Python as a Way Forward for VFP Developers

James Heuer
M-P Systems Services, Inc.
PMB #136
1631 NE Broadway
Portland, OR 97232, USA
503-335-8380 (voice):
www.mpss-pdx.com:
jsheuer@mpss-pdx.com:

Python is a powerful open source object oriented programming language with many features to appeal to Visual FoxPro programmers looking for a way to extend their applications without being tied to Microsoft .NET technology.

Contents

Introduction

Python is a powerful open source object oriented programming language with many features to appeal to Visual FoxPro programmers looking for a way to extend their applications without being tied to Microsoft .NET technology.

Having its own version of a "Command Window" where Python expressions can be typed in and immediately evaluated, directly executable Python source code files (no compilation step), and even an equivalent to the & macro operator, the EXECSCRIPT() function, and a powerful suite of TEXTMERGE-type functions, Python provides a comfortable environment for VFP programmers.

In this overview we will provide a window into the vast Python ecosystem of libraries, platform options (there is even a version called "Iron Python" which is compiled to .NET CLR code), and the key technologies like COM which make it interoperable with Visual FoxPro. We'll cover some of the things that Python can do that VFP struggles with and touch on those VFP features that Python likewise struggles with.

The white paper will end with a brief introduction to my company's work in building Python modules that can access DBF tables concurrently with VFP, interoperate with WestWind tools, and enable smooth interoperability with Visual FoxPro applications.

Topics we'll cover include:

- Basic features, strengths and limitations of Python as a programming partner to Visual FoxPro
- What Python code looks like and how programming in it is both similar and different from VFP
- Interoperating with Visual FoxPro via COM interfaces
- Options for GUI interfaces for console applications and their strengths and weaknesses
- Basics of deploying Python applications on customer computers
- Popular Python library modules that will help in your development with VFP and Python
- Where to get more information on Python

What is Python?

Python is a programming language. Python is a development ecosystem consisting of thousands of tools and 3rd party modules. Python is a tool for learning programming, widely adopted by universities world-wide.

Python was developed by Guido Van Rossum starting in 1989 in the Netherlands, and he has guided the development of the language ever since. Today the language's development is managed by the Python Software Foundation, Inc., including many major corporate participants like Google, Apple, and Microsoft. But despite corporate funding for some development projects, Python remains, and will always be, Open Source.

More specifically, Python is:

- Object oriented, but comfortably supports procedural coding

- Is semi-compiled (byte-code interpreted), but provides for options to compile to machine language modules
- Is broadly cross-platform, with implementations for Windows, MAC, MS-DOS, Unix, Linux, and other major operating systems, including mobile apps. It is packaged directly with all Linux distributions.
- It is offered in both 32-bit and 64-bit versions for most operating systems
- It is "loosely typed" in that any variable can be assigned any type of value at any time, but many operations expect to be applied to a single type (or a group of pre-defined types)
- It is completely Open Source, and there is no charge for the use of the language and its core libraries and most of the thousands of third party library modules
- Python can be used for web development, GUI console applications, commercially distributed software, and scientific/analytical programming, and many more.

Python and Visual FoxPro a Parallel History

VFP	Python
FoxPro 1.0 for DOS – 1989, by Fox Software, in Ohio	Version 0.9 released in the Netherlands, 1991, by Guido van Rossum
FoxPro Windows 2.6, March, 1994	First production release, January, 1994
Visual FoxPro 3.0, June, 1995, introduced object orientation. Some developers continue with FP 2.6.	Python Software Foundation created March, 2001, controls new development
Visual FoxPro 8.0, 2003	Version 3.0 released, incompatible with Version 2.x, 2008
VFP 9.0, SP 2 final release, October 2007	Version 3.7 release, June, 2018, Version 2.7 supported through 2020
Microsoft suspends support 2015	2018, Guido van Rossum steps down as BDFL

Figure 1 - VFP and Python Timelines

Though thought of as a "modern" programming language, Python has its roots in the same period as the launch of FoxPro 1.0 for DOS. As shown in **Figure 1 - VFP and Python Timelines** the first full production version of Python appeared a couple of months before Fox Software released FoxPro 2.6 for Windows. Both products experienced a major mid-life crisis. The FoxPro community was jolted by Microsoft's introduction of Visual FoxPro 3.0 in 1995, which had challenges with compatibility with the 2.x versions of FoxPro, and which introduced object orientation to the language, which was a major paradigm shift for many old VFP hands.

Python was faced with an even greater challenge when its version 3.0 was introduced in 2008, as it broke backward compatibility with all prior versions of Python. Further, the initial 3.0 version was markedly slower than 2.6, which it replaced, and didn't have the

rock-solid stability of the 2.x versions. Fast forward to 2018, by which time Python has weathered the transition to version 3.x, now up to 3.7, and the community is awaiting the release of Python 4.0, expected to be introduced in late 2019 or 2020. And the earlier problems with performance and reliability have been fully addressed in versions 3.6 and higher.

One thing to note about the development of Python compared to VFP is that Python's development has been guided since 1989 by its creator Guido Van Rossum. The Python community has unofficially designated Guido as the "Benevolent Dictator for Life" (BDFL), a post from which he unofficially retired in 2018, leaving the future of Python in the hands of the Python Software Foundation, Inc., which will manage the development into the future.

Python and Visual FoxPro Compared

As a long-time VFP coder, I was referred to Python by a fellow VFP veteran as being particularly welcoming to VFP folks used to the power, responsiveness and flexibility of FoxPro. There are both clear parallels ("loose typing", for example) and differences as illustrated below.

Table 1 - VFP and Python Compared

Feature	Visual FoxPro	Python
Programming Paradigm	Object oriented or procedural	Object oriented or procedural (also can support "functional programming")
Operating System Platform	Windows, 32-bit or 64-bit OS, 32-bit implementation	Windows, Linux, etc., 32-bit and 64-bit implementations
Variable typing	Loosely typed	Loosely typed
Compilation	Semi-compiled. VFP program code is automatically compiled to intermediate byte-code (.fxp files for example) when first run or compiled. C API available for compiling machine language modules to use in VFP applications.	Semi-compiled. Python .py files are automatically compiled to intermediate byte-code .pyc files when first run. C API available for compiling machine language modules to use in Python applications. Other compilation options are available.
Text Handling	Text merge and macros allow dynamic creation of text as well as code for immediate execution with EXECSCRIPT()	Powerful templating system to populate variables in text blocks as well as generating code for immediate execution with exec()
Dynamic code execution	Command Window for immediate code execution and testing	Python interpreter embedded in multiple environments allows immediate code

		execution and testing
Structured programming	Blocks of code are set off with "BEGIN" and "END" type syntax: e.g. FOR/ENDFOR	Blocks of code are set off with indentation, NOT punctuation or BEGIN/END syntax
Library Support	Several hundred modules in CodePlex and VFPX on GitHub and elsewhere	Python Package Index (https://pypi.org) has 152,000 projects
GUI Support	Proprietary built-in GUI forms builder with integrated data binding	Built-in GUI form system but no WYSIWYG builder. Third party GUI tools are available with WYSIWYG support.
Database support	Fast, native handling of VFP type DBF files, ODBC and OLEDB access to SQL databases and other data sources	No native database handling. Libraries available on PyPi for DBF, most SQL databases, and No-SQL databases like MongoDB
Report Writer	Built-in WYSIWYG report builder and output printing capability	No native report writer. Many output modules in PyPi, including powerful PDF generation.

Python Versions

Like FoxBase, FoxPro, and ultimately Visual FoxPro, Python has evolved through several versions, the latest of which as of fall, 2018, is 3.7.0. With the huge level of interest in Python, numerous off-shoots have been developed, and (as with FoxPro) earlier versions remain in active use.

CPython, 3.7.0 – The latest "canonical" version of Python. This is what is normally just referred to as "Python". Available in both 32-bit and 64-bit versions. Versions 3.4, 3.5, and 3.6 are largely interoperable, but with new features added to each succeeding version. All other versions are compared to this version for compatibility. ***This is the version you should probably start with – specifically the 32-bit version to allow COM communication with Visual FoxPro.*** This version supports the Python C API for compiling C or C++ code into modules directly usable by Python programs and the CTypes module that enables access to standard Windows DLLs written in C or compatible languages.

CPython, 2.7.15 – The last of the 2.x versions of Python. (32-bit and 64-bit are available too.) Officially replaced by version 3.0 in 2008, 2.x versions are still in widespread use, but lack some important features of the 3.x versions. Code written in 2.x Python generally must be re-factored to compile and run correctly under 3.x versions. While a 2-to-3 conversion utility is available, it is not completely foolproof. The C API and CTypes are also available here. For more on 2.x vs 3.x, see the section following this.

Iron Python – This is the .NET implementation of Python and can be tightly integrated

with Visual Basic and other .NET languages. If you have already embarked on major enhancements to your VFP application using .NET, this may provide a useful path into Python; however, Iron Python performance is generally not as good as regular Python, and some 3rd party modules may not be supported. Originally developed by Microsoft, but now supported by an Open Source team on GitHub which has been continuing development now that Microsoft seems to have lost interest.

Jython – Version of Python which is integrated with the Java language system. Java libraries can be "imported" into Jython code as can many regular Python libraries. Provides for convenience in interoperating between Java and Python where Java is the dominant partner.

PyPy – A fast, highly compatible implementation of the Python language with "just in time" compilation to machine language code… skipping the interpreted code feature in CPython. Many 3rd party libraries work just fine with PyPy, and PyPy is up to 10-50 times faster than CPython on compute-bound processes. Third party libraries that rely on the C API may or may not work. I have *not* tested the Python Win 32 Extensions (see below) in PyPy for COM interoperability.

Python 2.x vs 3.x

Many people may have heard about the upheavals created in the Python world when Version 3.0 came out and was not backwards compatible with Version 2.x. To answer the inevitable question: "Yes, the transition was difficult, but the popularity of Python has remained strong." The early limitations from important 3rd party library modules not supporting 3.x have largely disappeared. The overwhelming majority of the Python Package Index modules are now available for 3.x. Further, university computer science departments, which almost universally teach Python to newbie programmers, have all switched to Python 3.x.

But just how do 2.x and 3.x differ? The main difference is in the handling of string variables and the integration of Unicode into the language. In Python 2.x all strings are ASCII by default, and they must explicitly be converted to Unicode type strings when Unicode is required. Python 2.x has a "bytearray" for strings of bytes that don't represent human language text, but it is rarely used. Python 3.x, on the other hand, has a native string type which is *always* Unicode. It makes a very clean distinction between strings representing human language text (Unicode) and bytearrays, which are simply ordered collections of bytes with no particular meaning and are accessed byte-by-byte. This has several implications:

- Python 3.x smoothly handles HTML output that may need to show text for different languages, including multi-byte Asian languages commingled with English or other European languages (or some newly developed characters like emoticons).
- Python 3.x file write functions are sensitive to the type of the variable passed with bytes to be written: bytearrays require files opened for binary output, Unicode strings require files opened for text output. The file read functions are similarly sensitive to the file open mode.
- One "character" in a 3.x string may have a byte value greater than 255 where 2, 3, or

4 bytes are required to represent it. Your code doesn't need to know about this internal representation of your strings.

There are many other impactful but less dramatic changes to the language. For example in Python 2.x an expression like 2 / 3 evaluates to 0 because integer division always evaluates to an integer, and the decimal fraction portion of the result is truncated. When 3.x was being designed, this "feature" of 2.x was considered to be a mistake, so in 3.x the expression 2/3 now "correctly" evaluates to a float with a value of 0.6666667.

It is possible with improvements in the 2.x series to write code that is compatible with both 2.x and 3.x, and if you are developing libraries for the Open Source community you should consider doing so. Otherwise, embrace the modern world with version 3.7.

Installing Python

FoxPro users who intend to build components in Python that interoperate with VFP will need the 32-bit version of Python downloaded here:
https://www.python.org/downloads/windows/

You will want the Python 3.7.0 version Windows x86 executable installer download. This is a standard Windows installer package. It will put its executables and libraries in the c:\python37 directory by default, and you should generally allow it to do so (or at least put it in its own top level directory where you can get to it easily). Note: some 3.7 installers seem to want to put the Python executables in the c:\Users\All Users\Application Data directory. This is simply annoying and gets in the way of frequent access to this directory. You can change this to C:\Python37 or a directory of your choosing for convenience of access if you find that default being suggested.

In order to exploit Microsoft COM technology to interoperate between VFP and Python, you will need the Python Windows 32-bit Extensions. To install this feature you'll need to use the Python pip utility. To do this, open a Windows Command prompt and navigate to the c:\python37 directory (or wherever you installed Python). Then `cd scripts`. The command:

```
C:\Python37\Scripts>pip install pywin32
```

should result in the downloading and installation of the complete win 32 extensions for your version of Python. If you are using a Python 2.x version, you will need to find pywin32 on GitHub and download the appropriate version.

Using the Python Package Index

The on-line repository of 3rd-party libraries and tool modules contains some 152,000 projects. Yes, you read it right. And this number is growing daily. This mind-blowing array of tools is found at https://pypi.org, a website owned and managed by the Python Software Foundation. The sheer variety of topics listed in the module browse screens is staggering. Just some examples include:

- Ham Radio
- Telephony
- Home Automation

- 3D Rendering
- CD Music Readers/Players
- Artificial Intelligence
- Atmospheric Science
- OS System Administration
- And hundreds of others

And these tens of thousands of modules are available in a total of 55 different spoken languages including Urdu, Marathi, and Macedonian! (Not to mention Mandarin Chinese, English and other European languages.)

All of the tools mentioned in the following pages are available for download from PyPI. This is the result of a remarkable effort by the Python Software Foundation to wrangle tens of thousands of developers into using a common platform for module distribution. To obtain a module that you find in the PyPI index, you simply determine its correct name from the PyPI listing, and then open a command window and navigate to the `scripts` subdirectory of your Python install directory, for example it may be `c:\python37\scripts`. Then you use the pip command to obtain the package:

```
C:\python37\scripts>pip install somepackage
```

The PIP application then takes over, checking the modules you already have installed, looking for possible dependencies for your new module. If it finds that your version of Python is not supported by the module you are attempting to install, it will tell you that, so you don't waste your time further. Then it imports all the components you need from PyPI, including modules required as dependencies of your requested module that have not yet been installed on your system.

By the way, in the Python 2.x world, things were not quite so well organized, and some 2.x versions must be downloaded from other sources, but PyPI will usually inform you of where that might be.

Coding in Python

Some Basics

Four features of Python programming will be unfamiliar and potentially uncomfortable for VFP programmers, and I mention this first to get it out of the way:

Python, unlike VFP, is case sensitive. That means that the function helloWorld() is NOT the same as Helloworld(). It also means that you should adopt some conventions for how you inject capital letters into your variable, function and class names. Python has some standard practice guidelines to help you do that.

Python, uses the "import" command to bring modules into scope for the current program. This is similar to the "using" code in .NET. This differs from the VFP "SET PROCEDURE TO" and "SET LIBRARY TO" in that the import command has effect only for

the .py file in which it is contained.

Python function names and class names are "first class objects", and their object references can be stored to variables. There is no parallel capability in Visual FoxPro, and it may take some time to wrap your head around this concept. For example, you might define a function as "`def foo(cSomeParm):`". You can then assign `foo` to another variable like `fum = foo`, and then invoke the function by `fum("some parm")`.

Python uses white space (indentation) to delineate blocks of code. For example a "for each" type loop might look like this (note the colon ':' that marks a break in the structure also the lack of any indicator to mark the end of the structure block other than the change in indentation):

```
for cItem in myList:
     if cItem == "X":
          print("Found One")
     else:
          print("Bad One")
```

Examples from "Hello World" to Introspection!

The following is directly copied from a Python interpreter window:

Figure 2 - Idle, the Most Basic IDE

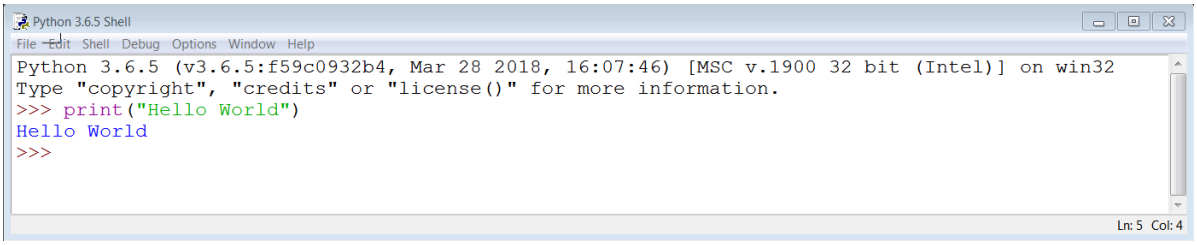


Figure 2 shows the Python Idle IDE screen, the simple, default cross-platform shell that provides the interactive environment to type in Python directly and see the results of executing the code. In this example the print() function was typed in, and the blue "Hello World" output appeared directly below it.

In the following examples, we show the source code as displayed in Idle followed by the output which results from running it. A brief explanation provides context for each example.

Note that in the example above I used Python 3.6.5 for this simple example. Since our company is currently still using Python 2.7.10, the examples which follow were run in that version of Idle. The code in these examples should work the same in version Python 3.x.

