A gentle introduction to PythonTEX

Andrew Mertz and William Slough

Abstract

The PythonTEX package allows authors to combine computational and typesetting tasks by embedding Python code in TEX documents. This package allows access to many powerful Python modules, providing support for such things as symbolic mathematics, plotting, arbitrary precision numerical calculations, and networking. Python's intuitive syntax, popularity, and extensibility along with TEX's formatting strengths make them a logical combination for programming documents. By examining a variety of examples, we will provide an overview of the capabilities and possibilities of PythonTEX.

1 Motivation and overview

As widely appreciated by its users, TEX is a typesetting system with numerous strengths and capabilities, providing the ability to create beautiful documents. Although it provides a capability for the definition of macros, designing and implementing them can be a daunting experience, especially for non-experts.

Python [9] is a programming language which has attracted a large number of users. As a further enhancement, scientific and technical computing is supported by an extensive collection of modules and utilities. These provide capabilities for numerical integration, linear algebra, linear programming, sparse matrix manipulation, symbolic mathematics, and plotting, for example.

PythonTEX [8] allows authors to combine the computational power of Python with the typesetting capabilities of TEX. This marriage of computational and typesetting worlds yields some exciting possibilities, as we intend to show in this paper.

Using the PythonTEX package, Python code may be placed directly into a LATEX document. During processing of this document — and "behind the scenes" — wherever Python code appears, a Python interpreter is executed, producing results which are then injected into the document in place of that code.

PythonTEX provides a variety of macros and environments with various optional arguments. It also has a tool, `depythontex`, for creating merged documents consisting of the original LATEX source and the Python output. This resulting document can then be processed without PythonTEX. Our aim is to provide an introduction, so we limit ourselves to a small, yet powerful, subset of PythonTEX.

2 Getting started

To begin, some installation will probably be needed. The PythonTEX package can be found at CTAN and installed, for example, by use of a package manager such as TEX Live's `tlmgr`.

In addition, a Python installation is needed, with Python 2.7 being the recommended version. The exact details of how this is done depend on your system, but one relatively simple way to obtain it is to download and install Anaconda [1]. (We are grateful to Richard Koch, from whom we learned about this resource.) Anaconda is a free Python distribution which supports GNU/Linux, Windows, and Mac OS X.

Not surprisingly, a document to be processed with PythonTEX will need to indicate this in its preamble:

```
\usepackage{pythontex}
```

A number of optional arguments can be supplied, though none of these are needed for what is being described in this introduction. For full details, refer the PythonTEX documentation.

Three steps are needed to process a PythonTEX document: first, LATEX, then PythonTEX, and finally LATEX. (Various engines are possible, including pdfLATEX, LuaLATEX, and XƎLATEX.) For example, the document `sample.tex` could be processed with the following sequence of commands.

```
pdflatex -interaction nonstopmode \
         -draftmode sample.tex
pythontex sample.tex
pdflatex sample.tex
```

The first step extracts the Python code from the document (to the file `sample.pytxcode`). In the second step, this code is given to the Python interpreter and the results are saved to a variety of files within the subdirectory `pythontex-files-sample`. In the final step, the results from Python are merged with the original document.

3 Fundamental PythonTEX

We begin our exploration by considering two macro commands intended for inline code: `\py` and `\pyc`. To use `\py`, a single-line Python expression is supplied as an argument:

`\py{`*expression*`}`

In response, the Python interpreter evaluates the expression, computes the result and stores the result as a string. This string then takes the place of the `\py` command which is subsequently typeset. For example, `\py{2**26}` produces 67108864, the value of 2^{26}.

```
\begin{pycode}
print(r"\begin{tabular}{c|c}")
print(r"$m$ & $2^m$ \\ \hline")
print(r"%d & %d \\" % (1, 2**1))
print(r"%d & %d \\" % (2, 2**2))
print(r"%d & %d \\" % (3, 2**3))
print(r"%d & %d \\" % (4, 2**4))
print(r"\end{tabular}")
\end{pycode}
```

```
\begin{tabular}{c|c}
$m$ & $2^m$ \\ \hline
1 & 2 \\
2 & 4 \\
3 & 8 \\
4 & 16 \\
\end{tabular}
```

Figure 1: Generation of a table using the `pycode` environment; resulting LaTeX code shown on the right

A related macro is `\pyc`, which has a subtle yet important difference. To use `\pyc`, a single-line Python statement is given:

`\pyc{`*statement*`}`

Here, the given statement is executed and anything *printed* by it takes the place of the `\pyc` command, which is subsequently formatted by TeX. As an example, `\pyc{print(2**26)}` yields 67108864.

