Interactive Notebooks for Python

An IPython project for Google's Summer of Code 2005

Fernando Pérez*

6th June 2005

Abstract

This project aims to develop a file format and interactive support for documents which can combine Python code with rich text and embedded graphics. The initial requirements only aim at being able to edit such documents with a normal programming editor, with final rendering to PDF or HTML being done by calling an external program. The editing component would have to be integrated with IPython.

This document was written by the IPython developer; it is made available to students looking for projects of interest and for inclusion in their application.

1 Project overview

Python's interactive interpreter is one of the language's most appealing features for certain types of usage, yet the basic shell which ships with the language is very limited. Over the last few years, IPython[1] has become the de facto standard interactive shell in the scientific computing community, and it enjoys wide popularity with general audiences. All the major Linux distributions (Fedora Core via Extras, SUSE, Debian) and OS X (via fink) carry IPython, and Windows users report using it as a viable system shell.

However, IPython is currently a command-line only application, based on the readline library and hence with single-line editing capabilities. While this kind of usage is sufficient for many contexts, there are usage cases where integration in a graphical user interface (GUI) is desirable.

In particular, we wish to have an interface where users can execute Python code, input regular text (neither code nor comments) and keep inline graphics, which we will call *Python notebooks*. This kind of system is very popular in scientific computing; well known implementations can be found in Mathematica™ [2] and Maple™ [3], among others. However, these are proprietary (and quite expensive) systems aimed at an audience of mathematicians, scientists and engineers.

The full-blown implementation of a graphical shell supporting this kind of work model is probably too ambitious for a summer project. Simultaneous support for rich-text editing, embedded graphics and syntax-highlighted code is extremely complex, and likely to require far more effort than can be mustered by an individual developer for a short-term project.

*Fernando.Perez@colorado.edu

[1] http://ipython.scipy.org

[2] http://www.wolfram.com/products/mathematica

[3] http://www.maplesoft.com

This project will thus aim to build the necessary base infrastructure to be able to edit such documents from a plain text editor, and to render them to suitable formats for printing or online distribution, such as HTML, PDF or PostScript. This model follows that for the production of LaTeX documents, which can be edited with any text editor.

Such documents would be extremely useful for many audiences beyond scientists: one can use them to produce additionally documented code, to explore a problem with Python and maintain all relevant information in the same place, as a way to distribute enhanced Python-based educational materials, etc.

Demand for such a system exists, as evidenced by repeated requests made to me by IPython users over the last few years. Unfortunately IPython is only a spare-time project for me, and I have not had the time to devote to this, despite being convinced of its long term value and wide appeal.

If this project is successful, the infrastructure laid out by it will be immediately useful for Python users wishing to maintain 'literate' programs which include rich formatting. In addition, this will open the door for the future development of graphical shells which can render such documents in real time: this is exactly the development model successfully followed by the LyX[4] document processing system.

2 Implementation effort

2.1 Specific goals

This is a brief outline of the main points of this project. The next section provides details on all of them. The student(s) working on the project would need to:

1. Make design decisions for the internal file structure to enable valid Python notebooks.
2. Implement the rendering library, capable of processing an input notebook through reST or LaTeX and producing HTML or PDF output, as well as exporting a 'pure-code' Python file stripped of all markup calls.
3. Study existing programming editor widgets to find the most suitable one for extending with an IPython connector for interactive execution of the notebooks.

2.2 Complexity level

This project is relatively complicated. While I will gladly assist the student with design and implementation issues, it will require a fair amount of thinking in terms of overall library architecture. The actual implementation does not require any sophisticated concepts, but rather a reasonably broad knowledge of a wide set of topics (markup, interaction with external programs and libraries, namespace tricks to provide runtime changes in the effect of the markup calls, etc.)

While raw novices are welcome to try, I suspect that it may be a bit too much for them. Students wanting to apply should keep in mind, if the money is an important consideration, that Google only gives the $4500 reward upon *successful completion* of the project. So don't bite more than you can chew. Obviously if this doesn't matter, anyone is welcome to participate, since the project can be a very interesting learning experience, and it will provide a genuinely useful tool for many.

[4] `http://www.lyx.org`

3 Technical details

3.1 The files

A basic requirement of this project will be that the Python notebooks shall be valid Python source files, typically with a `.py` extension. A renderer program can be used to process the markup calls in them and generate output. If run at a regular command line, these files should execute like normal Python files. But when run via a special rendering script, the result should be a properly formatted file. Output formats could be PDF or HTML depending on user-supplied options.

A reST markup mode should be implemented, as reST is already widely used in the Python community and is a very simple format to write. The following is a sketch of what such files could look like using reST markup:

```
from notebook.markup import rest

rest.title('This is a Python Notebook')
rest.heading(1,'A first-level heading')
rest.text("""\
Some plain text, without any special formatting.

Below, we define a simple function to add two numbers.""")

def add(x,y):
    return x+y
```

Additionally, a LaTeX markup mode should also be implemented. Here's a mockup example of what code using the LaTeX mode could look like.

```
from notebook.markup import latex

latex.document_class('article')
latex.title('This is a Python Notebook')

latex.section('A section title',label='sec:intro')

latex.text(r"""Below, we define a simple function
\begin{equation}
f: (x,y) \rightarrow x+y^2
\end{equation}""")

def f(x,y):
    return x+y**2

# since the .text method passes directly to latex, all markup could be input
# in that way if so desired
latex.text(r"""
\section{Another section}

More text...

And now a picture showing our important results...""")

latex.include_graphic('foo.png')
```

At this point, it must be noted that the code above is simply a sketch of these ideas, not a finalized design. An important part of this project will be to think about what the best API and structure for this problem should be.

