Data Munging with Python

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 a multi-gigabyte table.

But even that solution won't work in some situations. Another component of that project involved a multi-gigabyte text file that 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.

In this article, I'll introduce another tool that should be in your toolkit, and in subsequent articles, show you how to attack this type of problem. The tool I'm referring to is Python, an open-source programming language that has become very popular over the last decade.

I started on the Python journey a while back, wanting to expand my repertoire of development environments. Many VFP developers feel that .NET is the only choice available to them; as a former MVP and Lifetime Achievement award winner who got into big lots of trouble with Microsoft for championing Linux a decade ago, I knew better.

More importantly, I foresaw the day where I needed to be able to build applications on more platforms than just Windows. These days, customers are requesting work done for desktop Macs, Linux servers, and Android mobile platforms. VFP and .NET were not going to be able to satisfy that demand.

Python can.

For those of you unfamiliar, Python, created two decades ago, is an interpreted language with a syntax and flavor that VFP developers will find oddly familiar. Not identical, mind you; there are clear differences that will catch you unawares. But by and large, you can be up and running with a simple but productive Python script (think 'program') in an hour or two, unlike the learning curve required for popular competitors like C++ or Java.

Things You Need to Know

Most of you readers are VFP old-timers, and, as such, used to doing things 'the VFP way'. Changing paradigms, while not difficult, does require a virtual smack upside the head occasionally in order to shake out the old ways of thinking. For example, those of you who have done client-server apps may remember the shift in thinking from table-based navigation to set-based ways of doing things. "No, we're NOT going to bring the entire 370,000 record data set down the wire to display in a grid."

So what new ways of thinking are required for Python?

Don't look back

There are two major versions of Python out in the wild today, the 2.x release tree and the 3.x release tree.

As a VFP developer spoiled by Fox Team's excellent track record on compatibility, you will recoil in shock when you find out that 3.x is not backwards compatible with 2.x.

That begs the question, should you use 2.x or start fresh with 3.x?

Unless you've got an overwhelming reason to use 2.x (such as you've got a job that has a big 2.x code base), start with 3.x. It's been around long enough to have a good following, a lot of libraries converted from 2.x, and completely stable.

No kitchen sink here

First, Python is not an 'all-in-one' toolbox like VFP is. You're used to firing up VFP and having everything you need, except for perhaps an external SQL database, or (shudder), a handful of ActiveX controls. Right there in the IDE, you've got your Editor, your Form Builder, your Menu Builder, your Debugger, your Code References, your Object Browser, well, you get the idea.

With Python, you double-click on the icon and you get.... a command line, much like the dot prompt of dBASE II days. See **Figure 1**.

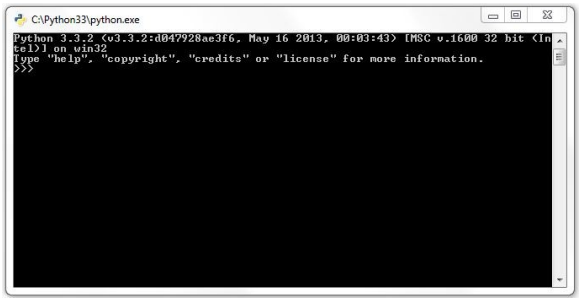

Figure 1. The Python Command Line.

To be sure, there are tools more advanced than ">>> ". When you install Python, IDLE comes along for the ride, Python's native **I**ntegrated **D**eve**L**opment **E**nvironment. Alas, IDLE is "advanced" in the same way that Notepad++ is more advanced than Windows Notepad. See **Figure 2**.

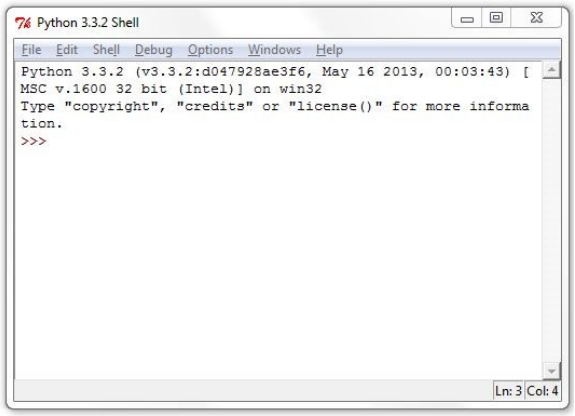

Figure 2. The IDLE interface, more advanced... than what?