Objects, Classes, and Flow Control

Figure 3 - Creating an Object Class

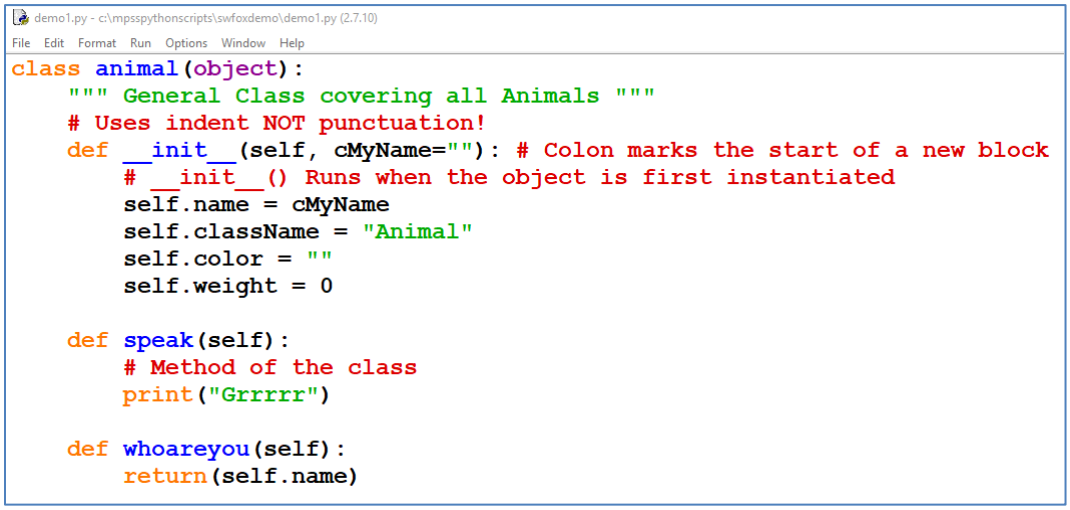
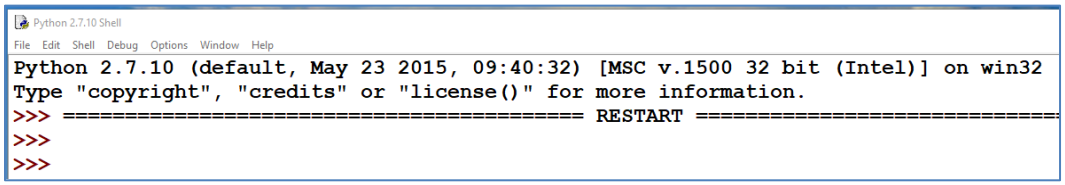


Figure 4 - Output from Creating an Object Class



The code in **Figure 3** actually produced no output as shown in **Figure 4**, only the header from the Python interpreter. This is because in a Python program, class definitions are executed (the definition is loaded into memory) but no action is taken at that point. Note the use of indentations to indicate where the functions are defined, and the hierarchy of indents to indicate levels. It is a Python convention to define all the instance properties of a class in the __init__(), which is executed when an object is first instantiated from the class.

The triple quotes at the top of the class definition are called the "doc string", and are considered best practice. The three quotes are also the highest level of quotation (the individual single and double quotes being lower) that allows practically anything to be included in the resulting string literal, including carriage returns and line feeds and other quotation symbols.

Figure 5 - Creating and Running a Class Instance

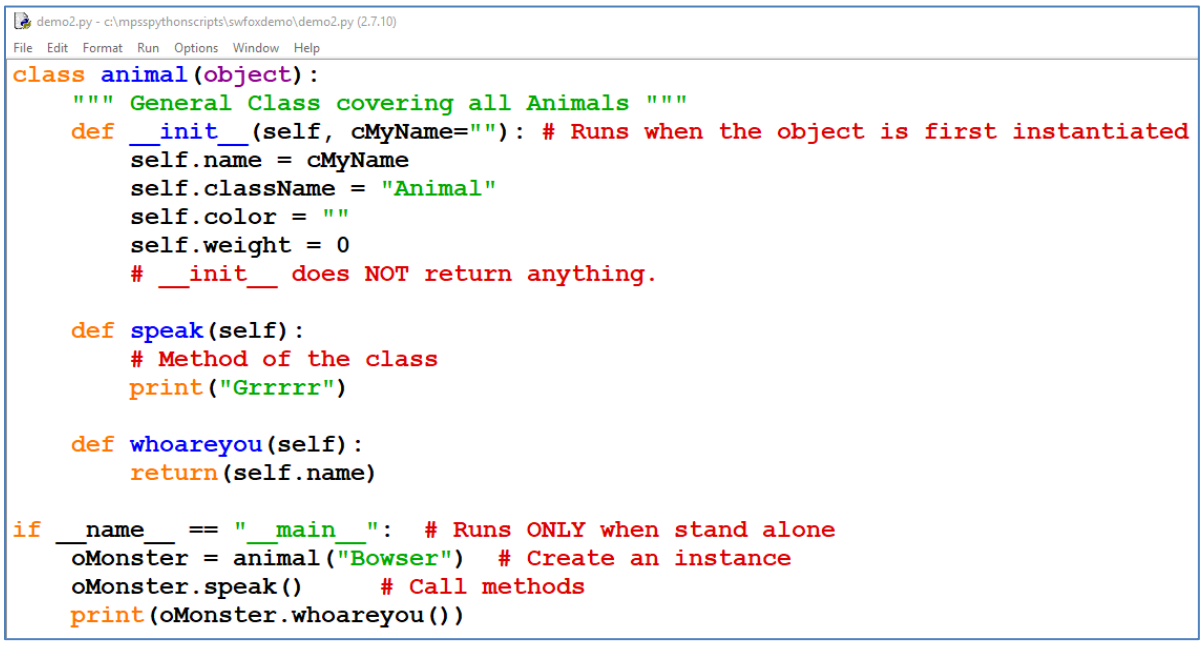
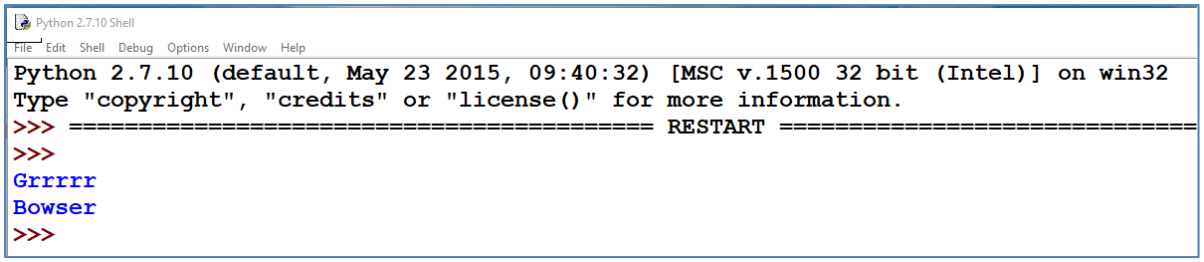


Figure 6 - Output from Creating and Running a Class Instance



By adding the `__name__ == "__main__":` code in **Figure 5** we have told Python to execute that block of code if this is running as a stand-alone program in the interpreter (its name will be the reserved word "__main__" (those are double underscores). This is a powerful feature that allows us to easily set up testing code for any module and run that code interactively in an IDE to test our code as we write it. In this example, we created an instance of animal, calling it oMonster and then invoked two of its functions, which both printed output in the Python Idle console, **Figure 6**. Note that unlike FoxPro, Python __init__() methods do not return anything, and short of a crash, the object will be created.

Arrays and Complex Data Types

In addition to simple scalar values, Python has a rich assortment of complex data types built into the language. Some of these have rough parallels in Visual FoxPro, but many do not.

Figure 7 - Dictionary Type in Python

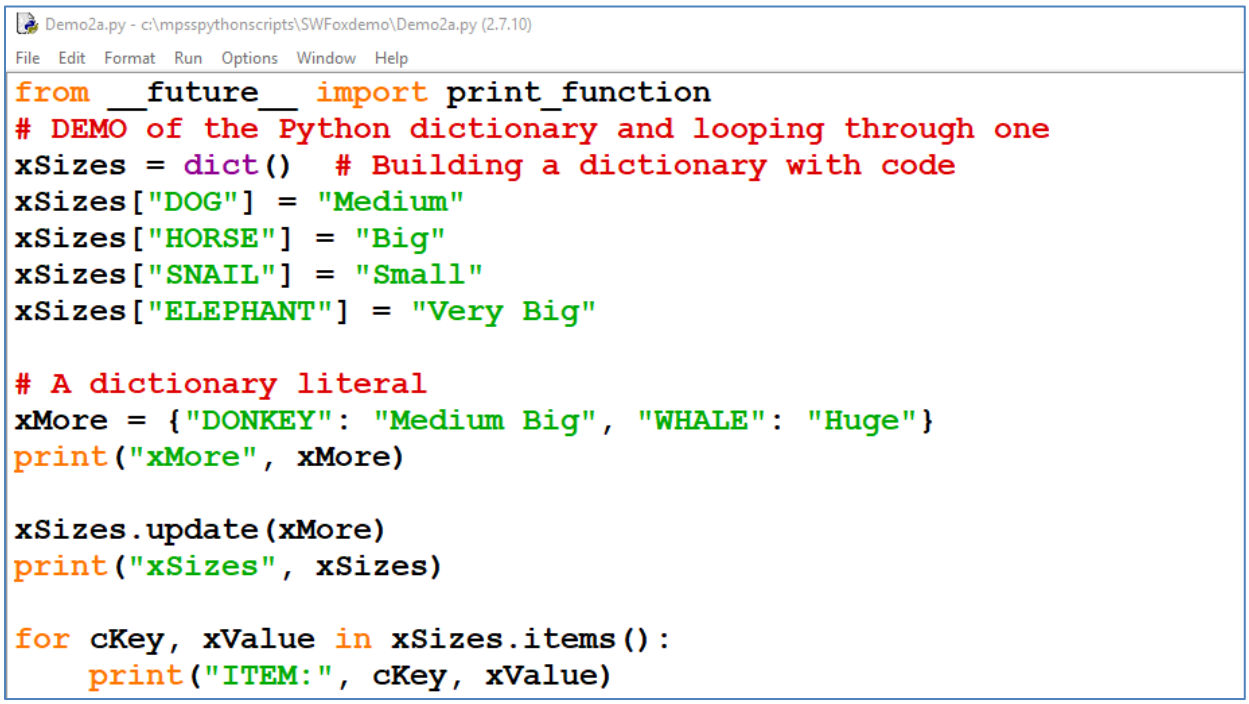
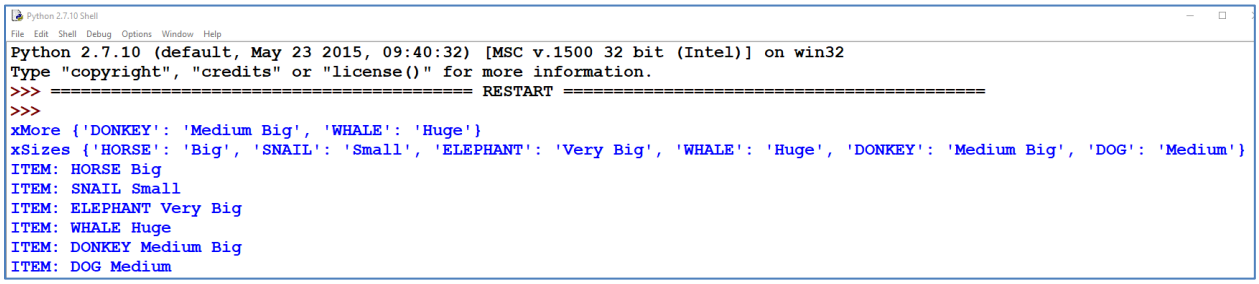


Figure 8 - Output from Dictionary Types in Python



The block of procedural code in **Figure 7** simply executed when we "ran" it from the Idle IDE and produced the output in **Figure 8**. At the top of the code is "`from __future__ import print_function`" one of several constructs that allow Python 2.x applications to function more like Python 3.x applications, thereby easing the transition to 3.x in the future.

In the body of the code I'm introducing the "dictionary". A dictionary is a set of name/value pairs. This works very much like a collection class in Visual FoxPro, but has some subtle differences. First the "name" may be virtually any type of Python variable provided it cannot be changed during the life of the dictionary. In the example we used text string "keys", but they could numbers or other objects, and they don't all need to be the same type. Dictionaries in Python can be defined either in code as shown or as dictionary literals. The update() method of a dictionary allows the elements of one dictionary (the parameter passed to the function) to be added to another.

Dictionaries are considered "iterables" in Python. This means that you can use a "for… each" (as in VFP or "for… in" in Python) type of construct to step through their elements in code. In this example, we use the items() iterator on the dictionary to return both a name (cKey) and a value (xValue) each time through the loop. This also illustrates another difference from VFP, Python functions can return multiple values!

Figure 9 - Demo of the Python List Object

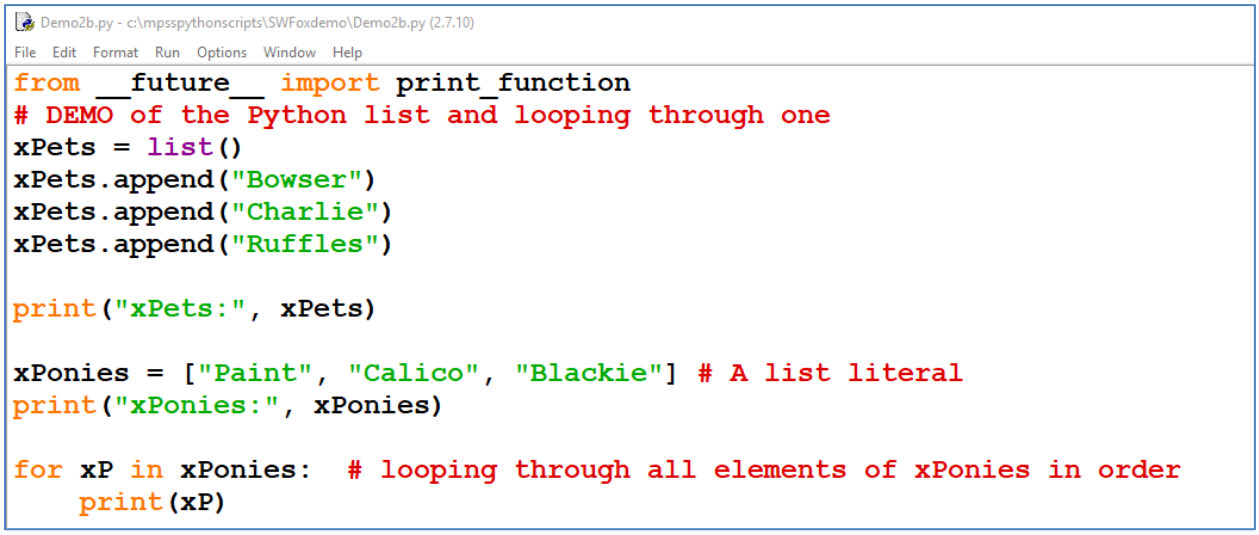
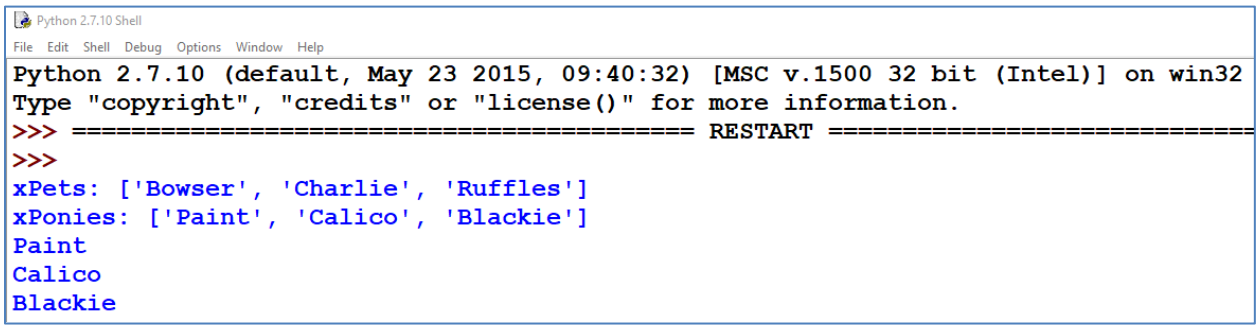


Figure 10 - Output from Demo of the Python List Object

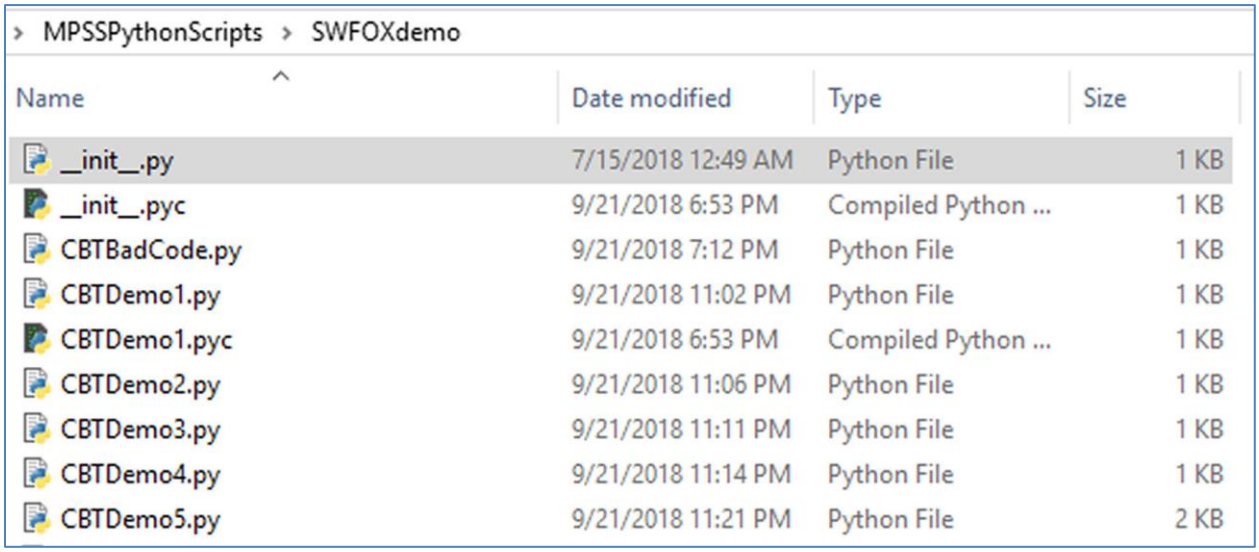


As illustrated in **Figure 9**, a list may be thought of as an ordered, single dimensional array. It is similar to FoxPro single dimensional arrays in that it may take values of any type in its elements, but different in that Python lists start out empty and grow as elements are added with the append() method as shown. It is also possible to define list literals as is shown with the xPonies list in the example. Lastly, the list differs from a VFP array in that its elements are addressed by a 0-based index, not a 1-based index.

While it is possible to use a for-next type of loop to iterate through a list (and we show that in an upcoming example) it is easiest to simply use the "for… in" iteration which returns every element of the list in turn as shown in **Figure 10**. Another nice feature of lists (as well as dictionaries and other compound Python objects and iterables) is that they can be printed or turned into strings for display of their content for debugging or other purposes.

