PythonTeX: Fast Access to Python from within LaTeX

Geoffrey M. Poore[‡*]

Abstract—PythonTeX is a new LaTeX package that provides access to the full power of Python from within LaTeX documents. It allows Python code entered within a LaTeX document to be executed, and provides access to the output. PythonTeX also provides syntax highlighting for any language supported by the Pygments highlighting engine.

PythonTeX is fast and user-friendly. Python code is separated into user-defined sessions. Each session is only executed when its code is modified. When code is executed, sessions run in parallel. The contents of stdout and stderr are synchronized with the LaTeX document, so that printed content is easily accessible and error messages have meaningful line numbering.

PythonTeX simplifies scientific document creation with LaTeX. Plots can be created with matplotlib and then customized in place. Calculations can be performed and automatically typeset with NumPy. SymPy can be used to automatically create mathematical tables and step-by-step mathematical derivations.

Index Terms—LaTeX, document preparation, document automation, matplotlib, NumPy, SymPy, Pygments

Introduction

Scientific documents and presentations are often created with the LaTeX document preparation system. Though some LaTeX tools exist for creating figures and performing calculations, external scientific software is typically required for these tasks. This can result in an inefficient workflow. Every time a calculation requires modification or a figure needs tweaking, the user must switch between LaTeX and the scientific software. The user must locate the code that created the calculation or figure, modify it, and execute it before returning to LaTeX.

One way to streamline this process is to include non-LaTeX code within LaTeX documents, with a means to execute this code and access the output. That approach has connections to Knuth's concept of literate programming, in which code and its documentation are combined in a single document [Knuth]. The noweb literate programming tool extended Knuth's work to additional document formats and arbitrary programming languages [Ramsey]. Sweave subsequently built on noweb by allowing the output of individual chunks of R code to be accessed within the document [Leisch]. This made possible dynamic reports that are reproducible since they contain the code that generated their results. As such, Sweave and similar tools represent an additional, complementary approach to reproducibility compared to makefile-based approaches [Schwab].

* *Corresponding author: gpoore@uu.edu*
‡ *Union University*

Several methods of including executable code in LaTeX documents ultimately function as preprocessors or templating systems. A document might contain a mix of LaTeX and code, and the preprocessor replaces the code with its output. The original document would not be valid LaTeX; only the preprocessed document would be. Sweave, knitr [Xie], the Python-based Pweave [Pastell], and template libraries such as Mako [MK] function in this manner. More recently, the IPython notebook has provided an interactive browser-based interface in which text, code, and code output may be interspersed [IPY]. Since the notebook can be exported as LaTeX, it functions similarly to the preprocessor-style approach.

The preprocessor/templating-style approach has a significant advantage. All of the examples mentioned above are compatible with multiple document formats, not just LaTeX. This is particularly true in the case of templating libraries. One significant drawback is that the line numbers of the preprocessed document, which LaTeX receives, do not correspond to those of the original document. This makes it difficult to debug LaTeX errors, particularly in longer documents. It also breaks standard LaTeX tools such as forward and inverse search between a document and its PDF (or other) output; only Sweave and knitr have systems to work around this. An additional issue is that it is difficult for LaTeX code to interact with code in other languages, when the code in other languages has already been executed and removed before LaTeX runs.

In an alternate approach to including executable code in LaTeX documents, the original document is valid LaTeX, containing code wrapped in special commands and environments. The code is extracted by LaTeX itself during compilation, then executed and replaced by its output. Such approaches with Python go back to at least 2007, with Martin R. Ehmsen's python.sty style file [Ehmsen]. Since 2008, SageTeX has provided access to the Sage mathematics system from within LaTeX [Drake]. Because Sage is largely based on Python, it also provides Python access. SympyTeX (2009) is based on SageTeX [Molteno]. Though SympyTeX is primarily intended for accessing the SymPy library for symbolic mathematics [SymPy], it provides general access to Python. Since these packages begin with a valid LaTeX document, they automatically work with standard LaTeX editing tools and also allow LaTeX code to interact with Python.

