Advances in PythonTEX with an introduction to fvextra

Geoffrey M. Poore

Abstract

The PythonTEX package allows Python and several other programming languages to be embedded within LATEX documents. It also typesets code with syntax highlighting provided by the Pygments library for Python. Code typesetting has been improved with the new fvextra package, which builds on fancyvrb by allowing long lines of code to be broken and by providing several additional features. Code execution has been improved by a new set of commands and environments that perform variable substitution or string interpolation. This makes it easier to mix LATEX with Python or other languages while avoiding errors due to expansion, tokenization, and catcodes.

1 Limitations with code typesetting and execution

In 2012, I released the first version of the PythonTEX package [13] with the goal of making it simpler to write mathematical and scientific LATEX documents. PythonTEX allows the LATEX source of a document to contain both a mathematical result and the Python code that calculated it, or both a plot and the Python code that generated it. Originally, PythonTEX only allowed Python code in a LATEX document to be executed, with the output included in the document. It is now possible to execute Ruby, Octave, Sage, Bash, and Rust code as well. From the beginning, PythonTEX has also allowed general code typesetting with syntax highlighting.

PythonTEX's code typesetting has been functional but relatively basic. It uses the Pygments library [15] for Python to perform syntax highlighting. (Pygments is also used for syntax highlighting by the minted [14] package, which I maintain, as well as the verbments [18], texments [3], and pygmentex [5] packages.) Pygments supports over 300 languages and can perform highlighting that would not be practical in a pure LATEX solution such as the listings package [2]. Yet Pygments is not without drawbacks. It uses the fancyvrb package [16] to perform the actual code typesetting. fancyvrb lacks many of the advanced features found in listings, such as the ability to break long lines of code. Unfortunately, attempting to use listings instead of fancyvrb brings its own set of issues; among other things, listings lacks built-in support for UTF-8 and other multi-byte encodings under the pdfTEX engine.

This paper introduces fvextra [12], a new package I have created to address these limitations in code typesetting. fvextra extends and patches fancyvrb. It provides most of the features that fancyvrb lacks compared to listings, including line breaking with fine-grained control over break locations. The fvextra package also provides additional features, such as the ability to highlight specific lines or line ranges based on line numbers. The most recent versions of PythonTEX and minted require fvextra and fully support all new features.

Another longstanding drawback in PythonTEX relates to code execution rather than typesetting. Documents that use PythonTEX are valid LATEX documents; there is no preprocessing step to produce LATEX source. A PDF or other output is created by running LATEX (code is saved to a temporary file), then running the `pythontex` executable (code is executed), and finally running LATEX again (code output is brought into the final PDF). The advantage of this approach is that it is possible to create macros that mix LATEX with Python or other languages. Since LATEX handles all code before it is executed, code can be assembled using macros. In a preprocessor approach such as that used by Sweave [4], knitr [17], and Pweave [9], this is generally not possible because LATEX only receives a copy of the document in which code has been replaced by its output.

The disadvantage of PythonTEX not being a preprocessor is that LATEX does indeed process everything. For example, it is not possible to use a PythonTEX command to insert Python output in the midst of a `verbatim` environment; the command would appear literally. Similarly, PythonTEX commands can cause errors within `tikzpicture` environments or in other situations in which characters do not have their normal meanings (catcodes) or in which other special processing is applied.

This paper introduces a new solution for these scenarios. New commands and environments perform variable substitution or string interpolation. These effectively allow the preprocessor approach to be applied to the argument of a command or the contents of an environment. It is now simpler to mix LATEX with Python or another language while avoiding errors due to expansion, tokenization, and catcodes.

2 A brief overview of PythonTEX

Before describing new PythonTEX features, I will briefly summarize PythonTEX usage to provide context. General PythonTEX usage has been explored in greater detail previously in *TUGboat* [6] and elsewhere [10, 11].

