Combining LaTeX with Python

Uwe Ziegenhagen

Abstract

Even older than Java, Python has achieved a lot of popularity in recent years. It is an easy-to-learn general purpose programming language, with strong capabilities, including in state-of-the-art topics such as machine learning and artificial intelligence. In this article we want to present scenarios where LaTeX and Python can work jointly. We will show examples where LaTeX documents are automatically generated by Python or receive content from Python scripts.

1 Introducing Python

Python has steadily grown to be one of the most widely used programming languages. Invented in 1991 by Guido van Rossum at the Centrum Wiskunde & Informatica in the Netherlands, Version 1.0 appeared in 1994. The current versions are 2.7 and 3.x. For people who wish to start with Python, Python 3 is strongly recommended.

```
print('Hello' + ' ' + 'World')
```

Listing 1: The unavoidable "Hello World" example

Python has a strong emphasis on code readability by making whitespace significant. In contrast to other programming languages, Python uses whitespace and indentation to define code blocks; a first example is in Listing 2.

```
def addTwo(a, b):
    return a+b

print(addTwo(5,3))     # gives 8
print(addTwo('U','S')) # gives 'US'
```

Listing 2: Basic function definition example

Python supports various programming paradigms, such as procedural, object-oriented and functional programming. Listing 3 shows an example for the functional programming paradigm, using a lambda function to filter those integers from a list that are divisible by 2.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8]
result = filter(lambda x: x % 2 == 0, my_list)
print(list(result))
```

Listing 3: Using functional programming to filter a list

Listing 4 shows an example for the OO-programming paradigm. Here we define a class with two properties that is then instantiated.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_age(self):
        print(self.name + ', ' + str(self.age))

john = Person('John', 50)
john.print_age()
```

Listing 4: Using object-oriented programming

Excellent literature is available for Python learners on- and offline; we can recommend [1].

2 Writing LaTeX files with Python

After that brief introduction we will now focus on the creation of LaTeX files using Python. The recommended approach is to use a so-called "context managers", as it will handle the management of the file references as well as errors in case the file is not accessible or writable.

Listing 5 shows an example on how to write a simple LaTeX file. Backslashes need to be escaped, the line endings need to be added. Depending on the platform the code is executed, they will be replaced by the system's line ending. The resulting file is then UTF-8-encoded and can easily be processed further.

```
with open('sometexfile.tex','w') as file:
    file.write('\\documentclass{article}\n')
    file.write('\\begin{document}\n')
    file.write('Hello Palo Alto!\n')
    file.write('\\end{document}\n')
```

Listing 5: Writing a TeX file

Processing, e. g., the compilation by pdfLaTeX and display by the system's PDF viewer can also be triggered from Python, as Listing 5 shows. We create the LaTeX file and use Python's subprocess module to call pdfLaTeX. When this process has a non-error exit code, the platform's PDF viewer is launched.

```
import subprocess, os
with open('sometexfile.tex','w') as file:
    file.write('\\documentclass{article}\n')
    file.write('\\begin{document}\n')
    file.write('Hello Palo Alto!\n')
    file.write('\\end{document}\n')

x = subprocess.call('pdflatex sometexfile.tex')
if x != 0:
    print('Exit-code not 0, check result!')
else:
    os.system('start sometexfile.pdf')
```

Listing 6: Writing & processing TeX files

When LATEX files are created programmatically the goal is often to create bulk letters or other dynamically adjusted documents. Python offers various ways to assist in this process. The most intuitive way is probably to use search & replace to eplace placeholders with text; Listing 7 shows an example for this approach. The example should be self-explaining, note the nested context managers to read and then write the LATEX file.

```
place = 'Palo Alto'

with open('place.tex','r') as myfile:
    text = myfile.read()
    text_new = text.replace('$MyPlace$', place)

    with open('place_new.tex', 'w') as output:
        output.write(text_new)
```

Listing 7: Replacing text

While this approach works fine, it is not recommended when more complicated documents need to be created. Fortunately Python offers a variety of template engines — either built-in or easily installable with the help of Python's package manager — that improve the workflow and avoid "re-inventing the wheel". Among the different template engines, we have successfully worked with Jinja2. It offers full Unicode support, sandboxed execution, template inheritance and many more useful features. Listing 8 shows a non-LATEX example for Jinja2, which tells us the following:

1. Syntax is (easily) understandable
2. Jinja2 brings its own notation for looping, etc.
3. Extensive use of {, %, }

```
from jinja2 import Template

mytemplate = Template("Hello {{place}}!")
print(mytemplate.render(place="Palo Alto"))

mytemplate = Template("Some numbers: {% for n
    in range(1,10) %}{{n}}{% endfor %}")
print(mytemplate.render())
```

Listing 8: A non-LATEX Jinja2 template example

So, to make Jinja2 work well with LATEX we need to modify the way a template is defined. Listing 2 shows[1] how this reconfiguration can be made. Instead of braces, we use two LATEX commands, `\BLOCK` and `\VAR`. Both commands will later be defined as empty LATEX commands in the LATEX file to have the file compile without errors.