At this point, it may appear that `\pyc` does not add much beyond what `\py` provides. However, this is far from the truth, as we intend to show. But before we can illustrate the power of `\pyc`, we need to discuss some additional features of PythonTeX and Python itself.

An analog of the `\pyc` command is the `pycode` environment:

```
\begin{pycode}
    Python statements
\end{pycode}
```

Unlike `\pyc`, this environment allows multiple-line Python statements to appear. As with `\pyc`, all of the printed output gets inserted into the document at that point, to be subsequently typeset.

To illustrate, consider Figure 1. This example shows how a `pycode` environment may be utilized to generate a table of values consisting of the pairs $(m, 2^m)$, using Python to generate powers of 2. Although this example does not use sophisticated Python code, it does illustrate an important idea: the code embedded within a `pycode` environment should generate the appropriate typesetting markup to achieve the desired effect. The typeset result of this code follows:

m	2^m
1	2
2	4
3	8
4	16

Like the C language, Python uses escape sequences (such as `\\`, `\n`, `\f`, etc.) to describe certain characters. Python uses “raw” strings, denoted with an `r` prefix, to disable escape sequences, allowing their content to appear verbatim. So, for example,

```
print("\\")
```

would output a single `\`, whereas

```
print(r"\\")
```

outputs `\\`. Since Python is being used to generate LaTeX code, the use of raw strings is often needed.

Another feature of Python being used here is the `%` operator, which is used for formatting strings. The `%d` specifies a placeholder, to be filled by a decimal integer value obtained from an expression. For example,

```
"%d and %d" % (3, 2**3)
```

produces the string `"3 and 8"`. These two features, raw strings and formatted strings, allow for the understanding of the example shown in Figure 1.

With this background, we can improve the code by introducing a loop which iterates over the desired values of m. The Python `range` function generates a list of integer values over a specified interval. Given integers l and h, `range(`l, h`)` generates a list of the integers from l to $h-1$. The example code shown in Figure 2 produces the same tabular output as before, but adds flexibility. The multiple assignment

```
lo, hi = 1, 4
```

allows an arbitrary range of table values to be specified; the `for` loop generates the table entries, one row per iteration. As a side note, Python provides arbitrary precision integer arithmetic; thus, tables of powers of 2 involving a large number of digits can be produced by simply adjusting `lo` and `hi`. For example, with `lo` $= 100$ and `hi` $= 102$ the following table is produced:

m	2^m
100	1267650600228229401496703205376
101	2535301200456458802993406410752
102	5070602400912917605986812821504

Python provides the ability to define functions as a way to promote program modularity. By defining a function within a `pycode` environment, we can

```
\begin{pycode}
lo, hi = 1, 4
print(r"\begin{tabular}{c|c}")
print(r"$m$ & $2^m$ \\ \hline")
for m in range(lo, hi + 1):
  print(r"%d & %d \\" % (m, 2**m))
print(r"\end{tabular}")
\end{pycode}
```

Figure 2: Generation of a table using a loop

subsequently evaluate it using `\py` or a similar command. To illustrate this capability, consider the well-known Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

defined by $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-2}+F_{k-1}$ for $k \geq 2$. Figure 3 provides the definition of a function which computes the n^{th} Fibonacci number.

Since `range(`n`)` generates the list of integers from 0 through $n - 1$, the `for` loop in `fib` makes exactly n iterations. One way to understand this function is to imagine scrolling across the Fibonacci sequence with a window capable of exposing two adjacent numbers in the sequence. Initially, the window is positioned over F_0 and F_1; to expose F_n the window is advanced n times.

```
\begin{pycode}
def fib(n):
  a, b = 0, 1
  for i in range(n):
    a, b = b, a + b
  return a
\end{pycode}
```

Figure 3: A Python function to compute the n^{th} Fibonacci number

With this function definition in place, we can use `\py` to evaluate arbitrary values of the Fibonacci sequence. For example, `\py{fib(10)}` produces 55, the value of F_{10}. As we saw earlier, arbitrary precision arithmetic is available "out of the box". so we can use this function with equal ease on larger values. For example, the claim

$$F_{100} = 354224848179261915075$$

is produced by `$F_{100} = \py{fib(100)}$`.