Python.sty, SageTeX, and SympyTeX illustrate the potential of a close Python-LaTeX integration. At the same time, they leave much of the possible power of the Python-LaTeX combination untapped. Python.sty requires that all Python code be executed every time the document is compiled. SageTeX and SympyTeX separate code execution from document compilation, but because all code is executed in a single session, everything must be executed whenever anything changes. None of these packages provides

comprehensive syntax highlighting. SageTeX and SympyTeX do not provide access to stdout or stderr. They do synchronize error messages with the document, but synchronization is performed by executing a `try/except` statement on every line of the user's code. This reduces performance and fails in the case of syntax errors.

PythonTeX is a new LaTeX package that provides access to Python from within LaTeX documents. It emphasizes performance and usability.

- Python-generated content is always saved, so that the LaTeX document can be compiled without running Python.
- Python code is divided into user-defined sessions. Each session is only executed when it is modified. When code is executed, sessions run in parallel.
- Both stdout and stderr are easily accessible.
- All Python error messages are synchronized with the LaTeX document, so that error line numbers correctly correspond to the document.
- Code may be typeset with highlighting provided by Pygments [Pyg]—this includes any language supported by Pygments, not just Python. Unicode is supported.
- Native Python code is fully supported, including imports from `__future__`. No changes to Python code are required to make it compatible with PythonTeX.

While PythonTeX lacks the rapid interactivity of the IPython notebook, as a LaTeX package it offers much tighter Python-LaTeX integration. It also provides greater control over what is displayed (code, stdout, or stderr) and allows executable code to be included inline within normal text.

This paper presents the main features of PythonTeX and considers several examples. It also briefly discusses the internal workings of the package.

PythonTeX environments and commands

PythonTeX provides four LaTeX environments and four LaTeX commands for accessing Python. These environments and commands save code to an external file and then bring back the output once the code has been processed by PythonTeX.

The **code** environment simply executes the code it contains. By default, any printed content is brought in immediately after the end of the environment and interpreted as LaTeX code. For example, the LaTeX code

```
\begin{pycode}
myvar = 123
print('Greetings from Python!')
\end{pycode}
```

creates a variable `myvar` and prints a string, and the printed content is automatically included in the document:

Greetings from Python!

The **block** environment executes its contents and also typesets it. By default, the typeset code is highlighted using Pygments. Reusing the Python code from the previous example,

```
\begin{pyblock}
myvar = 123
print('Greetings from Python!')
\end{pyblock}
```

creates

```
myvar = 123
print('Greetings from Python!')
```

The printed content is not automatically included. Typically, the user wouldn't want the printed content immediately after the typeset code—explanation of the code, or just some space, might be desirable before showing the output. Two equivalent commands are provided for including the printed content generated by a block environment: `\printpythontex` and `\stdoutpythontex`. These bring in any printed content created by the most recent PythonTeX environment and interpret it as LaTeX code. Both commands also take an optional argument to bring in content as verbatim text. For example, `\printpythontex[v]` brings in the content in a verbatim form suitable for inline use, while `\printpythontex[verb]` brings in the content as a verbatim block.

All code entered within code and block environments is executed within the same Python session (unless the user specifies otherwise, as discussed below). This means that there is continuity among environments. For example, since `myvar` has already been created, it can now be modified:

```
\begin{pycode}
myvar += 4
print('myvar = ' + str(myvar))
\end{pycode}
```

This produces

myvar = 127

The **verb** environment typesets its contents, without executing it. This is convenient for simply typesetting Python code. Since the verb environment has a parallel construction to the code and block environments, it can also be useful for temporarily disabling the execution of some code. Thus

```
\begin{pyverb}
myvar = 123
print('Greetings from Python!')
\end{pyverb}
```

results in the typeset content

```
myvar = 123
print('Greetings from Python!')
```

without any code actually being executed.

The final environment is different. The **console** environment emulates a Python interactive session, using Python's `code` module. Each line within the environment is treated as input to an interactive interpreter. The LaTeX code

```
\begin{pyconsole}
myvar = 123
myvar
print('Greetings from Python!')
\end{pyconsole}
```

creates

```
>>> myvar = 123
>>> myvar
123
>>> print('Greetings from Python!')
Greetings from Python!
```

PythonTeX provides options for showing and customizing a banner at the beginning of console environments. The content of all console environments is executed within a single Python session, providing continuity, unless the user specifies otherwise.

