Data Munging with Python, Part 2 – Handling 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 that multi-gigabyte table.

But even that solution won't work in some situations. Another component of that project involved several multi-gigabyte text files, some of which contained thousands of 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 introduced Python and showed you how to create and run the traditional Hello World script in several environments. This month, we're going to start building scripts that handle files.

There are two ways to learn a new language. One approach is to learn a ton of syntax withno context, in other words, without an end goal in sight. That's an educational style, learning for learning sake, and if you've got the time and patience, fine!

The other is to learn a narrow set of capabilities in pursuit of a specific goal, and then adding tools to your toolkit as needed. The danger here, of course, is the possibility of learning bad practices becoming narrow minded, ending up doing things the long way around.

Since Python can be used to do most anything, I think it makes sense to have a specific end game in mind. Working with these files is the end game for this article. Specifically, we're going to break up that multi-thousand column file into smaller chunks.

To get started, let's learn some useful mechanisms.

Some Basic Python Philosophy

A lot of VFP developers like Python because of its similarity to Fox. That said, it's not identical. Let's look at the Python frame of mind for a moment.

There's SO MUCH to Learn!

You know how overwhelmed someone new to VFP feels? Python is the same, even more so. Python doesn't have the panoply of design surfaces and embedded data access that VFP does, but Python's core language has much more depth than VFP's broad but shallow suite of commands and functions. Thus, it will take several passes at the language to develop comfort with the richness of the constructs and nuances of the syntax.

Being Pythonic

Unless you've only worked on your own code during your career, you've undoubtedly seen a variety of coding styles. You've also seen that there are some generally accepted ways of doing things in VFP, at least, in the visible community. Adhering to standards makes it easier for someone else to read, understand, help and maintain your code.

The Python community has adopted a set of generally accepted principles as well, and taken it further, to become almost a personality, and even a written set of guidelines. Thus, you'll often see people offer a solution to a problem, and then follow up with an improvement, saying, "Actually, this way is more Pythonic."

Unless you want to code in a cave the rest of your life, it's a good idea to try to become Pythonic as well. Code you see will make more sense, and it'll be easier for others to help you.

See PEP 8 for the definitive style guide.

```
http://www.python.org/dev/peps/pep-0008/
```

Spelling Things Out

While many programmers may delight in writing constructs that attempt to jam as much functionality into a single line as possible, their cleverness comes at the expense of readability, and thus, maintainability. I prefer to create intermediate variables that will be used throughout the routine and can be inspected as an aid to debugging and making later modifications easier and more reliable.

Thus, instead of this:

```
    geoid = line[:line.index("|")]
```

(the Python version of

```
substring = Substring(line, AT("|", line))
```

in VFP, I would do this:

```
    firstpipepos = line.index("|")
    geoid = line[:firstpipepos]
```

While this example may seem trivial, it's actually useful, because the 'firstpipepos' variable will be used throughout the subsequent code, and thus being able to debug it separately will be useful.

Sample Code Notes

This article has several exercises, each built in a separate project. For each project, I'll be using PyCharm as discussed in my last article. The PyCharm projects all use a script called **main.py**, but since you might be working in the Command Window or IDLE, those main.py files have been renamed in order to be included with this article. Each main.py file will contain all of the code segments for the first project in this article. Each code segment will be numbered separately and marked off with “IF” control structures. To run one sample code segment, just flip the flag for that IF statement to True.

Some Basic Python Constructs

Now we need to learn a few basic mechanisms that we'll use throughout our projects in this article.

Logic Structures - IF

It's pretty difficult to write any kind of useful program without doing a logical test, and, like VFP, you use the IF statement to do so. The most basic form, similar to “if .t.” in VFP, is this:

```
if True:
    <some code>
<some code after the IF segment
```

Several notes of interest. The Python equivalent to '.t.' is True (and, similarly, False.) These are case sensitive, you can't use 'true' or 'TRUE'.

Second, the IF statement must be terminated with a colon.

Third, if you need an 'else' condition, use 'elif'. Nifty about this statement, you can use multiple iterations. Additionally, close the entire segment with 'else', like so:

```
if x = 0:
    <some code>
elif x = 1
    <some code>
elif x= 2
    <some code>
else
    <some code>
```

Finally, you don't need an 'endif' statement. Remember from last time, indentation controls code blocks, returning to a non-indented statement finishes the code block.

Logic Structures - Loops

If you want to iterate through a a list, use “for”. Different from other languages, Python's “for” allows you to iterate through any type of list, not just a sequence of numbers. In VFP, you'll do this:

```
dimension animals[5]
animals[1] = 'aardvark'
animals[2] = 'beaver'
animals[3] = 'cat'
animals[4] = 'dog'
animals[5] = 'egret'
for I = 1 to 5
    ? len(animals[i]), animals[i]
next
8 aardvark
6 beaver
3 cat
3 dog
5 egret
```

In Python, you can do this:

```
animals = [aardvark, beaver, cat, dog, egret]
for critter in animals:
    print(len(critter), critter)
8 aardvark
6 beaver
3 cat
3 dog
5 egret
```

(Just like “if”, you'll need a colon at the end of “for” statement.) It's a little unnerving to see 'critter' seemingly to be used as a counter just like 'I' is used in VFP “for” constructs, so instead, think of a VFP 'for each' construct instead.

