file exceeded VFP's 2 GB/.DBF capacity. We used SQLite to import thaa multi-gigabyte table.

But even that solution won't work in some situations. Another component of that project involved several multi-gigabyte text files, several of which contained several thousand columns. You read that right, several *thousand*. In case you're wondering, yes, E.F. Codd is rolling in his grave right now, and Chris Date is muttering, “Shoot me now.” Even SQLite chokes on the CREATE TABLE statement that includes 4,300 column defs, surprise, surprise.

Last issue, I showed you how to build scripts that handled text files in a variety of ways. Being able to manipulate them is in and of itself not very useful unless you're doing basic statistical gathering. This time around, we're going to write the results of our manipulations back to disk, creating both text files as well as .DBFs.

Last month we worked with a file that contained data for multiple industries, side by side, and learned to grab a chunk for a single industry out of a single line of that file. Once we were able to parse the data, the next step is to write the results out to disk. Specifically, we want to create separate files for each industry, just like they should have been created in the first place. We have two ways to go about this.

One way is to roll through the file and process each line for the first industry, writing out a subset of that line for that industry, line after line, until a file for that industry has been completely processed. Then go back to the top of the file and roll through the file a second time for the second industry. Then repeat again for the third industry, and so on. While we are only processing each input line once, we're spinning through the input file multiple times, once for each industry. You can think of this as serial processing, one file at a time.

Another way is to process each line in the file once, parsing out a part of the line for the first industry, writing to that industry's file, then parsing out another part of that line for the second industry and writing to *that* industry's file, until that line has been completely processed. Then and only then move to the second line, and repeat the process.

The tradeoff is that while we're only processing the input file once, we are writing to each industry file for each line. Is there a cost to switching handles? This is more akin to parallel processing, where we're working with multiple files at the same time.

We'll use our million row file to acquire some timing data. Finally, we'll also write those output files to DBFs, since that's the format our user is expecting. And we'll look at a bit of double-checking our results as well.

We're also going to ease you out of the JetBrains IDE and into a more “Pythonic” way of development, using IDLE, that is, with an editor in one window and running the results in another.

Let's do that first.

## Setting up our IDLE environment

So before we get to the coding stuff, we're going to address the development environment again. This is an important topic, because it's where you're going to live, and there are a multitude of choices. I picked a full-featured IDE like JetBrains to give you a comfortable starting point, but it's generally not how the Python community works.

Instead, Python developers typically use several programs working in concert, using the operating system as the “development environment” as opposed to a single program running inside the operating system. The first program is a text editor, of course. A second is the Python interpreter, used to run the program you've just entered. A third would be a debugger, perhaps, or a separate tool to display results.

A nice intermediate step to getting to that type of distributed development environment is to use the Python wrapper (they call it a 'GUI', but you and I will snicker) called IDLE. (I mentioned it briefly in the first article in this series.) IDLE comes along for the ride when you install Python. It functions very much like the bare Python interpreter, but with a few bells and whistles that are nice.

Start IDLE via the Start Menu | Python | IDLE as shown in Figure 1.