4 Getting fancier

So far, we have considered just two commands, `\py` and `\pyc`, and one environment, `pycode`. Even with this limited collection, we have many possibilities.

However, an awareness of a few more features of PythonTEX will allow for improved processing times and additional flexibility. One such feature is the concept of *sessions*. Without naming sessions, as we have done up to this point, all Python code runs sequentially in one default session. This can have several advantages. For example, variables and functions defined in one `pycode` environment are available to subsequent `pycode` environments, which avoids redundant code.

On the other hand, running all Python code in one session has the disadvantage that multiple cores are not utilized. As the amount of Python code in a document increases, there is a time penalty to be paid. By utilizing multiple Python sessions, code can be executed in parallel, providing a welcome speedup. All of the PythonTEX commands and environments provide for an optional session name. If no such specification appears, it runs in the default session. Judicious use of sessions can have dramatic improvement in processing time.

Another speed-related benefit of sessions derives from the fact that Python will run only for those sessions where the code has recently changed. This allows the user to place time-intensive Python code in their own sessions — and if that code doesn't need to be modified, then it is executed just once.

Multiple sessions are independent. They do not share variables or function definitions, for example. Sometimes this will be exactly what we want, but other times not. It is for these latter situations that the `pythontexcustomcode` environment exists. This environment allows a code block to be specified, which is then made available to *all* sessions, irrespective of session name. Let's look at an example, shown in Figure 4, to explore this idea further.

```
\begin{pythontexcustomcode}{py}
def makeTable(lo, hi):
  print(r"\begin{tabular}{c|c}")
  print(r"$m$ & $2^m$ \\ \hline")
  for m in range(lo, hi + 1):
    print(r"%d & %d \\" % (m, 2**m))
  print(r"\end{tabular}")
\end{pythontexcustomcode}
```

Figure 4: Informing all sessions how to generate a table of powers of 2

As a small detail, we first note that the custom code in this example specifies `py`, which indicates the `py` *family* of commands and environments to which it applies. As an introduction to PythonTEX, we have chosen to focus exclusively on the `py` family, but more sophisticated uses of PythonTEX may benefit from other families of commands. Full details are given in the PythonTEX manual.

Notice that the custom code provided here is simply an abstraction based on the earlier example from Figure 2. In the present case, starting and ending rows can be specified. So,

```
\pyc{makeTable(1, 4)}
```

would generate a table of powers with m between 1 and 4, whereas

```
\pyc{makeTable(4, 10)}
```

would generate a similar table, with m between 4 and 10. Using independent sessions would allow for potential speedup:

```
\pyc[one]{makeTable(1, 4)}
\pyc[two]{makeTable(4, 10)}
```

The session names, `one` and `two`, are arbitrary. Admittedly, the time gains for this example are likely to be negligible, but these examples were chosen for their simplicity and ability to illustrate sessions.

As a further illustration, we can extend our code so that it generates tables of arbitrary functions. To do this, we include two additional parameters: one to specify the function and one for the desired table heading. These two parameters, `f` and `hd`, appear in the revised version shown in Figure 5.

```
\begin{pythontexcustomcode}{py}
def makeTable2(lo, hi, f, hd):
  print(r"\begin{tabular}{c|c}")
  print(r"$m$ & %s \\ \hline" % hd)
  for m in range(lo, hi + 1):
    print(r"%d & %d \\" % (m, f(m)))
  print(r"\end{tabular}")
\end{pythontexcustomcode}
```

Figure 5: Informing all sessions how to generate a table for an arbitrary function

With this abstraction, we can produce a portion of the Fibonacci sequence displayed as a table, using the command

```
\pyc{makeTable2(4, 8, fib, "$F_m$")}
```

This call produces the table:

m	F_m
4	3
5	5
6	8
7	13
8	21

Python has a wealth of predefined functions, made available from its library of modules. These can be accessed with an appropriate `import` statement. For example, to make the factorial function available to all sessions, we could write:

```
\newif\ifprime \newif\ifunknown % booleans
\newcount\n \newcount\p \newcount\d
 \newcount\a % integer variables
\def\primes#1{2,~3% assume #1 is at least 3
\n=#1 \advance\n by-2 % n more to go
\p=5 % odd primes starting with p
\loop\ifnum\n>0
  \printifprime\advance\p by2 \repeat}
% we will invoke \printp if p is prime
\def\printp{,
 \ifnum\n=1 and~\fi % "and~" precedes last value
 \number\p \advance\n by -1 }
\def\printifprime{\testprimality
  \ifprime\printp\fi}
\def\testprimality{{\d=3 \global\primetrue
\loop\trialdivision
  \ifunknown\advance\d by2 \repeat}}
\def\trialdivision{\a=\p \divide\a by\d
\ifnum\a>\d \unknowntrue\else\unknownfalse\fi
\multiply\a by\d
\ifnum\a=\p \global\primefalse\unknownfalse\fi}
```