Critter is the value of each element in animals, one at a time, as driven by the for iterator. In fact, if you're iterating (looping) through a collect of objects, critter can be a full-fledged object, and not just a scalar value.

UDFs

Remember the thrill you experienced when UDFs were added to the xBASE language? A whole new world opened for us. Now we take them for granted. So how do you create your own UDF in Python?

The 'def' construct defines a function, including the parameters passed, like so:

```
def MyFunc(firstname,lastname)
   fullname = lastname + ', ' + firstname
   return fullname
```

In a script, you'll want to include the def before the code that uses it.

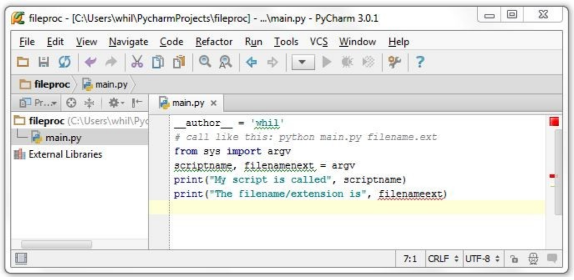
Passing Parameters

Let's talk about how to create a Python script that takes an external parameter.

Fire up PyCharm and create a project named **fileproc**. Create a script file named **main.py**, and enter the following code in it (the '#' char is one way to identify a comment):

```
# call like this: python main.py filename.ext
from sys import argv
scriptname, filenamenext = argv
print("My script is called", scriptname)
print("The filename/ext is", filenameext)
```

Figure 1 displays what your editor window should look like.


fileproc - [C:\Users\whil\PycharmProjects\fileproc] - ...\main.py - PyCharm 3.0.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
fileproc
main.py
External Libraries
7:1 CRLF UTF-8

```
__author__ = 'whil'
# call like this: python main.py filename.ext
from sys import argv
scriptname, filenamenext = argv
print("My script is called", scriptname)
print("The filename/extension is", filenameext)
```

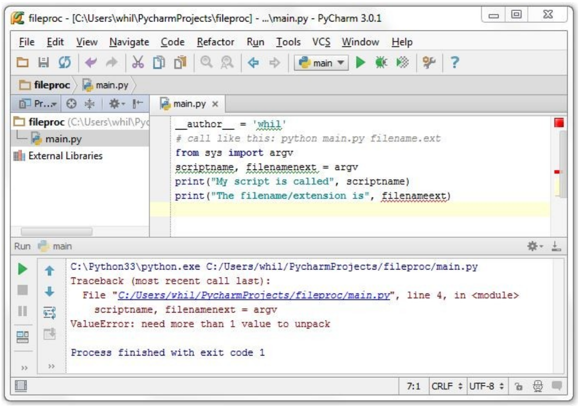
Figure 1. A script that accepts two parms.

Remember how I mentioned in the last article that you have to import modules to add functionality? Here's an example. We're importing the **argv** module from the sys library for use to pass parameters (arguments) to the script. Then we define the arguments, and use them.

Now just run the script (via the Run menu or the toolbar button.)

Oh, wait.

Clicking the Run button throws an error and opens the Debug pane at the bottom of the window to show you what is is. See **Figure 2**. I bet you knew that was coming, didn't you? After all, there wasn't any place to define what the argument being passed to the script was.


fileproc - [C:\Users\whil\PycharmProjects\fileproc] - ...\main.py - PyCharm 3.0.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
fileproc
main.py
External Libraries
Run main
7:1 CRLF UTF-8

```
__author__ = 'whil'
# call like this: python main.py filename.ext
from sys import argv
scriptname, filenamenext = argv
print("My script is called", scriptname)
print("The filename/extension is", filenameext)
```

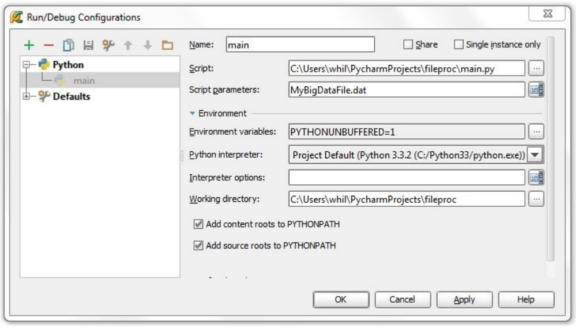
```
C:\Python33\python.exe C:/Users/whil/PycharmProjects/fileproc/main.py
Traceback (most recent call last):
  File "C:/Users/whil/PycharmProjects/fileproc/main.py", line 4, in <module>
    scriptname, filenamenext = argv
ValueError: need more than 1 value to unpack

Process finished with exit code 1
```

Figure 2. Running without parms throws an error.