Introducing "Modules"

Figure 11 - Example of a Python "Module"



We are all familiar with the init() method in FoxPro objects and the corresponding __init__() method in Python, but there is still another level of programming code aggregation that can have its own __init__() method in Python: the Module. A Module in Python is typically a set of related .py files contained in a single directory in the file system. In **Figure 11** above, the directory SWFOXdemo has become a Python Module by virtue of the presence of the __init__.py file contained in it. That is a signal to the Python interpreter that this is a Module and it should be treated as such at run-time. Now when an external Python component imports any object from any of the .py files in the directory, the code in the __init__.py program executes. This allows variables to be defined, resources to be allocated, etc. automatically. If you execute the code in the examples section of the session files, you'll find that there is a bit of code in this __init__.py program shown in

Figure 12 that pops up the little message box shown in **Figure 13**. There is also an essential initialization of the random module's number generator.

Figure 12 - Code in the __init__.py File

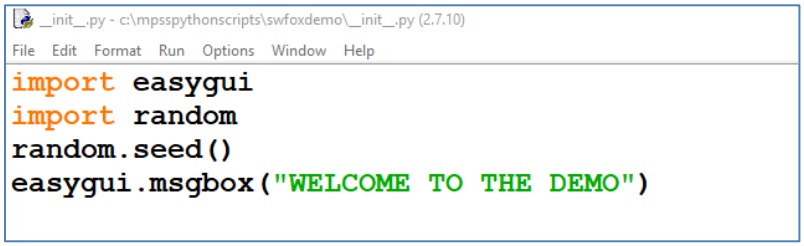
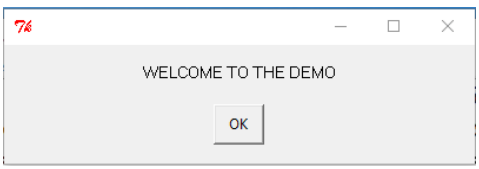


Figure 13 - The Resulting Message Box



The easygui module is downloaded from the Python Package Index repository and provides simple GUI tools like message boxes, file finders, etc., using TKinter but without any of the complexity of TKinter. (See GUI programming later in this paper.)

Figure 14 - More Complex Python Objects

demo2c.py - c:\mpsspythonscripts\swfoxdemo\demo2c.py (2.7.10)

File Edit Format Run Options

```
from __future__ import print_function  # compatibility with V. 3.7
# Lists of dictionaries - compound objects
from SWFOXdemo.DEMO2 import animal # This is a module -- see the module __init__ at work!

xMenagerie = list() # a List of Dictionaries containing names, their objects and owners
xMenagerie.append({"Name": "Bowser", "Object": animal("Bowser"), "Owner": "Bob"})
xMenagerie.append({"Name": "Charlie", "Object": animal("Charlie"), "Owner": "Terry"})
xMenagerie.append({"Name": "Ole Paint", "Object": animal("Ole Paint"), "Owner": "Larry"})

print("by number") # Iterate by number:
nLen = len(xMenagerie)
for jj in range(0, nLen):
    xM = xMenagerie[jj]
    print(xM["Name"])

print("")
print("by for") # Iterate with 'for...in'
for xM in xMenagerie:
    print(xM["Name"], xM["Object"].whoareyou())
```

Figure 15 - Output from More Complex Python Objects

Python 2.7.10 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================================== RESTART ================================
>>>
by number
Bowser
Charlie
Ole Paint

by for
Bowser Bowser
Charlie Charlie
Ole Paint Ole Paint
```

In **Figure 14** we first import the animal class from the DEMO2 program in the SWFOXdemo module. We again have specified a bit of code that allows this Python 2.7 version program to behave like Python 3.x: the import of the print function rather than the print command as found in Python 2.x. The list xMenagerie is created programmatically with the append() method adding a dictionary to each element. These dictionaries are defined by dictionary literals. Note that the 2nd element of each dictionary is an item named "Object", and the value is actually an instance of the animal() class, each given its own name when invoked (and passed to the __init__() method). This illustrates how not just scalar values but also objects of all kinds can be values in dictionaries.

There are two forms of iteration illustrated here with output in **Figure 15**. The first one demonstrates the classic for….next loop with an indexing variable, jj. The Python way of doing this is to use a range() iterator object which returns an integer each time it executes. Note that it starts with 0, since all lists and other indexable iterables use 0-based indexes in Python. The range() iterator also supports increments by values greater than 1 and backwards iteration. The second iteration example uses the simpler "for…in" type of loop, and actually calls a method on each of the objects referenced by the "Object" item in the dictionary.

Sub-Classing and Introspection

Python, like VFP, allows your custom classes to be further sub-classed. The rules for this are similar to VFP. However, there is one big difference: Python supports multiple inheritance. In other words, you can define a sub-class with two or more parent classes. Many computer science gurus believe this is a bad idea due to the complexity it introduces to your code, but Python allows it, and there are some potentially useful programming patterns that take advantage of it. Due to the complexity of multiple inheritance in Python I won't go into it further beyond this mention.

Figure 16 - Example of Sub-Classing in Python

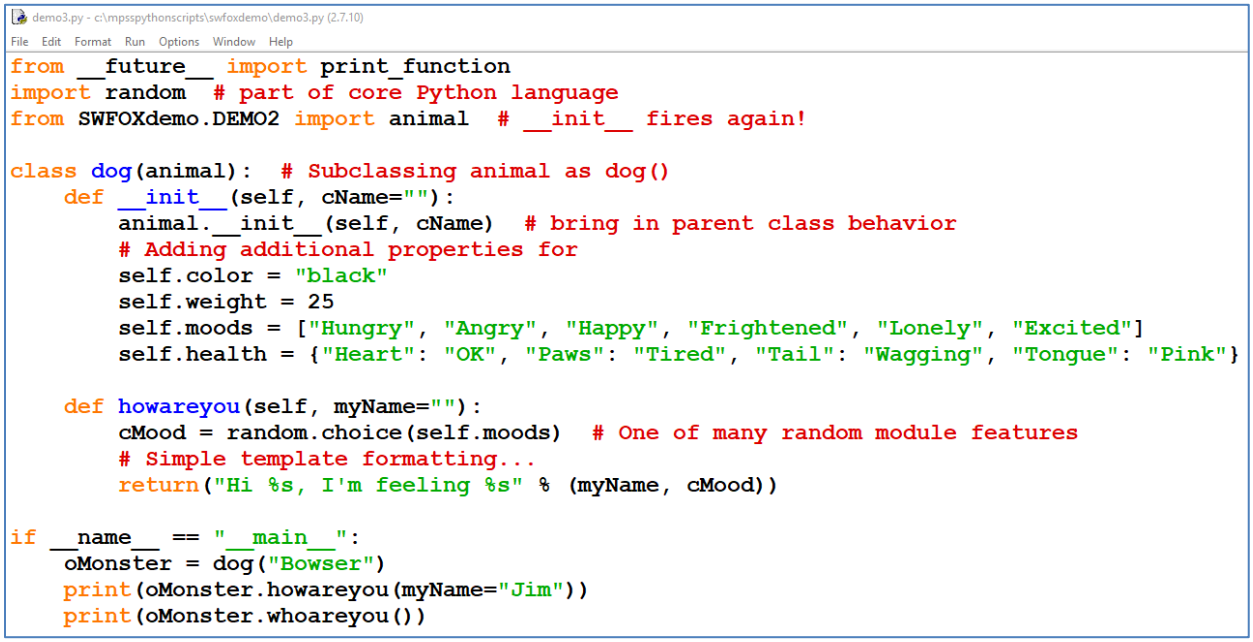
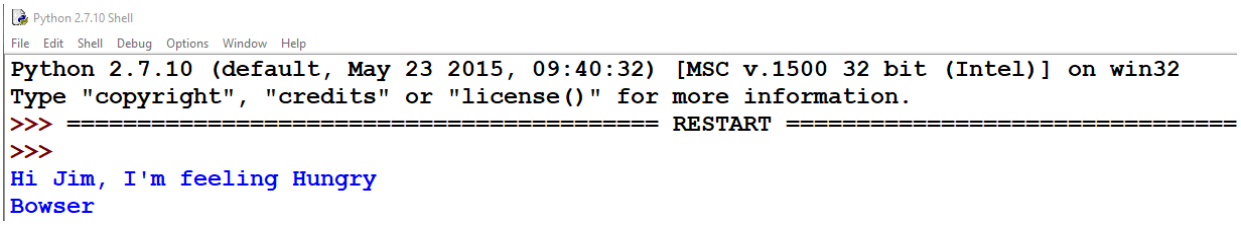


Figure 17 - Output from Example of Python Sub-Classing



The example in **Figure 16** shows a number of key features of Python. I create a class

named “dog” and make it a subclass of the previously defined “animal”. Note that I have simply referred to the animal class by importing it from DEMO2.py. The __init__() of the animal class is partially overridden by the dog class. As in FoxPro, when a method in the parent is also defined in a subclass, the parent class method of that name does NOT execute unless explicitly called. In this case, the animal.__init__(self) line accomplishes that like DODEFAULT() in VFP. Subsequently, there are additional properties defined for this subclass.

This example also introduces an example of the many powerful tools that are part of the main Python distribution. The random module is available with a simple import statement. One cool feature of the random module is the choice() method which takes as its argument any iterable (list, dictionary, string, etc.) and returns a randomly chosen member of that iterable, an example of which is displayed in the output in **Figure 17**.

Figure 18 - Introducing Introspection and Customization

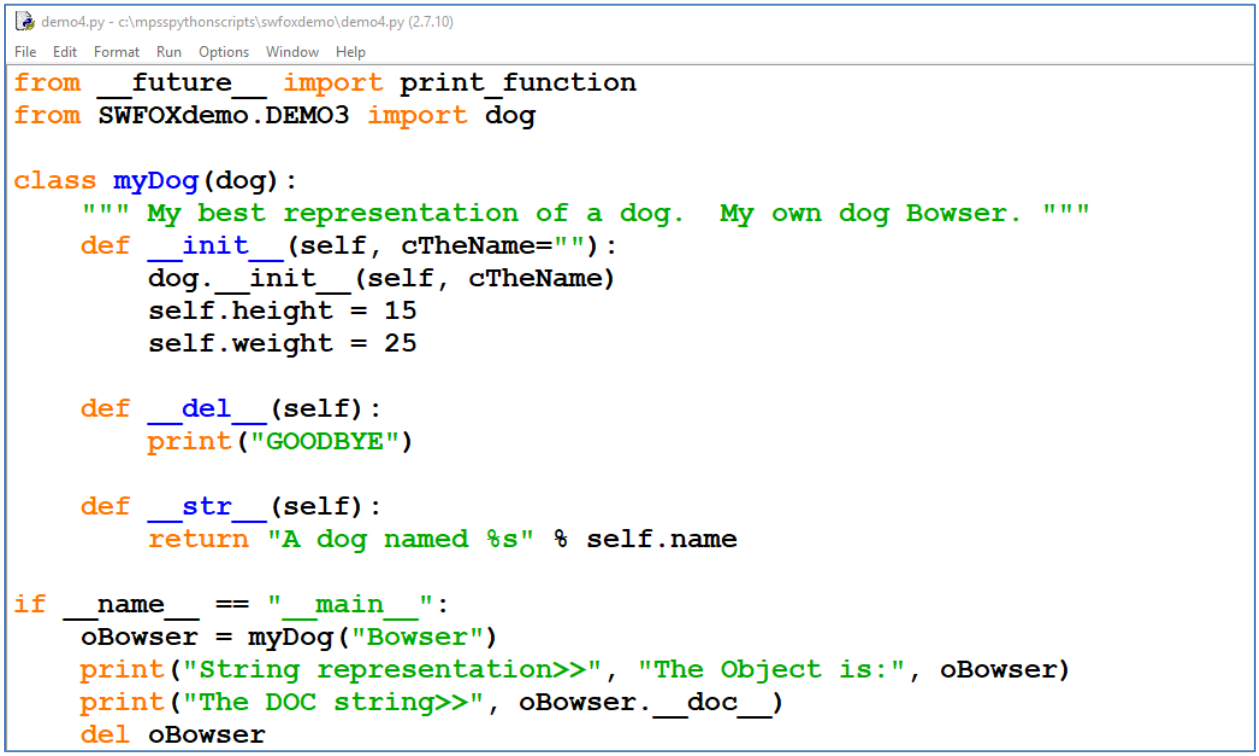
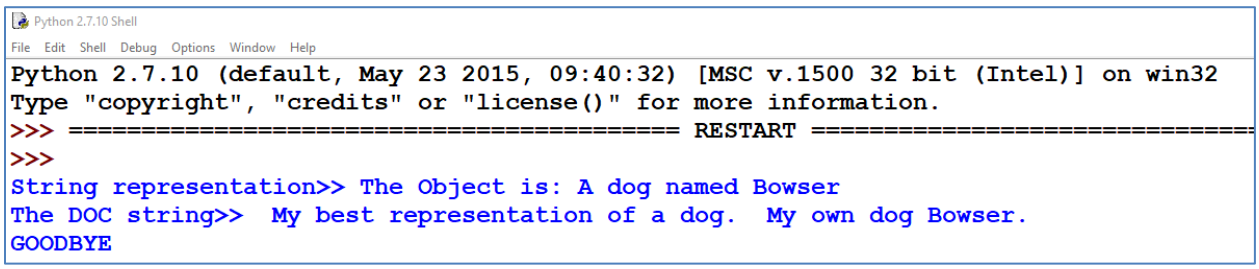


Figure 19 - Output for Introducing Introspection and Customization



Introspection in programming is the ability of the language to expose information about itself and its operations to the programmer. Basically, the programmer is able to discover

information about the program itself at run-time. Visual FoxPro has a limited capacity for this, but in Python it's an integral part of the language. Python also allows the programmer an impressive amount of control over the behavior of objects and modules.

In **Figure 18** I further subclassed the dog() class, which is imported from the DEMO3.py program in the SWFOXdemo module. The doc string which explains what myDog() is all about is not only useful to the programmer reading the code, but is actually stored in the __doc__ property of the class, so that it can be accessed by the programmer. This is the foundation of a very sophisticated suite of library modules to extract doc strings from Python applications and build documentation manuals automatically from them. More on this in *Documenting Your Code* below.

Another cool feature is the ability to override default class behavior. Any object like a string or a number in Python can be turned into a string for display in the interactive interpreter or for presentation to the user or debugger. But when an object instance of a class is rendered into a string, by default it's a not very attractive technical reference "<__main__.myDog object at 0x0370A750>". But by specifying a __str__() method, the programmer can control how the object is represented when converted to a text string, in this example "A dog named Bowser", which appears in the output in **Figure 19**.

Finally, in this example we included a __del__() method of the class. Like the DESTROY() method in VFP, this fires when the class is removed from memory, as happens when its last object reference, oBowser, is removed with the del command – appearing in the last line of the output.

GUI Console Applications in Python

One of VFP's greatest strengths has been its integrated visual form designer with seamless built-in databinding for all controls, making what has come to be called CRUD (Create, Read, Update, Delete) data table operations easy to program. While .NET has adopted many of the lessons from VFP development and IDE design, there's still nothing that beats VFP for this task.

In the Python eco-system, there is no single monolithic GUI development platform for console applications. Typical of the Open Source world, there are many options, some better than others, but none that stand out as "the" choice. Indeed some sites claim to have discovered over 30 GUI screen packages available for Python!

Still, there are major products developed with Python GUI interfaces, and some of them have been deployed to hundreds of thousands of sites. Two examples are the TortoiseHG (https://tortoisehg.bitbucket.io/) Windows user interface for the Mercurial Version Control System, and the local user interface for the DropBox™ cloud data storage system.

There are three top of the list GUI interface options for Python which are described below. Your choice of one of them will depend on how you intend to use it, whether cross platform operations are important to you, and how you intend to deploy your finished application.

TKinter – Just the Basics

Included with all Python distributions and functional in all Python versions, TKinter

(usually pronounced "kinter") is a highly capable tool for building forms and screens manually. There is no built-in WYSIWYG form designer, and the couple of modules that claim that capability for TKinter don't get very good reviews. Still, the layout mechanisms (though quite different from VFP and its concept of containers and "anchors") are robust, it has a broad selection of widgets, and is as far as many developers ever go for their console apps. It is the only GUI tool for Python that really and truly is available on all of the Python OS platforms that support user interfaces.

PyQT and PySide

QT is a library of C language routines that has been ported to multiple operating systems. At its core on Windows is a .DLL file with all kinds of functions to produce screens and UI widgets. There are implementations of QT for several languages including Python, Java, and others. There are two Python GUI modules that provide access to QT. The oldest and the one with the best documentation is PyQT, which was developed with help from Nokia Corp, the creator of QT itself (now distributed by the QT Corporation).

PyQT has a well-designed WYSIWYG form designer tool, provides some support for data binding, and generally looks good on all OS platforms it supports (Windows, Linux, and the Apple OS). The one downside of PyQT is that it is licensed for Open Source use under the fairly restrictive GNU GPL license which requires that any derived products be delivered to their users with all the source code of the derived product. Unless you are developing a product for internal company or personal use or are committed to the concept that all software should be free, this is probably NOT what you want to do. To bypass the GNU GPL, you can buy a license from Riverbank Software (https://www.riverbankcomputing.com/commercial/buy), which costs about $550 per developer. QT itself is an Open Source product, but it is available under the GNU Lesser GPL license which does NOT require release of your source code for derived products. However if you want hands-on support for QT itself, you can buy a support license.

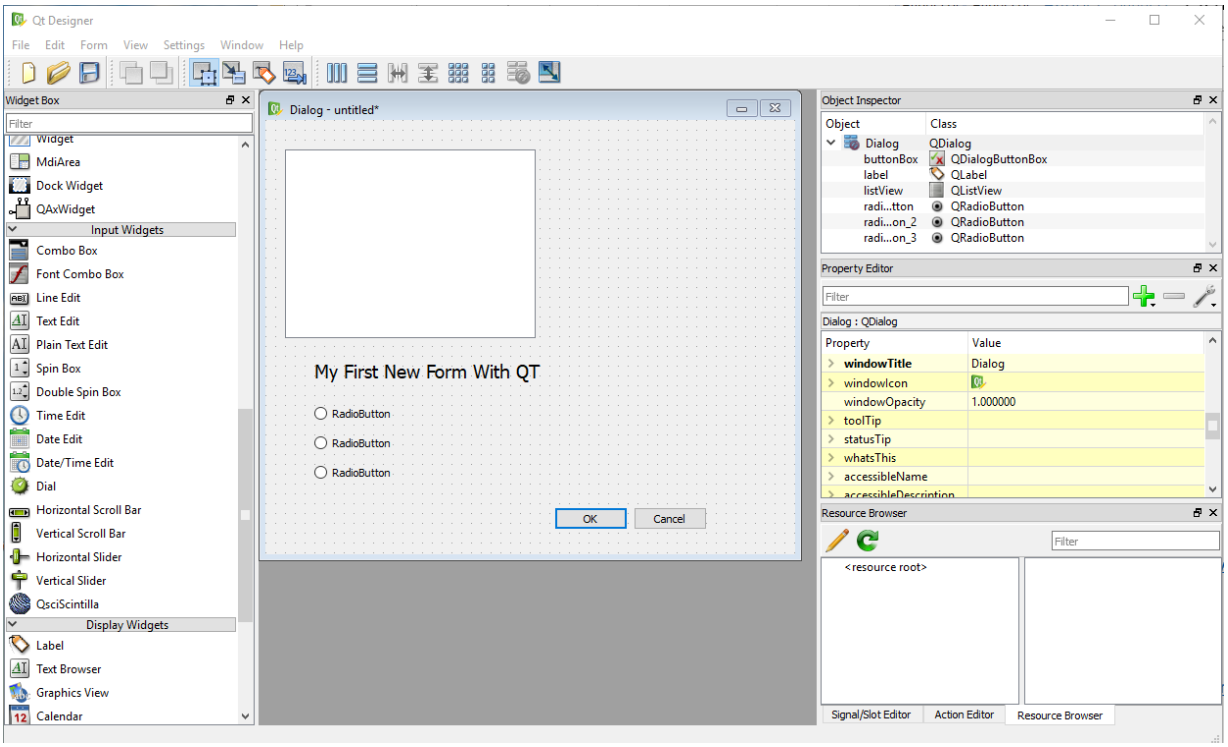
The PySide module also wraps the QT components, but uses a less restrictive license which doesn't force your product to be Open Source too. It too has a WYSIWYG form designer and provides access to QT's extensive array of screen widgets. However users report it has less friendly documentation, and for some things they wind up referring to PyQT documentation and examples.

Both PyQT and PySide are supported by PyInstall, which is discussed below as one of the best choices for building installable .EXE versions of your application for delivery to customer sites. The TortoiseHG program mentioned above is built using PyQT and delivered by a Windows installer program to many thousands of users.

An example screen shot of the QT form designer is shown below in

Figure 20.

Figure 20 - PyQT Form Designer



WXPython

Like PyQT and PySide, WXPython is built around an underlying cross-platform C library of screen/window building and display components. WXPython has its own WYSIWYG tools and is generally well regarded. Like PySide, it has a non-restrictive Open Source license, so does not require paying for the privilege of distributing closed-source versions of your apps. WXPython doesn't seem to support as many platforms as PyQT, but that may not be important to you. Check the latest docs if it is. The biggest difference is that WXPython doesn't have built-in data binding of any kind. There are some modules in PyPI which claim to provide that capability, but I have no experience with them.

Interoperating with Visual FoxPro Using COM

Considering that Python is proudly cross-platform, with versions that run not only under Windows, but also Linux, Solaris, Apple OS, and even IBM Mainframe systems, it may surprise you that Python's support for Windows-specific technologies is robust and mature. Early on, it was realized that Python needed a full suite of tools for Windows development, and the Win32 Extensions for Python were created. These provide a broad range of tools, including:

- COM Clients
- COM Servers
- WINSock access
- Named Pipes
- Python wrappers for almost the entire Win32 API, encapsulating the difficult pointer-based access to many of these functions
- DCOM implementation
- FTP, HTTP, and other communication technologies supported by the Win32 API.

It is the COM capability that makes Python a convenient partner for Visual FoxPro, as it allows VFP components to provide services to Python modules and vice versa, and to do so smoothly and with very high performance.

Accessing a VFP COM Server from Python

To illustrate the use of COM both as server and client there are a couple of simple examples. The first example makes use of a VFP COM object that I developed for this SWFox session. It has just three methods. One takes a simple string and translates it into a hexadecimal representation of the string (each character is replaced by the hex representation of its ASCII value). Another method reverses the process.

I also added a little method that returns a VFP array with a mixed bag of data elements, including numbers, a date value, text strings and logicals to demonstrate the flexibility of this type of connection.

As to performance, I have found that in either direction, the overhead of a COM call to an in-process COM server is roughly 0.0001 second or less. By comparison, there is price to be paid for going through inter-process communications, and the overhead of a COM call to an out-of-process COM server is closer to 0.001 to 0.002 seconds… Still fast, but worth considering when you are trying to process and generate HTML pages for a browser in sub-second times.

Figure 21 - Accessing a VFP COM Server

