

PAPER

PythonTeX: reproducible documents with LaTeX, Python, and more

To cite this article: Geoffrey M Poore 2015 *Comput. Sci. Discov.* **8** 014010

View the [article online](#) for updates and enhancements.

You may also like

- [Studying a physics problem with the help of open source software](#)
Andrea Mandanici
- [LaTeX allergy](#)
Mark Blamire
- [pygiftgenerator: a python module designed to prepare Moodle-based quizzes](#)
Jon Sáenz, Idoia G Gurtubay, Zumbeltz Izaola et al.

PythonTeX: reproducible documents with LaTeX, Python, and more

Geoffrey M Poore

Department of Physics, Union University, Jackson, TN 38305, USA

E-mail: gpoore@uu.edu

Received 31 January 2014, revised 16 July 2014

Accepted for publication 4 May 2015


Published 30 July 2015

Computational Science & Discovery **8** (2015) 014010

doi:[10.1088/1749-4699/8/1/014010](https://doi.org/10.1088/1749-4699/8/1/014010)

Abstract

PythonTeX is a LaTeX package that allows Python code in LaTeX documents to be executed and provides access to the output. This makes possible reproducible documents that combine results with the code required to generate them. Calculations and figures may be next to the code that created them. Since code is adjacent to its output in the document, editing may be more efficient. Since code output may be accessed programmatically in the document, copy-and-paste errors are avoided and output is always guaranteed to be in sync with the code that generated it. This paper provides an introduction to PythonTeX and an overview of major features, including performance optimizations, debugging tools, and dependency tracking. Several complete examples are presented. Finally, advanced features are summarized. Though PythonTeX was designed for Python, it may be extended to support additional languages; support for the Ruby and Julia languages is already included. PythonTeX contains a utility for converting documents into plain LaTeX, suitable for format conversion, sharing, and journal submission.

 Online supplementary data available from stacks.iop.org/csd/8/014010/mmedia

Keywords: Python, LaTeX, reproducibility, dynamic report generation, Ruby, Julia

1. Introduction

Although the concept of reproducible research has received increasing attention in recent years (see for example [1]), computational reproducibility is a long-standing concern [2]. There have

been two major approaches to making scientific and technical documents reproducible. In the first approach, a reproducible document contains results that may be reproduced via a makefile or similar system [2]. Madagascar [3], VisTrails [4], and similar software are more recent and sophisticated examples of this general strategy. This type of document is reproducible in the sense that its contents are the product of a reproducible workflow. The actual writing process is relatively similar to the unreproducible case, except that it is easier to ensure that figures and other results are up-to-date.

In the second approach to reproducible documents, executable analysis code is embedded in the document. This is common among users of the R language and environment for statistical analysis. Sweave [5] has allowed R to be embedded in LaTeX [6] documents since 2002. Its mixing of code and text has roots in literate programming, through noweb [7] back to Knuth's original concept [8]. knitr [9] has provided a similar but more powerful system for embedding R in LaTeX, markdown, HTML, and several other formats since its release in 2011. It also provides limited support for several additional languages. This type of document is reproducible in the sense that it actually contains some or all of the code required to generate results (typically from external data). The writing process can be very different from the unreproducible case, because document and code may be tightly integrated. For example, numerical results may be brought into the document's text via code, eliminating copy-and-pasting that could introduce errors. Or the document may constitute a dynamic report, automatically 'rewriting' itself to accommodate a given external dataset.

These two approaches to reproducible documents need not be mutually exclusive. Though the embedded-code approach is impractical for research involving an extensive codebase, it might still be useful to embed code that creates figures or performs final calculations. Similarly, even when all code could be embedded, it may be more practical to externalize some tasks via makefiles or a similar system. It is even possible to use a sort of hybrid of the two approaches (for example, Dexy [10]).

This paper introduces PythonTeX, an open-source Python-based system for creating reproducible LaTeX documents. Though PythonTeX primarily follows the embedded-code approach to reproducible documents, it also incorporates aspects of the makefile-style approach. PythonTeX focuses on the Python language [11] due to the many mathematical, scientific, and visualization libraries that are available [12–14]. It also supports Ruby [15] and Julia [16], and may be extended to support additional languages. Readers should be aware that I wrote this paper using PythonTeX—and used it to create the plain LaTeX source that I submitted for publication.

Python tools for reproducible documents.

A number of Python-based tools exist for creating reproducible documents with embedded code.

- Sphinx [17] was designed for creating Python documentation. Depending on the extensions employed, it can execute embedded Python code to provide at least basic reproducibility.
- Pweave [18] is essentially a Python version of Sweave. It allows Python to be embedded in reST, Sphinx, LaTeX, and markdown documents. Like Sweave and knitr, Pweave works by extracting all code from a document, executing the code, and then creating a copy of the original document in which code is replaced by its output. This makes compatibility with a range of document formats relatively simple, but typically does not provide close integration with any one document format.

- The IPython [19] notebook provides a browser-based, interactive interface in which code, output, and text may be combined. The notebook works with a single primary programming language, which is usually Python (although Ruby, Julia, and Haskell variants are in development). The Python-based notebook has the ability to execute independent snippets of code in several additional languages. The recently added `nbconvert` utility can convert the notebook into static formats including HTML, LaTeX, markdown, and reStructuredText.
- The Python package [20] for LaTeX provides a very minimal system for executing independent chunks Python code and including the output. It was the first LaTeX package for running Python code (2007).
- The SageTeX package [21] for LaTeX, released in 2008, provides a full-featured system for reproducible documents using the Sage [22] mathematics software, which is largely built on Python. The SymPyTeX package [23] (2009) is largely derivative and works with Python rather than Sage. SageTeX provided inspiration for PythonTeX when I started development in the spring of 2011.