While the PythonTeX environments are useful for executing and typesetting large blocks of code, the PythonTeX commands are intended for inline use. Command names are based on abbreviations of environment names. The **code** command simply executes its contents. For example, `\pyc{myvar = 123}`. Again, any printed content is automatically included by default. The **block** command typesets and executes the code, but does not automatically include printed content (`\printpythontex` is required). Thus, `\pyb{myvar = 123}` would typeset

```
myvar = 123
```

in a form suitable for inline use, in addition to executing the code. The **verb** command only typesets its contents. The command `\pyv{myvar = 123}` would produce

```
myvar=123
```

without executing anything. If Pygments highlighting for inline code snippets is not desired, it may be turned off.

The final inline command, `\py`, is different. It provides a simple way to typeset variable values or to evaluate short pieces of code and typeset the result. For example, `\py{myvar}` accesses the previously created variable `myvar` and brings in a string representation: 123. Similarly, `\py{2**8 + 1}` converts its argument to a string and returns 257.

It might seem that the effect of `\py` could be achieved using `\pyc` combined with `print`. But `\py` has significant advantages. First, it requires only a single external file per document for bringing in content, while `print` requires an external file for each environment and command in which it is used. This is discussed in greater detail in the discussion of PythonTeX's internals. Second, the way in which `\py` converts its argument to a valid LaTeX string can be specified by the user. This can save typing when several conversions or formatting operations are needed. The examples below using SymPy illustrate this approach.

All of the examples of inline commands shown above use opening and closing curly brackets to delimit the code. This system breaks down if the code itself contains an unmatched curly bracket. Thus, all inline commands also accept arbitrary matched characters as delimiters. This is similar to the behavior of LaTeX's `\verb` macro. For example, `\pyc!myvar = 123!` and `\pyc#myvar = 123#` are valid. No such consideration is required for environments, since they are delimited by `\begin` and `\end` commands.

Options: Sessions and Fancy Verbatims

PythonTeX commands and environments take optional arguments. These determine the session in which the code is executed and provide additional formatting options.

By default, all code and block content is executed within a single Python session, and all console content is executed within a separate session. In many cases, such behavior is desired because of the continuity it provides. At times, however, it may be useful to isolate some independent code in its own session. A long calculation could be placed in its own session, so that it only runs when its code is modified, independently of other code.

PythonTeX provides such functionality through user-defined sessions. All commands and environments take a session name as an optional argument. For example, `\pyc[slowsession]{myvar = 123}` and

```
\begin{pycode}[slowsession]
myvar = 123
print('Greetings from Python!')
\end{pycode}
```

Each session is only executed when its code has changed, and sessions run in parallel (via Python's `multiprocessing` package), so careful use of sessions can significantly increase performance.

All PythonTeX environments also accept a second optional argument. This consists of settings for the LaTeX `fancyvrb` (Fancy Verbatims) package [FV], which PythonTeX uses for typesetting code. These settings allow customization of the code's appearance. For example, a block of code may be surrounded by a colored frame, with a title. Or line numbers may be included.

Plotting with matplotlib

The PythonTeX commands and environments can greatly simplify the creation of scientific documents and presentations. One example is the inclusion of plots created with matplotlib [MPL].

All of the commands and environments discussed above begin with the prefix `py`. PythonTeX provides a parallel set of commands and environments that begin with the prefix `pylab`. These behave identically to their `py` counterparts, except that matplotlib's `pylab` module is automatically imported via `from pylab import *`. The `pylab` commands and environments can make it easier to keep track of code dependencies and separate content that would otherwise require explicit sessions; the default `pylab` session is separate from the default `py` session.