Figure 6: Knuth's code for generating prime numbers (editorial changes to line breaks and comments)

```
\begin{pythontexcustomcode}{py}
from math import factorial
\end{pythontexcustomcode}
```

With this import in effect, the table generation call

```
\pyc{makeTable2(10, 17, factorial, "$m!$")}
```

produces the following result:

m	$m!$
10	3628800
11	39916800
12	479001600
13	6227020800
14	87178291200
15	1307674368000
16	20922789888000
17	355687428096000

These examples illustrate how a wide assortment of tables can be generated and typeset with relatively little effort.

5 A table of primes

A recent post appeared on TEX Stack Exchange [12] asking how one might generate a collection of prime numbers using LATEX. Among the responses was the comment that Knuth himself had provided code for this [5, p. 218]. Figure 6 shows his implementation.

Knuth's code is not for the timid — he gives it his most difficult rating, a double dangerous bend. Some years later, Roegel [10] explains this 16-line macro, in the span of four pages, providing further evidence of the subtlety involved in its implementation.

```
\begin{pythontexcustomcode}{py}
from sympy import prime

def generatePrimes(n):   # Assume n >= 3
  for i in range(1, n):
    print("%d, "  % prime(i))
  print("and %d%%" % prime(n))
\end{pythontexcustomcode}
```

Figure 7: How to generate the first n prime numbers using PythonTEX

For comparison, we show an equivalent using PythonTEX in Figure 7. This code hides the computational details within a function `prime` which computes the i^{th} prime number. To use this we might write

```
Thirty primes: \pyc{generatePrimes(30)}.
```

which generates the following output:

> Thirty primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, and 113.

We think this provides a nice illustration of the suitability of PythonTEX for documents that can benefit from programmed output, especially for those TEX users who do not intend to become highly skilled in the art of macro writing.

This function has a small subtlety. The final line of output produced by `generatePrimes(30)` is

```
and 113%
```

followed by a newline supplied by the `print` statement. (A single `%` is produced by the Python specifier `%%`.) We want the terminating `%` to appear so `generatePrimes` can be included in the context of other text — such as the terminating period in the preceding example.

6 Python data structures

Python has several useful native data structures, such as lists, sets, and dictionaries. Furthermore, list objects have methods that allow them to be treated as stacks or queues. This section demonstrates the basic syntax for working with lists and dictionaries. These capabilities will be needed for examples in subsequent sections.

A list is an integer-indexed sequence of items, possibly of different types. In other words, a list can contain a mixture of numbers, strings, and other objects. Items can be removed from, added to, or retrieved from lists. Lists can be defined with a comma separated sequence of items within square brackets. For example:

```
\pyc{myList=["Iris", "Azalea", "Rose"]}
```

defines a new list named `myList` that contains three strings. The indexing operator, `[ ]`, can be used to retrieve items from a list. Lists in Python are indexed from zero. Thus, `\py{myList[0]}` is "Iris" and `\py{myList[1]}` is "Azalea".

A `for` loop can be used to iterate over any sequence. For example:

```
\begin{pycode}
for name in myList:
  print(name)
\end{pycode}
```

becomes "Iris Azalea Rose".

The `enumerate` function may be used if both the index and the value of an item are needed. For example,

```
\begin{pycode}
for index, name in enumerate(myList):
  print(r"%d: %s" % (index, name))
\end{pycode}
```

prints both the index and value each item in the list, that is, "0: Iris 1: Azalea 2: Rose".

While lists are indexed by integers, dictionaries can be indexed by any immutable type, often strings. Dictionaries can be thought of as sets of key-value pairs where the keys are unique. Dictionaries can be defined with a comma-separated sequence of key-value pairs within braces. For example,

```
\begin{pycode}
myDict = {"Illinois": "Violet",
          "New Mexico": "Yucca",
          "Indiana": "Peony"}
\end{pycode}
```

defines a dictionary named `myDict` with three entries. The indexing operator, `[ ]`, can be used to retrieve values from a dictionary with keys used as the index. So, `\py{myDict["Illinois"]}` yields "Violet".