There are more full featured alternatives to the Command Line and IDLE, and I'll discuss them in a bit. The point here is that Python doesn't fire up a widget-filled IDE like VFP or Visual Studio. So that's the first difference, and it takes a bit of tinkering to get your Python development environment set up the way you want it to work. That's a large part of what we'll be talking about in this article.

Inside Python

Once you've loaded Python, you'll begin to program by writing code in a text file (yes, code, none of those visual drag and drop 'code-less' methodologies for us!)

Unlike VFP, where you can pretty much format your code any way you want, all the way to just using the first four characters of language elements (who does THAT?), Python has several specific requirements.

Python scripts are simple text files with a .py extension. Inside those text files...

First, Python is case sensitive. As a result, 'amount' is different from 'Amount':

```
>>> amount = 100
>>> Amount = 95
>>> amount - Amount
5
```

Another example is

```
>>> sendinvoice = true
```

generates a "NameError: name 'true' is not defined" response while this works just fine:

```
>>> sendinvoice = True
```

Second, indentation matters. Indentation defines code blocks, like so:

```
if amount > 0
   sendinvoice = True
   print("sent")
else
  sendinvoice = False
```

But all lines must be indented the same amount:

```
if amount > 0
   sendinvoice = True
     print("sent")
else
  sendinvoice = False
```

generates an error.

Third, the basic Python interpreter just interprets your code. Unlike VFP, with its thousands of built-in functions as well as the object model, the classes, and the builders, the Python interpreter by itself doesn't come with a lot of goodies. You need to **include** modules to add functionality, much like you would include procedure files and libraries in VFP. For example, in VFP, you can issue the command

```
? cd
```

and the current directory is displayed. In Python, to determine the current directory, you need to import the 'os' module and then call its getcwd() function:

```
>>> import os
>>> os.getcwd()
c:\users\<you>\devp
```

Installing Python

Download Python from python.org. Since you're likely running Windows, click on "Download" on the left sidebar, and look for

```
Python 3.3.2 Windows x86 MSI Installer
(Windows binary -- does not include source)
```

You'll download a file named Python-3.n.n.msi that's about 20 megabytes. Double-clicking will run the Python installer, except for one optional step described next, there's nothing to see here

folks, move along, just click through until you're done. When finished, you'll have a pair of icons on your desktop, the Python command line and IDLE.

The one step that you might want to change from the default is Customizing the features to be installed. Click on the red X next to Add python.exe to Path and have the installer add python to the Windows path. See **Figure 3.**

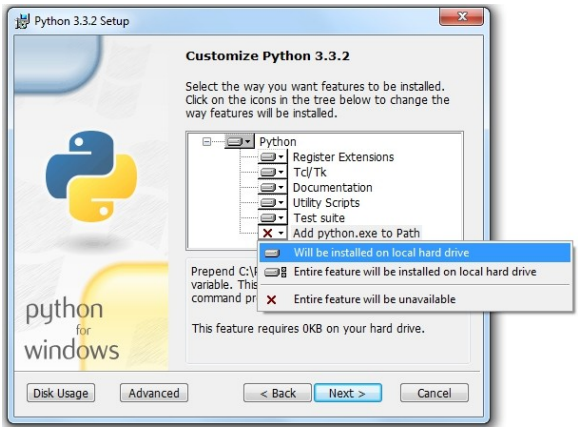

Figure 3. Adding python to the Windows path.

You'll end up with a new string in your Windows path, like so:

```
C:\Python33\;C:\Windows\system32;C:\Windows;
C:\Windows\System32\Wbem;C:\Windows\System32
\WindowsPowerShell\v1.0\;
```

Using the Python Command Line

The Command Line program can be found under Programs\Python 3.3. You can run it from there, or send it to your desktop, or pin it to your task bar. Whichever you do, run it in order to open the window (as shown earlier in Figure 1.)

Just like the venerable dBASE II dot prompt, the Python Command Line allows you to type commands and echo the results back. Since this is a single window, there's no luxury of a "Command Window" and a separate output area (the screen) as we're used to in VFP.

Using the Command Window works just like you'd expect:

```
>>> 2+4
6
>>> 'hello!'
'hello'
>>> 'hello' + 'world!'
'helloworld!'
>>> 'hello' + ' ' + 'world!'
'hello world!'
```

It's not the most efficient use of printed space here to go through a bunch of exercises to teach you Python syntax. I'd recommend an excellent online tutorial called "Learn Python the Hard Way!" It's written for Python 2.7, so a few commands don't work quite the same way, but the general philosophy – you gain the first level of mastery of a language by teaching your fingers to type the language elements – is sound. Go ahead and check it out at LearnPythonTheHardWay.org. I'll wait.