A moment of panic, when you think, "Do I have to open up a Command Prompt to run Python scripts if I want to pass a parameter? How kludgy!" But then you realize that an IDE this sophisticated must have planned for that contingency. And indeed they did.

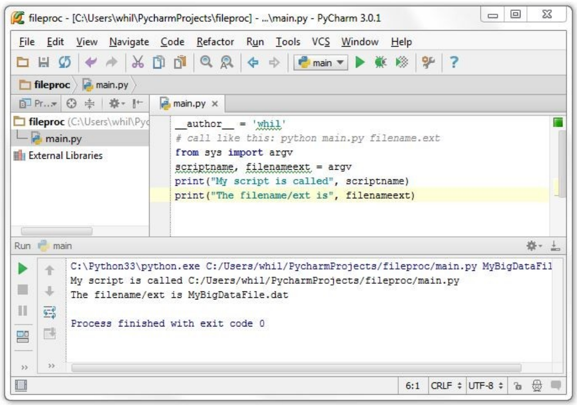
Select the Run | Edit Configuration menu option to display the Edit Config dialog, and enter a parameter to be passed to the script in the 'Script parameters' text box, specifically, the name of the data file ("MyBigDataFile.dat"), as shown in **Figure 3**.



Name: main ☐ Share ☐ Single instance only
Script: C:\Users\whil\PycharmProjects\fileproc\main.py
Script parameters: MyBigDataFile.dat
Environment variables: PYTHONUNBUFFERED=1
Python interpreter: Project Default (Python 3.3.2 (C:/Python33/python.exe))
Interpreter options:
Working directory: C:\Users\whil\PycharmProjects\fileproc
☑ Add content roots to PYTHONPATH
☑ Add source roots to PYTHONPATH

Figure 3. The Edit Configuration dialog.

Click Apply and OK, and then click the Run button on the toolbar again. This time, as shown in **Figure 4**, success!


fileproc - [C:\Users\whil\PycharmProjects\fileproc] - ...\main.py - PyCharm 3.0.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
fileproc
main.py
External Libraries
Run main
6:1 CRLF UTF-8

```
__author__ = 'whil'
# call like this: python main.py filename.ext
from sys import argv
scriptname, filenameext = argv
print("My script is called", scriptname)
print("The filename/ext is", filenameext)
```

```
C:\Python33\python.exe C:/Users/whil/PycharmProjects/fileproc/main.py MyBigDataFil
My script is called C:/Users/whil/PycharmProjects/fileproc/main.py
The filename/ext is MyBigDataFile.dat

Process finished with exit code 0
```

Figure 4. Successfully running the script with the proper parm.

Note that **scriptname** is a required parm in the

```
scriptname, filenameext = argv
```

statement, even if it isn't used.

So that's the first piece of the puzzle – we are now able to create a Python script that takes a parameter, such as a filename, and can do something with it.

As an aside, you may be wondering how to pass a parameter to a Python script in a DOS window.

```
F:> python main.py test.txt
```

If you're already in the Python interpreter, you'd think you could simply do something like this:

```
>>> main.py test.txt
```

but the answer is considerably more complex, so for the time being, we'll pass it by.

Project 1: Working with Files

(The code for this project is in main.py in the FileProc project.)

Our first project is to write a script that:

- reads in a file,
- manipulates the data in the file, and
- outputs various pieces.

In order to do so, we need to learn to write a script that takes parameters, returns values, and likely (we're just guessing here, but I'll bet it's a pretty good guess), contains one or more functions. We'll learn a couple more things along the way.

Opening Files

Since we've gone to so much trouble to pass the name of a file to the program, perhaps we should step up our game and do something with that file. Like... open it!

First, we'll need a data file to manipulate. Included in the source code for this article is a small text file named 'thisguy.txt'. It contains about a half dozen lines of text of varying lengths, typed into a text file using the Notepad ++ editor, and relying on the return key pressed to create line breaks with chr(13) and chr(10) line break characters.

We'll use the open function on our filename, and create an object from the result, like so (example 1 in main.py):

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

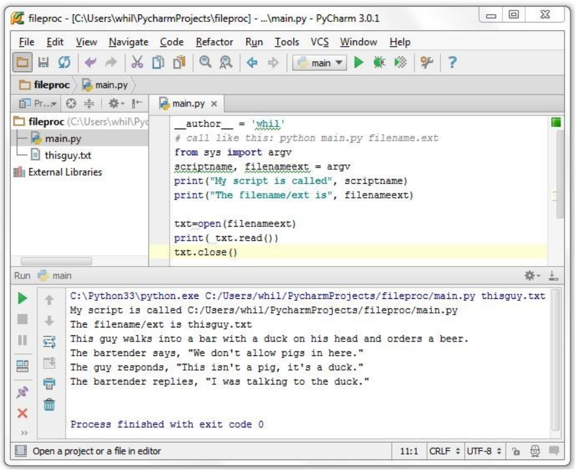
The **txti** variable is an object reference and has methods attached to it, such as read(). Thus, we can do this:

```
print( txti.read())
```

to print the entire file. When we're done with the file, be sure to close it, like so:

```
txti.close()
```