Combining PythonTeX with matplotlib significantly simplifies plotting. The commands for creating a plot may be included directly within the LaTeX source, and the plot may be edited in place to get the appearance just right. Matplotlib's LaTeX option may be used to keep fonts consistent between the plot and the document. The code below illustrates this approach. Notice that the plot is created in its own session, to increase performance.

```
\begin{pylabcode}[plotsession]
rc('text', usetex=True)
rc('font', **{'family':'serif', 'serif':['Times']})
rc('font', size=10.0)
rc('legend', fontsize=10.0)
x = linspace(0, 3*pi)
figure(figsize=(3.25,2))
plot(x, sin(x), label='$\sin(x)$')
plot(x, sin(x)**2, label='$\sin^2(x)$',
     linestyle='dashed')
xlabel(r'$x$-axis')
ylabel(r'$y$-axis')
xticks(arange(0, 4*pi, pi), ('$0$',
       '$\pi$', '$2\pi$', '$3\pi$'))
axis([0, 3*pi, -1, 1])
legend(loc='lower right')
savefig('myplot.pdf', bbox_inches='tight')
\end{pylabcode}
```

The plot may be brought in and positioned using the standard LaTeX commands:

```
\begin{figure}
\centering
\includegraphics{myplot}
\caption{\label{fig:matplotlib} A plot
created with PythonTeX.}
\end{figure}
```

The end result is shown in Figure 1.

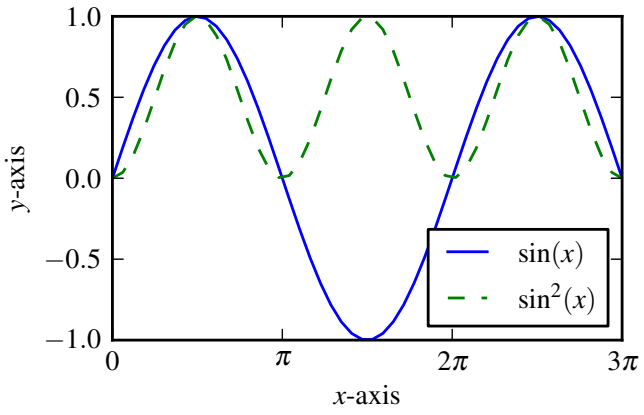

Fig. 1: *A matplotlib plot created with PythonTeX.*

Solving equations with NumPy

PythonTeX didn't require any special modifications to the Python code in the previous example with matplotlib. The code that created the plot was the same as it would have been had an external script been used to generate the plot. In some situations, however, it can be beneficial to acknowledge the LaTeX context of the Python code. This may be illustrated by solving an equation with NumPy [NP].

Perhaps the most obvious way to solve an equation using PythonTeX is to separate the Python solving from the LaTeX typesetting. Consider finding the roots of a polynomial using NumPy.

```
\begin{pylabcode}
coeff = [4, 2, -4]
r = roots(coeff)
\end{pylabcode}

The roots of $4x^2 + 2x - 4 = 0$ are
$\pylab{r[0]}$ and $\pylab{r[1]}$.
```

This yields

The roots of $4x^2+2x-4=0$ are -1.2807764064 and 0.780776406404.

Such an approach works, but the code must be modified significantly whenever the polynomial changes. A more sophisticated approach automatically generates the LaTeX code and perhaps rounds the roots as well, for an arbitrary polynomial.

```
\begin{pylabcode}
coeff = [4, 2, -4]
# Build a string containing equation
eq = ''
for n, c in enumerate(coeff):
    if n == 0 or str(c).startswith('-'):
        eq += str(c)
    else:
        eq += '+' + str(c)
    if len(coeff) - n - 1 == 1:
        eq += 'x'
    elif len(coeff) - n - 1 > 1:
        eq += 'x^' + str(len(coeff) - n - 1)
eq += '=0'
# Get roots and format for LaTeX
r = ['{0:+.3f}'.format(root)
     for root in roots(coeff)]
latex_roots = ','.join(r)
\end{pylabcode}

The roots of $\pylab{eq}$ are
$[\pylab{latex_roots}]$.
```

This yields

The roots of $4x^2+2x-4=0$ are $[-1.281,+0.781]$.

The automated generation of LaTeX code on the Python side begins to demonstrate the full power of PythonTeX.

Solving equations with SymPy

Several examples with SymPy further illustrate the potential of Python-generated LaTeX code [SymPy].