OK, now that you've done so, you'll see that the first exercise doesn't work at all – the print statement in 2.7 was changed to a print function in 3.x. So the exercise that prompts you to try

```
>>> print "Hello World!"
```

fails like so:

```
File "<stdin>", line 1
print "Hello World!"
                    ^
SyntaxError: invalid syntax
```

Instead, you need to pass the string as an argument to a function:

```
>>> print("Hello World!")
Hello World!
```

There's a great summary of the changes between 2.x and 3.x at

```
http://ptgmedia.pearsoncmg.com/imprint_downloa
ds/informit/promotions/python/python2python3.p
df
```

Anyway, we'll not be using the Command Line to create programs, simply to test command syntax that we'll put in our programs.

Your First Python Program

...will of course be "Hello World." We'll do it in the simplest way possible, creating a text file that contains our program, and then executing it via the Python interpreter, just to get a feel for the basics. Then we'll do the same using IDLE, and finally, with a full fledged IDE.

Setting up a work folder

In order to get started, we need to do a bit of housekeeping. First, we need to have a place to put our work. By default, both the Command Line and IDLE will perform its work in the Python 3.3 root folder. Not the best place to save our work. Let's create a folder for our Python programs. For the sake of this article, create a folder named 'devp' in your user profile, like so:

```
c:\users\<you>\devp
```

This will serve as a scratch folder while we're getting started. Later we'll learn how to create folders for separate projects.

Configure a Command Prompt

While I've talked about the Python Command Line, a great interactive tool, you can execute programs via the Python interpreter. Open a Windows Command Prompt, change to the folder that contains your Python program, and then type

```
c:\> cd \users\<you>\devp>
c:\users\<you>\devp> python yourprogram.py
```

Since it'll be a nuisance to keep executing the 'cd' command, why not change the 'Start In' property of the Command Prompt to the 'devp' folder so that the next time you fire it up, you're already where you want to be.

Creating and running Hello World!

Time to start.

Open up your favorite text editor. If you don't have one, might I suggest Notepad++? Some folks like gedit, and on Linux, it's my tool of choice, but on Windows, there's a noticeable lag to start up, so I use Notepad++ instead.

Enter the following line into your text editor

```
print("Hello, World!")
```

and save the file as 'hello.py' in your 'devp' folder.

Switch over to your Command Prompt and type

```
c:\users\<you>\devp> python hello.py
```

*\\\ redo this screen shot without 'admin'?

You should see "Hello, World!' echoed back under the DOS prompt, as shown in **Figure 4**.

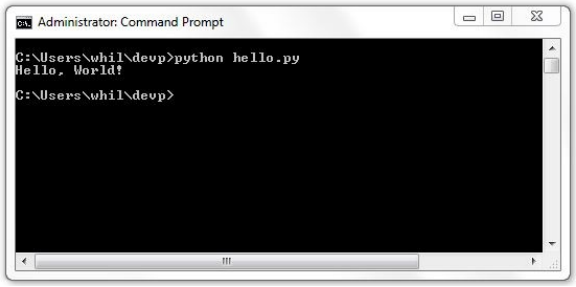

Figure 4. Setting up the Windows Command Prompt.

IDLE!

Yeah, I know what you're thinking. "What a nuisance, to have to have three separate programs running – a text editor, a Python interpreter, and a Windows Command Prompt. Doesn't really seem like the mid-2010s, does it?"

Python comes with it's own simple IDE, IDLE, that incorporates all of these functions in one environment.

If you followed the defaults when installing Python, you'll find

```
IDLE (Python GUI)
```

in the Python 3.3 program group. Copy it to your desktop or taskbar, open up the shortcut's properties by right-clicking, and change "Start In" to your 'devp' folder, just like you did with the Windows Command Prompt.

Once done, double-click on the shortcut and you'll be greeted with the IDLE UI as shown back in Figure 2. (They call it a "GUI", which has left me scratching my head.) This is the IDLE shell.

Create your first Python script in IDLE by selecting File | New Window. You'll see a text editing window open up along side the IDLE shell window, as shown in **Figure 5**.