Put them all together at the end of our 'main.py' script, run it, and the results will display in the Debugger pane, as shown in **Figure 5**.


fileproc - [C:\Users\whil\PycharmProjects\fileproc] - ...\main.py - PyCharm 3.0.1
File Edit View Navigate Code Refactor Run Tools VCS Window Help
main
fileproc
main.py
fileproc (C:\Users\whil\Pyc
thisguy.txt
External Libraries
Run main
Open a project or a file in editor
11:1 CRLF UTF-8

```
__author__ = 'whil'
# call like this: python main.py filename.ext
from sys import argv
scriptname, filenameext = argv
print("My script is called", scriptname)
print("The filename/ext is", filenameext)

txt=open(filenameext)
print( txt.read())
txt.close()
```

```
C:\Python33\python.exe C:/Users/whil/PycharmProjects/fileproc/main.py thisguy.txt
My script is called C:/Users/whil/PycharmProjects/fileproc/main.py
The filename/ext is thisguy.txt
This guy walks into a bar with a duck on his head and orders a beer.
The bartender says, "We don't allow pigs in here."
The guy responds, "This isn't a pig, it's a duck."
The bartender replies, "I was talking to the duck."


Process finished with exit code 0
```

Figure 5. Echoing back the source file.

So far, so good. Let's do something more interesting with our file than just spit it back out. How about if we parse a line?

Add the lines (as shown in example 2)

```
myline=txti.readline()
print("my line is:", myline, "!")
```

after the **print(txti.read())** statement, execute your script, and you'll find that the new print statement doesn't print anything in between "my line is" and "!" Some of you probably already came up with the answer – the read() function moves the 'pointer' in the file, and so when done printing the whole file, the command to print the results of readline() simply print what's at the end of the file – nothing!

Remove the **print(txti.read())** statement, so that the pointer is still located at the beginning of the file. Rerun, and you'll see the first line in the file printed out.

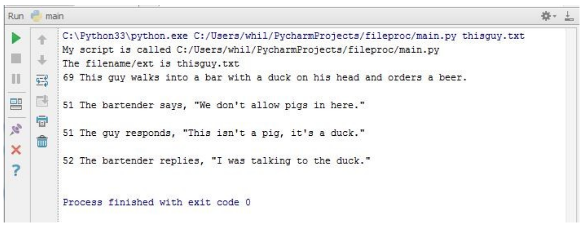
Processing a File

Our goal is to be able to manipulate the contents of the file in a variety of ways. We want to get comfortable moving through the file. One way to learn is to find out how long each line in the file is. To do so, we'll have to spin through the lines and perform some sort of line length operation on each line, and print out the length, one after another. Thinking ahead just a bit, we'd end up with a list of numbers after the program is done. Not really useful – and difficult to vet. How would we know if the results are correct? How about if we print out the line next to the length?

The construct we'll use (no surprise!) is the “FOR” loop, although with Python, we don't have to close it. Instead (remember the first article), indentation tells Python how long the construct is (example 3).

```
txti=open(filenameext)
for line in txti.readlines():
    print(len(line), line)
txti.close()
# this non-indented comment is not
# part of the FOR construct
```

Unfortunately, the result may not be... quite what we were expecting, as the lines each are double-spaced. See **Figure 6**.



```
C:\Python33\python.exe C:/Users/whil/PycharmProjects/fileproc/main.py thisguy.txt
My script is called C:/Users/whil/PycharmProjects/fileproc/main.py
The filename/ext is thisguy.txt
69 This guy walks into a bar with a duck on his head and orders a beer.

51 The bartender says, "We don't allow pigs in here."

51 The guy responds, "This isn't a pig, it's a duck."

52 The bartender replies, "I was talking to the duck."


Process finished with exit code 0
```

Figure 6. Echoing the length of each line in the source file.

What's this white space between the lines? When we grabbed the line, the newline chars were included, and they're also “printed.” How about if we knock the last byte off each line before printing? (Example 4)

```
txti=open(filenameext)
for line in txti.readlines():
    nn = len(line) - 1
    print(len(line), line[:nn])
txti.close()
```

Much better. (The [:nn] construct will be covered in the “Working with Strings” section later.)

Let's discuss a bit of what we've seen so far. The indentation rule is worth repeating, although for most of us, it'll become second nature almost immediately. Technically speaking, you don't need to have a non-indented line immediately after the construct. For example:

```
txti=open(filenameext)
for line in txti.readlines():
    print(len(line), line)

txti.close()
```

although that style isn't considered 'Pythonic'. Also, something that still bites me, flipping back and forth between Fox and Python, is the required ':' terminator after the FOR statement.