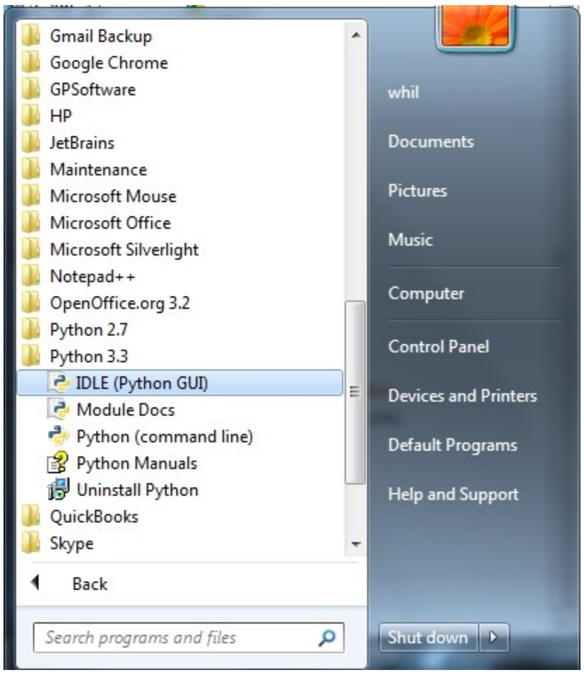
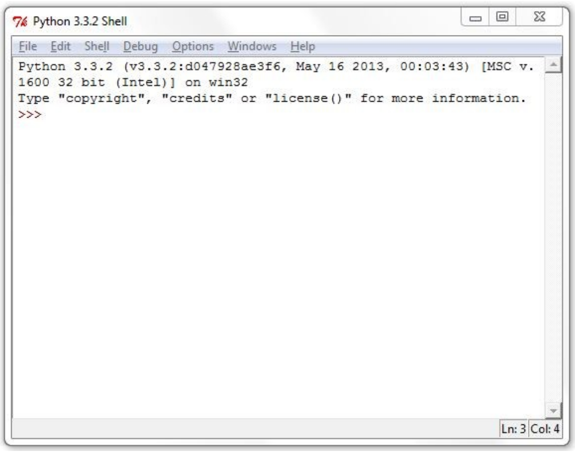


Figure 1. Calling IDLE.

Doing so will bring up IDLE, as shown in Figure 2.

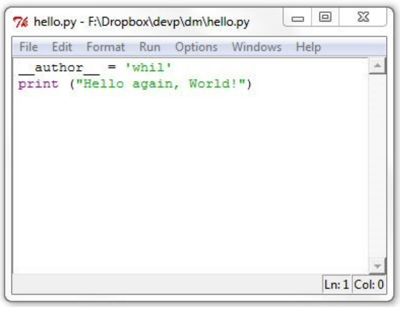


```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.
1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Figure 2.

You'll likely want to change IDLE's default folder; I have a subfolder in my 'dev' folder named 'devp' where all my Python projects live.

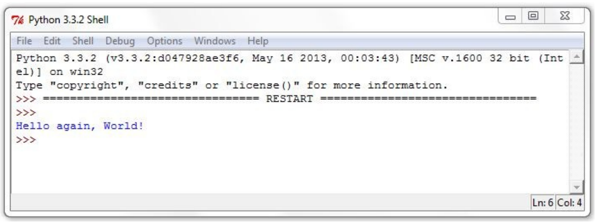
Doing so means that you can open a Python file with IDLE via the File menu. Doing so opens the file in the IDLE text editor, as shown in Figure 3.


hello.py - F:\Dropbox\devp\dm\hello.py
File Edit Format Run Options Windows Help
Ln: 1 Col: 0

```
__author__ = 'whil'
print ("Hello again, World!")
```

Figure 3. The IDLE editor.

One feature that's immediately useful is the fact that you can run your script simply by hitting F5. The results will display in the Python shell window, as shown in Figure 4.



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Int
el)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello again, World!
>>>
```

Figure 4. Running a script in IDLE produces the results in the Python shell.

Note that you don't have to have the Python shell window open in IDLE first – if it's not there, it'll be opened automatically.

The Python shell window will also display some debugging feedback.

## Why bother?

You might be wondering why it's important to be able to simply your development environment. Let me tell you a quick story.

As I was presenting this topic to a user group a while back, I ran into an interesting situation. My laptop wouldn't connect to the projector, and there I was, in front of room full of people who had come specifically to see this presentation. Yeah, not much pressure.

"How hard could it be to move your code to someone else's machine?" you're asking.

Well, in the VFP/Windows world where everyone has reasonably homogenous environments, it would potentially be easy. Sure, anyone who has had to struggle with all of the myriad Windows issues such as incompatible drivers, missing ActiveX controls, DLLs that are out of date, well, perhaps not as easy in real life. But still, we're all running the same IDE and can rely on the tools all being there.

But Python isn't like VFP on Windows. First, it runs on everything, so in that room there were Windows boxes, Mac boxes, and Linux boxes. I'm sure somewhere in the back, someone had a portable SparcStation or a Commodore 64, just because they could. And even on the same type of box, people were running a variety of environments.