To simplify SymPy use, PythonTeX provides a set of commands and environments that begin with the prefix `sympy`. These are identical to their `py` counterparts, except that SymPy is automatically imported via `from sympy import *`.

SymPy is ideal for PythonTeX use, because its `LatexPrinter` class and the associated `latex()` function provide LaTeX representations of objects. For example, returning to solving the same polynomial,

```
\begin{sympycode}
x = symbols('x')
myeq = Eq(4*x**2 + 2*x - 4)
print('The roots of the equation ')
print(latex(myeq, mode='inline'))
print(' are ')
print(latex(solve(myeq), mode='inline'))
\end{sympycode}
```

creates

The roots of the equation $4x^2+2x-4=0$ are $\left[-\frac{1}{4}\sqrt{17}-\frac{1}{4},\quad -\frac{1}{4}+\frac{1}{4}\sqrt{17}\right]$

Notice that the printed content appears as a single uninterrupted line, even though it was produced by multiple prints. This is because the printed content is interpreted as LaTeX code, and in LaTeX an empty line is required to end a paragraph.

The `\sympy` command provides an alternative to printing. While the `\py` and `\pylab` commands attempt to convert their arguments directly to a string, the `\sympy` command converts its argument using SymPy's `LatexPrinter` class. Thus, the output from the last example could also have been produced using

```
\begin{sympycode}
x = symbols('x')
myeq = Eq(4*x**2 + 2*x - 4)
\end{sympycode}

The roots of the equation $\sympy{myeq}$
are $\sympy{solve(myeq)}$.
```

The `\sympy` command uses a special interface to the `LatexPrinter` class, to allow for context-dependent `LatexPrinter` settings. PythonTeX includes a utilities class, and an instance of this class called `pytex` is created within each PythonTeX session. The `formatter()` method of this class is responsible for converting objects into strings for `\py`, `\pylab`, and `\sympy`. In the case of SymPy, `pytex.formatter()` provides an interface to `LatexPrinter`, with provision for context-dependent customization. In LaTeX, there are four possible math styles: displaystyle (regular equations), textstyle (inline), scriptstyle (superscripts and subscripts), and scriptscriptstyle (superscripts and subscripts, of superscripts and subscripts). Separate

`LatexPrinter` settings may be specified for each of these styles individually, using a command of the form

```
pytex.set_sympy_latex(style, **kwargs)
```

For example, by default `\sympy` is set to create normal-sized matrices in displaystyle and small matrices elsewhere. Thus, the following code

```
\begin{sympycode}
m = Matrix([[1,0], [0,1]])
\end{sympycode}

The matrix in inline is small:  $\sympy{m}$

The matrix in an equation is of normal size:
\[ \sympy{m} \]
```

produces

The matrix in inline is small: $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$
The matrix in an equation is of normal size:

$$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

As another example, consider customizing the appearance of inverse trigonometric functions based on their context.

```
\begin{sympycode}
x = symbols('x')
sineq = Eq(asin(x/2)-pi/3)
pytex.set_sympy_latex('display',
                      inv_trig_style='power')
pytex.set_sympy_latex('text',
                      inv_trig_style='full')
\end{sympycode}

Inline:  $\sympy{sineq}$

Equation:  \[ \sympy{sineq} \]
```

This creates

Inline: $\arcsin\left(\frac{1}{2}x\right) - \frac{1}{3}\pi = 0$
Equation:

$$\sin^{-1}\left(\frac{1}{2}x\right) - \frac{1}{3}\pi = 0$$

Notice that in both examples above, the `\sympy` command is simply used—no information about context must be passed to Python. On the Python side, the context-dependent `LatexPrinter` settings are used to determine whether the LaTeX representation of some object is context-dependent. If not, Python creates a single LaTeX representation of the object and returns that. If the LaTeX representation is context-dependent, then Python returns multiple LaTeX representations, wrapped in LaTeX's `\mathchoice` macro. The `\mathchoice` macro takes four arguments, one for each of the four LaTeX math styles display, text, script, and scriptscript. The correct argument is typeset by LaTeX based on the current math style.