Only Pweave, IPython, and SageTeX have a relatively complete built-in feature set for reproducible documents with embedded code. PythonTeX was designed to provide features that are missing in some or all of these programs, with a focus on ease of use, performance, and superior LaTeX integration.

- Embedded code may be divided into independent user-defined sessions. When code is executed, sessions run in parallel. Different sessions may use different languages. Pweave, IPython, and SageTeX all work with a single primary session (and thus a single primary language), which can be inefficient for performing independent calculations.
- Built-in utilities are provided for tracking dependencies such as data. Code is re-executed when dependencies change. Utilities are also included for tracking created files such as figures. When file names are changed or files are no longer used, old versions are automatically cleaned up.
- Errors and warnings are synchronized with the LaTeX document, so that the user knows not just where in the code things went wrong, but also where that code is located in the document. Synchronization is less of an issue in IPython due to its interactive nature. SageTeX includes synchronization, but its system is based on Python's `try/except` exception handling and thus fails whenever errors prevent execution (such as syntax errors). Pweave lacks synchronization.
- As a LaTeX package, PythonTeX allows users to write valid LaTeX documents in which the full power of LaTeX is immediately accessible, rather than hybrid documents that contain LaTeX markup. Unlike in IPython and Pweave, it is simple and straightforward to pass information such as page dimensions from LaTeX into Python. It is even possible to create LaTeX macros that mix LaTeX with other languages.

Over the last few years the IPython notebook has arguably become the dominant tool for reproducible documents with Python. PythonTeX lacks the notebook's interactivity and simple compatibility with multiple output formats, but it does have some significant advantages. The notebook does not have built-in support for hiding code or for treating code output as part of the normal text of the document. It also does not allow variable values to be accessed

within text cells. PythonTeX allows code, code output, or both to be displayed. Code output is interpreted as LaTeX by default (with built-in options for showing output verbatim), so it is easy to generate parts of the document text using code. Variable values may be accessed within normal text; inline executable code is also allowed. Writing a paper with IPython to conform to a journal's LaTeX standards may require editing the default templates and ensuring that the converted LaTeX output is acceptable. With PythonTeX, this is very simple since everything is written in LaTeX directly. Also, since the document is written entirely as valid LaTeX, any LaTeX editor may be used. This can significantly aid the writing process, since many editors provide forward and inverse search between the document source and the PDF output, autocompletion for references and citations, citation information on mouse hover, and similar features. Such tools are not available when writing in the IPython notebook.

2. Basic examples

2.1. *Hello, world!*

I will begin with several basic examples of PythonTeX usage. The LaTeX source for most of the examples in this paper is available as supplementary materials. First, consider a 'Hello, world!' document, `hello.tex`.

```
\documentclass{article}
\usepackage{pythontex}
\begin{document}

\begin{pycode}
print(r'\emph{Hello}, world!')
\end{pycode}
\end{document}
```

Using PythonTeX simply involves adding the `\usepackage{pythontex}` command in the preamble. (PythonTeX is part of TeX Live [24], version 2013 and later; the package also includes an installation script for other TeX distributions.) All code in the `pycode` environment will be executed, and anything that is printed will be brought into the document automatically. Compiling the document requires a three-step process:

- (i) Run LaTeX. For example, `pdflatex hello.tex`. PythonTeX is compatible with the pdfTeX, XeTeX, and LuaTeX engines, so the document may be compiled with the `latex`, `pdflatex`, `xelatex`, or `lualatex` commands. In this step, the PythonTeX package saves all code, along with information about where it was located in the document, to a temporary file.
- (ii) Run the PythonTeX script. In a typical installation, this may be done with the command `pythontex hello.tex`. In this step, code is loaded from the temporary file and executed, and its output is saved in a form accessible to LaTeX.
- (iii) Run LaTeX again. All code output is brought in to create the final document.

In this case, the resulting PDF document contains the following text: *'Hello, world!'*

Notice that the printed content is interpreted as LaTeX code; the `\emph` command does not appear as literal text but rather acts to emphasize the word ‘Hello’. Also be aware that the three-step compile process is *only* necessary when code needs to be executed. For example, if I were to add some normal LaTeX text to the document, running LaTeX once would be sufficient to update the PDF.

Using the `pycode` environment is sufficient if I simply want to execute code. In some situations, I might want to show the code in the document as well. In this case, the `pyblock` environment may be used. Its contents are executed and the code is typeset in the document verbatim (with optional syntax highlighting via Pygments [25]). Any printed content is not automatically included, however, since it might be desirable to insert additional text or LaTeX commands between the typeset code and its output. Instead, printed content may be accessed via the `\printpythontex` command. Thus, the following code

```
\begin{pyblock}
print(r'\emph{Hello}, world!')
\end{pyblock}
\printpythontex
```

produces

```
print(r'\emph{Hello}, world!')
Hello, world!
```

For completeness, a `pyverbatim` environment is also included. It typesets code with optional syntax highlighting, but executes nothing.