Fortunately, having transitioned from PyCharm projects to simple .py scripts, I was able to copy my scripts to a thumb drive, stuff it into the host's Mac, and with a little bit of keyboard fumbling (did they have to rename *every* key?), run each of my scripts right off of the thumb drive. If I had been wedded to PyCharm, I would have been sunk.

## This Month's Challenge

To reiterate, we have a text file with zillions of lines of similar data. Each line consists of two columns that act as a composite primary key for the entire row, and then several sets of fields whose format and structure are identical. Each set represents a different industry. The sample data file included with this month's source code has two lines, both of which have 11 sets of fields. The file in toto has approximately 26,000 characters, meaning each row is over 13,000 characters long.

We want to create 11 files, each of which has the two columns, and then one industry's set of fields.

The input file is named 'test002rows.dat'. We'll parse this file two different ways. First we'll spin through the file once for each industry, producing a single industry file each time, in essence, serial processing. Those output files will be named 'serial_indNNN.txt' where the NNN is an identifier for an industry.

Then we'll spin through the file just one time, processing a row and writing to each of the 11 industry files while processing that row, in essence, parallel processing. These files will be named 'parallel_indNNN.txt'.

## Serial Processing (one industry file at a time)

We have all the tools we need to process the input file. Let's look at the pseudo-code for this routine.

```
1 initialize starting variables
2 initialize a list that contains the
    industry identifiers
3 for each item in that list,
    perform the following
4   - create a filename for the industry
5   - open the input file
6   - open the output file, with a write flag
7   - for each line in the input file,
      perform the following
8     - extract the first two columns
9     - extract the columns for the current
        item's industry
10    - write the composite key columns
        and the industry columns
        to the output file
11  - close the output file
12  - close the input file
```

### 1 Initialize starting variables

We'll need some basic data to get started. First, stuff the input filename to a variable. In the production version, naturally, this program spins through a series of input files. For expediency's sake, I've hard-coded the name and stored it to a variable.

Since one of the goals of this article is to determine which process, serial or parallel, is faster, we'll set a timer.

And, finally, it's inevitable that we'll run into problems as we write this routine. Including print() statements throughout the code will help us trace the progress and debug what is going wrong. However, it's a waste of time to write a quick print() statement, use it while debugging a certain problem, and then delete it. It's quite likely that we'll want to be able to use print() statement again later. So I'll initialize a GoPrint variable that can will enable us to turn our print() statements on and off in a single location.

```
filenameext = 'test002rows.dat'
startsec = time.time()
GoPrint = 1
```

### 2 Initialize the industry identifier list

In the production version of this application, there were dozens of input files, each of which contained data for a set of industries. One set was professional services, another was heavy manufacturing, a third was farming/fishing, and so on. Thus, there was a master file that contained the industry identifiers. Continuing our quest for expediency, I've hard-coded the identifiers in a list instead of reading them in from that master file.

Remember, when you see 'Python list', think 'one-dimensional array'.

```
Indlist = ['401','402','403','404','405',
  '406','406a','407','408','408a','408b']
```

### 3 For each item in that list

Next, we'll spin through each item in the list. In this example, we know how many items are in the list, but in the production version, the number of items will vary, so let's grab that value programmatically instead of hard-coding it.

```
howmanychunks = len(indlist)
print('Serial process, go through input file
  multiple times')
for index in range(howmanychunks):
```

I've included a print statement that will echo to the Python shell, as a starting breadcrumb so that we can trace the execution of our program.

In the following sections, I've included either one or two hyphens in the section name, to provide a visual clue as to the level of the hierarchy we're in: one hyphen for items in the industry list, two hyphens for items in the input file.

### 4 - Create a filename for the industry

First, grab the item out of the industry list. Next, build a filename from that item. Finally, in order to aid in debugging, print the details of the newly created filename.

```
indname = indlist[index]
filenameextout = 'yserial_ind_%s.txt' %
(indname)
if GoPrint:
  print('industry #:', index+1, 'indname:',
  indname, 'filenameextout:', filenameextout)
```

### 5 - Open the input file

We're opening the input file each time we process an industry.

```
txti=open(filenameext)
```

### 6 - Open the output file, writeable

Similarly, open the output file each time we process an industry.

```
txto=open(filenameextout, 'w')
```

### 7 - - For each line in the input file

Then, work through each line in the input file.

```
linenum = 1
for oneline in txti:
```

For each line ('oneline') in the input file...