Using PythonTEX involves loading the package in the preamble:

```
\usepackage{pythontex}
```

and modifying the compile process. As mentioned above, PythonTEX requires a three-step compile. For a document `doc.tex`, this might look like

```
pdflatex doc.tex
pythontex doc.tex
pdflatex doc.tex
```

The `pythontex` executable is typically installed along with the package when PythonTEX is installed with a TEX distribution's package manager. The second and third steps of the compile process are only necessary when code needs to be executed or highlighted. PythonTEX caches all results to maximize performance, and the `pythontex` executable will not actually do anything unless it detects changes.

2.1 Code typesetting

PythonTEX provides a `\pygment` command and a `pygments` environment for general code typesetting. These are similar to the `\mintinline` command and `minted` environment provided by the minted package. Colorizing is automatic (but grayscaled here for the printed *TUGboat*). For example,

```
Inline: \pygment{python}{var = "string"}
```

results in

Inline: `var = "string"`

The code may be delimited by a pair of curly braces, as shown, or by a single pair of identical characters (like `\verb`). Similarly,

```
\begin{pygments}{python}
def func(var):
    return var**2
\end{pygments}
```

produces

```
def func(var):
    return var**2
```

Code typesetting may be customized using fancyvrb's `\fvset`, which applies document-wide options, or the `\setpygmentsfv` command, which restricts options to `\pygment` and `pygments`.

There are also language-specific commands and environments that do not need the language to be specified. For example, `\pyv` and `pyverbatim` could be substituted in the examples above if "`{python}`" were deleted. Typesetting may be customized with `\fvset` or `\setpythontexfv`.

2.2 Code execution

There is a `\pyc` command and a `pycode` environment that may be used to execute Python code. Similar commands and environments exist for Ruby, Octave, Sage, Bash, and Rust. By default, anything that is printed or written to `stdout` will automatically be included in the document, just as if it had been saved in an external file and then brought in via `\input`. For example,

```
\begin{pycode}
print("Python says hello to \\tug!")
\end{pycode}
```

produces

Python says hello to TUG!

Notice that by default printed text is interpreted as normal LATEX input, not as verbatim.

There is also a `\py` command for conveniently inserting string representations of Python expressions in a document. It would be possible to use the `\pyc` command for this purpose. For example, to insert the value of 2^8 into the document, this would suffice:

```
\pyc{print(2**8)}
```

However, `\py` is more convenient:

```
\py{2**8}
```

It is also possible to use `\py` to insert the value of a previously defined variable. For instance, if `\pyc{x = 2**8}` had been used previously to set the value of `x`, then `\py{x}` would produce 256.

2.3 Code typesetting and execution

There is a `\pyb` command and a `pyblock` environment that both typeset and execute code. Anything printed or written to `stdout` by the code is not automatically inserted in the document, since it might not be desirable to have typeset code immediately next to its output. Instead, anything printed by the most recent command or environment may be inserted using the `\printpythontex` command.

3 An introduction to fvextra

The fancyvrb package was first publicly released in 1998 at version 2.5. A few bugs were fixed and a few features added later that year in version 2.6. Since then, the documentation lists two bug fixes, with version 2.8 released in 2010. The stability of fancyvrb speaks to its success as a fancy verbatim package.

I released the first version of the fvextra package at the end of June 2016. The package focuses on adding features to fancyvrb that improve code typesetting, especially when used with syntax highlighting provided by Pygments. fvextra also implements a few patches to the fancyvrb internals and makes a few changes to default fancyvrb behavior. All patches and changes to defaults are detailed in the fvextra documentation. At the end of July 2016, I released PythonTEX version 0.15 and minted 2.4. These require the fvextra package and support all features described below.

3.1 Single quotation marks

By default, LaTeX verbatim commands and environments convert the backtick (`) and single typewriter quotation mark (') into the left and right curly single quotation marks (‘’). This behavior carries over into fancyvrb. In typeset code, these characters should be represented literally. This is typically accomplished by manually loading the upquote package [1].

That approach has two drawbacks. First, not using upquote by default means that it is easily forgotten. I have had the experience myself of reviewing the final proofs of a paper, only to realize that I had forgotten to load upquote. Second, when upquote is loaded, obtaining the curly quotation marks is inconvenient if they are ever legitimately desired in a verbatim context.