2.2. Preventing copy-and-paste errors

Copy-and-pasting numerical values is a significant source of potential error when writing a document. To prevent this, PythonTeX includes commands that may be used within normal text to access variable values. Suppose the following code is used to find the length of the hypotenuse of a right triangle.

```
\begin{pycode}
from math import *
a = 3.0
b = 4.0
c = sqrt(a**2 + b**2)
\end{pycode}
```

If I later wish to refer to the values of the variables, I may use the `\py` command, which returns a string representation of whatever it is given. The LaTeX code

When $a = \text{\py{a}}$ and $b = \text{\py{b}}$, then $c = \text{\py{c}}$.

results in

When $a = 3.0$ and $b = 4.0$, then $c = 5.0$.

It is also possible to use `\py` to perform simple calculations and return the result inline. For example, `\py{2**16}` yields 65536.

Command versions of the three environments are provided as well. The `\pyc` command executes code. Anything that is printed will be included automatically by default. The `\pyb`

command executes code and also typesets it, with optional syntax highlighting. Anything that is printed may be accessed via `\printpythontex`. Finally, the `\pyv` command only typesets code, with optional highlighting; nothing is executed. All of these commands (including `\py`) take an argument delimited by a pair of curly braces or by a pair of matched symbols (like LaTeX's `\verb`), so that character escaping may be avoided. Thus, `\py{x}` and `\py#x#` would both be valid.

Accessing variable values may be less straightforward when complex calculations are involved. The code required may be long enough that it is not practical or efficient to include it within the LaTeX document. In such cases, the code may be developed in an external file (perhaps as a module) and then imported. This can keep the code that is actually in the document at a manageable length. External code could also be executed using the `subprocess` module, though that will typically be less desirable. Any external code can still be typeset in the document if desired, using the `\VerbatimInput` command from `fancyvrb` [26], the `\inputpygments` command provided by `PythonTeX`, or similar commands from other packages for typesetting source code.

2.3. Reproducible interactive console sessions

Another potential source of error when writing a document is interactive console sessions. If an author fails to copy the complete session, code will be missing. If a library changes, necessitating new syntax, console examples will become outdated, and updating them may be tedious.

To prevent this, `PythonTeX` provides a `pyconsole` environment. All Python commands entered within the environment are treated as the input to an interactive console session, via Python's `code` module. Since the code is actually executed, and consists only of commands, it is easy to make modifications to keep examples current and correct errors.

The following commands could be used to recalculate the hypotenuse of the right triangle.

```
\begin{pyconsole}
from math import *
a = 3.0
b = 4.0
c = sqrt(a**2 + b**2)
c
\end{pyconsole}
```

This is the result:

```
>>> from math import *
>>> a = 3.0
>>> b = 4.0
>>> c = sqrt(a**2 + b**2)
>>> c
5.0
```

As before, it is possible to access variable values via a command; in this case, `\pycon` must be used. Additional special-purpose commands and environments for working with console code are also provided.

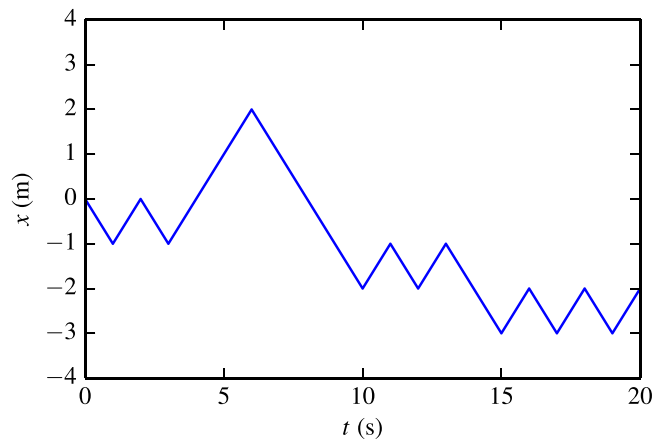


Figure 1. A plot of a random walk, created with PythonTeX. The random walker moved between -3.0 and 2.0 on the x -axis. The walker crossed the point $x = 0$ on 2 separate occasions.

3. Additional examples

3.1. Plotting and automated analysis

PythonTeX can simplify figures by allowing plots to be adjacent to the code that created them. And data analysis can be automated. Suppose I have a data file, `rand_walk.csv`, that gives the trajectory of a random walk:

```
t(s),0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
x(m),0,-1,0,-1,0,1,2,1,0,-1,-2,-1,-2,-1,-2,-3,-2,-3,-2,-3,-2
```

This could be plotted and some basic analysis performed with the following code. The code assumes that the data file is located in the current working directory, which is the document directory by default. A different working directory may be specified using the `\setpython-texworkingdir` command in the document preamble, or using Python's `os.chdir()` within the code. The result is shown in figure 1.

```
\begin{pycode}
from matplotlib import pyplot as plt
from matplotlib import rc

with open('rand_walk.csv') as f:
    data = f.readlines()
t = [float(t_n) for t_n in data[0].split(',') [1:]]
x = [float(x_n) for x_n in data[1].split(',') [1:]]

rc('text', usetex=True)
rc('font', family='serif', serif='Times', size=10)
```

```

(Continued.)
plt.figure(figsize=(4, 2.5))
plt.plot(t, x)
plt.xlabel('$t$ (s)')
plt.ylabel('$x$ (m)')
plt.ylim(-4, 4)
plt.savefig('figure1.pdf', bbox_inches='tight')
plt.savefig('figure1.eps', bbox_inches='tight')

zerocross = 0
for n, x_n in enumerate(x):
    if n != 0 and n != len(x) and x_n == 0 and x[n-1] != x[n+1]:
        zerocross += 1
\end{pycode}

\begin{figure}[h!]
\centering
\includegraphics{figure1}
\caption{\label{fig:random-walk} A plot of a random walk,
created with PythonTeX. The random walker moved between
 $\py{\min(x)}$  and  $\py{\max(x)}$  on the  $x$ -axis. The
walker crossed the point  $x=0$  on  $\py{zerocross}$ 
separate occasions.}
\end{figure}