```
pwtest.py - c:\mpsspythonscripts\SWFoxdemo\pwtest.py (2.7.10)
File  Edit  Format  Run  Options  Window  Help
""" Testing a VFP COM object in SWFOXdemo.dll """
from __future__ import print_function
from win32com.client import Dispatch
from pythoncom import com_error
from MPSSCommon import MPSSBaseTools as mTools

try:
    oHEX = Dispatch("swfoxdemo.makehexstring")
except com_error:
    print("Dispatch Failed.  Did you forget to register the DLL?")
    oHEX = None
except:
    print("Unidentified error occurred.")
    oHEX = None

if oHEX is not None:
    cSourceString = "TESTING 12345"
    cHexVersion = oHEX.strtohex(cSourceString)
    print(("Source '%s' Converted to HEX: '%s'\n" % (cSourceString, cHexVersion)))

    cTestString = oHEX.hextostr(cHexVersion)
    print("HEX string '%s' Restored to String: '%s'\n" % (cHexVersion, cTestString))

    xArray = oHEX.gettestarray() # demo function in the VFP COM object, returns a "tuple"
    print("ARRAY", xArray)
    print("")
    tCorrectedTime = mTools.PyTime2datetime(xArray[1])
    print("Corrected DateTime", tCorrectedTime)

del oHEX
```

Figure 22 - Output from Accessing a COM Server

```
Python 2.7.10 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================================== RESTART ================================
>>>
Source 'TESTING 12345' Converted to HEX: '54455354494E47203132333435'

HEX string '54455354494E47203132333435' Restored to String: 'TESTING 12345'

ARRAY (u'TEST1', <PyTime:12/31/2018 12:00:00 AM>, 4930, u'ANOTHER TEST', True)

Corrected DateTime 2018-12-31 00:00:00
```

As shown in **Figure 21**, accessing a COM server requires just one module, the Dispatch component of the win32com.client library. A single call to the Dispatch() function works exactly like CREATEOBJECT() in VFP when accessing a COM ProgID (like "Excel.Application"). Methods on the COM server are called normally as with methods of any native Python object, and public properties of the COM object are accessed similarly.

Since we are making this code available (and the swfoxdemo.dll COM server) to attendees, it is possible that someone will try to run it on their own computer before they have registered the .DLL file with regsvr32.exe. Accordingly we wrapped the Dispatch() call in a try…except block to trap that error and provide a meaningful explanation. The error that is triggered by a bad ProgID is com_error, which is a special error object found in the pythoncom module, and which had to be imported to make it available for the exception trapping.

The little demo function gettestarray() provides some insights into COM technology here.

You can see the output where the line starts with the word "ARRAY" in **Figure 22**

Also note in the output that the string constants passed back are prefixed by a 'u' character. This signifies that the COM interface has converted the standard VFP strings into Unicode. This is part of the Windows COM technology. Since we are using Python 2.7 in this example, the 'u' makes this explicit. In Python 3.x, all standard text strings are automatically Unicode. Of course the numbers and logical values came across normally ("True" is the semantic equivalent of .T. in VFP), but the date value came across as a type PyTime. For technical reasons the Win32 components do not pass the VFP date values as Python date or datetime values. We have a simple function in our tools library to make this conversion.

In this example I also introduce another kind of "iterable", the tuple, which is the type of the result from the gettestarray() method. The tuple is like a list in that it is an ordered set of objects (of any kind), but it is different in that it is considered to be "immutable". In other words, it is not possible to alter a value in an element of a tuple or to append() another value to the end. Tuples are useful in situations like this where the "owner" of the object in question (this array) is actually a VFP program, and we don't have access to be able to change it directly.

While powerful, there is one thing missing from this COM implementation due to an issue in the VFP COM service… It is not possible for a VFP COM object to pass an object reference to some other object that it controls or creates. It can only pass scalar values and arrays. This means that complex objects must first be coded into strings or other strategies must be employed to provide that functionality.

Building a Python COM Object and Accessing It from VFP

Once you have switched to doing most of your new development in Python, you'll find yourself creating many COM objects in Python so that your legacy VFP components can access the new capabilities. Fortunately, this is surprisingly easy.

In our working example, we first create a Python COM object that performs one simple task: it has one method that accepts a comma-separated set of text strings and returns an array with each element of the set in a separate element of the array. This will demonstrate how a Python COM object can return not only a simple scalar value, but also an array.

Note that this program to create a Python COM object is completely self-contained. It contains the object class definition, which is a standard Python class based on the "object" parent class and only has a few special features to provide information as to how the COM object should behave. The code in the __name__ == "__main__" block is all that is needed to register this as a COM server. Once it is run, it is not necessary for Python to be running in order for a COM client to access this server… Python is invoked behind the scenes.

The only external action required of the programmer is to separately create a unique GUID for this new COM server. This can be done by accessing any form of the Python interactive interpreter like IDLE, and using cut-and-paste to put the new value into the program as in

Figure 23:

Figure 23 - Creating a New Unique GUID

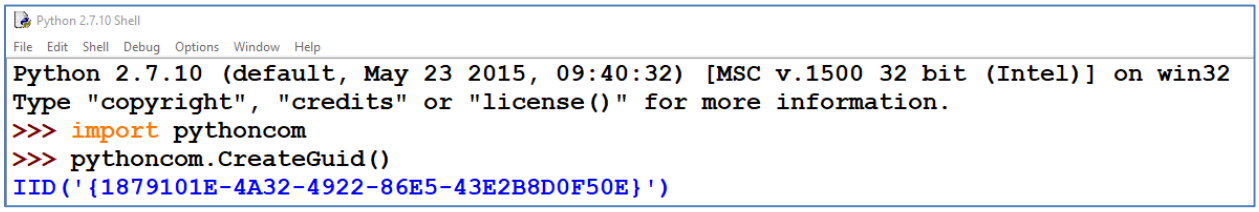


Figure 24 - Creating a Python COM Server

DemoCOMServer.py - c:\mpsspythonscripts\swfoxdemo\DemoCOMServer.py (2.7.10)

File Edit Format Run Options Window Help

```
"""
Demonstration of a Python COM server to be accessed by a VFP client.
"""
from __future__ import print_function
import win32com.server.register


class COMDemonstration(object):
    # Only methods and attributes declared as public here will be visible to client
    _public_methods_ = ['str2array']
    _public_attrs_ = ["errormessage", "version", "name"]
    _reg_progid_ = "democom.stringtool"  # This is the progID that VFP will use.
    _reg_clsid_ = "{9A25B1FC-7F54-4A47-9741-DEB829C38881}"
    # the clsid, a.k.a. GUID is created with the pythoncom.CreateGuid() method

    def __init__(self):
        self.name = "COM Demo"
        self.errormessage = ""
        self.version = 1.0

    def str2array(self, cTheString=""):
        # Simple function that breaks up a comma delimited string into a list in Python
        # and passes that result back to the client (VFP)
        if "," in cTheString:
            return cTheString.split(",")  # Returns a list with each word as an element
        else:
            self.errormessage = "zero length string passed"
            return None


if __name__ == "__main__":
    print("Registering Object")
    win32com.server.register.UseCommandLine(COMDemonstration)
    # This is required to be run at least once to register the COM object.
    print("DONE")
```

In

Figure 24 - Creating a Python COM Server demonstrating the creation of a Python COM object, the properties defined between the class definition line and the __init__() function are called class properties in Python as opposed to instance properties. We have seen instance properties in earlier examples, where properties are defined in the __init__() method. Those instance properties are only available to the specific instance itself, and if there are multiple instances of the same class in existence at one time, each has its own set of instance property values. In contrast, these class properties are values that are shared by ALL instances of the object. In this case, they are required by the COM Python

subsystem to define which methods and attributes of the object should be exposed via the COM interface. They also indicate what the ProgID will be (to be referenced in the VFP CREATEOBJECT() function call), and define the globally unique GUID previously set up by the programmer (See **Figure 23**). Win32com has many configuration properties that can be customized here, including indicating whether the COM process will be in-process or out-of-process. Mark Hammond's book *Python Programming on Win32* has the details.

Running this program on a stand-alone basis causes the __name__ == "__main__" block to execute. This allows the programmer to immediately register the object with the operating system. This step need be done only once on the target machine but doing it repeatedly doesn't hurt. Note that to deploy this COM object on a client or production machine, it will be necessary to set up a little batch process to run each of your COM server programs individually to register them with the local operating system.

***Helpful Hint*:** There are two important Python support files that are required to make Python COM servers work. In Python 3.7 they are named pythoncom37.dll and pywintypes37.dll. For these dlls to be accessible to VFP when acting as a COM client, they must be in the c:\windows\SysWOW64 directory. Some Python installers fail to copy them into this directory, resulting in non-Python COM clients failing to load Python COM objects. To fix this, copy these files from the Python site-packages directory. If your Python main directory is c:\Python37 this will be:

C:\Python37\Lib\site-packages\pywin32_system32

Note that this problem primarily affects in-process COM servers, but you should test it out and correct the problem anyway after installing Python.

Finally, you may wonder if you can have COM objects active running on more than one version of Python. In other words can you have a COM object coded for Python 2.7 and registered with Python 2.7 accessible to your VFP apps at the same time as a COM object coded for Python 3.7 and register with Python 3.7. The answer is "yes", provided that they have unique GUIDs and unique ProgID values.

Figure 25 - Output from Registering a Python COM Object

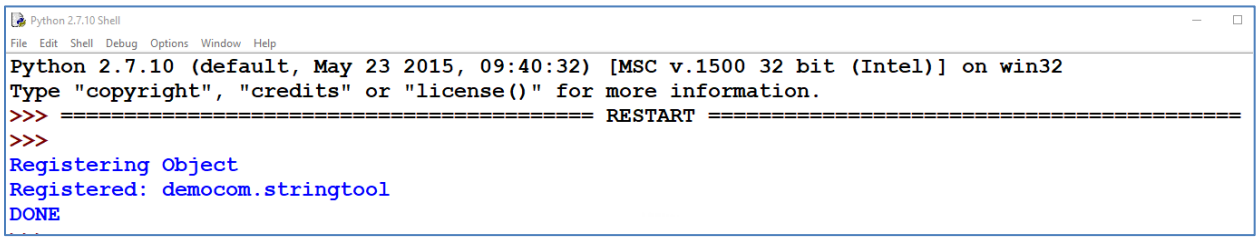
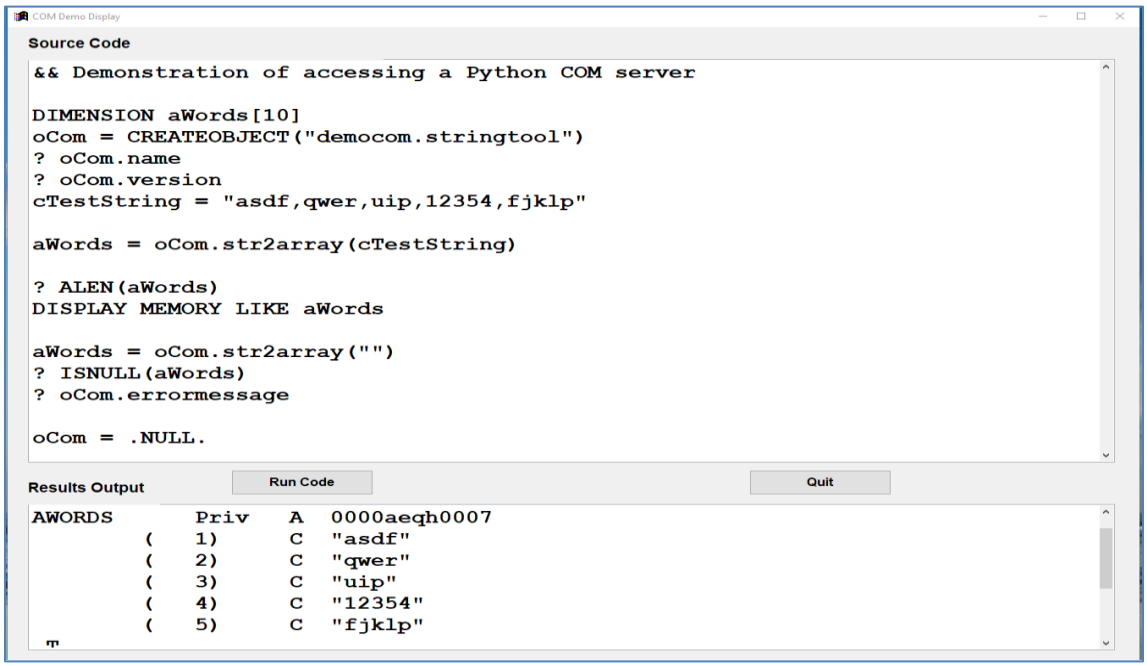


Figure 26 - Accessing a Python COM Server from VFP



Accessing this Python COM object is handled by VFP like any other COM object access. I wrote a simple VFP application to demonstrate this as shown in

Figure 26: comserverdemo.exe in the source code library for this White Paper and the related presentation.

In this case we know we will be receiving an array as the output of the Python COM server, so we first define the array with a DIMENSION statement. Strictly speaking, this should not be necessary, but we have found sometimes unpredictable results if we don't first create the VFP array and make it big enough to handle the data we are expecting back. If the array sent by Python is smaller, the VFP array will be truncated to match.

The source code windows shows the standard CREATEOBJECT() call to the ProgID we have created. We set up the comma separated value string and invoke the str2array() method on the COM object. Since VFP can't display the contents of an array using the ? command, we output the current memory to a file (SET ALTERNATE TO is defined outside the program code displayed to save screen space) and read it back in for display in the Results Output window, which shows a 5-element single dimensional array AWORDS with each of the substrings in an element.

Just so you know, it is also possible to return a Python dictionary in a COM call, which is translated in VFP into a 2 dimensional array with two columns, the first element being the key and the second being the associated value from the dictionary.

While this example has been necessarily simplistic, you can imagine much more complex scenarios where Python is returning JSON or HTML or other complex content for use in web pages or other services.

Python COM Objects as Object Factories

Unlike VFP, Python COM objects can have methods which return native Python objects which can be manipulated in VFP just like other COM objects. This allows the client to be given "smart" objects that already know about the state of the main COM object and can communicate with it when necessary. For example, in our applications, we have a very powerful configuration object that keeps track of the special requirements of each of our many customers, each of which has a separate database used for their information and many hundreds of customer-specific parameters. We have a Python services object that has an instance of the configurator at its heart, and hands out configurator-aware objects as-needed by VFP modules. Thus the VFP module only has to signal what customer it's working with to the shared configurator, and then any COM object the services object hands back already is aware of that customer's specific requirements.

A detailed explanation of how this works is beyond the scope of this White Paper (Mark Hammond's book covers it extensively), but here is some code from a live example to show you the highlights. First, in **Figure 27**, you see a large number of import statements bringing into the local scope the many service objects that we have defined and which VFP modules can make use of. Then in

Figure 28 there is a Python dictionary, which maps a text service name to a reference to the object class that provides the service.

This dictionary can work because class names and function names are considered "first class objects" in Python. In other words if you have defined a class like "class foo(object)",

then "foo", the class itself, is an object which can stored in another variable, made a part of a list, or (as in this case) a value in a dictionary. This capability, not found in VFP, is a huge differentiator between Python and VFP, as we mentioned earlier.

Figure 27 - Imports Supporting COM Object Factory