The fvextra package requires upquote, so that the default behavior is correct for typesetting code. It also defines a new `curlyquotes` option that restores the default LaTeX behavior. For example, using fancyvrb's `Verbatim` environment,

```
\begin{Verbatim}[curlyquotes]
`Single quoted text'
\end{Verbatim}
```

yields

```
‘Single quoted text’
```

This eliminates one of the most common mistakes in code typesetting while still providing convenient access to the normal LaTeX behavior.

3.2 Math in verbatim

The fancyvrb package allows typeset mathematics to be embedded within verbatim material. Pygments builds on this with its `mathescape` option, which enables typeset math within code comments. That can be useful when typesetting code that implements mathematical or scientific algorithms.

A close examination of fancyvrb's typeset math within verbatim reveals that the result differs from normal math mode. Spaces are significant and appear literally, rather than vanishing. The `\text` command provided by the amstext package [7] and loaded as part of amsmath [8] uses the verbatim font rather than the normal document font. The single quotation mark (') causes an error rather than becoming a prime (that is, being converted into `^\prime`). fvextra modifies typeset math within verbatim so that all of these behave as expected. For example,

```
\begin{Verbatim}[commandchars=\\\{\},
                 mathescape]
Verbatim $x^2 + f_\text{sub}(x) = g''(x)$
\end{Verbatim}
```

now correctly produces

Verbatim $x^2 + f_{\text{sub}}(x) = g''(x)$

The `commandchars` option used in this example is defined by fancyvrb and allows macros within otherwise verbatim material. The `mathescape` option is a new feature added by fvextra that serves as a shortcut for giving the dollar sign, underscore, and caret their normal math-related meanings. When fvextra's `mathescape` is used with code highlighted by Pygments, it reduces to Pygments' `mathescape`, only producing typeset math in comments.

3.3 Tabs and tab expansion

By default, fancyvrb converts tabs into a fixed number of spaces, which may be controlled with the `tabsize` option. It also offers tab expansion to tab stops with the `obeytabs` option.

Tab expansion involves a clever recursive algorithm. Each tab character causes everything that precedes it in the current line of text to be saved in a box, and the width of the box is compared to the tab stop size to determine the needed width for the current tab. (For those who would like more details, this is defined in the `\FV@@ObeyTabs` and `\FV@TrueTab` macros in `fancyvrb.sty`.)

The tab expansion algorithm works excellently in normal verbatim contexts. Unfortunately, it also fails spectacularly (and silently) when tabs occur within macro arguments, which is common in Pygments output. In a multiline string or comment that is indented with tabs, `obeytabs` typically causes all lines except the first and the last to vanish, with no errors or warnings.

fvextra patches tab expansion so that it will never cause lines to vanish. Tab expansion for tabs that are only preceded by spaces or tabs is always correct, even for tabs that are in macro arguments. This covers the most common case of tabs used for indentation. Unfortunately, tab expansion is not guaranteed to be correct for tabs within macro arguments that are preceded by non-tab, non-space characters. The limitations of the new tab expansion algorithm are discussed in detail in the documentation.

Tabs are also improved in fvextra with the addition of the `tab` and `tabcolor` options. For example,

```
\begin{Verbatim}[showtabs,
                 tab=\rightarrowfill,
                 tabcolor=orange]
        A tab-indented line of text
\end{Verbatim}
```

produces

```
————→A tab-indented line of text
```

The original fancyvrb treatment of visible tabs was modified so that variable-width symbols such as `\rightarrowfill` expand to fill the full tab width.