```

The advantage of this approach is that it is very convenient to edit the plot. For instance, if a dimension needs to be adjusted or a color changed, there is no need to search for an external file of code that created the plot, open that file, edit the code, and then execute it. Instead, the plot code may be edited in the document, and then the document may be recompiled with PythonTeX.

This example also illustrates how documents can be programmed to adapt automatically to whatever external data is provided. The figure caption gives the minimum and maximum positions reached by the random walker on the x -axis, as well as the number of times that the walker crossed $x = 0$. These values are calculated from the data, so they would automatically adjust if data for a different random walk were used. Such automation would be particularly beneficial in cases when reports must be generated for several similar data sets.

3.2. Automated mathematics

Another application is the automatic creation of mathematical tables or derivations. The following example uses SymPy [13], a Python library for symbolic mathematics, to perform symbolic differentiation. SymPy's `Derivative` class creates an unevaluated derivative; the `doit()` method actually evaluates the derivative. The `latex()` function returns a LaTeX version of its argument. Since SymPy uses an italic d for differentials, but IOP journals use a Roman d , all d 's are replaced by the `\rmd` macro.

```

\begin{pycode}
from sympy import *
x = symbols('x')
print('\begin{eqnarray*}')
for func in [sin(x), sinh(x), csc(x)]:
    left = Derivative(func, x)
    right = Derivative(func, x).doit()
    eq = latex(left) + '=' + latex(right) + '\\\\'
    print(eq.replace('d', '\\rm{d}'))
print('\end{eqnarray*}')
\end{pycode}

```

$$\frac{d}{dx} \sin(x) = \cos(x)$$

$$\frac{d}{dx} \sinh(x) = \cosh(x)$$

$$\frac{d}{dx} \csc(x) = -\cot(x) \csc(x)$$

It is also possible to use SymPy to perform step-by-step solutions. In the case below, a double integral is evaluated in two steps. SymPy's `Integral` class creates an unevaluated integral, while the `doit()` method evaluates the integral.

```

\begin{pycode}
from sympy import *
x, y = symbols('x, y')
func = x**2*sin(y)**2
step1 = Integral(func, x, y)
step2 = Integral(Integral(func, y).doit(), x)
step3 = step1.doit()
print('\begin{eqnarray*}')
print(latex(step1).replace('d', '\\rm{d}') + '&=')
print(latex(step2).replace('d', '\\rm{d}') + '\\\\')
print('&= ' + latex(step3).replace('d', '\\rm{d}'))
print('\end{eqnarray*}')
\end{pycode}

```

$$\iint x^2 \sin^2(y) \, dx \, dy = \int x^2 \left(\frac{y}{2} - \frac{1}{2} \sin(y) \cos(y) \right) dx$$

$$= \frac{x^3}{3} \left(\frac{y}{2} - \frac{1}{2} \sin(y) \cos(y) \right).$$

3.3. Code examples with errors

PythonTeX can include error messages next to the code that created them. This can be particularly useful for documentation or instructional purposes. When PythonTeX is loaded with the `makestderr` option (`\usepackage[makestderr]{pythontex}`), versions of error messages appropriate for inclusion in the document are saved. (This involves the system

described in section 6.) These may then be accessed via the `\stderrpythontex` command. Consider this example.

```
Code:
\begin{pyblock}
a = 1
b = a + 2 +
\end{pyblock}
Error:
\underline{\stderrpythontex}
```

The output is shown below.

```
Code:
a = 1
b = a + 2 +
Error:
  File "<file>", line 2
    b = a + 2 +
                ^
SyntaxError: invalid syntax
\underline{}
```

PythonTeX provides a `stderrfilename` package option that allows the form of the file name that appears in error messages to be customized.

3.4. LaTeX macro programming

It is possible to program LaTeX macros that mix LaTeX with other languages. This is one area where PythonTeX's close LaTeX integration is particularly evident. Mixing languages can be vastly simpler than programming an equivalent with pure LaTeX.

LaTeX commands are created with code in the form

```
\newcommand{\<cmd name>}[<no. of args>][<default 1st arg>]
{<body>}
```

In the `<body>` definition, arguments are referred to as `#1`, `#2`, etc. The easiest way to use Python in LaTeX macros is to use the `\py` command. For example, the following code creates a `\reverse` command that reverses the order of whatever text it is given, using Python's string indexing.

```
\newcommand{\reverse}[1]{\py{"#1"[::-1]}}
```

Using `\reverse{1234567890}` yields `0987654321`.

Notice that the `#1` is wrapped in quotation marks in the command definition, so that Python will treat it as a string; otherwise, it would be interpreted as a variable or a number. Also keep in mind that the `#1` argument could have been passed to a function (perhaps defined in a previous pycode environment).