7 Symbolic mathematics

While the preceding sections attempt to be a relatively simple introduction to Python and PythonTEX, the remaining sections are more complex. The goal is to demonstrate some of the cases where PythonTEX can provide useful capabilities that would be difficult using only LATEX.

Aside from the built-in functionality, Python has many powerful modules for mathematics. For instance, SciPy [11] is a rich collection of open source Python-based software for science, engineering, and mathematics. SciPy includes SymPy [13], a Python module for symbolic mathematics with features similar to other computer algebra systems like Mathematica [15] and Maple [6]. Using SymPy with LATEX allows mathematics not only to be beautifully typeset, but also to be manipulated and evaluated.

While a full exploration of SymPy is beyond the scope of this paper, an introduction to its features will be provided to highlight how well it can integrate with TEX through PythonTEX. For more information see the SymPy tutorial [14].

Like other modules, SymPy must be imported before use. For example:

```
\begin{pythontexcustomcode}{py}
from sympy import *
\end{pythontexcustomcode}
```

imports all of the functions and objects defined in the `sympy` module. SymPy defines numerous functions, many with common names such as `sin`, `cos`, and `var`. This can interfere with other modules, such as the plotting module `pylab`, which will be discussed in the next section. Thus, it can be safer to import the module with an `import sympy` statement:

```
\begin{pythontexcustomcode}{py}
import sympy
\end{pythontexcustomcode}
```

or to import inside of a session that will be used only for SymPy:

```
\begin{pycode}[sympy-session]
from sympy import *
\end{pycode}
```

PythonTEX also defines macros (`sympy`, `sympyc`) and related environments (such as `sympycode`) which simplifies this process.

The `var` function can be used to declare symbolic variables. For example, `\pyc{var("x, y")}` declares two symbolic variables `x` and `y`. Such variables can be used to form symbolic expressions that can be manipulated by SymPy. The examples in this section assume this variable declaration has been performed and that a

```
from sympy import
```

statement was used.

The `latex` function returns the LATEX code representing a given SymPy expression. For example,

```
$\py{latex((x + y)**5)}$
```

yields $(x+y)^5$. Without the `latex` function,

```
$\py{(x + y)**5}$
```

simply becomes $(x+y)**5$, since the Python exponentiation operator is not converted into its LATEX equivalent. This difference is more pronounced as the expressions become more complex. Also, LATEX code is just text to SymPy and cannot be manipulated as symbolic expressions can. It is important to remember to use the `latex` function only when typesetting SymPy expressions.

Symbolic expressions can be saved in ordinary Python variables. For example,

```
\pyc{z = (x + y)**5}
```

stores an expression in the variable `z`. SymPy has many functions for manipulating symbolic expressions, including: `simplify`, `factor`, `collect`, and `expand`. These functions are applied to symbolic expressions like any other function call. For instance, `z` can be expanded with

```
\[ \py{latex(expand(z)) + "."} \]
```

which becomes

$$x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5.$$

Figure 8 shows a more complex example that forms a table of binomials and their expansions. This example also shows how to build a list of expressions and iterate over them.

SymPy exports many trigonometric and calculus related functions, some of which are illustrated in Figure 9. Several SymPy objects, such as limits, integrals, sums and products are not evaluated automatically. To evaluate such objects the `doit` method may be used.

SymPy also includes combinatorial functions, for such things as Bernoulli, Catalan, Fibonacci, and Stirling numbers. Figure 10 details the creation and use of a function for formatting tables of Stirling numbers of the second kind, denoted $\begin{Bmatrix}n\\k\end{Bmatrix}$, which counts the number of ways to partition a set of n elements into k nonempty subsets. This function relies on a macro to format Stirling numbers, for example:

```
\usepackage{amsmath}
\newcommand{\Stirling}[2]{
\begin{Bmatrix}#1\\#2\end{Bmatrix}}
```

```
\begin{pycode}
# Start with an empty list
binomials = []

# Add a few symbolic expressions to the list
for m in range(2, 6):
  binomials.append((x + y)**m)

# Start an align environment to hold the
# results
print(r"\begin{align*}")

# Add the original expressions and their
# expansions to the table
for expr in binomials:
  print(r"%s &= %s\\" % (latex(expr),
                latex(expand(expr))))

# End the align environment
print(r"\end{align*}")
\end{pycode}
```

$$\begin{aligned}
(x+y)^2 &= x^2 + 2xy + y^2\\
(x+y)^3 &= x^3 + 3x^2y + 3xy^2 + y^3\\
(x+y)^4 &= x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4\\
(x+y)^5 &= x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5
\end{aligned}$$