So now we can read through a file and manipulate individual lines. Now it's time to learn to write results of our processing to a second file.

Writing to a File

Instead of echoing our calculations to the screen via print(), let's send them to a second file. In order to do so, we'll need to open that second file, and do so in 'write' mode:

```
txto=open('outputfile.txt','w')
```

The 'txto' string (the trailing 'o' is for 'output') is an object reference that we can manipulate just like 'txti' was earlier. We'll do so thusly:

```
txto.write(str(len(line)))
txto.write(line[:10])
txto.write("\n")
```

We'll get the length of each line, grab the first few characters of that line (so that we double check to make sure we are doing the work correctly, e.g. make sure we are skipping through the file and not simply processing one line over and over again), and, finally, terminate the line. The salient part of the script (example 5) now looks like this:

```
txti=open(filenameext)
txto=open('outputfile.txt','w')
for line in txti.readlines():
    txto.write(str(len(line))+' ')
    txto.write(line[:10])
    txto.write("\n")
txti.close()
txto.close()
```

You'll notice a couple of things in this code snippet. First, the output file was opened with a second parm, 'w', identifying that the file is open to write to. Second, unlike print(), the 'write()' function can only take one parameter, so you can write multiple items either by concatenating them (as I did in the first write() statement), or by using multiple write() statements, as I did in the following lines.

The resulting 'outputfile.txt' looks like this:

```
69 This guy w
51 The barten
51 The guy re
52 The barten
```

Optimizing File Opening

(The code for this project is in main.py in the FileRead project.)

While txti.readlines() construct works fine to grab and process the entire file at once, it's less than optimal for several reasons. First, because it is processing the file 'live', it's difficult to go through the file more than once. You'd have to open the file and execute readlines() again, which is clearly wasteful.

Even worse, readlines() reads the entire file into memory all at once, and thus can perform poorly in terms of memory usage. Let's look at an alternative.

There is a second method, readline(), that allows you to move through a file line by line, like so (example 1):

```
txti=open(filenameext)
txti.readline()
<first line>
txti.readline()
<second line>
<etc>
```

However, you have to trap for the return of an empty string to tell that you've reached the end of the file.

Python has a concept called 'iterable objects'; we saw a glimpse of them in the 'animals' example earlier. A file is a natural target for demonstrating this concept, as the file data type has a built 'next' method that allows you to step through it line by line (example 2).

```
txti=open(filenameext)
txti.next()
<first line>
<etc>
```

However, the next() method, like the readline() method above, requires you to trap for the end of the line, this time raising a StopIteration exception that can be trapped.

Since the file object has this next() method built in, it can be stepped through with a for loop because the next() method and the end of file StopIteration except are automatically handled. Thus, this works quickly and elegantly (example 3):

```
txti=open(filenameext)
for line in txti:
    print(len(line), line)
txti.close()
```

With these tools, we can now begin to work on the monster 2000 column file mentioned at the beginning of this article.

Project 2: Working with Strings

(The code for this project is in main.py in the FileMinMax project.)

We have to know what is in our very, very, very, very large data file before we can work with it. For example, how many lines are in the file? What's the longest line? The shortest? How many of a specific character are in a line? Where is a specific character in a line? How can I pull a substring out of a line?

Create a project named **fileminmax** and add a main file called **main.py.**

Counting Lines

Determining how many lines exist in this file is, by now, anti-climactic. Example 1:

```
txti=open(filenameext)
howmany = 0
for line in txti.readlines():
    howmany += 1
print('howmany', howmany)
txti.close()
```

As an aside, this is more an example than practical, as there are easier ways than scanning through the whole list to determine the number of items in that list. For example, if we store the entire file to a list (named "lines"), the len() function will tell us how many items are in the list, as shown in example 2:

```
txti=open(filenameext)
lines=txti.readlines()
howmany=len(lines)
print('how many lines',howmany)
txti.close()
```

Even more 'pythonic' is this single line:

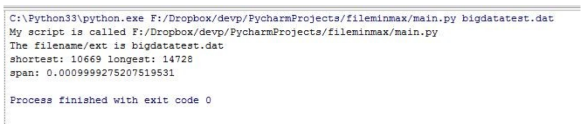
```
num_lines = sum(1 for line in
open('myfile.txt'))
```

Determining shortest/longest lines

We'll iterate through the file, line by line, and compare the length of each line to predefined minimum and maximum values. If the length of the current line exceeds either, the min or max is updated. Example 3:

```
txti=open(filenameext)
nummaxchar = 0
numminchar = 100000
sstart = time.time()
# iterate through list content
for line in txti:
    linelen = len(line)
    if linelen > nummaxchar:
        nummaxchar = linelen
    if linelen < numminchar:
        numminchar = linelen
print('shortest:', numminchar, 'longest:',
   nummaxchar)
sstop = time.time()
print('span:', sstop-sstart)
txti.close()
```

The results should look something like that shown in **Figure 7**.



```
C:\Python33\python.exe F:/Dropbox/devp/PycharmProjects/fileminmax/main.py bigdatatest.dat
My script is called F:/Dropbox/devp/PycharmProjects/fileminmax/main.py
The filename/ext is bigdatatest.dat
shortest: 10669 longest: 14728
span: 0.0009999275207519531

Process finished with exit code 0
```

Figure 7. The output of the FileMinMax project.