This approach to macro programming with PythonTeX will work in most cases except when literal percent and hash characters are involved, since these have a special meaning in LaTeX. Working around those cases can require LaTeX programming techniques involving verbatim arguments, catcodes, and tokenization that are beyond the scope of this paper.

4. The PythonTeX utilities class

PythonTeX includes a Python utilities class. An instance of this class called `pytex` is always created before any user code is executed, so that it is always available. The utilities class provides many options for customizing PythonTeX behaviour. Some of the key features are summarized below.

- Complete information about where in the document code originated, what type of command or environment it came from, etc is available as class attributes. This allows code to customize output based on its origin.
- The `before()` and `after()` methods are called before and after each chunk of user code is executed. These methods do nothing by default; they are included to be redefined by the user. For example, specific LaTeX commands could always be printed at the beginning and end of each environment. Or the `after()` method could check to see if any figures were created by the current chunk of user code, and if so, automatically save the figures and include them in the document. This makes possible significant automation. Details about working with these methods are provided in the main documentation [27].
- The `formatter()` method determines how `\py`'s argument is converted to a string. Redefining this method is particularly useful when working with a package like SymPy. SymPy's objects have a string representation, but they can also be converted to a LaTeX string via the `latex()` function. If `formatter()` is redefined to use `latex()`, then SymPy objects brought in via `\py` will automatically be in LaTeX form.

The utilities class also has a `context` attribute that deserves special attention. It serves as a gateway for passing contextual information such as page dimensions from the LaTeX side to the Python side. By default, no contextual information is available. The user defines what is passed via the `\setpythontexcontext` macro, which takes a comma-separated list of key-value pairs. For example, this paper had the following command in the preamble when I wrote it.

```
\setpythontexcontext{textwidth = \the\textwidth,
                    textheight = \the\textheight}
```

This allowed me to access the values of LaTeX's `\textwidth` and `\textheight` in Python. Since `pytex.context` is a dictionary that also has named attributes, these values may be accessed in either of the following ways:

```
pytex.context['textheight']
pytex.context.textheight
```

When I was writing this document, these contained the string `'672.0pt'`. (By default, everything in `context` is a string; PythonTeX provides a basic system for specifying the type of what is passed.) Since LaTeX dimensions are in units of TeX points, the utilities class provides methods for performing unit conversions.

The `context` attribute may be particularly useful when working with figures. Figures may be created on the Python side so that their dimensions automatically adjust to fit page margins on the LaTeX side.

Three additional methods provided by the utilities class are discussed in section 5.2.

5. Optimizing Performance

5.1. Sessions

By default, all code for a given language is executed in a single session. This is what allows a command like `\py` to access variables defined in a separate environment like `pycode`. As a result, whenever any code is modified, everything must be re-executed. That can quickly degrade performance when several independent, computationally intensive operations must be performed.

To address this, PythonTeX allows code to be divided into user-defined sessions. All commands and environments take an optional argument that specifies the session in which the code is executed. A slow calculation might be isolated in its own session like this:

```
\begin{pycode}[slow]
<code>
\end{pycode}
```

Later, the results could be accessed via `\py[slow]{<variable>}` or a similar command.

Separating independent calculations into their own sessions not only prevents code from being executed unnecessarily. It also allows additional performance gains. Whenever two or more sessions need to be executed, they automatically run in parallel on systems with multiple processors, using Python's `multiprocessing` module.

One disadvantage of dividing independent calculations into separate sessions is that any user code created for interacting with LaTeX (perhaps using the tools in section 4) is no longer shared. Such code could be copied into each session, but that would just create one more thing to get out of sync. For such cases, PythonTeX provides a `pythontexcus-tomcode` environment. This allows code to be added to all sessions of a given type. For example,

```
\begin{pythontexcus-tomcode}{py}
<code>
\end{pythontexcus-tomcode}
```

would add `<code>` to all sessions that use the commands and environments whose names begin with `py` (`pycode`, `\py`, etc).

5.2. File tracking

The embedded-code approach to reproducible documents can make it easy to update, say, a figure. The code for creating the figure can be right next to where the figure is included in the document. But what happens if the code for creating the figure relies on data in an external file, and the data is updated? More generally, how can external dependencies be tracked? It would be possible to re-execute all code to make sure that everything is up-to-date (using the methods described in section 5.3), but that could be slow and inefficient. This is a case where the makefile-style approach to reproducible documents might seem more attractive.

PythonTeX includes features for dependency tracking, so that at least in many situations, the use of an external makefile or similar system can be avoided. The PythonTeX `utilities` class (section 4) includes an `add_dependencies()` method that allows dependencies to be specified. For example, in Python dependencies may be specified as follows:

```
pytex.add_dependencies('data1.txt', 'data2.txt', 'data3.txt')
```

Once a dependency has been specified, PythonTeX will check it for modification on all subsequent runs. By default, modification is checked via file modification time (Python's `os.path.getmtime()`), since this is fast even for large files. An SHA-1 hash may be used instead via the `hashdependencies` package option.

Tracking dependencies addresses external files that are read. There is a related issue with external files that are written. Suppose a plot is created. Later, the plot's file name is changed, leaving the old file with the old name behind. Many unused files could easily accumulate in such a fashion. To avoid this, the utilities class provides an `add_created()` method. Whenever code is re-executed, all tracked files that it created on previous runs are deleted.

