EPS Import Functionality for ReportLab

A Proposal for the

By

Mark Peters

mark.peters@ivanhouse.com

Synopsis

ReportLab is a Python Library designed to easily implement PDF output functionality into Python programs. Currently, ReportLab can import bitmap images of a variety of types and has some native vector graphics capabilities, but has no support for external vector-based images. This proposal would add import support for Encapsulated PostScript (EPS) files to ReportLab, with usage comparable to the current image import functionality.

This PDF document was produced using Python and ReportLab. The logo above was produced from an EPS file and imported into ReportLab using a proof-of-concept of this functionality.

Benefits to Community

The primary benefit of this proposal will, of course, be increased functionality for the ReportLab project. The main increases in functionality will be:

- **Smaller PDF Output** - Vector format graphics, e.g. EPS, often produce smaller files than bitmap graphics of comparable quality.
- **Better Print Quality** - Vector format graphics print at the resolution of the output device. This means that printing a PDF containing vector format graphics will not result in jagged edges on lines or pixelization which can occur when printing low resolution bitmap images.
- **Better Support for Logos/Clipart** - Most high-quality clip art is available in EPS form. It is also a common format used by graphics professionals for company logos (the Google logo above is an example). EPS support in ReportLab will make it easier to produce professional documents.

Also, implementation of this proposal will involve much of the work required to implement a complete PostScript interpreter written in Python. The framework developed for this project, including a PostScript parser and EPS interpreter, could be useful to other projects in the community.

Finally, as the visibility of high-quality Python projects increases, the Python community as a whole will benefit.

Biographical Information

Software

Programming since 1982 (started on Commodore VIC-20).

Languages include BASIC, 65xx assembler, C, C++, Pascal, REXX, awk, unix shell, perl, and Python.

Worked six years for company providing supply-chain communication solutions, mainly fax, email, and forms integration into ERP (SAP, Baan, Peoplesoft, Oracle) systems. Specific accomplishments include:

- Implemented Unicode (Chinese) Streamserve forms project for worldwide electronics company.
- Conducted customer training (both in-house and on-site) in use of Streamserve forms package.
- Designed integration toolkit, written in Python, on which all company integrations were based.

Have used various open source programs over the years, but have never had a good opportunity to contribute beck to the community. I hope that this proposal will give me that opportunity.

Education

1987-1988 - Completed approximately one year at University of Cincinnati, College of Engineering

2005 - Enrolled at Columbus State Community College, Columbus, OH

Educational Goals - Associates degree in Computer Information Technology, Enterprise Developer Track with certificates in Purchasing and Logistics from the Logistics program. This is to position myself, upon graduation, as a Supply-Chain developer.

Beginning full-time classes Summer Quarter, 2005 (classes start June 27, 2005)

Summary of Project Schedule

June 2	Heard about Google: Summer of Code project
June 3	Selected Project
June 4-6	Planned project proof of concept
June 7-12	Coded proof of concept
June 12-14	Wrote Proposal
by June 24	Proposal Accepted
by July 7	Finish refactoring prototype language framework
July 7-15	Finish implementation of most PostScript operator functions
July 15-21	Flesh out ReportLab interface
July 21-31	Extra time to handle PostScript/PDF operator differences
July 31	Beta Release
Aug 1 - Aug 21	Bug Fixes and documentation
Aug 21	Release

Project Details

The ReportLab Toolkit is an open source library for generation of PDF documents. As suggested on the ideas page for the Google Summer of Code project, this project would add Encapsulated PostScript (EPS) file import capability to that library. The implementation would be similar to the existing image file import ability, which allows the importation of bitmap images into created PDF files.

Currently, to use a bitmap image in ReportLab, you would use a function similar to:

```
Image("replogo.gif",width=2*inch, height=1*inch)
```

to produce the following output:

This project would add the ability to use EPS files in a similar fashion. The following code:

```
EPSImage("google_summer.eps",width=2*inch, height=1*inch)
```

should produce the following output:

In order to describe the implementation of this EPS import functionality, a discussion of some of the differences between PostScript and PDF is necessary.