```
from win32com.server.util import wrap, unwrap  # important for passing object to VFP

from LoadOptWeb import FuelSurchargeCalculator
from LoadOptWeb import LoadOptRuntimeEstimator
from LoadOptWeb import ServiceIndicatorFlags
from LoadOptWeb.FiltersAndHelpers import ControllerDispatch
from LoadOptWeb.FreightDataManager import POUIDataManager
from LoadOptWeb.FreightRatingEngines import PoolRatingEngine
from LoadOptWeb.FreightRatingEngines import gxCfg
from LoadOptWeb.FreightRatingEngines.AccessorialCalculator import AccCalc
from LoadOptWeb.FreightRatingEngines.RatingServices import RatingService
from LoadOptWeb.OptimizationComponents import EngineManager
from LoadOptWeb.SystemDataManager import SystemManager
from MPSSCommon import EvosPythonRunner
from MPSSCommon import LOConfigurator
from MPSSCommon import MPSSBaseTools as mTools
from MPSSCommon.DBFXLStools2 import dbfxlTools
```

In this view of the program, we see the multiple imports that provide class object references for services to be provided to VFP via COM. We also have some functions to be imported from the win32com subsystem, which we'll use later.

Figure 28 - Mapping Service Names to Object Classes

```
self.xXref = {"CONFIG": LOConfigurator.LOConfig,
              "RATESERVICE": RatingService,
              "POOLRATER": PoolRatingEngine.PoolRater,
              "ENGMGR": EngineManager.SolverEngineManager,
              "FUELCALC": FuelSurchargeCalculator.FuelCalculator,
              "POUIMGR": POUIDataManager.POUIdata,
              "PYRUNNER": EvosPythonRunner.PythonRunner,
              "TIMEEST": LoadOptRuntimeEstimator.LOEstimator,
              "SERVICE": ServiceIndicatorFlags.SvcFlagMgr,
              "SYSMGR": SystemManager.ComSysAdmin,
              "DBFEXCEL": dbfxlTools,
              "ACCESSMGR": AccCalc,
              "CONTROLLER": ControllerDispatch.ControlDispatch}
```

VFP modules will be asking for specific services by name. For example, a VFP module might ask for the Truckload Accessorial Calculator (AccCalc) by its name here "ACCESSMGR" in

Figure 28. In each case the value associated with the service name key in the dictionary is not an actual instantiation of the object, but simply the class name itself, which is actually an object too.

Each of these objects have class properties which define what properties and methods to

expose through COM, so when the COM factory serves them up to VFP, the COM mechanism knows how to expose the services to the VFP client.

In the program excerpted here, a service object is only built once, and a dictionary keeps track of it and shares it with the VFP client as needed. In the code below in **Figure 29**, you see what happens when the object factory gets a request for a service object that has not previously been requested. Since this object factory is used both by VFP via COM and by Python as a standard Python object, we need to know whether the requester is a COM client or not, thus the variable bCOMservice that indicates the requester is a COM object.

Figure 29 - COM Object Factory Creating and Returning a Python Object

```
    # We haven't built one of these already, so we create an actual instance
    # now.
    oTemp = oObject(oVFP=self.oVFP, oCfg=self.oCfg, bIsCOM=bCOMservice)
    oTemp.oServiceFactory = self
    # we test for is this a COM call?
    if bCOMservice:
        # wrap is provided by the Win32COM extension to make the object
        # usable to pass to the COM client.
        oReturn = wrap(oTemp)
    else:
        # not COM, so we just return the basic object reference.
        oReturn = oTemp
```

In this last snippet of code, the oObject variable has been set equal to the object class specified in the dictionary we saw above in

Figure 28. Only now are we able to create an actual instance of the required object. Each of these service objects needs to be able to access the service factory object to get environment information, so we assign "self" the object reference for the service factory itself, to the newly created object's oServiceFactory property. Note that we are passing environmental objects to the new class when we create it. We have established a common calling pattern for all of these objects, so they all expect the same set of parameters, regardless of what kind of object is being created… at this point the factory code doesn't care, the object creation code is the same for all.

The final step to make this object suitable for handing off to VFP via COM is to use the wrap() function (see the import block above in **Figure 27**) which creates a COM-enabled version of the standard Python object. It is this "wrapped" object that is then passed to VFP, which handles it like any other COM object.

Handling Exceptions (Errors) in Python COM

Python does not support a global error handler as does Visual FoxPro. There is no "ON ERROR" command that can pass control to some mechanism to record the error condition and return control to some stable part of the application to allow it to keep running. "Best Practice" in Python is to use try…except blocks liberally to catch error conditions and handle them as they occur. That is nice in theory, but in very large applications, unexpected situations come up ("stuff happens"), error conditions are raised in unexpected places, and our applications have to keep on going despite that.

This is especially problematic in COM where there is no user interface, not even some kind of console output, and if a Python COM server object runs into an error, the amount of information reported on the VFP client side is very limited.

From the VFP side you should wrap any mission critical calls to the Python COM objects in Try…Catch…EndTry blocks to make sure you catch unexpected issues and provide code to handle them. That at least will keep your mission critical application from going down in spite of a glitch in the COM object. However, without a comprehensive error report from the COM object, debugging may be hit and miss even if your COM object code has a __name__=="__main__" section for testing and debugging, as there may be special circumstances that are not tested for when running stand alone.

To deal with COM module error trapping and reporting, we have constructed a special class that wraps the functioning of all its methods in error trapping code and then compiles an error output file with details on all object properties and current variables throughout the calling hierarchy. The actual COM object created from this class uses multiple inheritance – first from the error trapping wrapper, and then from the actual Python object class being exposed via COM. To make this work, Python features were required that are well beyond the scope of this white paper, but I am happy to share this code and how to use it for those who feel the need for this kind of error trapping.

Python and Databases, DBF and Others

In its basic distribution, Python makes very limited provision for what is sometimes referred to as "persistence", i.e. the ability to store information for future reference. There is no native access to any kind of database. The only mechanism for storing data is a process called "pickling" which involves turning Python native objects like dictionaries or more complex objects into text which can be saved into a file on disk. This is much like the rarely used VFP SAVE TO and RESTORE FROM commands that dump memory out to disk files or memo fields and retrieve them later.

SQL Database Support

But, not to worry, there is extensive database access support in the libraries found in PyPI. There are native bindings for nearly all major database products including:

- SQLite
- PostgreSQL
- MySQL
- MS SQLServer
- Firebird
- Generic ODBC
- And many others

These native bindings have the speed advantage of working directly with the low-level drivers to access the server directly. Their disadvantage is that if you are working with multiple database environments, you may have to program multiple incompatible dialects of SQL to get to your data. Python solves this problem with SQLAlchemy, which provides a wrapper for all the databases mentioned above and many more to support a common SQL

syntax and common calling conventions regardless of what underlying DB you are accessing.

No-SQL Database Support

If you have moved into the No-SQL world with products like MongoDB, as we have, then you have help there too. There is a module in PyPI called PyMongo which enables convenient access to MongoDB, once again making use of the native bindings. There is even a module called PQL, which provides translations of Python logical expressions into the peculiar and syntactically complex JSON-based query language understood by MongoDB.

What about DBF Files?

Until we began working with Python, the answer to this question was not very positive. There is one module in PyPI called simply DBF, that is written in pure Python and can access most DBF tables, but it is excruciatingly slow and is limited in the types of fields it can access. Basically, it is only useful for simple one-time transfers of basic DBF tables to other formats supported by Python, but not for any kind of production applications.

Our team explored a number of alternatives before finally deciding to create our own solution.

OLE-DB Access to VFP Tables

Since OLE-DB is a COM-compatible protocol, we tried this out early on. The advantage is that the VFP OLE-DB driver can access tables in VFP databases and recognizes standard VFP field types. The disadvantage is that it is a Microsoft product and hasn't been updated since 2008. This example code in Figure 30 demonstrates how to use OLE-DB to access a VFP table:

Figure 30 - Accessing a VFP Table with Python and OLE-DB

```
from win32com.client import Dispatch

oConn = Dispatch('adodb.connection') # Standard COM Client approach.
oRS = Dispatch('adodb.recordset')

cConnStrng = "Provider=vfpoledb;Data Source=e:\\loadbuilder2\\appdbfs\\clients.dbf"
xresult = oConn.Open(cConnStrng)  # One of a couple approaches to this.
print(xresult)

oRS.Open("SELECT * FROM clients WHERE client_num==11690", oConn, 2, 3, 1)
# Standard VFP-style SQL
print('Current value:', oRS.Fields("clientname").Value)
print('Record Count:', oRS.RecordCount)
nFldCnt = 0
for xf in oRS.Fields:
    cOutput = "%10s - %s"
    print(cOutput % (xf.Name, str(xf.Value)))
    nFldCnt += 1
    if nFldCnt > 10:
        break
oRS.Close()
oConn.Close()
```

Figure 31 - Output from VFP OLE-DB Example

```
Python 2.7.10 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================================== RESTART ================================
>>>
None
Current value: AFFINITY VAULTS
Record Count: -1
client_num - 11690
clientname - AFFINITY VAULTS
   datadir - AMJCAMP
   dirlock - False
  tms_code - AMJMISONT
filter_prg -
    addltl - True
 dfltclass - 85
  dfltdisc - 50
 ltloption - XXXXXXXX
  loctable - True
```

While this is a successful demonstration as shown by results in **Figure 31**, the OLE-DB approach here is cumbersome as it requires SQL code to be constructed in string variables and data to be managed in record sets. Known issues with OLE-DB access to VFP tables may also need to be considered in your application.

Advantage Database Server

The Advantage Database Server is known for its ability to access DBF tables via a client-server model. The product is now sold by SAP, which seems to have very little interest in selling it. There is a native Python binding for the Advantage Server downloadable from their website, but it has never been updated to Python 3.x compatibility. In our tests, the Advantage Server was highly compatible with VFP tables, but was disappointingly slow

compared to native VFP table access. Given the lack of 3.x support, I can't recommend Advantage as a path to using DBF tables with Python.

CodeBase and a DIY Solution

We actually had committed to using Python before we found a fully satisfactory solution for the DBF sharing issue. But early on we tried the CodeBase software product for accessing VFP tables, and had found it complicated to use, but workable. Accordingly, we acquired a full source-code license for the Windows 32-bit version of the product to try it out. In our initial tests, we found its performance was close to that of native VFP and markedly faster than other database solutions, including PostgrSQL, MySQL, etc.

Unfortunately, CodeBase, as a C-style DLL works at the lowest possible level for accessing tables. By itself, it would never have the convenience and expressiveness of VFP data table handling features. So we decided to build a system of Python components around the CodeBase underlying DLL to give ourselves a VFP-like development environment that would be able to share VFP tables concurrently with our VFP applications.

We completed most of our work by 2014, and then in 2015, Sequiter Software announced the discontinuance of the product! We elected to continue on with it, as we not only were highly satisfied with what we had built, but as we had the source code for the product and found we could build the DLL file using MS Visual Studio from the source code, we were confident in continuing with the product.

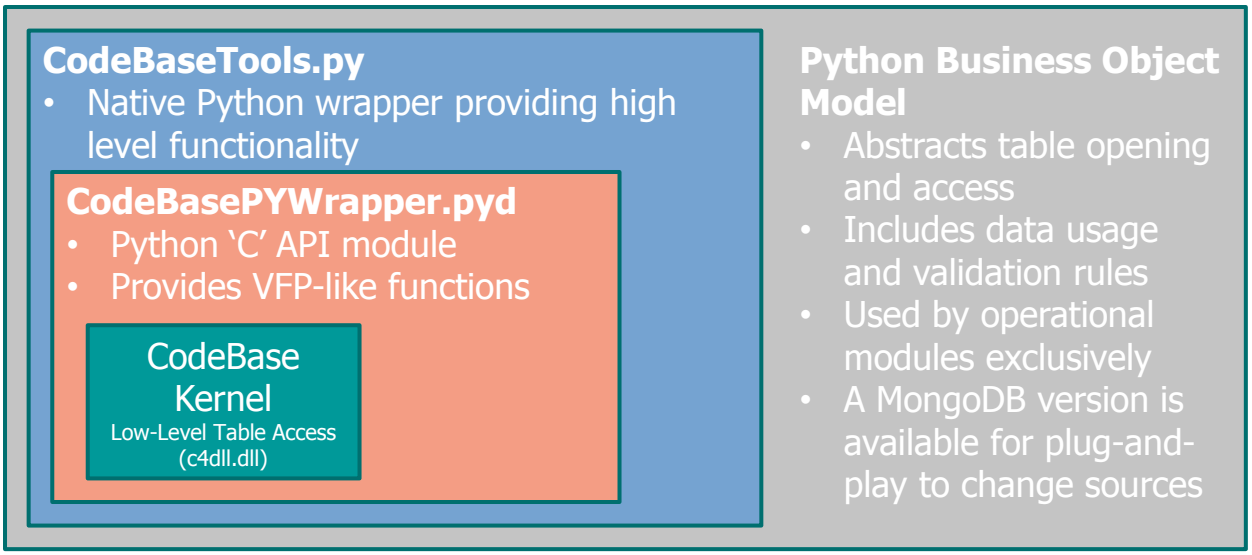
Fortunately, in the last month, I have obtained an agreement with Sequiter, Inc., now known as LegalDepot.com, to release the source code and compiled DLL into Open Source. By the time you read this, the complete product installation package should be freely available for download from GitHub! In that same spirit, we are including all the source code for our "Python Code Base Tools" in our distribution materials for the SW Fox Conference and hope to have it all available for download initially from GitHub in early 2019, and soon thereafter from PyPI.

Python CodeBase Tools

Our key objectives in developing these tools were 1) to mask the complexity of the raw CodeBase technology, and 2) to provide VFP-like functions that would work in familiar ways. For example, the 'USE' command we wanted to emulate in Python. To accomplish that we developed a layered approach that builds on the underlying DLL, provides a high-performance Python-based wrapper written using the Python C-API, and finally a user-friendly object allowing both object oriented and VFP procedural style access to VFP data tables. Finally, we built a business object class that could be adapted to connect to either a VFP table via the Tools or to a MongoDB collection via PyMongo. The overall framework is illustrated in

Figure 32.

Figure 32 - Python CodeBaseTools Framework



While these tools are serving us well, they are not a complete solution for every application. There are things that they do well, other places there are gaps. First the good news:

- Common VFP Constructs have been preserved with equivalent functions:
  - SELECT, USE, SCATTER, GATHER, ZAP for example
  - Navigational table access is the basic mode:
    - SEEK(), GOTO, RECNO(), NEXT, PREV, SKIP, etc.
    - LOCATE FOR, SCAN FOR
    - Information about tables and indexes is also available with equivalents for: ATAGS(), AFIELDS(), RECCOUNT(), ALIAS(), DBF(), DELETED(),
  - Upon USE, each open table has an ALIAS, and there is the concept of the "current selected table". Fields in tables not currently selected can be referenced with dot notation using the alias name as the prefix.
- Essential table handling functions are provided:
  - REINDEX
  - PACK
  - ZAP
  - DELETE FOR
  - SET DELETED
  - USE (with clauses that support exclusive, read-only, and shared)
  - COPY TO Excel (including memo fields and using XLSX format)
  - APPEND FROM Excel (including memo fields and using XLSX format)
  - COPY TO csv, tab separated, other formats
  - CURSORTOXML and XMLTOCURSOR
  - APPEND FROM csv, tab separated, other formats
  - Temporary indexes can be created, used, auto updated, and then automatically deleted when the table is closed.
- Record, table and table header locking are compatible with VFP locking mechanisms.
- For DBF tables that are **not** to be shared with VFP, Python CodeBaseTools support higher limits on many elements.

- Max fields per table: 2046
- Max width of a character field: 65,517 bytes
- Max table size (with "large mode" turned on): 8GB
- Max record size: 1GB

However, on the down side, some basic operations can only be performed on free tables, not tables contained in a VFP database container, and DBC functionality is not supported:

- INDEX ON, DELETE TAG, ALTER STRUCTURE, etc., since the CodeBaseTools don't recognize the existence of the DBC, and won't update the DBC with the new structure information. These capabilities work just fine on "free" tables, however.
- Long field names are not supported
- Triggers and database events are not supported
- Auto incrementing fields are not supported

Since the VFP DBC is actually just another VFP DBF table by different name, it should be relatively straightforward to add DBC recognition. (A job for the Open Source community?) Other issues may require more effort. The basic CodeBase engine does not support SQL queries. Sequiter did make a module that supported SQL, but that has not yet been released for Open Source, and Python CodeBase Tools doesn't make use of it. Another future enhancement?

To answer a question that many xBase programmers may ask, yes, the underlying CodeBase engine supports VFP, Clipper, and dBase IV table types, and index formats. Since we had no use for any DBF tables other than those we create and use under VFP, the Python CodeBase Tools do not have a mechanism for switching to the alternate DBF types. This multi-format option could be added relatively quickly, should it be asked for.

To illustrate the use of the Python CodeBase Tools, we provide some examples…

Figure 33 - Example DBF Table Create

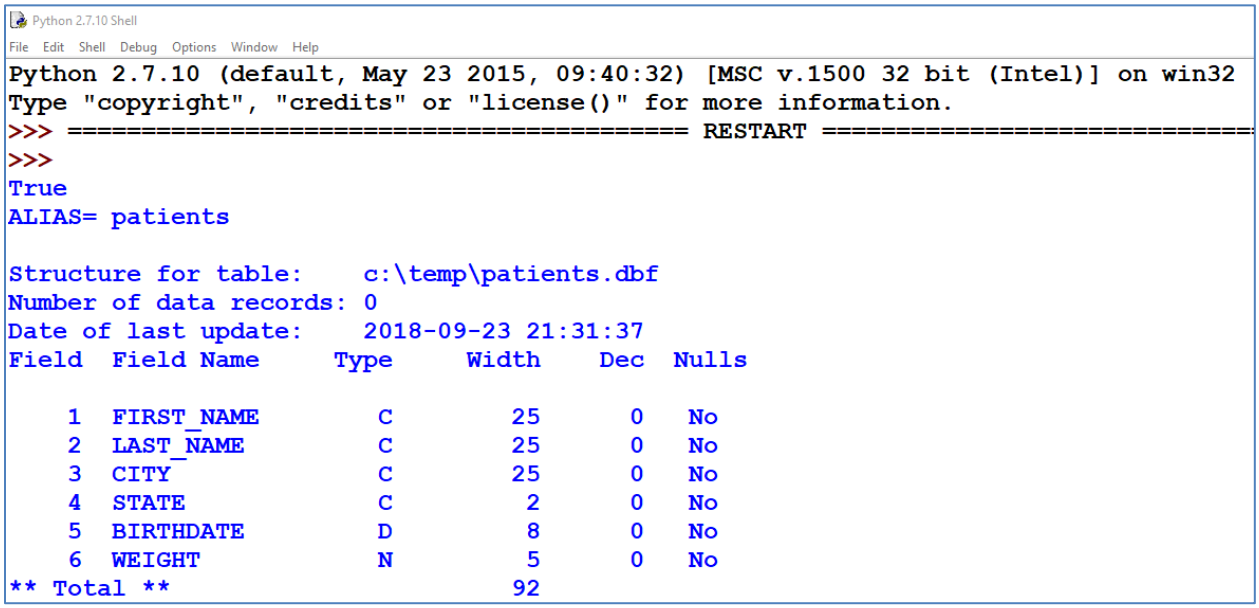
```
from __future__ import print_function
from MPSSCommon.CodeBaseTools import cbTools
oCB = cbTools() # Optionally create version that supports > 2GB tables