3.4 Line highlighting

When writing about code, it can be useful to highlight a specific line or range of lines based on line numbers. As far as I know, fvextra is the first LaTeX package to implement this with its `highlightlines` and `highlightcolor` options. For example,

```
\begin{Verbatim}[numbers=left,
                 highlightlines={1, 3-4}]
First line
Second line
Third line
Fourth line
Fifth line
\end{Verbatim}
```

results in

```
1 First line
2 Second line
3 Third line
4 Fourth line
5 Fifth line
```

By default, a `\colorbox` that uses `highlightcolor` is inserted around specified lines. Additional customization is possible when desired. fvextra defines macros that are applied to the first, last, and inner lines in a range, as well as to isolated highlighted lines and to unhighlighted lines. These macros may be redefined to produce fancier highlighting.

3.5 Line breaking

The ability to automatically break long lines of code is perhaps the most important feature present in listings but missing in fancyvrb.

The fvextra package adds an option `breaklines` that enables line breaking. Line breaking is turned off by default, to ensure that the standard behavior of fancyvrb is unchanged by fvextra. Perhaps it will also encourage users to consider a smaller font size, inserting hard line breaks, or otherwise modifying code as an alternative to automatic line breaking.

By default, `breaklines` indents continuation lines to the same indentation level as the start of the line (adjustable via option `breakautoindent`), and then adds a small amount of extra indentation to make room for a line continuation symbol on the left (adjustable via `breaksymbolindentleft`). For instance,

```
A line that would eventually end up in the
 ↪ margin without breaklines
      An indented line that would be too
       ↪ long without breaklines
```

Many options are provided for customizing break indentation and break symbols. One possibility is to use custom break symbols on both the left and the right. Break symbols could be defined:

```
\newcommand{\symleft}{%
  \ensuremath{\Rightarrow}}
\newcommand{\symright}{\raisebox{-1ex}{%
  \rotatebox{30}{\ensuremath{\Leftarrow}}}}
```

Then the following options could be added:

```
breaklines,
breaksymbolleft=\symleft,
breaksymbolright=\symright
```

An example using these settings is shown below.

```
A line that would eventually end up in ↙
 ⇒ the margin without breaklines
```

By default, line breaks occur only at spaces when the `breaklines` option is used. Breaks may also be allowed anywhere (between non-space characters) by turning on the additional option `breakanywhere`. In many cases, however, simply breaking anywhere will not be acceptable. Two more options, `breakbefore` and `breakafter`, allow specific characters to be specified as break locations. For instance, setting

```
breakafter={+-=,}
```

allows breaks after any of the specified characters (`+-=,`). This could be useful for allowing breaks when spaces are not present while avoiding breaks within variable names. Special LaTeX characters such as the percent sign and number sign must be backslash-escaped when passed to `breakbefore` and `breakafter`. When a given character is specified as a potential break location, by default breaks will not be inserted between identical characters; rather, runs of identical characters are grouped. This may be modified with the `breakbeforegroup` and `breakaftergroup` options.

All of the breaking options discussed so far apply equally well to both normal verbatim text and highlighted computer code output by Pygments. fvextra also provides two line breaking options which are specific to Pygments output, and thus intended for the PythonTeX and minted packages. The `breakbytoken` option prevents line breaks from occurring within Pygments tokens, such as strings, comments, keywords, and operators. A complete list of Pygments tokens is available at `pygments.org/docs/tokens`. Breaks are still allowed at spaces outside tokens. The `breakbytoken` option could be used in a case like

```
var = "string 1" + "string 2" + "string 3"
```

to prevent breaks from occurring inside the strings, while still allowing breaks at spaces elsewhere.

There is also a `breakbytokenanywhere` option that prevents breaks within tokens, but allows breaks

between immediately adjacent tokens. This could be used in a case like

```
var = "string 1"+"string 2"+"string 3"
```

to prevent breaks within the strings while still allowing breaks before and after the plus signs.

4 Variable substitution and string interpolation

As mentioned in the introduction, one of the advantages of PythonTEX is that it allows macro programming that mixes LATEX with Python or another language. For instance, I could define a command that swaps the first and last characters in a string:

```
\newcommand{\swapfirstlast}[1]{%
  \pyc{s = "#1"}%
  \py{s[-1] + s[1:-1] + s[0]}}
```

This stores the argument of the command as a Python string, and then uses the character indices to swap the first and last characters. Invoking

```
\swapfirstlast{0123456789}
```

yields

9123456780

Such convenience is only possible because PythonTEX does not function as a preprocessor; LATEX handles all text before code is seen by Python (or another language) for evaluation, and then the result of evaluation is brought in during the next compile. In this case, LATEX macros are used to assemble Python code that is subsequently evaluated.