PostScript and PDF are both page description languages, optimized for describing how data should appear on a printed page. Since both formats were created by Adobe, the languages are quite similar. In theory, this should make converting from EPS (which is a subset of PostScript) fairly easy. For example, let's look at the PostScript for drawing a line:

```
50 100 moveto 150 200 lineto
```

This will draw a line from coordinates 50,100 to 150,200. Let's examine the PDF equivalent of the same line:

```
50 100 m 150 200 l
```

One would think that it should be pretty easy to convert from one to the other. However, this is where things become complicated. While PDF and PostScript both describe pages similarly, PostScript is a full featured programming language, and PDF is designed to display static output. This means that one can write fairly complex programs in PostScript. PostScript supports stacks of namespaces, conditionals, looping structures, and procedure definitions. The ability to define procedures is what really causes problems converting from EPS to PDF. Most programs that generate PostScript (and EPS) write a lengthy prologue in the beginning of the file that defines procedures and assigns them to shortcut names. For example, a procedure might be defined that does a move and line together:

```
/moveandline {moveto  lineto} def
```

So that the actual code to draw the line now looks like:

```
150 200 50 100 moveandline
```

It now becomes impossible to know what the procedure 'moveandline' does without implementing at least enough of a PostScript interpreter to be able to build procedure definitions.

Project Details (continued)

A PostScript interpreter with full support for all the language features is a major undertaking. The Ghostscript project has been working on this for many years and is the definitive work in this area. However, the task at hand (interpreting an EPS file to extract PDF commands) is simplified in a number of ways:

- The interpreter does not need to actually render the PostScript into a raster image. This is one of the main features that Ghostscript has which is not necessary for this project. When a PostScript drawing command in encountered, the interpreter only needs to generate a corresponding PDF command (translating 'moveto' to 'm' in the example above).
- EPS is a simplified subset of full PostScript. EPS files by definition, are designed to be included inside other files. Because of this, they are limited to one page, with no device-specific operators allowed.
- Writing the interpreter in Python simplifies the project because most of PostScript's complex data structures map directly to Python data structures (arrays map to lists, dictionaries map to dictionaries, etc...). Python's introspection ability also allows easier and faster writing of PostScript operators.

This project will be implemented as a module in the ReportLab open source library. There should not be any dependencies on external libraries, and it should run without problem on any platform that ReportLab supports.

Documentation will be provided to the ReportLab open source library, describing usage and providing examples. This is simplified by the similarity in syntax to the existing image capability.

EPS files can be very complex. The compatibility goal of this project is to properly handle at least 90% of EPS files by the beta release. I plan to continually work to increase that number throughout the beta period and beyond into the first release. The framework in place will make it easy to maintain this code for updates and bug fixes.

Architecture

The architecture of the project is comprised of four primary components:

- **Parser/Tokenizer** - This component handles the conversion of an input stream of PostScript commands and parses it into a series of PostScript tokens of various types.
- **PostScript Language Framework** - This component will handle tasks specific to the various elements of the PostScript language. These tasks include maintenance of the dictionary, operand, and execution stacks. This component will handle definition and execution of procedures. It will be also responsible for looking up and calling the PostScript operator functions.
- **PostScript Operator Functions** - This component contains the actual definitions of the operator functions. Most of the work in a PostScript program happens in these operators. There are operators to manipulate the stacks, perform arithmetic, and perform graphic operations. The graphic operations will, for the most part, produce the equivalent PDF commands.
- **ReportLab Interface** - This component will provide the interface between the components listed above and the ReportLab library. It will handle the syntax of various functions exposed to the library. It will also provide support for preconverting EPS files in a similar manner to the proof-of-concept program. It will also provide support for storing the object once in the PDF, with subsequent references only generating links to that stored object.