xStru = list()
xStru.append(oCB.newfield("FIRST_NAME", "C", 25, 0, False))
xStru.append(oCB.newfield("LAST_NAME", "C", 25, 0, False))
xStru.append(oCB.newfield("CITY", "C", 25, 0, False))
xStru.append(oCB.newfield("STATE", "C", 2, 0, False))
xStru.append(oCB.newfield("BIRTHDATE", "D", 0, 0, False))
xStru.append(oCB.newfield("WEIGHT", "N", 5, 0, False))

bResult = oCB.createtable("c:\\temp\\patients.dbf", xStru)
print(bResult)
print("ALIAS=", oCB.alias())
oCB.dispstru()
oCB.closetable(oCB.alias())
oCB = None
```

The code in **Figure 33** illustrates the analog to the VFP process which can create a table from an array of field information. In this case, the source object is a Python list, and each

element has a field descriptor created by the newfield() method. It is also possible to create new DBF tables using plain text strings defining the fields. Incidentally, the list of fields returned by afields(), can be used directly for creating another table.

As seen in **Figure 34** the alias() method tells us what the "ALIAS" is for the table – as in VFP, it is the base name of the table (following the same naming conventions used in VFP). The dispstru() method performs the same job as the VFP DISPLAY STRUCTURE command.

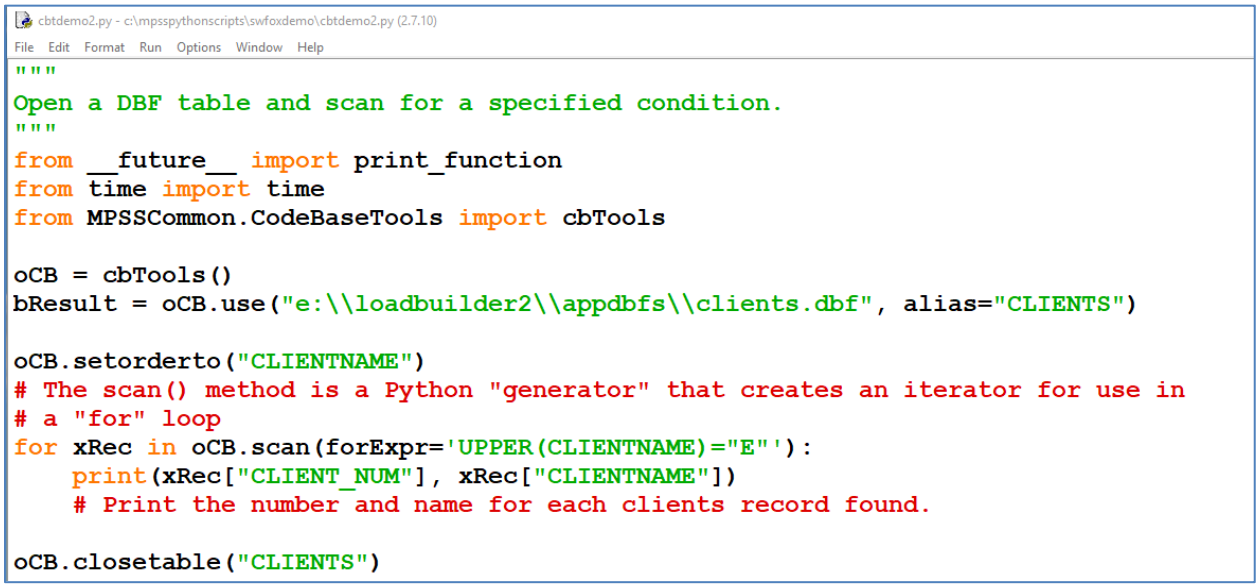
Figure 34 - Output from Table Create Demo



```
Python 2.7.10 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================================== RESTART ================================
>>>
True
ALIAS= patients

Structure for table:     c:\temp\patients.dbf
Number of data records:  0
Date of last update:     2018-09-23 21:31:37
Field  Field Name      Type      Width    Dec  Nulls

    1  FIRST_NAME        C         25      0   No
    2  LAST_NAME         C         25      0   No
    3  CITY              C         25      0   No
    4  STATE             C          2      0   No
    5  BIRTHDATE         D          8      0   No
    6  WEIGHT            N          5      0   No
** Total **                        92
```

Sequential scanning through a table, either in its entirety or based on some record selection criteria is a very common pattern. This capability is illustrated here in **Figure 35**

Figure 35 - Opening and Scanning a Table



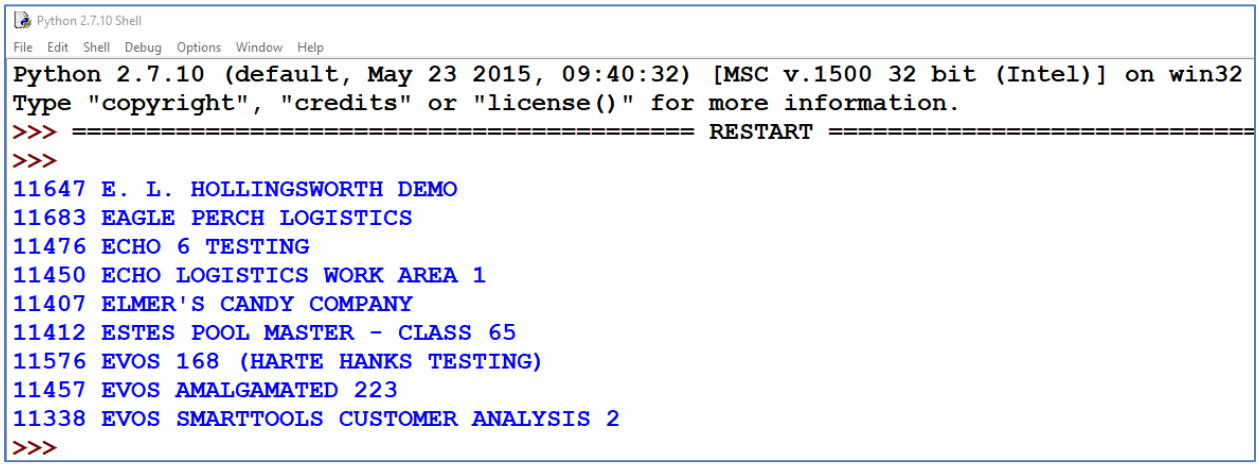
```
cbtdemo2.py - c:\mpsspythonscripts\swfoxdemo\cbtdemo2.py (2.7.10)
File  Edit  Format  Run  Options  Window  Help
"""
Open a DBF table and scan for a specified condition.
"""
from __future__ import print_function
from time import time
from MPSSCommon.CodeBaseTools import cbTools

oCB = cbTools()
bResult = oCB.use("e:\\loadbuilder2\\appdbfs\\clients.dbf", alias="CLIENTS")

oCB.setorderto("CLIENTNAME")
# The scan() method is a Python "generator" that creates an iterator for use in
# a "for" loop
for xRec in oCB.scan(forExpr='UPPER(CLIENTNAME)="E"'):
    print(xRec["CLIENT_NUM"], xRec["CLIENTNAME"])
    # Print the number and name for each clients record found.

oCB.closetable("CLIENTS")
```

Here the code is scanning a list of clients in order by the tag CLIENTNAME and looking only

for records where the name starts with "E". This uses the familiar "for…:" framework because the Python CodeBase Tools implements scan() as what is known as a "generator". A generator acts like an iterator but is able to return repeatedly to its function code to increment the record pointer and then return information on the next record, all transparently to the calling code.

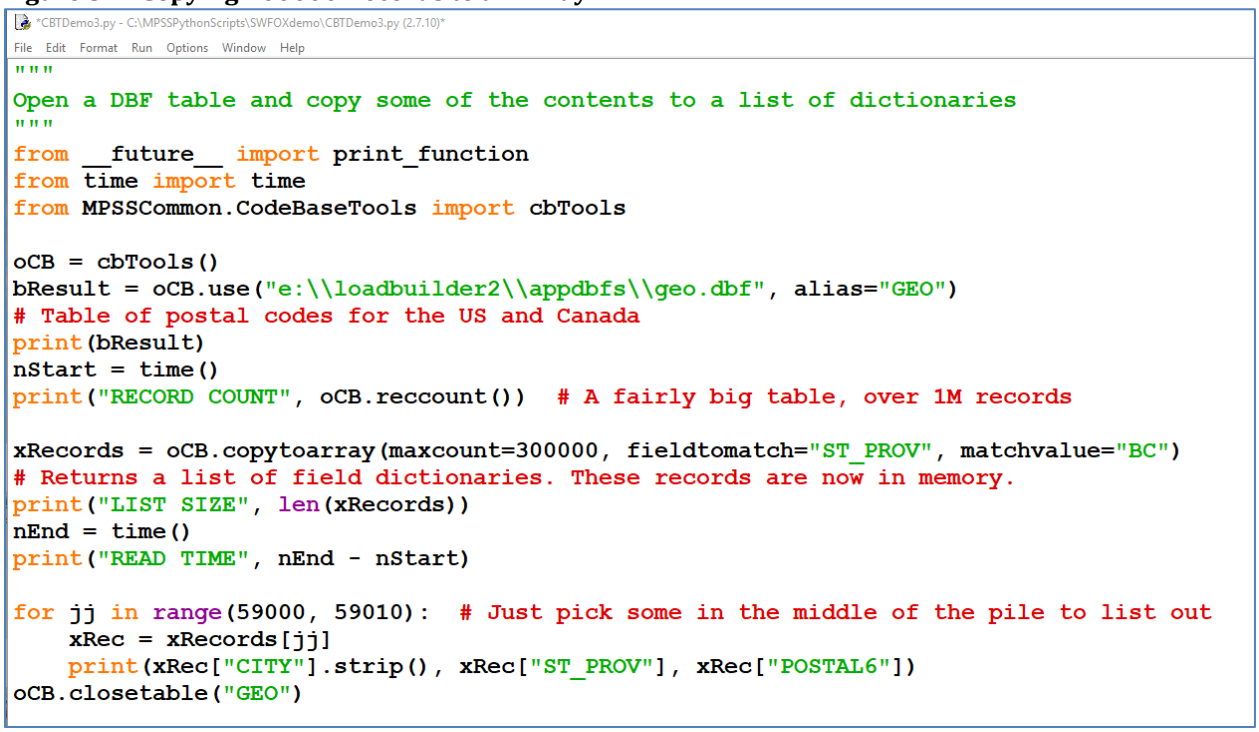
Note that the xRec value returned with each iteration through the scan() is a field data dictionary with the keys being the field names, with their values from the table. For the sake of speed, you can restrict the fields returned or even return no fields to let the code in the scan loop do its own reading of the current record data. Output from this example is in **Figure 36**.

Figure 36 - Output for Scan Demo



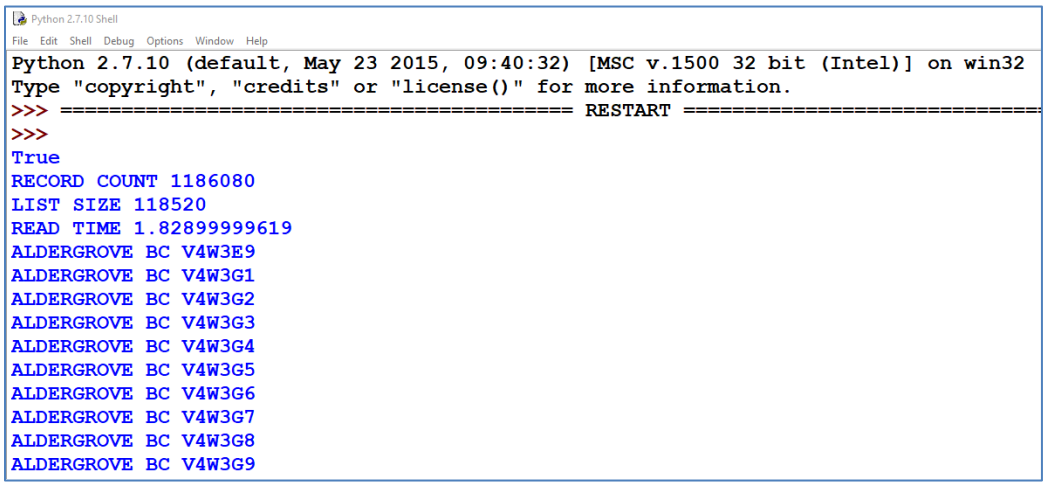
Now let's try accessing a fairly large table and reading a significant chunk of it into memory as a list of dictionaries. At first this sounds a lot like the VFP SQL SELECT into an array as a destination. That is deceptive, since to use the resulting 2-dimensional VFP array, you need to know what fields were extracted and in what order they appeared either in the SQL statement or natively in the table. To address a particular field in the resulting array, then you need to know its index in that array. Instead, what we'll be doing with Python is reading a large number of records into memory, but we'll store them in a list where each element of the list is a Python dictionary keyed by the field names… no need to know what order the fields appeared in, just refer to them by their names… The code to do this is shown in **Figure** 37.

Figure 37 - Copying 100000 Records to an Array



```
*CBTDemo3.py - C:\MPSSPythonScripts\SWFOXdemo\CBTDemo3.py (2.7.10)*
File  Edit  Format  Run  Options  Window  Help
"""
Open a DBF table and copy some of the contents to a list of dictionaries
"""
from __future__ import print_function
from time import time
from MPSSCommon.CodeBaseTools import cbTools

oCB = cbTools()
bResult = oCB.use("e:\\loadbuilder2\\appdbfs\\geo.dbf", alias="GEO")
# Table of postal codes for the US and Canada
print(bResult)
nStart = time()
print("RECORD COUNT", oCB.reccount())  # A fairly big table, over 1M records

xRecords = oCB.copytoarray(maxcount=300000, fieldtomatch="ST_PROV", matchvalue="BC")
# Returns a list of field dictionaries. These records are now in memory.
print("LIST SIZE", len(xRecords))
nEnd = time()
print("READ TIME", nEnd - nStart)

for jj in range(59000, 59010):  # Just pick some in the middle of the pile to list out
    xRec = xRecords[jj]
    print(xRec["CITY"].strip(), xRec["ST_PROV"], xRec["POSTAL6"])
oCB.closetable("GEO")
```

Figure 38 - Output from Copying 100000 Records to an Array

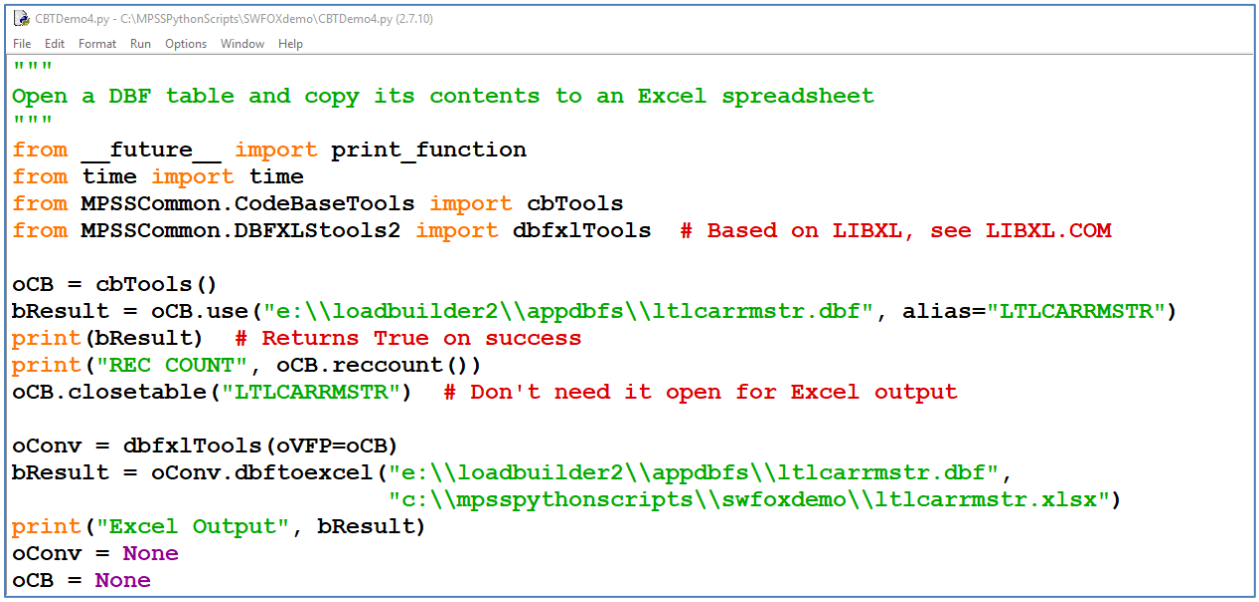