### 8 - - Extract the first two columns

First, determine the length of the current line. This has to be done for each line, because each line could be a different length, depending on the data in it. There will be the same number of columns, but the size of a data element in any given column may vary.

Calculate the positions of the pipes after the first and second columns, and then extract the data between them.

```
linelen = len(oneline)
firstpipepos = oneline.index("|")
secondpipepos
  = oneline.index("|",firstpipepos+1,linelen)
geoid = oneline[:firstpipepos]
occ = oneline[firstpipepos+1:secondpipepos]
```

### 9 - - Extract the current item's columns

The index is the number of industry. Use it as the multiplier to calculate the first and last pipes demarcating the range of columns for the industry. Then pull the whole slew of columns from the current line into a string.

The findnth function, discussed in last month's article and defined at the beginning of the script, is used to determine the positions of the start and end pipes.

```
startpipenum = 2+192*(index)
endpipenum = 2+192*(index+1)
startpipepos
  = findnth(oneline,'|', startpipenum)
endpipepos = findnth(oneline, '|', endpipenum)
thischunk = oneline[startpipepos+1:endpipepos]
```

### 10 - - Write the columns

Assemble the first two columns and all of the columns for the current industry into a string and write that to the output file.

```
strtodump =
  geoid + '|' + occ + '|' + thischunk +
  chr(13)
txto.write(strtodump)
```

### 11 - Close the output file

Once done processing all of the lines in the input file, close the output and input files.

```
txto.close()
```

### 12 - Close the input file

```
txti.close()
```

### Notes

You'll notice the variable 'linenum' initialized before processing each line in txti, and then incremented for each line. This variable isn't needed, strictly, for processing, but is used as a flag to aid in debugging.

## Parallel Processing (all industry files simultaneously)

While the serial processing works just fine, there's likely a little voice in the back of your head saying, "That seems wasteful, opening and closing the input file for each industry, and spinning through it over and over. Particularly if the file is really long. Would it be faster to just open the input file once and process each line once?"

That's a good question, and no fair skipping to the end to find out the answer.

There's another reason I bring this idea up, and that's to learn more about processing data files. The parallel processing routine is going to require slightly different tricks.

Let's look at the pseudo-code for this routine.

```
1  initialize starting variables
2  initialize a list that contains the
   industry identifiers
3  open the input file
4. for each line in the input file
5  - extract the first two columns
6  - for each item in the industry list
```

```
7    - determine start and end pipes for
       the industry columns
8    - extract the columns for the current
       item's industry
9    - create a filename for the industry
10   - open the output file, with a write flag
11   - write the composite key columns
       and the industry columns
       to the output file
12 close output file
13 close input file
```

## 1 Initialize starting variables

Just like with the serial routine, we'll initialize some variables. In this routine, we'll set two 'GoPrint' flags, one for each time a line is processed and the other for each industry within a line.

```
filenameext = 'test002rows.dat'
startsec = time.time()
GoPrintLine = 0
GoPrintIndustry = 0
```

## 2 Initialize industry identifiers list

In this routine, we're going to grab the number of industries right away.

```
Indlist = ['401','402','403','404','405',
  '406','406a','407','408','408a','408b']
howmanychunks = len(indlist)
```

## 3 Open the input file

Since we're processing the input file just once, we'll open it right away.

```
txti=open(filenameext)
```

## 4 For each line in the input file

Once open, we'll start to processing each line in it.

```
for oneline in txti:
```

Once we start looping, similar to the serial routine, in the sections that follow, I've included either one or two hyphens in the section name, to provide a visual clue as to the level of the hierarchy we're in. But this time, one hyphen is for items in the input file and two hyphens for items in the industry list.