It is relatively straightforward to automate the tracking of dependencies and created files. For example, the PythonTeX utilities class includes a custom `open()` method. If a file is opened for reading, it is added as a dependency and then opened; if it is opened for writing, it is added as a created file and then opened. When such custom functions are used for working with files, `add_dependencies()` and `add_created()` need not be used explicitly.

5.3. Caching and controlling execution

PythonTeX automatically caches all code output. This means that when no code needs to be executed, running PythonTeX (followed by LaTeX) is unnecessary; if PythonTeX is run under such conditions, it determines that there is nothing to do and exits immediately. Thus, as long as no code needs to be executed, PythonTeX adds essentially no overhead to compiling a document.

By default, PythonTeX executes code whenever it is new or modified, or if it produced errors when it was last executed. While that approach is often desirable, there may be times when it is inconvenient. For instance, if one session is producing errors, but the user is currently working on code in another session, it might be best only to execute modified code. This may be configured using the package option `rerun` (`\usepackage[rerun=<choice>]{pythontex}`). Five different thresholds for execution may be specified:

- `never`: never execute code. PythonTeX will issue a warning if there is modified code.
- `modified`: only execute code that has been modified (including code with modified dependencies; see section 5.2).
- `errors`: execute all modified code and all code that produced errors on the last run.
- `warnings`: execute all modified code and all code that produced errors or warnings on the last run.
- `always`: always execute code.

6. Debugging

Debugging can be a challenge in the embedded-code approach to reproducible documents. Consider the sample document below, which contains a syntax error in the second line of Python code (trailing '+').

```

\documentclass{article}
\usepackage{pythontex}
\begin{document}

\begin{pycode}
a = 1
b = a + 2 +
\end{pycode}

\end{document}

```

The syntax error is on line 2 of the code. But when the code is executed, the error will actually be reported on a much later line, because PythonTeX inserts user code into a template that begins with configuration code¹. Furthermore, users typically will not care about where in the *code* an error occurred; it is much more relevant to know where in the *document* the error was triggered.

PythonTeX addresses these debugging issues by returning an annotated version of all errors and warnings. For the document above, the PythonTeX output would include a section like this:

```

* PythonTeX stderr - error on line 7:
  File "<outputdir>\py_default_default.py", line 54
    b = a + 2 +
              ^
SyntaxError: invalid syntax

```

Each message begins with a line that tells where in the document the error occurred; in this case, it was line 7. This is followed by the raw error message generated by the code. In this case, the error occurred on line 54 of the code that was actually executed. (Remember that PythonTeX inserts user code into a template; this accounts for the first 52 lines.)

Even more complex situations are correctly synchronized. For example, suppose the *pycode environment* is saved in a file `code.tex`:

```

\begin{pycode}
a = 1
b = a + 2 +
\end{pycode}

```

Then we bring this into the main document, `main.tex`:

```

\documentclass{article}
\usepackage{pythontex}
\begin{document}

\input{code.tex}

\end{document}

```

¹ The insertion of user code into a template would seem to prevent the user from importing from Python's `__future__` module, since such imports are required to come before any other code. PythonTeX actually parses the beginning of user code and relocates these imports to the appropriate place.

PythonTeX is still able to synchronize the error message. In this case, it also specifies the file in which the error occurred, since it was not the main file:

```
* PythonTeX stderr - error on line 3 in "code.tex":
  File "<outputdir>\py_default_default.py", line 54
    b = a + 2 +
              ^
SyntaxError: invalid syntax
```

Synchronizing `stderr` is accomplished through a combination of two methods. When PythonTeX runs, it creates a record of where each chunk of user code was originally located in the document, and where it is inserted into the template code that is actually executed. In most cases, this is sufficient for calculating which document line number corresponds to which code line number. There are situations, however, when this approach fails. For example, when warnings are raised by Python's `warnings.warn()` in an imported module, the warning will only refer to the line number in the module, not to the line in the user's code where the module was accessed. So there is no line number to synchronize. To deal with these cases, PythonTeX writes a delimiter to `stderr` before each chunk of code is executed. This allows errors and warnings to be traced back to the code in a single PythonTeX command or environment even in the worst-case scenario.

While synchronized `stderr` is helpful, there may be times when a more powerful approach is needed. PythonTeX has a command-line option `--interactive` that runs a specified session (or the default session, if none is specified) in interactive mode. (Console sessions are not currently supported.) Both `stdout` and `stderr` are normally redirected to a file for parsing into a form suitable for inclusion in the document. In interactive mode, they are directed to the terminal instead, and user input is allowed. This provides the necessary conditions for running an interactive debugger. For example, the Python debugger `pdb` could be imported within a `pycode` environment at the beginning of a session, and then `pdb.set_trace()` could be used to enter the debugger at a desired point.

Commands that launch a debugger are problematic when code runs in standard mode. User input is not possible in standard mode and thus the presence of debugging commands would cause PythonTeX to freeze while waiting for input that is impossible. To address this, PythonTeX passes the command-line argument `--interactive` to code that is executed in interactive mode. The presence of this argument may then be used to invoke a debugger conditionally. For example,

```
if '--interactive' in sys.argv[1:]:
    pdb.set_trace()
```

PythonTeX also provides an approach to debugging that does not require any modification of the document source. The command-line option `--debug` passes a specified session (or the default session, if none is specified) to a debugger. This is analogous to running `python -m pdb <script>`. The default debugger is `syncpdb` [28]. `syncpdb` is a wrapper for `pdb` that is aware of the connections between the code and the document; it was specifically created for PythonTeX. Whenever `pdb` would give a file name and line number for the code that is being executed, `syncpdb` also gives the corresponding document name and line number (assuming the code did not originate in the template into which user code is inserted). Many `pdb` commands take a line number or a file name plus line number as an argument. In `syncpdb`,

these commands also take arguments with a percent symbol (%) prefix. If the prefix is present, then the file name and line number are interpreted as referring to the document, rather than the code. For example, the `pdb` command `break 70` would set a breakpoint in the code at *code* line 70. With `syncpdb`, `break %70` would set a breakpoint at the code that came from line 70 in the main *document*. If the code came from an inputted file `input.tex`, then `break %input.tex:70` would be used instead.