Determining columns in a line

Third, we want to figure out how many data columns are in a line. Columns are separated by the pipe ("|") character, so if there are 20 columns, there will be 19 pipes in the row (there is not a leading or terminated pipe.) The count() function will return the number of items in a list, like so (example 4):

```
txti=open(filenameext)
oneline=txti.readline()
howmany = oneline.count('|')
print('number of pipes:', howmany)
txti.close()
```

If you had more complex evaluations to do, you could wrap a test in a FOR loop and increment the counter, as in Example 5:

```
txti=open(filenameext)
oneline=txti.readline()
howmany = 0
charaprev = ''
for chara in oneline:
    if chara=='|' and charaprev<>'|':
        howmany +=1
    charaprev = chara
print('number of pipes:', howmany)
txti.close()
```

Where is a specific character in a line?

VFP devs, you're thinking 'at()', aren't you? Python has two mechanisms to determine where the first instance of a character is in a line: find() and index(). The difference is that find() returns a '-1' if not found while index() raises an error condition.

```
>>> oneline='abcdef'
ol.find('d')
3
ol.find('i')
-1
ol.index('i')
Traceback (most recent call last):
ValueError: substring not found
```

Thus, to find the first instance of a pipe character in a line:

```
firstpipepos = oneline.index("|")
```

Where to find the nth char in a line?

One of the niceties of VFP's at() is the third parm, where you can specific which occurrence of the string you're looking for. Python's find() and index() have no such parm. Instead (because we're going to need this in a little bit), it's time to write our first function.

```
def findnth(bigstr, lilstr, i):
    pos = bigstr.find(lilstr)
    while pos >= 0 and i > 1:
        pos = bigstr.find(lilstr,
                          pos+len(lilstr))
        i -= 1
    return pos
```

And there you go, you can see how, while the syntax is a bit different, you already intuitively understand every line of code.

To use it

```
txti= open(filenameext)
oneline=txti.readline()
begpipenum=7
begpipepos = findnth(oneline, '|', begpipenum)
print('pos', begpipenum, ' is', begpipepos)
txti.close()
```

How to pull a substring out of a line?

Next, we'll want to pull a substring, and now that we know how to find where a specific character is (say, the beginning of that substring), we can go to town!

The '[:nn]' (VFP devs, think 'substr()') is not just useful, it's quite flexible. For instance (see example 7 in main.py):

```
# prints entire line
print(line)
# prints first ten chars
print(line[:10])
# starts at 16th char, prints rest of line
print(line[15:])
# prints characters 3 through 16
print(line[2:15])
```

The careful reader may spot a seeming discrepancy between the second and third examples. Yes, Python is zero based, so [15:] starts with character number 16, as the index begins with 0.

We now have some tools to use with our very, very, very, very big data file.

Project 3: Splitting Chunks

(The code for this project is in main.py in the FileSplitChunks project.)

Armed with a rough idea of the size and scope of the file and the ability to extract sections of a line, we now want to analyze exactly what's in this file and break it down into manageable pieces. Remember, this file contains thousands of columns; just plain silly no matter which way you look at it.

Under the hood of the files

The business logic is trivial, and the algorithms aren't very complicated either. The value this exercise brings is experience using a new language, and learning new concepts. Let's look at the business and processing parts first.

We know a couple of things about this monster file, according to a file specification that was provided with the actual data file.

The file consists of one set of two columns and then a number of 192 column chunks, each of which is identical, and each of which contains data applicable to a specific industry. For example, one set of 192 columns contains data for industry code 402, the next set of 192 columns contains data for industry code 404a, and so on.

You can think of this structure as a set of identical spreadsheets placed one next to the other; the first two columns acting as a compound primary key that applies to the row in each of the spreadsheets.

```
CPK             Chunk1    Chunk2     Chunk3
geo     occ     b1..b192  c1..c192   d1..d192
0502344 1024    29   8.0  10   2.7   128  35.1
```

Thus, the compound primary key of 0502344 1024 applies to the data ranging from 29 to 8.0 in chunk 1, the data ranging from 10 to 2.7 in chunk 2, and the data ranging from 128 to 35.1 in chunk 3, and so on.

When we split this file apart, we'll create separate files for each chunk (each named with the industry code mentioned above), but will need to include the 'geo' and 'occ' columns as the first two columns in each file.

```
File from chunk 1
CPK
geo     occ    b1..b192
0502344 1024   29   8.0

File from chunk 2
CPK
geo     occ    c1..c192
0502344 1024   10   2.7

File from chunk 3
CPK
geo     occ    d1..d192
0502344 1024  128  35.1
```

More specifically, the first column is a geographic ID, a 14 character string that defines the type of region the data applies to (say, a state or a city) and then a PK for that region. The second column is an occupation code for the data, describing what job(s) the data applies to.