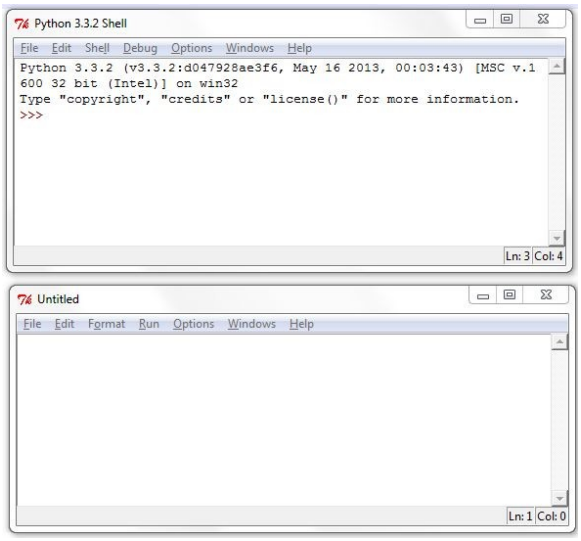

Figure 5. The IDLE shell and text editor.

Enter the following line into the editor window

```
print("Hello there, world!")
```

and save the file as hellothere.py. Then run the script via Run |Run Module or hitting F5. You'll see the results appear in the IDLE shell window, as shown in **Figure 6**.

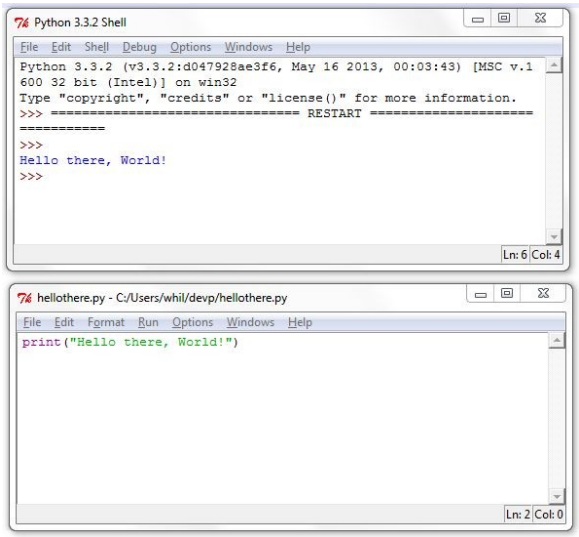

Figure 6. Running a program in IDLE.

Much better than our first option. While IDLE has a rudimentary debugger, we're used to something more robust. Let's keep looking.

*\\\your theme of IDEs being required for development. Running a couple of windows, linked or not, in order to do development, is the new/old normal. Your editor edits, your interpreter interprets, and your runtime application, perhaps a big app in dabo or a web app you view in a browser is a third. A debugger could be integrated or separate, just as the VFP debugger can be run in a separate frame (eliminating entire classes of Heisenbugs where focus goes ka-blouie when you click in an integrated debugger).

It's a hard lesson to learn, and I went through Eclipse (Java!) and Komodo and others before getting it. It's understandable you want to give VFP devs something familiar to work with (my two largest 'legacy' PHP apps still have VFP-standard naming conventions!) but you might check with Ed Leafe or Paul McNett to confirm this is yet another 'tab v. spaces' issue but the general consensus towards separate windows. Remember, they did name it Microsoft WINDOWS!

Installing PyCharm

While the command line or IDLE are all fine and good for simple tasks, as VFP developers, we're a little spoiled. Having to switch in and out of several applications just to run and test a chunk of code feels like we've regressed to the days of using BRIEF instead of the native FoxBase editor or watching FoxPro for DOS shell out to theWATCOM compiler.

Searching for Python IDEs brings up the usual list of options; while the ugly are easy to spot (last updated, 2007?), it's hard to parse out the good from the bad.

I'll suggest that you stay away from Komodo, while popular, it's getting increasingly too big for its britches, and suffering through their growing pains while you're just getting started will suck gumption from you faster than a user insisting "We don't need those fancy primary keys in our database."

Instead, PyCharm, by the Czech company JetBrains, walks that fine between 'too simple' and 'too complicated.'

Installing

Grab a 30 day evaluation from their website, jetbrains.com/pycharm. Installing the professional version is a matter of a few click-throughs, resulting in a 120 megabyte file (yeah, really?) and an opening screen as shown in **Figure 7**.

Figure 7. The PyCharm opening screen.

And configuring...

The first time you elect to create a project, you'll need to do some basic configuration. Figure 5 shows the Create New Project dialog, with spaces for the name, location, type of project and the interpreter. See **Figure 8**.