The downside of this approach is that it makes it more difficult to evaluate Python code in a verbatim or other special context. As should be expected,

```
\begin{Verbatim}
x = \py{2**16}
\end{Verbatim}
```

simply produces the literal text

```
x = \py{2**16}
```

Though that makes it convenient to write about PythonTEX, it certainly does make it more difficult to insert Python output in some situations.

In a case like this, it is possible to assemble all of the text as a Python template, and then print it:

```
\begin{pycode}
s = """
\\begin{{Verbatim}}
x = {x}
\\end{{Verbatim}}
"""
print(s.format(x=2**16))
\end{pycode}
```

That does give the desired result:

```
x = 65536
```

Unfortunately, a certain amount of complexity is required even for this simple case. The backslash must be escaped unless a raw string is used. Curly braces, which are of course everywhere in LATEX, must be doubled to appear literally when used with Python's string formatting.

To simplify these cases, PythonTEX now includes a `\pys` command and `pysub` environment that perform variable substitution or string interpolation. Equivalent commands and environments exist for Ruby, Octave, Sage, Bash, and Rust. Using the `pysub` environment, the last example becomes

```
\begin{pysub}
\begin{Verbatim}
x = !{2**16}
\end{Verbatim}
\end{pysub}
```

The content of the environment is passed verbatim to Python. Substitution fields take the form `!{`⟨*expression*⟩`}`. After ⟨*expression*⟩ is evaluated, a string representation of the result is returned to LATEX. If ⟨*expression*⟩ is simply a variable name, then it is replaced with a string representation of the variable value.

The form `!{`⟨*expression*⟩`}` was chosen because the exclamation point is one of the few ASCII punctuation characters without a special LATEX meaning. Using more common string interpolation syntax from other languages seemed unwise; `$`⟨*variable*⟩, `${`⟨*expression*⟩`}`, and `#{`⟨*expression*⟩`}` are constructs which commonly appear in LATEX. Likewise, using Python's string formatting syntax of `{`⟨*variable*⟩`}` would be problematic, since it would require all literal curly braces to be escaped by doubling.

The exact rules for delimiting and escaping `!{`⟨*expression*⟩`}` differ somewhat from standard LATEX syntax. If a literal exclamation point followed by an opening curly brace is desired, then the exclamation point is escaped by doubling (`!!`). A literal exclamation point only needs to be escaped when followed immediately by an opening curly brace. Curly braces never need to be escaped, since they only delimit a substitution field when they immediately follow an unescaped exclamation point.

If ⟨*expression*⟩ is delimited by a single pair of curly braces, `!{`⟨*expression*⟩`}`, then it may contain paired curly braces up to five levels deep. If the first or last character in ⟨*expression*⟩ would be a curly brace, then it must be separated from the delimiting braces by a space; leading and trailing spaces are stripped before ⟨*expression*⟩ is evaluated.

While ⟨*expression*⟩ will typically contain paired curly braces, there may be times when it does not. In these cases, it may be delimited by a sequence of curly braces up to six levels deep. Then ⟨*expression*⟩ must not contain an opening or closing sequence of the same depth as the delimiters, but may contain any combination of shorter sequences. For example, in

```
!{{{⟨expression⟩}}}
```

the ⟨*expression*⟩ could contain any combination of `{`, `}`, `{{`, or `}}`, paired or unpaired. It could not contain `{{{` or `}}}`, however; that would require delimiters of greater depth.

The `\pys` command is directly analogous to the `pysub` environment and follows the same rules. For instance,

```
\pys{\verb|x = !{2**32}|}
```

yields

```
x = 4294967296
```

Like the other PythonTEX commands, `\pys` takes an argument delimited by curly braces or by a single matched character, like `\verb`.