```
Python 2.7.10 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.10 (default, May 23 2015, 09:40:32) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========================================== RESTART ================================
>>>
True
RECORD COUNT 1186080
LIST SIZE 118520
READ TIME 1.82899999619
ALDERGROVE BC V4W3E9
ALDERGROVE BC V4W3G1
ALDERGROVE BC V4W3G2
ALDERGROVE BC V4W3G3
ALDERGROVE BC V4W3G4
ALDERGROVE BC V4W3G5
ALDERGROVE BC V4W3G6
ALDERGROVE BC V4W3G7
ALDERGROVE BC V4W3G8
ALDERGROVE BC V4W3G9
```

Figure 38 shows that Python CodeBase Tools took just under 2 seconds to read 118,520 records into memory as a list of dictionaries. Not up to VFP speeds, you say? Well possibly. A simple SQL select into an array takes about 1/10th that long in VFP. But remember that each element of that big list in Python is a name-addressable dictionary, as you can see in the little loop where we listed some of the Aldergrove, BC, zips. The VFP equivalent of that would be to create a large array, then SCAN through the table looking for the BC province records, scattering each of them to an object (to NAME) and storing that record object into the VFP array. That gives you the same name-addressable result in every element of the array. In case you are wondering how long VFP would take to do that… I tried it, and on the same machine that ran these examples, VFP required a **full 7 minutes** to accomplish that task of storing name addressable objects into the array!

We mentioned that we are able to output and append Excel files in the modern XLSX format. This is done using a proprietary tool called LIBXL, which can be purchased very reasonably at LIBXL.COM. The beauty of this solution is that it is many times faster than using Excel itself via a COM interface, and it avoids any licensing peculiarities Microsoft may throw into your path. An example of Excel output is shown in **Figure 39.**

Figure 39 - Code to Output a DBF as an Excel Spreadsheet



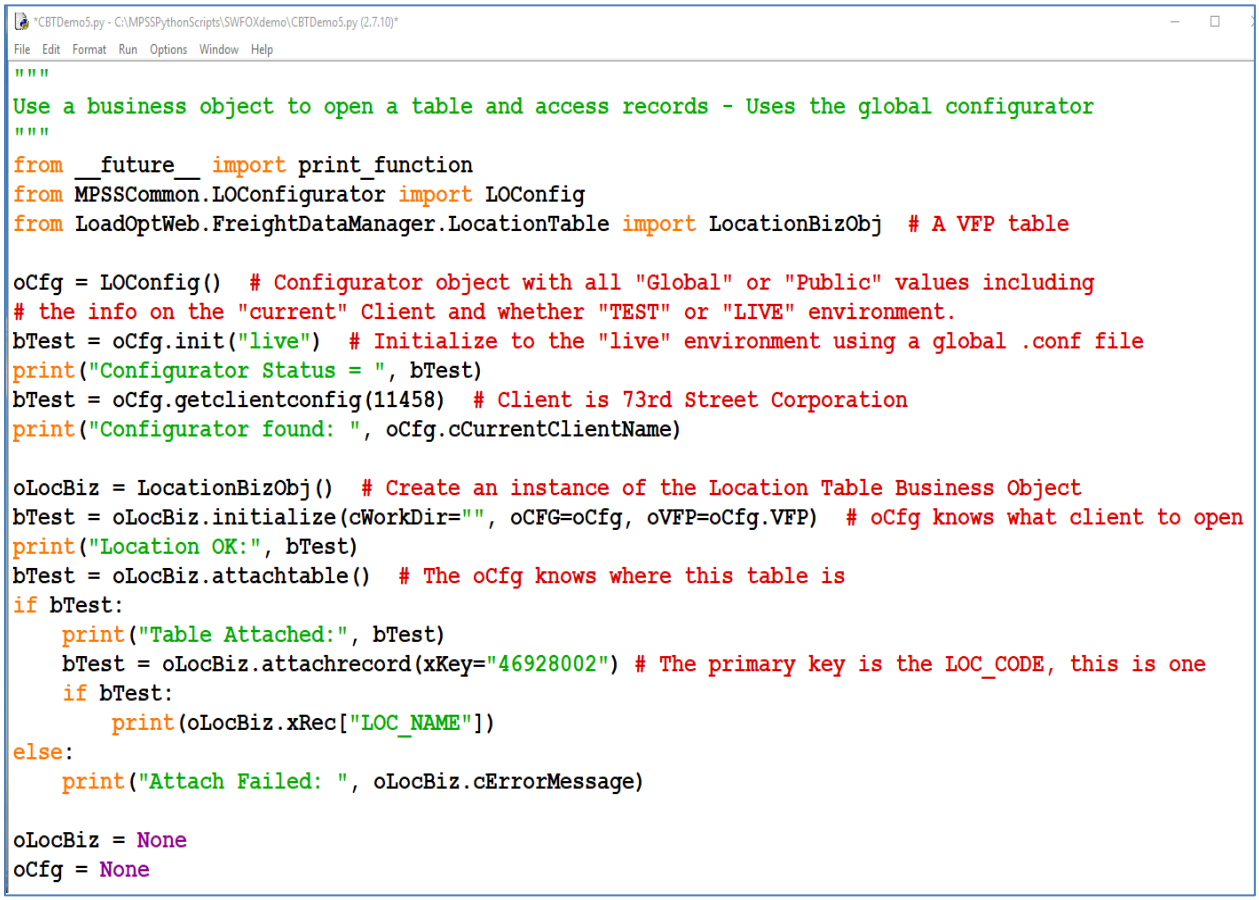
Our module that performs the output and input of Excel spreadsheets with DBF tables is dbfxlTools. We intend to make this module available in Open Source, and I have included the code in our Conference materials, BUT, since the LIBXL product is proprietary, we can't make that available. You'll need to buy your own copy and apply your license code to run it in your own applications with Python.

In the code above, we open the LTLCARRMSTR.DBF table and report the number of records. We really didn't need to do that, as the dbftoexcel() method will take care of opening the DBF table you specify. I haven't shown the results of this code, as it just shows that the converter returned a '1' for OK. The resulting ltlcarrmstr.xlsx is found in the Conference Materials.

Finally, I talked about creating a Business Object framework that allows us to open DBF or MongoDB tables using essentially the same logic throughout our Python applications (we don't attempt to open MongoDB tables in VFP, as we can easily do it without ODBC directly in Python). An example of using our Business Object framework is shown in

Figure 40.

Figure 40 - Using a Python Data Business Object



```
*CBTDemo5.py - C:\MPSSPythonScripts\SWFOXdemo\CBTDemo5.py (2.7.10)*
File  Edit  Format  Run  Options  Window  Help
"""
Use a business object to open a table and access records - Uses the global configurator
"""
from __future__ import print_function
from MPSSCommon.LOConfigurator import LOConfig
from LoadOptWeb.FreightDataManager.LocationTable import LocationBizObj  # A VFP table

oCfg = LOConfig()  # Configurator object with all "Global" or "Public" values including
# the info on the "current" Client and whether "TEST" or "LIVE" environment.
bTest = oCfg.init("live")  # Initialize to the "live" environment using a global .conf file
print("Configurator Status = ", bTest)
bTest = oCfg.getclientconfig(11458)  # Client is 73rd Street Corporation
print("Configurator found: ", oCfg.cCurrentClientName)

oLocBiz = LocationBizObj()  # Create an instance of the Location Table Business Object
bTest = oLocBiz.initialize(cWorkDir="", oCFG=oCfg, oVFP=oCfg.VFP)  # oCfg knows what client to open
print("Location OK:", bTest)
bTest = oLocBiz.attachtable()  # The oCfg knows where this table is
if bTest:
    print("Table Attached:", bTest)
    bTest = oLocBiz.attachrecord(xKey="46928002") # The primary key is the LOC_CODE, this is one
    if bTest:
        print(oLocBiz.xRec["LOC_NAME"])
else:
    print("Attach Failed: ", oLocBiz.cErrorMessage)

oLocBiz = None
oCfg = None
```

The power here comes from using a master configuration object that knows all about our hundreds of customer databases (most of our customers require by contract that their data not be comingled with other customer data), where they are, and what configuration parameters govern the way our software works with the customer. The first step is to create an instance of the LOConfig object (LO refers to “LoadOpt”, the old name for our PlanTools product) which knows where to look to find all of its application directories depending on whether it is running in our “live” or our “test” environment.

Then the Location Table business object is created. The Location table stores shipper and consignee location details, and every one of our customers has one. It happens to be a VFP table in this case. The Location Biz Obj is passed an object reference to the Config object, and as a result knows exactly where the table is that it needs, and it knows the rules for

managing that table that are specific to the customer. This allows the table to be attached (with use() for a VFP table or an openCollection() in MongoDB).

Once the table is attached, there are several ways to get at records in the table, one of which is the attach() method which can search by record number, primary key, or other conditions. In this case the named parameter "xKey" requires the primary key of the record. If the attach is successful, the xRec property of the Biz Obj is a dictionary of fields and field values for the attached record.

Like all classic data business objects, our BizObjects have table validation rules and multiple methods that run before and after saves, creation, and deletes among other things.

Open To-Dos for CodeBase Python Tools

We continue to add features from time to time to the CodeBaseTools components, but there are several tasks that call for time commitments we haven't been able to budget. I hope to generate enough enthusiasm in the VFP/Python communities to enhance these tools still further with the help of the community. Some possible projects:

- Implement CodeBase's transaction tracking feature. It works differently from transactions in VFP, but doesn't require the tables be part of a database container.
- Implement CodeBase's SQL query tool
- Change the 'C' engine to recognize long field names, and other characteristics of the VFP database container, including being able to create, alter structure, and add/alter indexes of those tables.
- Add recognition for VFP-style auto-incrementing fields. In its last version CodeBase introduced auto-incrementing integer fields, but didn't make them compatible with their VFP counterparts.
- Consider implementing triggers and database events with Python code in addition to or instead of VFP code.

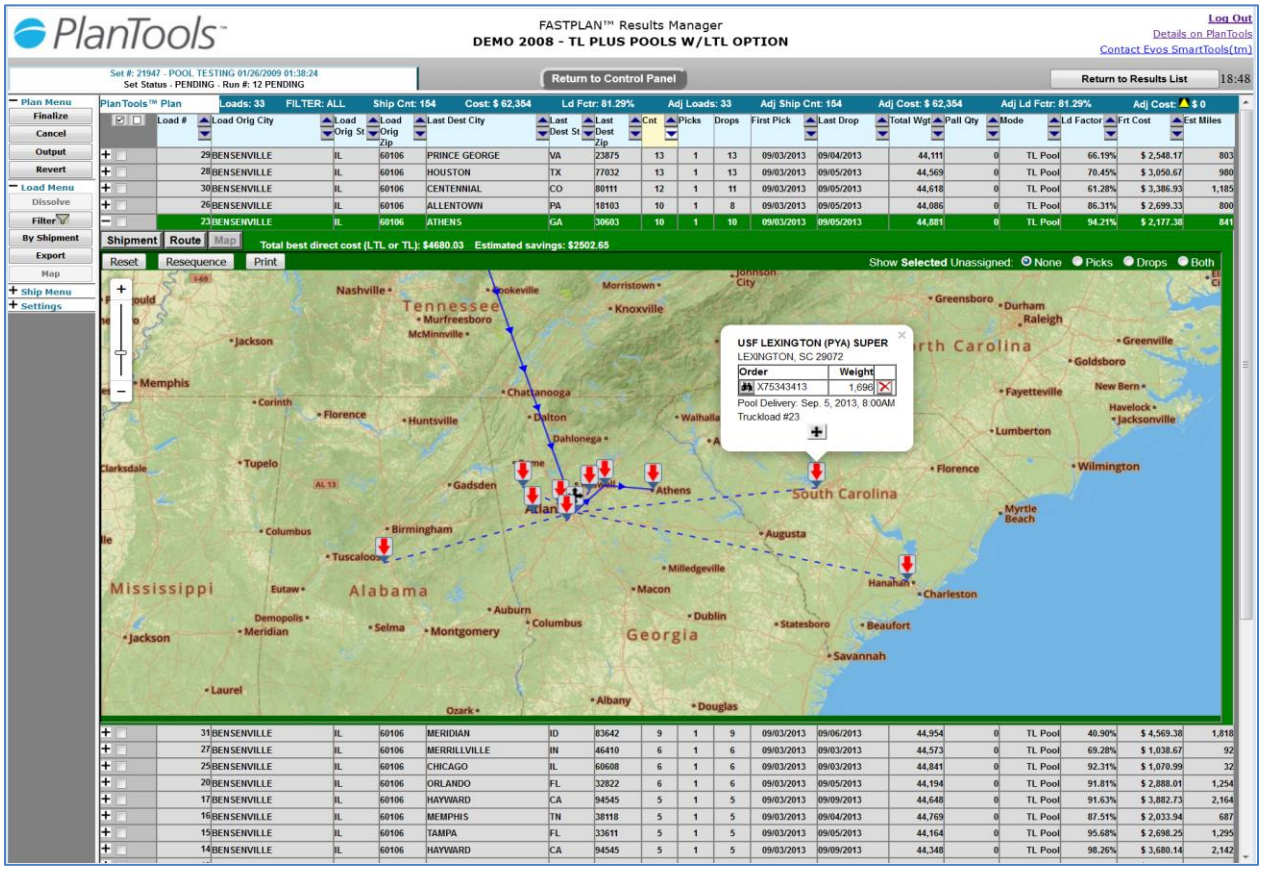
Building a Web Application with WestWind™, VFP, and Python

M-P System Services, Inc., my company, and our affiliate Evos SmartTools™ had adopted Rick Strahl's WestWind web tools very early in our move to web delivery of our product. When we decided to turn to Python as a strategic tool for building new capabilities, we had to come up with ways to integrate Python modules with WestWind to make the user experience completely seamless between VFP and Python-generated pages and page components.

Now, our customers navigate from page to page in our applications, oblivious as to how the pages are generated, be they legacy VFP pages or new Python pages. We use VFP or Python to generate table displays for VFP tables, VFP or Python to generate edit screens for individual records, and Python to generate table displays for MongoDB tables. We also use Python to generate live PDF and Excel output that instantly reflects users' most recent

changes in their data entries and setups.

Figure 41 - Example PlanTools(tm) Page with VFP and Python Elements



In this example in

Figure 41 above you see our FastPlan™ interface where our customers manage the multi-shipment loads built by our proprietary optimizer. The header frame is generated by a

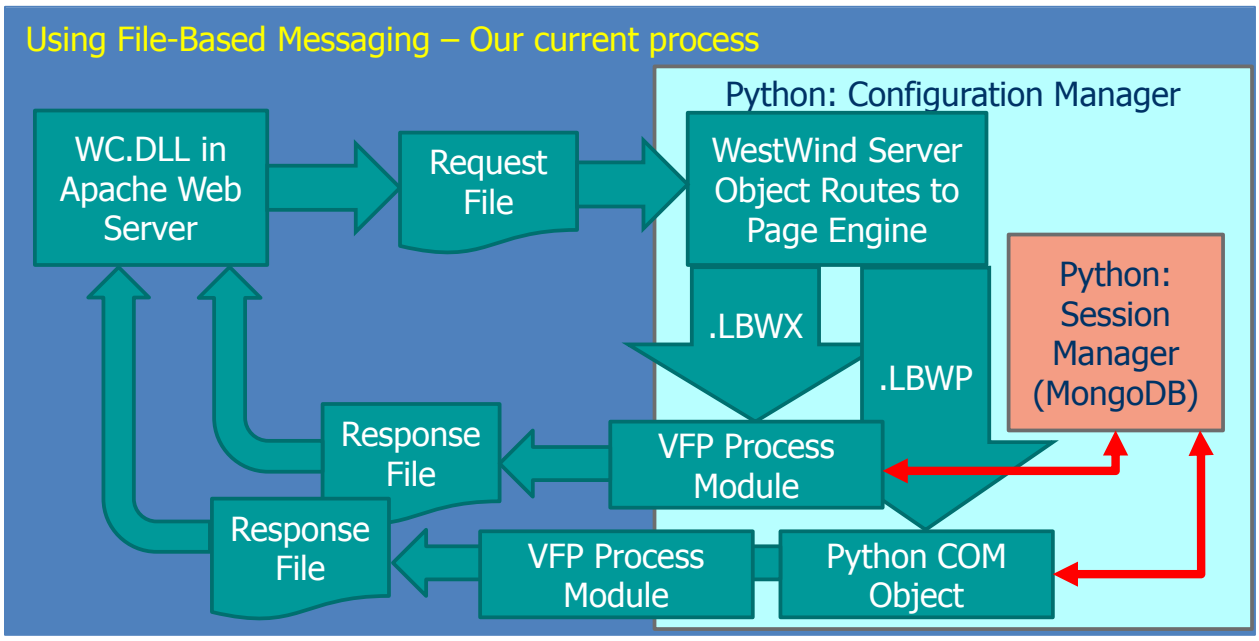
Python module that launches a JavaScript component to track the time since login and optionally display status messages collected periodically by AJAX calls to a Python module. The left-side menu and the list of "loads" in the gray table are created by legacy VFP components. The map is generated by a Python module that uses complex templates to generate the JavaScript code pushed out to the browser to generate the map and place the route and shipment drop points on it. The pop-up bubble defining the elements of one of the shipment drops is generated by a Python module called by AJAX from the browser.

File-Based Messaging and Python

How we accomplish all this integration is shown at a high level by the diagram below in

Figure 42:

Figure 42 - WestWind and Python Schematic



First, our system uses the "file based messaging" option supported by WestWind. We chose this originally for its simplicity over the COM-based messaging option and have never changed. Our activity volume is such (as an industrial, not a consumer product) that we rarely find a need to launch additional server instances to keep up with the volumes. Usually those high volume times are at the end of bidding events where motor carriers are rushing to upload their bids at the last minute before the event closes. Even there we

typically have managed just fine with 2 or three instances of the VFP server application.

To make the integration of Python and VFP/WestWind work we found we needed to build two key modules in Python. First was the configuration engine discussed previously in the context of controlling Biz Objects in a multi-customer setting. By having a common Config framework based in Python and accessed by VFP via COM, we were able to greatly expand the capabilities of the otherwise very adequate configuration tool provided by WestWind. (It may be worth mentioning that our Python Configurator COM component is wrapped in a VFP object that turns properties of the Python COM object into VFP PUBLIC variables for use throughout VFP modules.) Second, we wanted to have a common platform for our session management. This enables us to freely share session data from request to request regardless of whether the responder is Python or VFP.

At first we built our joint session manager (modeled after Rick Strahl's excellent example in WestWind) using PostgreSQL as the database. Ultimately we scrapped PostgreSQL in favor of the more flexible (and faster) MongoDB, the No-SQL database.

The key to making this joint page service possible is the use of the page name extensions to determine where to send the incoming requests. As shown in

Figure 42, .LBWX extension page requests are sent to a pure VFP process. Extensions of .LBWP actually go to a different VFP process that immediately hands the request off to what we call our "Python Page Engine", an out-of-process COM server which determines from the main page name which module to run across the dozens of .PY files containing page generation code. We elected to make our page engine an out-of-process server so we could kill it if it got into trouble and start a new instance without bringing down our main server application.

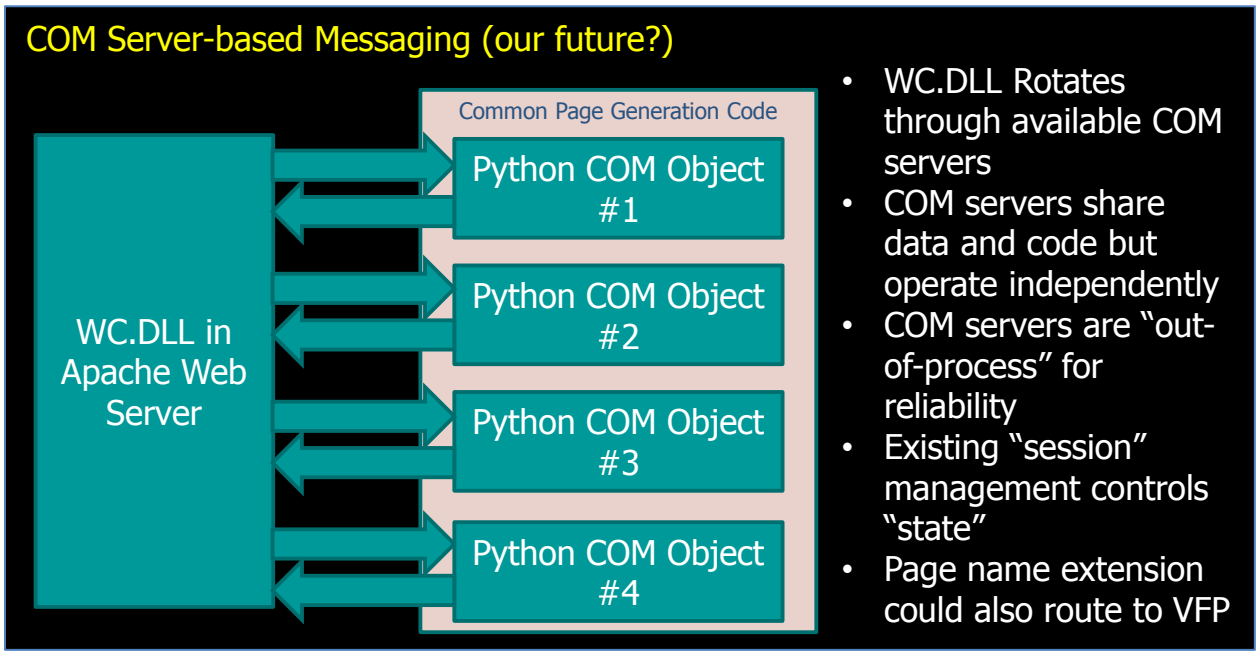
Our Python solution is more "WestWind-like" than most of the standard Python-based tools for building web servers (the most well-known of which is Django). Instead of using directory tree type notation in the URL to get to sub-levels of page generation logic, we simply follow the WestWind approach of using the full page name to find its logic in a function by the same name. (Actually, to be more specific, the master Python Page Engine looks for a function name that starts with "_PAGE" and then has the target page name for the rest of the function name.)

Future Option – Python COM Direct Integration

We aren't certain what the future will bring for our company and our technology, but one possible new direction could be direct requests to Python modules to generate pages. We have experimented with that using the same WC.DLL module that WestWind uses to manage page hits coming through the web server, either IIS or Apache (which we use).

The diagram below in **Figure 43** shows one approach we have already tested and found to be very robust.

Figure 43 - Python COM-based Messaging



When Rick developed WestWind components he envisioned his customized FOX ISAPI module called WC.DLL would be configured to call a set of VFP COM servers in round-robin fashion, exploiting its multi-threaded capability to handle high volumes of traffic. But as you've seen, Python makes a perfectly good COM server. We have tried this configuration with 4 Python COM servers, each with its own GUID, but with a common code platform. The result is robust and stable. Will this be our future, possibly, if our volumes grow to demand it… we'll see.

HTML Page Generation in Python

To be honest, we are HTML, CSS, and Javascript coders too. In addition to something like 100,000 lines of new Python code in our applications, we also have tens of thousands of lines of HTML and Javascript. We started using AJAX for console-like responsiveness in our browser-based displays just a year or so after the acronym was created, and now use it extensively in our web-based GUI.

Like most Python web developers, we use a combination of Python page logic which produces content with little or no actual HTML in it, which is then displayed using HTML template files that use a "templating language" to allow for smart generation of the output HTML page. To support this pattern of development the Open Source community has developed a number of HTML templating frameworks for Python. These vary in their syntax, their support of efficiency features like inheritance, and the speed of rendering. Among these we chose MAKO some years ago, and have never regretted the decision. See http://www.makotemplates.org/ for more information. If you wonder about the performance of a Python-driven HTML templating system, consider that MAKO is used by reddit.com, where it delivers billions of page hits per month.

Three key features of MAKO are important to us:

- It supports inheritance. This allows us to provide a standard base HTML document with all of our required library, script, css, and logo/image components, so none of that is ever repeated in our page logic.
- It supports embedded Python code for complex logic, most commonly used for looping constructs and conditional display of HTML code blocks.
- Its syntax for embedded Python is recognized by PyCharm our favorite Python IDE, so we can edit our HTML/Javascript files with PyCharm too (which it is very good at).

The HTML MAKO code snippet below in **Figure 44** is for part of one of our base templates:

Figure 44 - Base MAKO Template