Figure 8: Binomial expansions

```
\begin{pycode}
# Define a list of functions
functions = [sin(x), cos(x), tan(x)]

print(r"\begin{align*}")

# For each function build a symbolic expression of its derivative and integral
for f in functions:
  d = Derivative(f, x)
  i = Integral(f, x)

  # Print a row in the table displaying the derivative and integral. Note the
  # use of the "doit" method to evaluate the derivative and integral. Also
  # string concatenation, +, is used to join strings in this example.
  print(latex(d) + "&=" + latex(d.doit()) + "&" +
      latex(i) + "&=" + latex(i.doit()) + r"\\")

print(r"\end{align*}")
\end{pycode}
```

$$\begin{aligned}
\frac{d}{dx}\sin(x) &= \cos(x) & \int \sin(x)\,dx &= -\cos(x)\\
\frac{d}{dx}\cos(x) &= -\sin(x) & \int \cos(x)\,dx &= \sin(x)\\
\frac{d}{dx}\tan(x) &= \tan^2(x) + 1 & \int \tan(x)\,dx &= -\frac{1}{2}\log\left(\sin^2(x) - 1\right)
\end{aligned}$$

Figure 9: Building a table of derivatives and integrals

```
\begin{pycode}
# Import a function for computing Stirling numbers
from sympy.functions.combinatorial.numbers import stirling

# Define a function to print a table of all of the Stirling numbers for sets
# of size 1 to maxN.
def stirlingTable(maxN):

  # Print the start of the table using a triple quoted string. Triple quoted
  # strings can span lines and are useful when including long strings.
  print(r"""\begin{displaymath}
\begin{array}{c|*{%d}{c}} \hline
\multicolumn{%d}{c}{\textbf{Stirling's Triangle for Subsets}} \\ \hline
n""" % (maxN, maxN + 1))

  # Print each of the column headings using the previously defined Stirling
  # macro.
  for k in range(1, maxN + 1):
    print(r" & \Stirling{n}{%d} " % k)

  # Add some phantom space so the braces are not touching the hlines.
  print(r"\vphantom{\parbox[c][7ex]{0in}{}} \\ \hline")

  # Start each row with the current n value
  for n in range(1, maxN + 1):
    print("%d" % n)

    # Add each of the Stirling numbers to the row
    for k in range(1, n + 1):
      print("& %d" % stirling(n, k))

    # End the row
    print(r"\\")

  # End the table
  print(r"\hline \end{array}\end{displaymath}")

stirlingTable(8)
\end{pycode}
```

Stirling's Triangle for Subsets

n	$\left\{ {n \atop 1} \right\}$	$\left\{ {n \atop 2} \right\}$	$\left\{ {n \atop 3} \right\}$	$\left\{ {n \atop 4} \right\}$	$\left\{ {n \atop 5} \right\}$	$\left\{ {n \atop 6} \right\}$	$\left\{ {n \atop 7} \right\}$	$\left\{ {n \atop 8} \right\}$
1	1							
2	1	1						
3	1	3	1					
4	1	7	6	1				
5	1	15	25	10	1			
6	1	31	90	65	15	1		
7	1	63	301	350	140	21	1	
8	1	127	966	1701	1050	266	28	1

Figure 10: Building a table of Stirling numbers

```
\begin{pycode}
from pylab import *

# Define f(t), the desired function to plot
def f(t):
  return cos(2 * pi * t) * exp(-t)

# Generate the points (t_i, y_i) to plot
t = linspace(0, 5, 500)
y = f(t)

# Begin with an empty plot, 5 x 3 inches
clf()
figure(figsize=(5, 3))

# Use TeX fonts
rc("text", usetex=True)
rc("font", family="serif")

# Generate the plot with annotations
plot(t, y)
title("Damped exponential decay")
text(3, 0.15, r"$y = \cos(2 \pi t) e^{-t}$")
xlabel("time (s)")
ylabel("voltage (mV)")

# Save the plot as a PDF file
savefig("myplot.pdf", bbox_inches="tight")

# Insert LaTeX code to include the plot.
print(r"\begin{center}"
  + r"\includegraphics[width=\textwidth]{myplot.pdf}"
  + r"\end{center}")
\end{pycode}
```