We don't need to know anything more than that, but it's important to understand that these two data elements apply to all of the data in the row. Thus, when we pull chunks of columns out of the row, we need to include these two columns each time as well.

The first thing we'll need to do is determine how many 192-column chunks are in the file. We've already determined how many pipe characters are in a row; what we need to do now is, based on that number, determine how many 192 column chunks are in that file.

```
# pipes = # chunks * 192 + 1
```

or

```
# chunks = (# pipes-1)/192
```

So a data file with (for example) 2113 pipes has 11 chunks. After breaking this data file up, we'll end up with 11 files, each with 194 columns (the geo ID column, the occ column, and the 192 columns in the chunk) in it.

The project is **filesplitchunks**. Let's look at how to do it.

First, we'll need to use the code from the previous project to determine how many chunks we are creating and add the calculation to get the number of chunks:

```
howmanychunks = (howmanypipes-1)/192
print('number of chunks:', howmanychunks)
```

Armed with this, we'll need to create a series of files (fileN, where N ranges from 1 to 'howmanychunks') and then, as we iterate through each line in the very, very, very, very big data file, write segments of data to each of the files.

The p-code looks like this:

```
for line in lines:
    get the first 2 columns
    for chunk of howmanychunks
        get 192 columns
        write 2 cols + 192 cols to fileN
```

The implementation will be a bit more involved, of course. I'll spend some time talking about the syntax involved, because isn't that what drives you crazy – you have the algorithm working, but there's just **one** expression that doesn't seem to be working like you expect?

Building the industry files

First, let's talk about how to build the files that will hold the chunks. As mentioned earlier, we'll use the industry code to build a unique filename; the chunk of columns belonging to industry 402 will be placed in a file named ind_402.dat.

One of VFP's hidden gems is macro expansion, being able to substitute strings into variables that are interpreted at runtime. So we'd be able to do this:

```
dime laInd[11]
laInd[1] = '401'
laInd[2] = '402'
laInd[3] = '402a'
…
laInd[11] = '409'

for li = 1 to alen(laInd)
    lcNaFile = 'ind_' + laInd(li) + '.dat'
    strtofile(geoid+"|"+occ, lcNaFile)
next
```

In Python, we'd do something like this:

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

```
for ind in indlist:
    print('industry:', ind)
    feout = 'ind_%s.dat' % (ind)
    print('output fn for:', ind, feout)
    with open(feout, 'w') as out:
        out.write(ind)
```

See what I mean by "it's just syntax"? Let's walk through the code, line by line.

The first statement (spread over three lines here) introduces a new concept of 'lists'. We saw that we stored the names of the chunks to a one-dimensional array in VFP.

You might be thinking 'array!', as you would with VFP. But Python has different mechanisms to handle this type of work. First, Python's arrays are different than VFPs, and we won't use them much. Second, for many purposes, Python uses a mechanism called a 'list', which is exactly what it sounds like, a list of 'things'. Think "list = one-dimensional array."

So the statement

```
indlist = ['401', '402', '402a', '403'...]
```

creates a list of industry codes that we'll use shortly. By the way, lists, as you might have guessed, are zero-based. If you wanted to retrieve the fourth item in a list, you'd do this:

```
>>> indlist[3]
403
```

(You can start adding your own 'one-off error' jokes now.)

The second line should be comfortable to you by now; we're iterating through the items in the "indlist" list, referring to them by the variable named "ind". As a way to be comfortable, I've included a "print()" statement that simply displays the value of "ind" for each iteration of the "for" construct. It's not necessary for the working of the program.

The next line is where the magic starts. We're creating a variable named "feout" (for filename extension output), and stuffing a string that looks like (mostly) this:

```
ind_.dat
```

The one piece that is foreign is the "%s" string. It's a placeholder for a variable, sort of like how we used "laInd(li)" in our VFP assignment:

```
lcNaFile = 'ind_' + laInd(li) + '.dat'
```

earlier. The difference is that the '%s' is actually a formatting construct, while the variable being stuffed into the construct follows, via the

```
% (ind)
```

string. (I'll refer you to the docs for all the nuances of '%' format strings, as there are many, much like VFP's InputMasks and Format strings.) So, putting it all together, we're iterating through the "indlist" list, stuffing the values, one by one, into the 'feout' variable (Filename Extension OUTput):

```
    feout = 'ind_%s.dat' % (ind)
```

We now have a name of the file that looks like this:

```
ind_401.dat
```

But we just have a filename, the file itself doesn't really exist yet. (Again, the sample code prints the name, just to show you what it really looks like.)

The next statement actually creates the file:

```
    with open(feout, 'w') as out:
```

and the final statement writes data (in this case, just the industry code) to the file just created:

```
        out.write(ind)
```

So now we can create a variable number of files and write to them. Next, it's time to parse the two PK columns and the appropriate set of 192 columns for each chunk.