Step-by-step derivations with SymPy

With SymPy's LaTeX functionality, it is simple to automate tasks that could otherwise be tedious. Instead of manually typing step-by-step mathematical solutions, or copying them from an external program, the user can generate them automatically from within LaTeX.

```
\begin{sympycode}
x, y = symbols('x, y')
f = x + sin(y)
step1 = Integral(f, x, y)
step2 = Integral(Integral(f, x).doit(), y)
step3 = step2.doit()
\end{sympycode}

\begin{align*}
\sympy{step1} &= \sympy{step2} \\
              &= \sympy{step3}
\end{align*}
```

This produces

$$\begin{aligned}\iint x+\sin(y)\,dx\,dy &= \int \frac{1}{2}x^2 + x\sin(y)\,dy\\ &= \frac{1}{2}x^2 y - x\cos(y)\end{aligned}$$

Automated mathematical tables with SymPy

The creation of mathematical tables is another traditionally tedious task that may be automated with PythonTeX and SymPy. Consider the following code, which automatically creates a small integral and derivative table.

```
\begin{sympycode}
x = symbols('x')
funcs = ['sin(x)', 'cos(x)', 'sinh(x)', 'cosh(x)']
ops = ['Integral', 'Derivative']
print('\\begin{align*}')
for func in funcs:
    for op in ops:
        obj = eval(op + '(' + func + ', x)')
        left = latex(obj)
        right = latex(obj.doit())
        if op != ops[-1]:
            print(left + '&=' + right + '&')
        else:
            print(left + '&=' + right + r'\\')
print('\\end{align*}')
\end{sympycode}
```

$$\begin{aligned}\int \sin(x)\,dx &= -\cos(x) & \frac{\partial}{\partial x}\sin(x) &= \cos(x)\\ \int \cos(x)\,dx &= \sin(x) & \frac{\partial}{\partial x}\cos(x) &= -\sin(x)\\ \int \sinh(x)\,dx &= \cosh(x) & \frac{\partial}{\partial x}\sinh(x) &= \cosh(x)\\ \int \cosh(x)\,dx &= \sinh(x) & \frac{\partial}{\partial x}\cosh(x) &= \sinh(x)\end{aligned}$$

This code could easily be modified to generate a page or more of integrals and derivatives by simply adding additional function names to the `funcs` list.

Debugging and access to stderr

PythonTeX commands and environments save the Python code they contain to an external file, where it is processed by PythonTeX. When the Python code is executed, errors may occur. The line numbers for these errors do not correspond to the document line numbers, because only the Python code contained in the document is executed; the LaTeX code is not present. Furthermore, the error line numbers do not correspond to the line numbers that would be obtained by only counting the Python code in

the document, because PythonTeX must execute some boilerplate management code in addition to the user's code. This presents a challenge for debugging.

PythonTeX addresses this issue by tracking the original LaTeX document line number for each piece of code. All error messages are parsed, and Python code line numbers are converted to LaTeX document line numbers. The raw stderr from the Python code is interspersed with PythonTeX messages giving the document line numbers. For example, consider the following code, with a syntax error in the last line:

```
\begin{pyblock}[errorsession]
x = 1
y = 2
z = x + y +
\end{pyblock}
```

The error occurred on line 3 of the Python code, but this might be line 104 of the actual document and line 47 of the combined code and boilerplate. In this case, running the PythonTeX script that processes Python code would produce the following message, where `<temp file name>` would be the name of a temporary file that was executed:

```
* PythonTeX code error on line 104:
    File "<temp file name>", line 47
      z = x + y +
                ^
  SyntaxError: invalid syntax
```

Thus, finding code error locations is as simple as it would be if the code were written in separate files and executed individually. PythonTeX is the first Python-LaTeX solution to provide such comprehensive error line synchronization.

In general, errors are something to avoid. In the context of writing about code, however, they may be created intentionally for instructional purposes. Thus, PythonTeX also provides access to error messages in a form suitable for typesetting. If the PythonTeX package option `stderr` is enabled, any error message created by the most recent PythonTeX command or environment is available via `\stderrpythontex`. By default, stderr content is brought in as LaTeX verbatim content; this preserves formatting and prevents issues caused by stderr content not being valid LaTeX.