Both of the examples of `\pys` and `pysub` above involve verbatim. There are other situations in which they are useful. For example, in the `tikzpicture` environment provided by the tikz package, the `\py` command will typically conflict with tikz processing and result in an error. This may sometimes be avoided by using `\py` to output an entire line of tikz code, including the terminating semicolon, rather than just a snippet of text. The `pysub` environment provides a simpler alternative that avoids any guesswork regarding potential conflicts.

5 Conclusion

With the new fvextra package, it is now possible to typeset code using Pygments syntax highlighting without sacrificing advanced code typesetting features, such as line breaking. This should make PythonTEX (and minted) significantly better options for code typesetting in the future.

PythonTEX's code execution capabilities have also been improved by the new `\pys` command and `pysub` environment, and other commands and environments for variable substitution or string interpolation. These remove many of the remaining obstacles to document programming mixing LaTeX with Python, or with any of the other languages supported by PythonTEX, including Ruby, Octave, Sage, Bash, and Rust.

References

[1] Michael A. Covington, Frank Mittelbach, and Markus G. Kuhn. *upquote — upright-quote and grave-accent glyphs in verbatim.* `ctan.org/pkg/upquote`, 2012.

[2] Carsten Heinz, Brooks Moses, and Jobst Hoffmann. *The listings package.* `ctan.org/pkg/listings`, 2013.

[3] Marek Kubica. *The texments package.* `ctan.org/pkg/texments`, 2008.

[4] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in computational statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

[5] José Romildo Malaquias. *Testing the PygmenTEX package.* `ctan.org/pkg/pygmentex`, 2014.

[6] Andrew Mertz and William Slough. A gentle introduction to PythonTEX. *TUGboat*, 34(3):302–312, 2013. `tug.org/TUGboat/tb34-3/tb108mertz.pdf`.

[7] Frank Mittelbach and Rainer Schöpf. *The amstext package.* `ctan.org/pkg/amstext`, 2000.

[8] Frank Mittelbach, Rainer Schöpf, Michael Downes, and David M. Jones. *The amsmath package.* `ctan.org/pkg/amsmath`, 2016.

[9] Matti Pastell. *Pweave — reports from data with Python.* `mpastell.com/pweave`, 2010.

[10] Geoffrey M. Poore. Reproducible documents with PythonTEX. In Stéfan van der Walt, Jarrod Millman, and Katy Huff, editors, *Proc. of the 12th Python in Science Conference*, pages 78–84, 2013.

[11] Geoffrey M. Poore. PythonTEX: Reproducible documents with LaTeX, Python, and more. *Comp. Science & Discovery*, 8(1):014010, 2015.

[12] Geoffrey M. Poore. *The fvextra package.* `github.com/gpoore/fvextra`, 2016.

[13] Geoffrey M. Poore. *The pythontex package.* `github.com/gpoore/pythontex`, 2016.

[14] Geoffrey M. Poore and Konrad Rudolph. *The minted package: Highlighted source code in LaTeX.* `github.com/gpoore/minted`, 2016.

[15] The Pocoo Team. *Pygments: Python syntax highlighter.* `pygments.org`, 2016.

[16] Timothy Van Zandt, Denis Girou, Sebastian Rahtz, and Herbert Voß. *The 'fancyvrb' package: Fancy verbatims in LaTeX.* `ctan.org/pkg/fancyvrb`, 2010.

[17] Yihui Xie. *Dynamic Documents with R and knitr.* Chapman and Hall/CRC, Boca Raton, FL, 2013. ISBN 978-1482203530, `yihui.name/knitr`.

[18] Dejan Živković. *The verbments package: Pretty printing source code in LaTeX.* `ctan.org/pkg/verbments`, 2011.

◇ Geoffrey M. Poore
1050 Union University Dr.
Jackson, TN 38305
`gpoore (at) gmail dot com`
`https://github.com/gpoore/`