3.2 From notebooks to PDF, HTML or Python

Once a clean API for markup has been specified, converters will be written to take a python source file which uses notebook constructs, and generate final output in printable formats, such as HTML or PDF. For example, if `nbfile.py` is a python notebook, then

```
$ pynb --export=pdf nbfile.py
```

should produce `nbfile.pdf`, while

```
$ pynb --export=html nbfile.py
```

would produce an HTML version. The actual rendering will be done by calling appropriate utilities, such as the reST toolchain or LaTeX, depending on the markup used by the file.

Additionally, while the notebooks will be valid Python files, if executed on their own, all the markup calls will still return their results, which are not really needed when the file is being treated as pure code. For this reason, a module to execute these files turning the markup calls into no-ops should be written. Using Python 2.4's -m switch, one can then use something like

```
$ python -m notebook nbfile.py
```

and the notebook file `nbfile.py` will be executed without any overhead introduced by the markup (other than making calls to functions which return immediately). Finally, an exporter to clean code can be trivially implemented, so that:

```
$ pynb --export=python nbfile.py nbcode.py
```

would export only the code in `nbfile.py` to `nbcode.py`, removing the markup completely. This can be used to generate final production versions of large modules implemented as notebooks, if one wants to eliminate the markup overhead.

3.3 The editing environment

The first and most important part of the project should be the definition of a clean API and the implementation of the exporter modules as indicated above. Ultimately, such files can be developed using any text editor, since they are nothing more than regular Python code.

But once these goals are reached, further integration with an editor will be done, without the need for a full-blown GUI shell. In fact, already today the (X)Emacs editors can provide for interactive usage of such files. Using python-mode in (X)Emacs, one can pass highlighted regions of a file for execution to an underlying python process, and the results are printed in the python window. With recent versions of python-mode, IPython can be used instead of the plain python

interpreter, so that IPython's extended interactive capabilities become available within (X)Emacs (improved tracebacks, automatic debugger integration, variable information, easy filesystem access to Python, etc).

But even with IPython integration, the usage within (X)Emacs is not ideal for a notebook environment, since the python process buffer is separate from the python file. Therefore, the next stage of the project will be to enable tighter integration between the editing and execution environments. The basic idea is to provide an easy way to mark regions of the file to be executed interactively, and to have the output inserted automatically into the file. The following listing is a mockup of what the resulting file could look like

```
from notebook.markup import rest

rest.title('This is a Python Notebook')
rest.text("""\
Some plain text, without any special formatting.

Below, we define a simple function to add two numbers.""")

def add(x,y):
    return x+y

rest.text("Let's use it with x=2,y=3:")
# This simply means that all code until the next markup call is to be executed
# as a single call.  The editing screen should mark the whole group of lines
# with a single In[NN] tag (like IPython does, but with multi-line
    ...capabilities)
rest.input()
add(2,3)
# This output would appear on-screen (in the editing window) simply marked
# with an Out[NN] tag
rest.output("5")
```

Basically, the editor will execute `add(2,3)` and insert the string representation of the output into the file, so it can be used for rendering later.

4 Available resources

IPython currently has all the necessary infrastructure for code execution, albeit in a rather messy code base. Most I/O is already abstracted out, a necessary condition for embedding in a GUI (since you are not writing to stdout/err but to the GUI's text area).

For interaction with an editor, it will be necessary to identify a good programming editor with a Python-compatible license, which can be extended to communicate with the underlying IPython engine. IDLE, the Tk-based IDE which ships with Python, should obviously be considered. The Scintilla editing component[5] may also be a viable candidate.

It will also be interesting to look at the LyX editor[6], which already offers a Python client[7]. Since LyX has very sophisticated LaTeX support, this is a very interesting direction to consider for the future (though LyX makes a poor programming editor).

[5] `http://www.scintilla.org`
[6] `http://www.lyx.org`
[7] `http://wiki.lyx.org/Tools/PyClient`

5 Support offered to the students

The IPython project already has an established Open Source infrastructure, including CVS repositories, a bug tracker and mailing lists. As the main author and sole maintainer of IPython, I will personally assist the student(s) funded with architectural and design guidance, preferably on the public development mailing list. I expect them to start working by submitting patches until they show, by the quality of their work, that they can be granted CVS write access. I expect most actual implementation work to be done by the students, though I will provide assistance if they need it with a specific technical issue.

If more than one applicant is accepted to work on this project, there is more than enough work to be done which can be coordinated between them.

6 Licensing and copyright

IPython is licensed under BSD terms, and copyright of all sources rests with the original authors of the core modules. Over the years, all external contributions have been small enough patches that they have been simply folded into the main source tree without additional copyright attributions, though explicit credit has always been given to all contributors.

I expect the students participating in this project to contribute enough standalone code that they can retain the copyright to it if they so desire, as long as they accept all their work to be licensed under BSD terms.

7 Acknowledgements

I'd like to thank John D. Hunter, the author of matplotlib[8], for lengthy discussions which helped clarify much of this project. In particular, the important decision of embedding the notebook markup calls in true Python functions instead of specially-tagged strings or comments was an idea I thank him for pushing hard enough to convince me of using.

My conversations with Brian Granger, the author of PyXG[9] and braid[10], have also been very useful in clarifying details of the necessary underlying infrastructure and future evolution of IPython for this kind of system.

Thank you also to the IPython users who have, in the past, discussed this topic with me either in private or on the IPython or Scipy lists.

[8] `http://matplotlib.sf.net`
[9] `http://hammonds.scu.edu/~classes/pyxg.html`
[10] `http://hammonds.scu.edu/~classes/braid.html`