Python code and the error it produces may be typeset next to each other. Reusing the previous example,

```
\begin{pyblock}[errorsession]
x = 1
y = 2
z = x + y +
\end{pyblock}
```

creates the following typeset code:

```
x = 1
y = 2
z = x + y +
```

The stderr may be brought in via `\stderrpythontex`:

```
  File "<file>", line 3
    z = x + y +
              ^
SyntaxError: invalid syntax
```

Two things are noteworthy about the form of the stderr. First, in the case shown, the file name is given as `"<file>"`. PythonTeX provides a package option `stderrfilename` for controlling this name. The actual name of the temporary file that was executed may be shown, or simply a name based on the session (`"errorsession.py"` in this case), or the more generic `"<file>"` or `"<script>"`. Second, the line number shown corresponds to the code that was actually entered in the document, not to the document line number or to the line number of the code that was actually executed (which would have included PythonTeX boilerplate). To accomplish this, PythonTeX parses the stderr and corrects the line number, so that the typeset code and the typeset stderr are in sync.

General code highlighting with Pygments

The primary purpose of PythonTeX is to execute Python code included in LaTeX documents and provide access to the output. Once support for Pygments highlighting of Python code was added [Pyg], however, it was simple to add support for general code highlighting.

PythonTeX provides a `\pygment` command for typesetting inline code snippets, a `pygments` environment for typesetting blocks of code, and an `\inputpygments` command for bringing in and highlighting an external file. All of these have a mandatory argument that specifies the Pygments lexer to be used. For example, `\pygment{latex}{\pygment}` produces

```
\pygment
```

in a form suitable for inline use while

```
\begin{pygments}{python}
def f(x):
    return x**3
\end{pygments}
```

creates

```
def f(x):
    return x**3
```

The `pygments` environment and the `\inputpygments` command accept an optional argument containing `fancyvrb` settings.

As far as the author is aware, PythonTeX is the only LaTeX package that provides Pygments highlighting with Unicode support under the standard pdfTeX engine. The `listings` package [LST], probably the most prominent non-Pygments highlighting package, does support Unicode—but only if the user follows special procedures that could become tedious. PythonTeX requires no special treatment of Unicode characters, so long as the `fontenc` and `inputenc` packages are loaded and used correctly. For example, PythonTeX can correctly highlight the following snippet copied and pasted from a Python 3 console session, without any modification.

```
>>> var1 = 'âæéöø'
>>> var2 = 'ßçñðŠ'
>>> var1 + var2
'âæéöøßçñðŠ'
```

Implementation

A brief overview of the internal workings of PythonTeX is provided below. For additional details, please consult the documentation.

When a LaTeX document is compiled, the PythonTeX commands and environments write their contents to a single shared

external file. The command and environment contents are interspersed with delimiters, which contain information about the type of command or environment, the session in which the code is to be executed, the document line number where the code originated, and similar tracking information. A single external file is used to minimize the number of temporary files created, and because TeX has a very limited number of output streams.

During compilation, each command and environment also checks for any Python-generated content that belongs to it, and brings in this content if it exists. Python-generated content is brought in via LaTeX macros and via separate external files. At the beginning of the LaTeX document, the PythonTeX package brings in two files of LaTeX macros that were created on the Python side, if these files exist. One file consists of macros containing the Python content accessed by `\py`, `\pylab`, and `\sympy`. The other file contains highlighted Pygments content. The files are separate for performance reasons. In addition to content that is brought in via macros, content may be brought in via separate external files. Each command or environment that uses the print statement/function must bring in an external file containing the printed content. The printed content cannot be brought in as LaTeX macros, because in general printed content need not be valid LaTeX code. In contrast, `\py`, `\pylab`, and `\sympy` should return valid LaTeX, and of course Pygments-highlighted content is valid LaTeX as well.

On the Python side, the file containing code and delimiters must be processed. All code is hashed, to determine what has been modified since the previous run so that only new and modified code may be executed. Code that must be executed is divided by session, and each session (plus some PythonTeX management code) is saved to its own external file. The highlighting settings for Pygments content are compared with the settings for the last run, to determine what needs to be highlighted again with new settings.