Parsing the PK Columns

For each line, we'll need to grab the geoid and occ columns. Here's how:

```
txti=open(filenameext)
for line in txti:
    # get the first 2 columns
    linelen = len(line)
    firstpipepos = line.index("|")
    secondpipepos = line.index("|",
firstpipepos+1,linelen)
    geoid = line[:firstpipepos]
    occ = line[firstpipepos+1:secondpipepos]
txti.close()
```

Back to the construct that extracts **geoid** and **occ**. The first few lines in this code segment should be familiar by now. We grab an object reference to the input file, and then iterate through each line in that file via "for". For each line, we determine how long the line is, because we'll need that value shortly.

Next, we determine where the first pipe is, like so:

```
firstpipepos = line.index("|")
```

which enables us to grab the geoid column – all the data from the beginning of the row to the first pipe, like so:

```
geoid = line[:firstpipepos]
```

As mentioned earlier, the [:j] construct is similar to VFP's substring function, taking the first 'n' characters. (If there had been a value before the colon, the [i:j] construction would have taken the characters from "i" through "j". Again, remember this is zero-based.)

Parsing the 'occ' column is slightly more difficult, as we need to start at the first character after the first pipe, and continue until we reach the second pipe.

The stripped down format looks like this:

```
occ = line[from:to]
```

The expression

```
line.find("|")+1
```

begins the extraction at the position after the first pipe, and since we have the position of the first pipe:

```
firstpipepos+1
```

is where we start the extraction of the second column. Easy enough. The location of the second pipe is a bit trickier.

The "index" function can be passed additional parameters that define where it starts and ends. In order to find the second pipe, we want the index function to start searching after the first pipe, which we've found is

```
line.find("|")+1
```

Then we want the searching to end at the end of the line, which is the value

```
linelen
```

that we calculated earlier. Putting it all together, the location of the second pipe (the 'to' expression) is

```
secondpipepos = line.find("|",
firstpipepos+1, linelen)
```

(This all goes on one line, it's broken in order to fit.) Thus, the occ column is:

```
    occ = line[firstpipepos+1:secondpipepos]
```

We'll use "geoid" and "occ" repeatedly when writing to the industry files.

Parsing a 192 Column Chunk

Our next task is the parse a 192 column chunk of data. We'll do this 'n' times, where 'n' is the number of chunks in the very, very, very, very big data file.

Much like grabbing the 'occ' column, we can grab the 192 columns in one fell swoop by identifying where the pipe that comes before the string is, and then extracting all the data between that pipe and the 193rd pipe following.

```
begpipenum = 2+192*(index)
endpipenum = 2+192*(index+1)
```

Then we can find the position of those pipes using our 'findnth' function, described earlier in this article:

```
begpipepos = findnth(oneline, '|', begpipenum)
endpipepos = findnth(oneline, '|', endpipenum)
```

And, finally, grab the chunk, like so:

```
thischunk = line[begpipepos+1:endpipepos]
```

This particular construct uses the algorithm discussed earlier to identify the positions of the starting and ending characters for a specific chunk as identified by the 'ind' (index) counter. Now let's put it all together, spinning through the entire file.

```
txti=open(filenameext)
for oneline in txti:
    linelen = len(oneline)
    firstpipepos = oneline.find("|")
    secondpipepos = oneline.find("|",
              firstpipepos+1,linelen)
    geoid = oneline[:firstpipepos]
    occ = oneline[firstpipepos+1:
                 secondpipepos]
    index=0
    begpipenum = 2+192*(index)
    endpipenum = 2+192*(index+1)
    begpipepos
       = findnth(oneline, '|', begpipenum)
    endpipepos
       = findnth(oneline, '|', endpipenum)
    thischunk = oneline[begpipepos+1:
                        endpipepos]
txti.close()
```

Assembling the PK and 192 Column Chunks

Now that we can grab a chunk for a single industry, it's time to put this all together. We have two ways to go about this.

One way is to roll through the file and process each line for the first industry, then roll through the file a second time for the second 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.

Another way is to process each line in the file once, writing to each of the industry files in turn, for each line. 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?

Next time, as we learn to write to a series of files, we'll look at both ways, using our million row file to acquire some timing data. 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.

Source code

Project 1 (Working with Files):

fileProc: 1proc.py, thisguy.txt
fileRead: 2read.py, skeleton.txt

Project 2 (Working with Strings):

fileMinMax: 3minmax.py, test002rows.dat

Project 3 (Splitting Chunks)

fileSplitChunks: 4split.py

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

Here's a suggestion: get rid of all the negatives. Not, doesn't. Whatever. Rewrite any sentence with three commas in it. Remove one from any sentence with two. Your writing feels strained, like you're trying too hard to build complex sentences. Or maybe this is my problem, and I'm projecting. Refer to Strunk & White. I may have to go back and read that, myself. ???