[1] Source: https://web.archive.org/web/20121024021221/http://e6h.de/post/11/

```
import os
import jinja2 as j

latex_env = j.Environment(
   block_start_string = '\BLOCK{',
   block_end_string = '}',
   variable_start_string = '\VAR{',
   variable_end_string = '}',
   comment_start_string = '\#{',
   comment_end_string = '}',
   line_statement_prefix = '%-',
   line_comment_prefix = '%#',
   trim_blocks = True,
   autoescape = False,
   loader = j.FileSystemLoader(os.path.abspath('.'))
   )
```

The following Listing 9 shows an excerpt from the final code. It loads the template, fills the placeholders and writes the final document to the disk. One advantage of this approach is that it allows the template to be separated from the program logic that fills it; in more complex situations, the built-in scripting comes very handy.

```
template = latex_env.get_template('jinja-01.tex')
document = template.render(place='Palo Alto')
with open('final-02.tex','w') as output:
    output.write(document)
```

Listing 9: Rendering the document

3 Running Python from LATEX

In this section we want to address the reverse: not the creation of LATEX code but the execution of Python code from within LATEX. Several packages and tools are available to support this. Here we want to demonstrate two of them. One is derived from code posted to tex.stackexchange.com, the other, `pythontex`, is a well-maintained LATEX package.

The idea for the code given below came from the fact, that LATEX is a) able to write the content of environments to external files and b) is able to run external commands when `--shell-escape` is enabled. One just needs need to combine both to write and run external files. Based on our question on TSX, an easily implementable solution was given;[2] it is shown in Listing 10. When Python code is placed in a `pycode` environment inside a document, LATEX writes the code to the filename specified in the parameter of the environment, runs Python on this file and pipes its output to a `.plog` file. This `.plog` file is then read by LATEX and typeset with syntax highlighting provided by the `minted` package (which also uses Python internally).

The advantage of this approach is that it can be adjusted easily to different external programs, as

[2] https://tex.stackexchange.com/questions/116583

long as they are able to run in batch mode. One can easily adjust the way the code is included, e.g., we have worked successfully with a two-column setup in Beamer, where the left column shows the source code and the right column the result of the code execution. One disadvantage is that the programs are executed each time the LaTeX code is compiled.

```
\usepackage{minted}
\setminted[python]{frame=lines, framesep=2mm,
     baselinestretch=1.2, bgcolor=colBack,
     fontsize=\footnotesize, linenos}
\setminted[text]{frame=lines, framesep=2mm,
     baselinestretch=1.2, bgcolor=colBack,
     fontsize=\footnotesize, linenos}

\usepackage{fancyvrb}
\makeatletter
\newenvironment{pycode}[1]%
  {\xdef\d@tn@me{#1}%
   \xdef\r@ncmd{python #1.py > #1.plog}%
   \typeout{Writing file #1}%
   \VerbatimOut{#1.py}%
  }%
  {\endVerbatimOut %
   \toks0{\immediate\write18}%
   \expandafter\toks\expandafter1%
     \expandafter{\r@ncmd}%
     \edef\d@r@ncmd{\the\toks0{\the\toks1}}%
   \d@r@ncmd
   \noindent Input
   \inputminted{python}{\d@tn@me.py}%
   \noindent Output
   \inputminted{text}{\d@tn@me.plog}%
  }%
\makeatother
```

Listing 10: The `pycode` environment

The `pythontex` package [2] uses a more advanced approach: it can detect if the Python code has been edited or not. Only if an edit took place is the Python code rerun, thus saving time especially with more complicated Python code. The workflow is the following: first the LaTeX engine of your choice is run, followed by the `pythontex` executable, followed by another `latex` run. The package offers various LaTeX commands and corresponding environments; see the package documentation.

Let us show with an example (Listing 11) how the package can be applied. After loading the package `pythontex` we use the `\pyc` command, which only executes code and does not typeset it, for the first line of Python code. Here we instruct Python to load a function from the `yahoo_fin` library which allows us to retrieve stock information from Yahoo, given that an Internet connection is available.

In the following table we then use `\py` commands to specify which stock quote to be retrieved. This command requires the executed Python code to return a single expression.

```
\documentclass[12pt]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{pythontex}
\usepackage{booktabs}
\begin{document}

\pyc{from yahoo_fin import stock_info as si}

\begin{tabular}{lr}
\toprule
Company & Latest quote \\
\midrule
Apple  & \py{round(si.get_live_price("aapl"),2)} \\
Amazon & \py{round(si.get_live_price("amzn"),2)} \\
Facebook & \py{round(si.get_live_price("fb"),2)} \\
\bottomrule
\end{tabular}
\end{document}
```

Listing 11: Using `pythontex` to retrieve stock prices

Company	Latest quote
Apple	203.43
Amazon	1832.89
Facebook	190.16

Figure 1: Output resulting from Listing 11

The `pythontex` package provides many more features, among them even symbolic computation. It can thus be highly recommended.

4 Summary

We have shown how easy LaTeX documents can be enriched by Python, a scripting language that is easy to learn and fun to work with. Accompanying this article is the more extensive presentation held at TUG 2019, for which the interested reader is directed to the slides at `www.uweziegenhagen.de`.

References

[1] M. Lutz. *Learning Python.* O'Reilly, 2013.

[2] G. M. Poore. PythonTeX: Reproducible documents with LaTeX, Python, and more. *Comput. Sci. Disc.* 8(1), 2015. `ctan.org/pkg/pythontex`

◇ Uwe Ziegenhagen
Escher Str. 221
50739 Cologne, Germany
`ziegenhagen (at) gmail dot com`
`www.uweziegenhagen.de`