```
%if cHTTPheader:
<%include file="stdResponseHeader.html" />
%endif
<!DOCTYPE HTML>
## base2.html
<% SVER = "" if 'cScriptRandomizer' not in context.keys() else cScriptRandomizer %>

<html>
    <meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
    <head>
        <title>${appTitle} ${fCodeClear(self.title())}</title>


        <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
        <script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/jquery-ui.min.js"></script>
        <script type="text/javascript" src="Scripts/ww.jquery.js"> </script>
        <script type="text/javascript" src="Scripts/wwScriptLibrary.js?VERNO=${SVER}"> </script>
        <script language="javascript" src="Scripts/alphanum/jquery.alphanum.js?VERNO=${SVER}"></script>
```

This example shows the loading of multiple jQuery and other javascript and css libraries that are used throughout our applications. There is much more logic here, but down in the body of this template is a tag like this: ${self.body()}, where the actual page HTML is incorporated into the template.

A portion of an HTML page inheriting from this base template is shown in

Figure 45 - Python Script in a MAKO Template below. The HTML in this module includes conditional sections plus a loop. The one concession to the very different environment that HTML provides is that in MAKO, python blocks do require ending tags, which they don't in standard Python coding. The ${somevalue} construct calls for a replacement of that code with the value of a dictionary item with a key of "somevalue". This is the basic mechanism for pushing Python data into the page. Note too that the value wrkCols is actually the list which happens to be the value in the dictionary for a key value of "wrkCols" in the controlling Python code.

Figure 45 - Python Script in a MAKO Template

```
    <tr>
        <td><b>Excel Column</b></td>
        <td><b>First Row Data</b></td>
        % if bViewAsLabels == "CHECKED":
            <td><b>Label</b></td>
        % endif
        <td><b>Assigned Field</b></td>
    </tr>
    % for xCol in wrkCols:
    <tr>
        <td align="center">${xCol[0]}</td>
        <td>${xCol[1]}</td>
        % if bViewAsLabels == "CHECKED":
            <td>${xCol[2]}</td>
        % endif
        <td align="center">
            <select size="1" name="COL_${xCol[0]}_SEL" value="${xCol[3]}" id="${xCol[4]}"
             title="Target Field to Map To">
            <option value="IGNORE">Ignore Field</option>
        </select></td>
    </tr>
    % endfor
```

Distributing your Console Applications

In Visual FoxPro you have two options for production distribution. In a few cases, VFP programmers literally copy their PRG and other source files onto a disk and install them on the customers' computers. But generally, we bundle our applications into .EXE files starting with the VFP Project Manager.

Alas, there is nothing exactly like the VFP Project Manager in Python. In Python you are responsible for organizing your code into file system directory trees that make sense and setting up directories with your common code where all of your applications can find them. From there you have a couple of options for how to distribute your work:

Creating a Windows .EXE and the Installer

For the most "Windows-like" product to hand to users, you can use one of two modules from PyPI: PyInstaller or py2exe. In both products, you point them at your application code and launch them to build a windows .EXE containing your application. Optionally, you can create a setup.exe or use a third party Windows installation builder if you need more sophisticated installation features.

A feature of these tools is that at most they will distribute some .pyc or .pyd files which don't expose your source code to your customers. However, if you don't care about that, and like the ability to quickly replace just one .py file that needs a fix, then you can create a Windows installer with pynsist (also download from PyPI). It will install the Python run-time in an application specific directory, and will distribute all of your .py and .pyc programs in a directory tree to allow the runtime to find them. It supplies a launcher .exe file which starts up your application, so it feels to users like a regular Windows executable.

Distributing to Users with Python Installed

There are several use cases where you want to provide your code modules to customers or

fellow programmers who already have Python installed, and simply want to add your module functionality to their system. Naturally an example of this is distributing your module via PyPI where the users will always have some version of Python installed.

There are very strict rules for how to organize files and directories for modules to be distributed either to individual programmers using standard Python tools or to the Open Source community via PyPI. The underlying mechanism makes use of a suite of capabilities referred to as Python distutils. The inner workings of distutils are beyond the scope of this White Paper, but there are ample on-line examples and tutorials to help you get started. And the cool thing is that once you've packaged your module with distutils, deploying it to PyPi or some private repository is very straightforward.

Documenting Your Code

Of course you liberally document your code with comments to help yourself and your team mates understand how your code works. But you can go much farther with that in Python. In **Figure 18 - Introducing Introspection and Customization** we showed how comments can be enclosed in quotation marks, becoming "DOC Strings" which are accessible by Python code itself. The Python ecosystem has taken advantage of that capability to create several powerful module documentation systems which can tap into these DOC Strings to build nicely formatted and indexed help both in HTML and PDF format.

The most basic access to DOC strings in Python is via the pydoc module (download from PyPI), which allows you to view on screen or dump to an HTML file, all the documentation on any module, object class, or function. See the example PyDocDemo.py for an example of dumping out all the documentation for the built in os.path module. Note that while there is limited formatting available, pydoc can be used easily from the interactive interpreter and will work not only for built-in objects and modules, but also for your own code modules (provided you have thought to insert DOC strings at appropriate places in your code!)

The OpenSource\Documentation directory in my Session materials has HTML formatted documentation for all the modules included. You'll find the information nicely indexed, searchable, and (relatively) well organized. This documentation was all extracted from DOC Strings in the Python code itself using the sphinx module from PyPI with the autoapi.sphinx extension and the sphinx-rtd-theme controlling the display (both accessible from PyPI). Sphinx is best for creating and maintaining code for your own applications such that you can easily regenerated your docs after any major code update. Sphinx also allows you to incorporate your own text documents written in the "markdown" language called reStructuredText (.rst files). If you have followed Rick Strahl's writing on Markdown, you know how easy it is to create formatted text, and the Markdown he describes can quickly be converted to rst by one of several Python utilities.

Getting Started with Python

A little over 8 years ago I stood at the precipice of the Python world and decided to jump in. It wasn't a solo dive, fortunately. Here's some suggestions for making your plunge as easy as possible:

- Download Version 3.7 of Python from www.python.org, the website of the Python Software Foundation and follow the instructions above for installation.
- Explore the rich trove of training, tutorial, and documentation options here: https://wiki.python.org/moin/BeginnersGuide/Programmers
- Prepare yourself for feeling overwhelmed. It's completely normal. Just resolve to press on anyway. It may take you a year or more to get up to speed with Python development.
- Get the "community" free version of PyCharm from https://www.jetbrains.com/pycharm or pop for the commercial version. The free version is designed just for Python. The commercial version supports editing HTML, Javascript, MAKO, and other languages. You'll want it eventually if you are doing web development.
- Find a local "mentor" who has learned Python and can be a sounding board. He or she may make finding solutions much easier than your fruitless Google searches. If your organization has the resources, hire that person!
- Buy the books (on-line PDF or in paper – your choice):
    - *Python Programming on Win32*, Mark Hammond – The official documentation for the Python Win32 extensions and essential if you are going to be doing COM programming in Python
    - *Python in a Nutshell*, Alex Martelli – At 772 pages hardly a "nutshell", but this is recognized as one of the fundamentally important books on Python. Not for learning programming, but for both experienced and new Python programmers it's an invaluable reference.
    - *Rapid GUI Programming with Python and QT*, Mark Summerfield – Worth getting if you are even considering building console applications with Python. (The Martelli book also has a good section on Tkinter, the built-in Python GUI engine.)
- Don't be afraid to create or find VFP-like code to make Python feel more familiar. I've included MPSSBaseTools.py in the session materials which has a number of functions like STRTOFILE() and JUSTSTEM() that do the same thing as their VFP equivalents.
- Don't be embarrassed if your Pythonista colleagues accuse you of programming Python with a "VFP accent". Of course you will. So what! We all are influenced by all the programming languages we have learned over our careers.

PyPI Modules of Interest

The Python Package Index (PyPI) with its 152,000 modules is a great source of tools to make your programming work easier. Here are some example modules referenced in my presentation and in this document that you may want to explore further:

- **Win32com** – The Windows 32-bit extensions by Mark Hammod. You get the latest from PyPI.
- **Configobj** – the foundation for the configurator objects that we built for our applications.
- **Zeep** – A SOAP client tool to access Microsoft SOAP-based web services.
- **Soap2Py** – SOAP server tools to create SOAP services with compliant WSDLs

without using .NET

- **Easygui** – Simple GUI components like message boxes, confirmation boxes, file finders, and so on. Based on Tkinter, it's completely cross platform and can be invoked even if your Python program doesn't officially have a user interface.
- **Mailer** – A wrapper that greatly simplifies use of the powerful but terribly complex Python built-in tools for sending emails.
- **ReportLab** – Powerful tools for generating PDF files under program control. Very fast performance. The basic product is Open Source, but to get sophisticated templating for elaborately formatted output you have to pay a license fee.
- **XHTML2PDF** – To create more sophisticated PDF output consider this module which converts HTML, including CSS referenced by it, into formatted PDF output.
- **BeautifulSoup** – parses any HTML page into an object hierarchy which can be easily searched by tagname, name, id, and other object properties.
- **Pyconcrete** – Encrypts your .pyc files and requires a passkey to import them within your applications. Prevents reverse engineering your compiled Python code.
- **Sphinx** – Compiles program documentation from text files plus doc strings in your code. Explore the extensions that read your doc strings directly and build indexed and cross referenced HTML document output.
- **Requests and Requests-toolbelt** – Companion modules that greatly simplify accessing HTTP and HTTPS based web services. They handle secure connections much more simply than the core Python networking and sockets components which are 1000% complete but crazy complex to use.

Other Useful Links

Here are some web URLs you may find useful too:

- GitHub repository for the CodeBase™ Open Source product: https://github.com/MPSystemsServices/CodeBase-for-DBF
- GitHub repository for the Python CodeBase Tools Open Source modules: https://github.com/MPSystemsServices/Python-CodeBase-Tools
- PCWorld Magazine guide to Python Training Resources: https://tinyurl.com/BestPythonTraining
- Hitchhiker's Guide to Python, an on-line book: https://docs.python-guide.org/
- PythonBytes, a weekly podcast about Python: https://pythonbytes.fm/
- Test and Code, another regular podcast focused on testing and software validation in Python: https://pythonbytes.fm/
- PythonOutLoud, a podcast directed at Python learners with many smart discussions about tough Python features: http://pythonoutloud.com/
- A discussion about Python application deployment installers: https://fernandofreitasalves.com/distributing-python-apps-windows-desktops/
- Home page for PyInstaller: https://www.pyinstaller.org/
- Home page for the MAKO templating tools: http://www.makotemplates.org/

Biography

Jim Heuer has had a long career applying computing technology to business problems,

starting in the mid 1960s with the first edition of Fortran-66 running on an IBM 1401 computer. He continued in the late 1970s with a DataFlex application built to run on an MPM-86 "micro-computer". After experience with DataFlex, Knowledgeman, SPSS, and numerous other data management tools, he jumped into FoxPro for DOS 2.5 in 1990 and started building applications for his clients as well as supporting his logistics management consulting with custom-built analytical tools. From FoxPro DOS, he migrated his skills to FoxPro for Windows 2.5 and then all the versions of Visual FoxPro through the last, lamented version 9.0.

In 2002, Jim developed logistics analysis software implementing advanced AI problem solving technology built in Visual FoxPro with a high-performance computing component built in C for heavy number crunching. Over the years that software has grown in capability to today where it is marketed as PlanTools by Jim's company Evos SmartTools. Jim has continued as the primary designer and developer of the product as it currently deployed as a web-only application serving the needs of large corporations in the U.S. and Canada to manage their inbound and outbound freight. Starting in 2012, facing the demise of Microsoft support for Visual FoxPro and pressure from potential investors to begin a migration to a more active platform, he explored several options for a way to build on the current application while retaining full compatibility with the 200,000 lines of existing VFP code. After considering .NET, Java, PHP, and others, he settled on Python as the most compatible and VFP-like of the options.

In the years since, Evos SmartTools has built scores of new modules for new capabilities in Python—all smoothly interoperating with the existing VFP code, much of which will remain functional indefinitely as new features are built out in Python. Today, Evos has over 100,000 lines of Python code which extend the original VFP applications seamlessly.

Over the years, Jim has attended numerous VFP conferences, including several SW Fox events, but never, so far, as a speaker.