Next, Python's `multiprocessing` package is used to perform all necessary tasks. Each of the session code files is executed within a separate process. The process executes the file, parses the stdout into separate files of printed content based on the command or environment from which it originated, and parses the stderr to synchronize it with the document line numbers. If specified by the user, a modified version of the stderr is created and saved in an external file for inclusion in the document via `\stderrpythontex`. Two additional processes are used, one for highlighting code with Pygments and one for evaluating and highlighting all console content (using Python's `code` module).

Finally, all LaTeX macros created by all processes are saved in one of two external files, depending on whether they contain general content or content highlighted by Pygments (again, this is for performance reasons). All information that will be needed the next time the Python side runs is saved. This includes the hashes for each session. Any session that produced errors is automatically set to be executed the next time the Python side runs. A list of all files that were automatically created by PythonTeX is also saved, so that future runs can clean up outdated and unused files.

PythonTeX consists of a LaTeX package and several Python scripts. A complete compilation cycle for a PythonTeX document involves running LaTeX to create the file of code and delimiters, running the PythonTeX script to create Python content, and finally running LaTeX again to compile the document with Python-generated content included. Since all Python-generated content is saved, the PythonTeX script only needs to be run when the document's PythonTeX commands or environments are modified. By default, all files created by PythonTeX are kept in a subdirectory within the document directory, keeping things tidy.

Conclusion

PythonTeX provides access to the full power of Python from within LaTeX documents. This can greatly simplify the creation of scientific documents and presentations.

One of the potential drawbacks of using a special LaTeX package like PythonTeX is that publishers may not support it. Since PythonTeX saves all Python-generated content, it already provides document compilation without the execution of any Python code, so that aspect will not be an issue. Ideally, a PythonTeX document and its Python output could be merged into a single, new document that does not require the PythonTeX package. This feature is being considered for an upcoming release.

PythonTeX provides many features not discussed here, including a number of formatting options and methods for adding custom code to all sessions. PythonTeX is also under active development. For additional information and the latest code, please visit https://github.com/gpoore/pythontex.

References

[Leisch] F. 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. http://www.statistik.lmu.de/~leisch/Sweave/.

[Ehmsen] M. R. Ehmsen. "Python in LaTeX." http://www.ctan.org/pkg/python.

[Drake] D. Drake. "The SageTeX package." https://bitbucket.org/ddrake/sagetex/.

[Molteno] T. Molteno. "The sympytex package." https://github.com/tmolteno/SympyTeX/.

[SymPy] SymPy Development Team. "SymPy." http://sympy.org/.

[Pyg] The Pocoo Team. "Pygments: Python Syntax Highlighter." http://pygments.org/.

[FV] T. Van Zandt, D. Girou, S. Rahtz, and H. Voß. "The 'fancyvrb' package: Fancy Verbatims in LaTeX." http://www.ctan.org/pkg/fancyvrb.

[MPL] J. D. Hunter. *Matplotlib: A 2D Graphics Environment*, in Computing in Science & Engineering, Vol. 9, No. 3. (2007), pp. 90-95. http://matplotlib.sourceforge.net/.

[NP] Numpy developers. "NumPy." http://numpy.scipy.org/.

[LST] C. Heinz and B. Moses. "The Listings Package." http://www.ctan.org/tex-archive/macros/latex/contrib/listings/.

[IPY] The IPython development team. "The IPython Notebook." http://ipython.org/notebook.html.

[Pastell] M. Pastell. "Pweave - reports from data with Python." http://mpastell.com/pweave/.

[Knuth] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. Stanford, California: Center for the Study of Language and Information, 1992.

[Ramsey] N. Ramsey. *Literate programming simplified*. IEEE Software, 11(5):97-105, September 1994. http://www.cs.tufts.edu/~nr/noweb/.

[Schwab] M. Schwab, M. Karrenbach, and J. Claerbout. *Making scientific computations reproducible*. Computing in Science & Engineering, 2(6):61-67, Nov/Dec 2000.

[Xie] Y. Xie. "knitr: Elegant, flexible and fast dynamic report generation with R." http://yihui.name/knitr/.

[MK] M. Bayer. "Mako Templates for Python." http://www.makotemplates.org/.