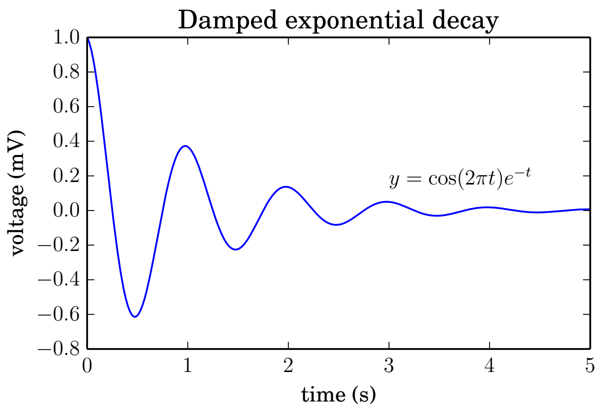

Figure 11: Plotting a function with Matplotlib

8 Plotting with Matplotlib

Matplotlib is a two-dimensional Python plotting library with an object-oriented interface and a set of functions similar to MATLAB [7]. To use it, the `pylab` module must be imported. Figure 11 shows an example of plotting a function with annotations. This example is inspired by a plot from the Matplotlib gallery [4] which contains many examples and tutorials. Such plots are desirable since they can use fonts which blend with the rest of the document.

Matplotlib's plots are saved in an external file such as a PDF, which can then be included in the current document. However, this can be problematic as the file may not exist the first time the document is processed. To avoid this problem, the Python code generates the required `\includegraphics` statement. In this way, the `\includegraphics` is not present the first time TeX processes the document, but is on subsequent processing.

9 Web services

There are many powerful and freely available web services which return JSON (Java Script Object Notation) or another easily parsed format. Python's excellent parsing and networking libraries make using such services relatively simple. Furthermore, some web services have Python modules made specifically for them.

Accessing such a web service typically requires some authorization. This may involve requesting an account with the service provider and agreeing to their terms of service. Often a key is provided to identify a client to the service and this key must be presented each time the service is used. Using the service can normally be broken into following tasks.

- Encoding the request as a URL
- Fetching the URL
- Parsing the response
- Using the result

```
\begin{pycode}
# Import functions to save the contents of a URL to a file and
# encode a dictionary as a string suitable for URL requests
from urllib import urlretrieve, urlencode

# See Google's documentation for service usage information and examples
# https://developers.google.com/maps/documentation/staticmaps/
def showGoogleMap(address, filename, zoom=10, width=640, height=680):
  # Build a dictionary with the key-value pairs required by the service
  query = {"key":googleKey, "center": address, "zoom": zoom,
           "size": "%dx%d" % (width, height), "sensor": "false"}

  # Convert the query into a url
  url = "https://maps.googleapis.com/maps/api/staticmap?" + urlencode(query)

  # Save the image to a file and include it in the document
  urlretrieve(url, filename + ".png")
  print(r"\begin{center}\includegraphics[width=0.45\textwidth]{%s}\end{center}"
        % filename)
\end{pycode}
```

Figure 12: Displaying a map of Tokyo using Google's Static Maps web service

The code of Figure 12 illustrates how the Google Static Maps API [3] can be used. This web service returns a PNG image of a map centered at a given address. The `showGoogleMap` function defined in this example has default values for the `zoom`, `width`, and `height` arguments. As a result, when invoking this function these arguments do not need to be specified. The results of invoking

```
showGoogleMap("Tokyo", "tokyoMap")
```

are shown in Figure 13.

To use the Google Static Maps service, an API key is needed. Such keys can be created with the Google APIs console (`https://code.google.com/apis/console`). The examples assume such a key has been stored in a global variable `googleKey`. For instance:

```
\begin{pycode}
googleKey = "Put API Key Here"
\end{pycode}
```

As a second example, Google's "URL Shortener" [2] web service takes long URLs and converts them into URLs with fewer characters, yielding links that can be easier to share. This service sends and receives data as JSON (a simple plain text format for transmitting information as key-value pairs). This format has dictionaries, lists, numbers, and strings for data types. Figure 14 shows an example of what JSON looks like, while Figure 15 shows a sample response from the URL Shortener. Note the similarity to the declaration of a Python dictionary. See Figure 16 for the details of using this service.