For example, when processing the following PostScript command:

```
50 100 add
```

The Parser/Tokenizer will convert this to the following PostScript tokens:

```
Integer             50
Integer            100
Executable Name    add
```

When processing these tokens, the language framework will push the two integers onto the operand stack, look up the name 'add' on the dictionary stack, find an operator function of that name in the system dictionary, and call that function. The code in that function will pull the two values from the operand stack, add them together, and push the result back onto the stack.

Project History

While browsing the Python Daily News site on the morning of June 2, I found a link to the Google Summer of Code project. Looking at the ideas page, the ReportLab idea caught my eye, as I had played around with that library some and liked what I saw. I decided to make a real attempt to get my proposal accepted.

I started looking at some EPS files and quickly determined that a simple text scraping solution was not going to cut it for even simple EPS files. Also, I have had some experience with scraping text from a PostScript file. Important safety tip: Doing that with regular expressions is the road to madness. The only option for a real solution was to write a PostScript interpreter.

To get a feel for the complexities involved in writing a PostScript interpreter, I decided to write a proof-of-concept prototype. My goal was to be able to parse a simple EPS file. I opened CorelDraw, drew a rectangle, and exported that as an EPS file. My proof-of-concept goal was to display that in the output of a PDF file created in ReportLab.

EPS files usually include a prologue which defines a set of procedures used by that file. Because of the PostScript prologue that is included in the EPS output from CorelDraw, parsing the EPS file to extract the individual PostScript tokens yielded 3,774 tokens for a simple rectangle. Note that Corel Draw's prologue varies from prologues written by other applications and those prologues also differ from each other.

I built a framework which includes a Python class to handle the PostScript dictionary stack and which behaves like a Python dictionary. Also, operand and execute stacks are implemented as Python lists. PostScript objects are implemented as Python tuples in the form (type,value). I used Python's introspection ability to simplify the creation of functions to handle PostScript operators.

Merely defining the function with a certain name automatically creates an operator with that name in the system dictionary. When an executable name is encountered, it is looked up on the dictionary stack which returns a reference to that function. That function then gets called to handle that operator. An error results if an unknown operator name is encountered.

Work on the prototype continued, by processing the rectangle EPS file until it encountered an unknown operator token, then writing code to handle that operator. After implementing at least minimal definitions for 67 operators, I was able to interpret the rectangle file without errors.

Next, I needed to create the PDF commands which corresponded to the PostScript ones. With the PostScript and PDF language reference manuals side by side, I implemented code in each of the PostScript operators to output the appropriate PDF operator to a file. This gave me a file with all of the PDF operators needed to describe the graphic in the EPS file.

I then wrote a function to be used in ReportLab to import the list of PDF operators, with options to scale and translate the graphic.

This completed my proof-of-concept and proved to me that writing a limited-functionality PostScript interpreter was within the scope of a summer project.

Finally, I wrote the proposal that you are reading now. It was written using Python and ReportLab, using EPS files converted with the proof-of-concept program.

Details for Deliverables and Milestones

by July 7

Finish refactoring proof-of-concept interpreter - The interpreter developed as a proof-of-concept worked well, but began to show some design flaws as functionality was added and more was learned about the PostScript language. This is not a problem, that is the purpose of a proof-of-concept: to learn these things as early as possible. The first step toward completion, though, will be to refactor the interpreter with the following specific objectives:

- **Rewrite parser as Python Generator** - The code which parses PostScript input and creates tokens is currently a bit of a kludge and should be rewritten as a Python generator function, allowing either a string or a file object for creation.
- **Create a PostScript object class** - Currently, PostScript objects are tuples of a type string and a value represented by a native Python object. PostScript objects have a number of attributes which are not handled in the current model. These include access and execute attributes. This will bring the capabilities and behavior of objects in this interpreter more in line with the PostScript specifications. Also, add support for Python arithmetic methods, where appropriate.
- **Bring names in line with PostScript specifications** - Currently, the names of object types do not match the names as returned by the PostScript 'types' operator. This change would bring those names in line with the specification. Also, create Python exceptions which match in name to the official PostScript error types.
- **Create Template for Operator Functions** - Before refactoring the existing operator functions, create a template for what one should look like, including style for doc strings, comments, variable naming conventions, assertions, and exceptions.
- **Better Comment Handling** - Current support for comments is minimal. This should be improved as EPS files actually store important information in comments.