An example of part of a `syncpdb` session is shown below. This was produced by taking the example document from the beginning of this section and removing the trailing '+' so that the syntax error would not interrupt the debugger. Note that the user code begins on line 6 of this document. The path to the temporary script that is executed has been abbreviated for clarity. Line numbers for both the document and the executed code are shown by `syncpdb`.

```
(Pdb) break %6
Breakpoint 1 at ../py_default_default.py:53 (main.tex:6)
(Pdb) list %6,7
main.tex:
    6   53 B      a = 1
    7   54      b = a + 2
```

7. De-PythonTeX: removing PythonTeX dependence

A potential disadvantage of a LaTeX package like PythonTeX is that documents written with it cannot be compiled without it. Some journals place limitations on the packages that manuscripts may use. Likewise, utilities for converting LaTeX documents to other formats (for example, Pandoc [29]) often support only a subset of standard LaTeX commands, much less a package like PythonTeX.

To deal with these situations, PythonTeX includes a `depythontex` utility that creates a copy of a document in which all PythonTeX content has been replaced by a plain LaTeX equivalent. Consider a simple file `unconverted.tex`.

```
\documentclass{article}
\usepackage[depythontex]{pythontex}
\begin{document}

\begin{pyblock}
print(r'\emph{Hello}, world!')
\end{pyblock}
\printpythontex

\begin{pyconsole}
var = 2
var**4
\end{pyconsole}
\end{document}
```

As long as this document has been compiled with the `depythontex` package option, then it can be converted to plain LaTeX with a command such as the following:

depythontex --listing=fancyvrb -o converted.tex unconverted.tex
The result, converted.tex, is shown below.

```
\documentclass{article}
\usepackage{fancyvrb}
\begin{document}

\begin{Verbatim}
print(r'\emph{Hello}, world!')
\end{Verbatim}
\emph{Hello}, world!
\begin{Verbatim}
>>> var = 2
>>> var**4
16
\end{Verbatim}

\end{document}
```

By default, all code that was originally typeset by PythonTeX will be typeset in the converted document using LaTeX's `\verb` and `verbatim`, since this is most universal. In this example, I have used `depythontex`'s `--listing` option to specify that the `fancyvrb` package [26] should be used instead. It provides a more powerful `\Verb` and `Verbatim`. Notice that the `\usepackage{pythontex}` was removed, and `\usepackage{fancyvrb}` inserted. PythonTeX is no longer needed, but now `fancyvrb` is.

The `depythontex` utility is also compatible with the popular `listings` [30] and `minted` [31] LaTeX packages for syntax highlighting. When one of these is specified with the `--listing` option, any syntax highlighting as well as line numbering will carry over to the converted document. This allows the converted document to contain everything present in the original.

Currently, the `depythontex` utility is only officially compatible with built-in PythonTeX commands and environments. User-defined LaTeX code that uses these internally is not officially supported, but that is an objective for future development.

8. Support for additional languages

Although I created PythonTeX for Python, from the beginning I have tried to make the design as language-agnostic as possible, so that other languages may be supported. PythonTeX currently provides built-in support for the Ruby [15] and Julia [16] languages. This is nearly identical to Python support. Interactive console sessions are not yet available. The Python commands and environments all begin with the prefix `py`. Similarly, Ruby uses either `rb` or `ruby`, and Julia uses either `j1` or `julia`. Ruby and Julia support must be loaded with the `usefamily` package option. Enabling the `ruby` and `julia` families of commands and environments would require the following command:

```
\usepackage[usefamily={ruby,julia}]{pythontex}
```

Ruby and Julia versions of *'Hello, world!'* would then be

```
\begin{rubycode}
puts "\\emph{Hello}, world!"
\end{rubycode}
```

and

```
\begin{juliacode}
println("\\emph{Hello}, world!")
\end{juliacode}
```

An instance of the utilities class is always available in both languages. It is called `rbtex` in Ruby and `jltex` in Julia, with almost all of the same methods and attributes as the Python utilities class.

PythonTeX support for additional languages is based on a template system. Adding full basic support for Ruby and Julia involved templates with approximately 100 and 150 lines of code, respectively. The `stderr` synchronization algorithm also required modification to accommodate the different forms of error messages. The template system and `stderr` synchronization code will be streamlined in the future to simplify the process of adding support for more languages.

9. Conclusion

PythonTeX makes possible reproducible LaTeX documents in which Python, Ruby, and Julia code is embedded. This can mitigate the potential for copy-and-paste errors, simplify the creation of figures, and allow significant portions of documents to be generated automatically. Built-in utilities for dependency tracking reduce the need for makefiles or similar systems. Tight LaTeX integration allows code to adapt to its context in a document. User-defined sessions that run in parallel provide high performance. Synchronization of errors and warnings with document line numbers mean that writing and coding remain efficient. Finally, the `depythontex` utility ensures that PythonTeX may be used even when plain LaTeX documents are needed.