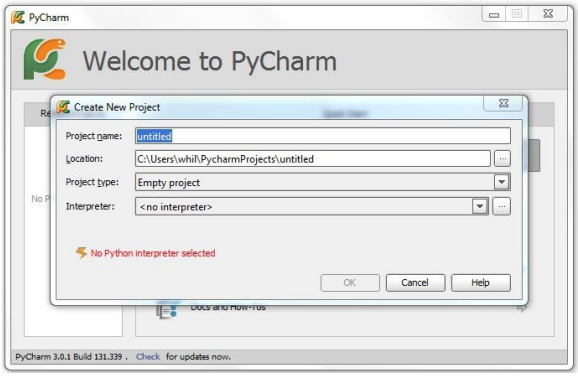

Figure 8. Starting a new project in PyCharm.

The first two are obvious, the third can be left as is (the other options provide wizards to create projects for various Python frameworks), and the last, well, therein lies the rub.

As installed, PyCharm doesn't know where Python is – you'll need to tell it. Click on the ellipsis to the right to open a dialog that allows you to match up your Python install and add in packages. See **Figure 9**.

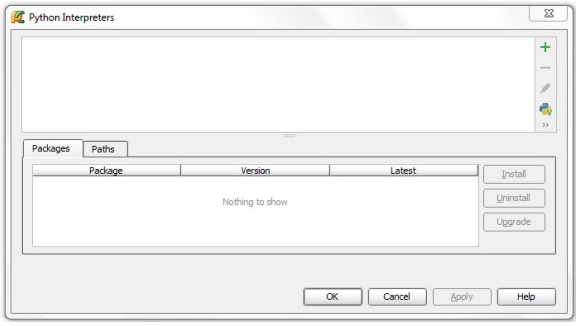

Figure 9. Getting ready to identify your Python install.

Click on the green (+) sign in the upper right to open the Select Interpreter Path as shown in **Figure 10**.

Figure 10. Selecting the Python interpreter path.

Once the Python interpreter is installed (you'll see it in the top listbox), the configure routine will complain via a yellow warning at the bottom that some packages are missing, specifically, the package management tools. See **Figure 11**. Click the friendly hyperlink and you'll be guided through the installation.

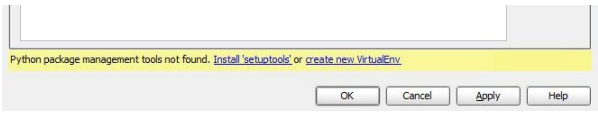

Figure 11. Warning that a package isn't installed.

If no packages are listed, be sure to highlight the item in the top listbox, as shown in **Figure 12**.

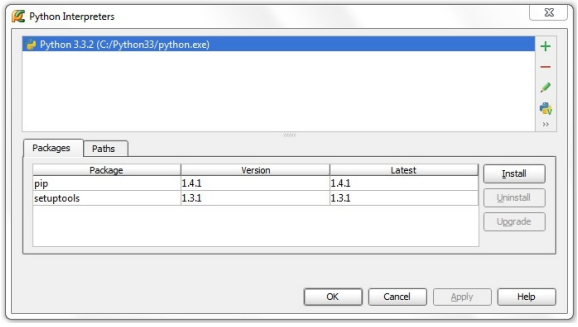

Figure 12. Displaying packages for a particular install.

By the way, when you see a blue arrow next to an item in the 'Latest' column, select that row to enable the Upgrade button. If all goes well, you'll be greeted with a green label (**Figure 13**); click the Apply and OK buttons and you're ready to start your first project.

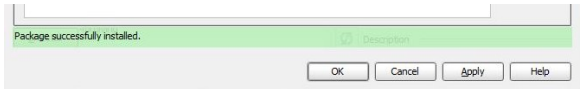

Figure 13. Packages successfully installed.

Your First PyCharm Project

Unlike the simple scripts we created earlier, PyCharm allows us to create and manage projects. You should be back to the Create New Project dialog shown earlier in Figure 8. Name your project 'helloworld' and click OK. You'll be greeted with the dialog as shown in **Figure 14**.

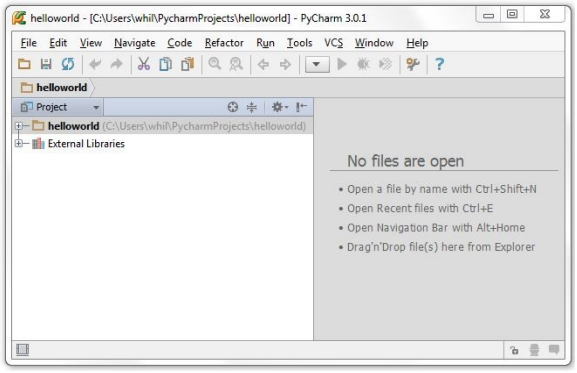

Figure 14. Your first Python project.