Upload Work - By this date, work out with mentoring organization where data should be kept (probably ReportLab's Subversion server), as well as policies regarding updates. Also, decide which version(s) of ReportLab to support.

Details for Deliverables and Milestones (continued)

July 7 - 15	**Finish implementation of most PostScript operators** - This time will be spent fleshing out PostScript operator functions using the template developed previously. The objective here is to support all of the easy-to-implement functions, even if they have not been encountered. Also, start collecting EPS files from various sources and developing new operator functions as needed to process those files. Any operators not handled after this step should be documented.
July 15 - 21	**Finish ReportLab Interface** - This time will be spent integrating the PostScript parser with ReportLab. Some specific objectives: • **Support Object Caching** - As the ReportLab Image routines do, the EPS support should cache the imported graphic in the document to save on space in the output. • **Support Object Conversion** - Since the conversion step is time consuming, provide support to preconvert EPS files, or to save the converted file to disk (much as the proof-of-concept does) similar to the way Python treats compiled (*.pyc) files.
July 21 - 31	**Padding** - Some extra time to deal with any remaining issues before Beta release. I will possibly need to spend extra time on differences between some PostScript and PDF operators. Specifically, there are differences in the commands for stroking and filling arbitrary paths.
July 31	**Beta Release**
Aug 1 - Aug 21	**Bug Fixes** - As soon as this program hits beta, I expect a variety of "This won't work with my EPS file" bug reports. Find those bugs and squash them! **Documentation** - Finish the user documentation and examples.
Aug 21	**Release** - Make sure mentoring organization does not have any outstanding issues and release it to the world.

Development Methodology

While every task is approached differently, the methodology used to solve this problem is similar to how I approach any problem. The methodology used for this problem is summarized below.

- Analyze problem
- Plan and code proof-of-concept programs
- Re-analyze problem, making changes to design as necessary
- Rewrite or refactor proof-of-concept programs
- Systematically complete program features
- Release Beta
- Write Documentation
- Fix Bugs
- Release

Examples of use

The "Google: Summer of Code" logo used throughout this document was made from an EPS file imported into CorelDraw, adding the "Summer of Code" text, and exporting as an EPS file. Then, using the proof-of-concept interpreter, I converted it to the equivalent PDF commands which are saved in the file 'google_summer.out'. Note: I had to tweak a couple color space operators in the output to get the colors right due to an unresolved bug (all colorspace is being interpreted as RGB, but the EPS has a mix of RGB and CYMK). All other operators are untouched. This will be fixed during the refactoring.

The following code is used to display the Google logo below:

```
google = EPSImage("google_summer.out")
google.hAlign = "CENTER"
getStory().append(google)
```

Now lets display the logo one inch wide and left justified

```
google = EPSImage("google_summer.out",width = 1 * inch)
google.hAlign = "LEFT"
getStory().append(google)
```

Finally, the rectangle that started it all. Here's the rect.out contents:

```
%%BoundingBox: 105 624 155 667
105.42756 666.44929000000002 m
154.60015999999999 666.44929000000002 l
154.60015999999999 625.04079000000002 l
105.42756 625.04079000000002 l
105.42756 666.44929000000002 l
h
0.0 0.0 0.0 RG
S
```

And lets put it 1/2 inch left of the left margin.

```
rect = EPSImage("rect.out",x = -0.5 * inch)
rect.hAlign = "LEFT"
getStory().append(rect)
```