## 5 - Extract the first two columns

In this routine, we just have to grab the first two columns that act as a primary key once.

```
linelen = len(oneline)
# extract first two cols
firstpipepos = oneline.index("|")
secondpipepos =
  oneline.index("|",firstpipepos+1,linelen)
geoid = oneline[:firstpipepos]
occ = oneline[firstpipepos+1:secondpipepos]
```

We'll store the columns to 'geoid' and 'occ' variables and use them over and over again.

## 6 - For each item in the industry list

Unlike the serial routine, 'index' will roll through one item in the industry list. It acts as a counter. 'index' will range from 1 to however many industries there are in this particular input file.

```
for index in range(howmanychunks):
```

## 7 - - Determine start and end pipes

There are 192 columns in an industry chunk, so the startpipenum and endpipenum can be calculated by using the counter that represents the industry.

The findnth function, again, is used to determine the positions of the start and end pipes.

```
startpipenum = 2+192*(index)
endpipenum = 2+192*(index+1)
startpipepos =
  findnth(oneline,'|', startpipenum)
endpipepos = findnth(oneline, '|', endpipenum)
```

## 8 - - Extract current item's columns

Once we have the starting and ending positions, the string containing all 192 columns for the current industry can be extracted.

Then that string is added to the 'geoid' and 'occ' primary key fields to create the string to write to an output file.

```
thischunk =
  oneline[startpipepos+1:endpipepos]
strtodump =
  geoid + '|' + occ + '|' + thischunk +
  chr(13)
```

## 9 - - Create a filename for the industry

Since we're processing each output file, we need to determine what the name of the output file for the industry in question is.

```
indname = indlist[index]
filenameextout =
  'bparallel_ind_%s.txt' % (indname)
```

## 10 - - Open the output file

Once we have the output filename, open it for writing. Use the 'a' flag to append instead of overwriting.

```
txto=open(filenameextout, 'a')
```

## 11 - - Write the columns

Then write the string for the industry to the output file.

```
txto.write(strtodump)
```

## 12 Close output file

The output file is closed each time the next one is opened, but the last one stays open. Once done with every line in the input file, close the last output file.

```
txto.close()
```

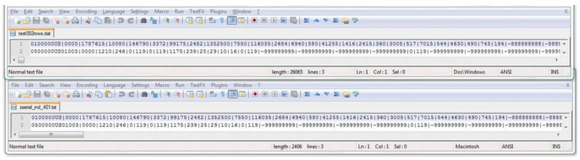
### 13 Close input file

```
txti.close()
```

## Verifying Results

Tis an amateur programmer who assumes that, once his program runs without throwing errors, he's done. It may be writing to the various industry files without choking, but how do we know it's writing the correct data? I'm sure each of you has had the experience where you look at an output file and find some sort of silly results, like the same row being replicated 107,000 times. Let's make sure that we didn't do something foolish like truncate the last character of each chunk due to a one-off error.

The first thing we can do is a visual inspection. Difficult with an input file that has over 13,000 characters in each row, but not impossible. Open up test002rows.dat and serial_ind401.txt in matching text editor windows, and scan through, as shown in Figure 5. The key numbers to look for are the first industry column and the last.



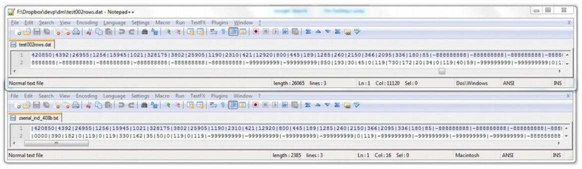
```
test002rows.dat
0100000US|0000|1787615|10080|146790|3372|99175|2482|1352500|7550|116035|2684|6940|580|41255|1416|2415|360|3005|517|7015|544|4630|490|745|194|-888888888|-888
0500000US01003|0000|1210|246|0|119|0|119|1175|239|25|29|10|16|0|119|-999999999|-999999999|-999999999|-999999999|0|119|-999999999|-999999999|-999999999|-9999
Normal text file    length : 26065   lines : 3    Ln : 1   Col : 1   Sel : 0    Dos\Windows   ANSI   INS

zserial_ind_401.txt
0100000US|0000|1787615|10080|146790|3372|99175|2482|1352500|7550|116035|2684|6940|580|41255|1416|2415|360|3005|517|7015|544|4630|490|745|194|-888888888|-8888
0500000US01003|0000|1210|246|0|119|0|119|1175|239|25|29|10|16|0|119|-999999999|-999999999|-999999999|-999999999|0|119|-999999999|-999999999|-999999999|-99999
Normal text file    length : 2406   lines : 3    Ln : 1   Col : 1   Sel : 0    Macintosh   ANSI   INS
```

File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?