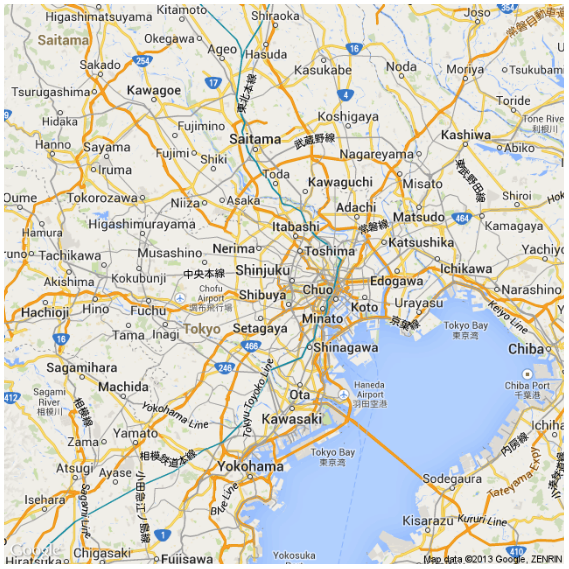

Figure 13: Map output from Static Maps service

10 Conclusions

The ability to include arbitrary computations within a LaTeX document holds much appeal. As a programming language, Python has relative simplicity with broad expressive power. The examples presented in this paper provide a glimpse of what is possible with this software combination.

```
{
  "debug": "on",
  "window": {
    "title": "Main View",
    "width": 640,
    "height": 480
  },
  "image": {
    "src": "Images/Icon.png",
    "hOffset": 10,
    "vOffset": 10
  }
}
```

Figure 14: JSON sample

```
{
 "kind": "urlshortener#url",
 "id": "http://goo.gl/fbsS",
 "longUrl": "http://www.google.com/"
}
```

Figure 15: Response for a successful use of the URL Shortener API

```
\begin{pycode}
# A function and object for fetching URLs and
# customizing requests
from urllib2 import urlopen, Request

# JSON/Python conversion functions
from json import load, dumps

def shortenURL(longURL):
 # The base URL for shortening requests
 url = ("https://www.googleapis.com/" +
        "urlshortener/v1/url")

 # For this service the query is sent as JSON.
 # So the query is converted to JSON and the
 # content type is set in the request's header.
 query = dumps({"longUrl": longURL,
                "key": googleKey})
 request = Request(url, query,
     {"Content-Type": "application/json"})

 # Fetch the request and parse the returned JSON
 result = load(urlopen(request))

 # Retrieve the shortened URL from the result
 shortURL = result["id"]

 # Add the shortened URL to the document
 print(r"\url{%s}%%" % shortURL)

shortenURL("http://mirror.ctan.org/macros/latex/" +
           "contrib/pythontex/pythontex.pdf")
\end{pycode}
```

Figure 16: Shortening a long URL to http://goo.gl/sfT8S5

References

[1] Continuum Analytics. Anaconda. http://store.continuum.io/cshop/anaconda/.

[2] Google. The URL Shortener API. https://developers.google.com/url-shortener/.

[3] Google. The Google Static Maps API. https://developers.google.com/maps/documentation/staticmaps/.

[4] John Hunter. Matplotlib gallery. http://matplotlib.org/gallery.html.

[5] Donald E. Knuth. *The TEXbook*. Addison-Wesley Professional, 1984.

[6] Maplesoft. Maple. http://www.maplesoft.com/products/maple/.

[7] MathWorks. Matlab. http://www.mathworks.com/products/matlab/.

[8] Geoffrey Poore. PythonTEX. http://www.ctan.org/pkg/pythontex.

[9] Python Software Foundation. Python. http://python.org.

[10] Denis Roegel. Anatomy of a macro. *TUGboat*, 22:78–82, 2001. http://tug.org/TUGboat/tb22-1-2/tb70roeg.pdf.

[11] SciPy Developers. SciPy. http://scipy.org.

[12] TEX Stack Exchange. How to produce a list of prime numbers in LATEX. http://goo.gl/903u75.

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

[14] SymPy Development Team. Sympy tutorial. http://docs.sympy.org/latest/tutorial/.

[15] Wolfram Research. Mathematica. http://www.wolfram.com/mathematica/.

◇ Andrew Mertz and William Slough
Department of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920
aemertz (at) eiu dot edu,
waslough (at) eiu dot edu