Under the hood, PyCharm created a folder where all the project's files are going to go. When you later call up a project, PyCharm will handle pathing within the folder.

Select File | New to display the New Options menu as shown in **Figure 15**, and select Python file.

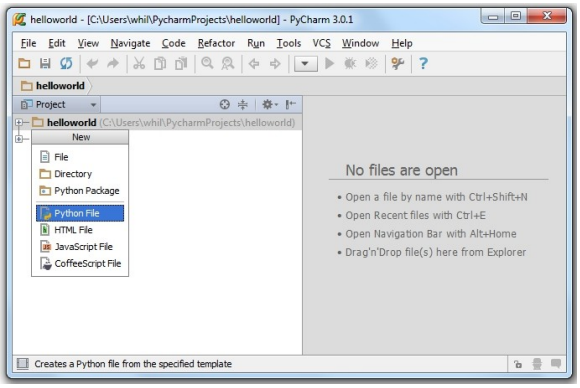

Figure 15. Adding a Python templated file to the project.

A new file will display in the editor pane on the right, with a brief template statement already built for you. See **Figure 16**.

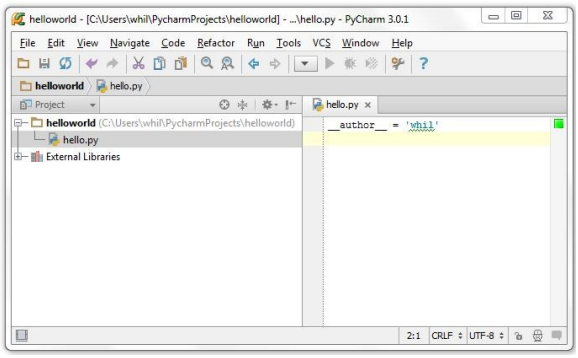

Figure 16. Adding a script file to the project.

This time, though, instead of adding a syntactically correct print statement, we're going to purposely make an error, so you can see the advantage of the PyCharm IDE.

The statement you'll type on the second line of the editor is the 2.7 version of print:

```
print "Hello again, World!"
```

Then select Run | Run 'hello' or click on the green arrow in the toolbar. The debugger pane will open and you'll see the error displayed, as shown in **Figure 17**.

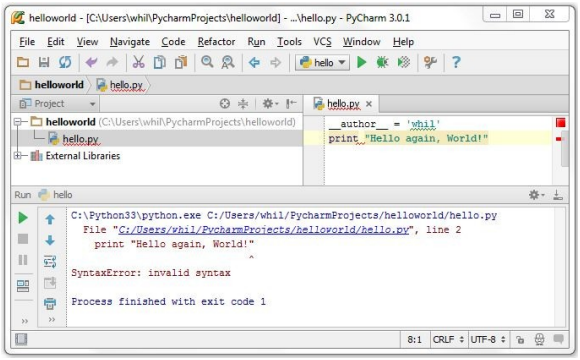

Figure 17. The PyCharm debugger displaying an error.

Make the correction (adding parens), Run again, and you'll see positive reinforcement in the Debugger pane. **Figure 18**.

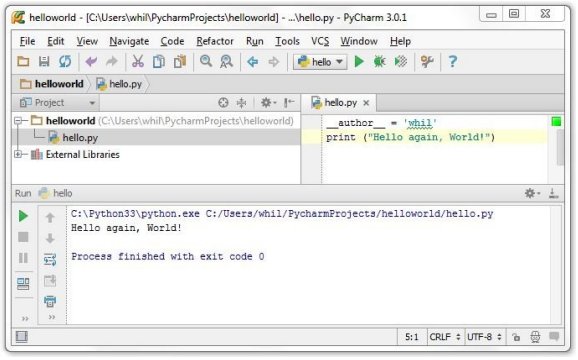

Figure 18. Your hello.py script running successfully.

Python, while syntactically similar to Visual FoxPro in many ways (compared to most other languages), still has enough differences to make getting started a challenge. This article has shown you how to set up Python's development environment so that you can begin creating projects and writing code in a half hour, rather than a couple of days.

In the next article in this series, I'll show you how to use Python to manipulate files and write to Visual FoxPro tables. Along the way, we'll learn more about creating Python scripts and maybe even take advantage of a little bit of Python object orientation!

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