Figure 5. Comparing the original input file and one of the industry output files.

Next, open up the **last** industry file, serial_ind408b.txt, and compare those values, as shown in Figure 6.



```
F:\Dropbox\devp\dm\test002rows.dat - Notepad++
test002rows.dat
420850|4392|26955|1256|15945|1021|328175|3802|25905|1190|2310|421|12920|800|445|189|1285|260|2150|366|2095|336|180|85|-888888888|-888888888|-888888888|-8888
8888888|-888888888|-888888888|-888888888|-888888888|-888888888|-999999999|-999999999|850|193|30|45|0|119|730|172|20|34|0|119|40|59|-999999999|-999999999|0|1:
Normal text file    length : 26065   lines : 3    Ln : 1   Col : 11120   Sel : 0    Dos\Windows   ANSI   INS

zserial_ind_408b.txt
|420850|4392|26955|1256|15945|1021|328175|3802|25905|1190|2310|421|12920|800|445|189|1285|260|2150|366|2095|336|180|85|-888888888|-888888888|-888888888|-888
|0000|390|182|0|119|0|119|330|162|35|50|0|119|0|119|-999999999|-999999999|-999999999|-999999999|0|119|-999999999|-999999999|-999999999|-999999999|-888888888
Normal text file    length : 2385   lines : 3    Ln : 1   Col : 16   Sel : 0    Macintosh   ANSI   INS
```

File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?

Figure 6. Comparing the end of the original input file and the last industry file.

Now we've verified that the first and last industries are good, at least in the first row. It's possible that formatting changes in the middle of the file might have caused problems, so next,

The second test we can perform is matching up the file produced for a single industry with both routines. The files should be identical, so compare the file sizes. They may be off by one character due to an EOF marker in one file but not the other, so if they're close, or if they're not, open them both in text editor windows and see what the difference is.

So now we've verified that the files are the same. But visual verification only goes so far, particularly because these files are so wide and so long.

If the input file has a total row, it would be possible to compare the totals of the input file and the output files, but in this case, there are no total rows, so we're out of luck.

One final test would be to do a scan through the contents of each column, ensuring that there is data in at least some of the rows. A million row file with zeros in every row might be a clue that something went wrong.

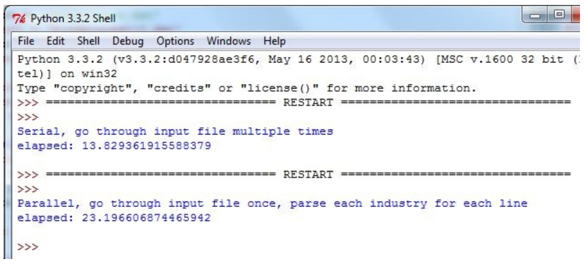
There's really no more verification that can be done on this level. In the next part of this series, we'll move the data from text files into a different repository that can be examined more robustly.

## Timing Results

The reason we did the file extraction in two different ways was to dig up empirical evidence about which way would be faster. Interestingly enough, my original hypothesis was wrong.

I ran the serial and parallel routines against a 600 row file with 11 industry sets, long enough gain some timing but not long enough to need to go get popcorn.

Processing the input file over and over again was actually faster as shown in Figure 7.



```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Serial, go through input file multiple times
elapsed: 13.829361915588379

>>> ================================ RESTART ================================
>>>
Parallel, go through input file once, parse each industry for each line
elapsed: 23.196606874465942

>>>
```

Figure 7. Comparing the processing times of the serial and parallel routines.

Next time, we'll move write these files to DBFs.

## Author Profile

*Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at [whil@whilhentzen.com](mailto:whil@whilhentzen.com)*