Several features may be improved in the future. LaTeX macro programming would benefit from additional tools, including `depythontex` support for user macros. The template system for adding additional languages will need refinement as more languages are added. While thresholds for code execution may currently be set at the document level, it would be nice to be able to do this at the session level.

More broadly, although PythonTeX's tight LaTeX integration is one of its strengths, it also constrains PythonTeX as lightweight markup languages such as markdown [32] become increasingly popular. Meanwhile, programs that do support multiple document formats, such as `knitr` and `Pweave`, use an approach that limits what is possible with LaTeX. A major objective for future development is refactoring PythonTeX to separate the LaTeX-related functionality from a core that manages code execution. This will allow PythonTeX to maintain its LaTeX integration while adding support for additional document formats.

Acknowledgments

Thanks to Nicholas Lu Chee Seng for assistance in testing the earliest versions of PythonTeX. Thanks to Øystein Bjørndal for several suggestions and for assistance with OS X compatibility. Thanks to William G Nettles for suggestions and helpful discussions.

References

- [1] Yale Law School Roundtable on Data and Code Sharing 2010 Reproducible research *Comput. Sci. Eng.* **12** 8–12
- [2] Schwab M, Karrenbach N and Claerbout J 2000 Making scientific computations reproducible *Comput. Sci. Eng.* **2** 61–67
- [3] Madagascar Development Team 2012 *Madagascar Software, Version 1.4* <http://ahay.org>
- [4] Silva C T, Freire J and Callahan S P 2007 Provenance for visualizations: reproducibility and beyond *Comput. Sci. Eng.* **9** 82–89
- [5] Leisch F 2002 Sweave: dynamic generation of statistical reports using literate data analysis *Compstat 2002—Proc. in Computational Statistics* ed W Härdle and B Rönz (Heidelberg: Physica Verlag) pp 575–580
- [6] Lamport L 1994 *LaTeX: A Document Preparation System* 2nd edn (Reading, MA: Addison-Wesley) updated for LaTeX 2 ϵ
- [7] Ramsey N 1994 Literate programming simplified *IEEE Softw.* **11** 97–105
- [8] Knuth D E 1992 *Literate Programming (CSLI Lecture Notes Number 27)* (Stanford, CA, USA: University of Chicago Press)
- [9] Xie Y 2013 *Dynamic Documents with R and knitr* (London, Boca Raton, FL: Chapman and Hall, CRC Press)
- [10] Nelson A 2014 *Dexy: The Most Powerful Flexible Documentation Tool Ever* <http://dexy.it>
- [11] van Rossum G *et al* 1991 *Python Programming Language* <http://python.org/>
- [12] Oliphant T E 2007 Python for scientific computing *Comput. Sci. Eng.* **9** 10–20
- [13] SymPy Development Team 2013 *SymPy: Python Library For Symbolic Mathematics* <http://sympy.org>
- [14] Hunter J D 2007 Matplotlib: A 2D graphics environment *Comput. Sci. Eng.* **9** 90–95
- [15] Matsumoto Y *et al* 1995 *Ruby: A Programmers Best Friend* <http://ruby-lang.org>
- [16] Bezanson J, Karpinski S, Shah V B and Edelman A 2012 *The Julia Manual* <http://julialang.org>
- [17] Brandl G 2008 *Sphinx: Python Documentation Generator* <http://sphinx-doc.org>
- [18] Pastell M 2010 *Pweave—Reports from Data with Python* <http://mpastell.com/pweave>
- [19] Pérez F and Granger B E 2007 IPython: a system for interactive scientific computing *Comput. Sci. Eng.* **9** 21–29
- [20] Ehmsen M R 2007 *Python—Embed Python code in LaTeX* <http://ctan.org/pkg/python>
- [21] Drake D 2008 *The SageTeX Package* <https://bitbucket.org/ddrake/sagetex>
- [22] Stein W A *et al* 2013 *Sage Mathematics Software (Version 6.0). The Sage Development Team* <http://www.sagemath.org>
- [23] Molteno T 2009 *SympyTeX* <https://github.com/tmolteno/SympyTeX/>
- [24] Berry K *et al* 2013 *TeX Live Documentation* <http://tug.org/texlive>
- [25] The Pocoo Team 2014 *Pygments: Python Syntax Highlighter* <http://pygments.org>
- [26] van Zandt T, Girou D, Rahtz S and Voß H 2010 *The ‘fancyvrb’ Package: Fancy Verbatims in LaTeX* <http://www.ctan.org/pkg/fancyvrb>
- [27] Poore G 2014 *PythonTeX: Fast Access to Python from within LaTeX* <https://github.com/gpoore/pythontex>
- [28] Poore G 2014 *Synchronized Python Debugger (syncpdb)* <https://github.com/gpoore/syncpdb>
- [29] MacFarlane J 2014 *Pandoc: A Universal Document Converter* <http://pandoc.org>
- [30] Heinz C, Moses B and Hoffann J 2014 *The Listings Package* <http://ctan.org/pkg/listings>
- [31] Poore G and Rudolph K 2013 *The Minted Package: Highlighted Source Code in LaTeX* <https://github.com/gpoore/minted>
- [32] Gruber J 2004 *Markdown* <http://daringfireball.net/projects/markdown>