Dokumentenautomation mit PythonTEX

Karsten Brodmann

Oktober 2018

Automatisierung schützt uns vor langweiliger, manueller Arbeit und spart in der Regel wertvolle Zeit. Außerdem, und das ist ganz wichtig, senkt Automation die Fehlerquote. PythonTEX von Geoffrey M. Poore macht es einfach, LATEX-Dokumente mit Hilfe von Python zu dynamisieren, quasi beliebige Dokumentinhalte, von mathematischen Formeln über Grafiken bis hin zu Tabellen aus Datenbankabfragen, automatisch zu generieren.

1 PythonTEX beschaffen

Üblicherweise ist PythonTEX in den gängigen TEX/LaTeX-Distributionen bereits enthalten. Wenn nicht, kann es bei GitHub unter der folgenden URL heruntergeladen und installiert werden: `https://github.com/gpoore/pythontex`. Bevor man PythonTEX jedoch auch tatsächlich nutzen kann, muss freilich Python installiert sein. Es werden sowohl Python 2.7 sowie auch die aktuellen Python-Versionen ab 3.2 unterstützt. Aufgrund bestehender Abhängigkeiten zu Python-Erweiterungen müssen die entsprechenden Pakete gegebenenfalls auch noch installiert werden. In jedem Fall wird `pygments` benötigt.

PythonTEX beinhaltet eine einigermaßen umfangreiche Dokumentation. Sie besteht aus einer Schnellstart-Einführung, einem Beispieldokument und der typischen Dokumentation von LATEX-Paketen, die hier nicht ganz 150 Seiten umfasst.

2 LATEX-Dokumente mit PythonTEX übersetzen

PythonTEX ist kompatibel zu pdfTEX, XeTEX oder auch LuaTEX, so dass zum Compilieren der Quelltexte `latex`, `pdflatex`, `xetex` oder `luatex` verwendet werden können. Dabei besteht ein Übersetzungsvorgang bei Verwendung von PythonTEX aus mindestens drei Schritten:

1. übersetzen des Dokuments mit `pdflatex` (oder dessen Verwandten)
2. `pythontex` ausführen
3. nochmaliges Übersetzen des Dokuments mit `pdflatex`, um die Ergebniss von `pythontex` ins Dokument zu integrieren

Mit der Option `--interpreter` von `pythontex` kann gesteuert werden, welche Python-Version zur Ausführung des Pythoncodes verwendet werden soll.

Schritt 2 kann entfallen, wenn sich an den Dokumentinhalten, die durch Pythoncode bestimmt werden, nichts geändert hat. Weil die Anzahl der notwendigen Übersetzungsläufe immer etwas nervig ist, kann man sich das Leben einfacher gestalten. PythonTEX kann in Verbindung mit `latexmk` genutzt werden. Dazu muss man sich lediglich eine entsprechende Konfigurationsdatei schreiben.

Eine einfache Konfigurationsdatei kann so aussehen:

```
1 add_cus_dep('pytxcode', 'tex', 0, 'pythontex');
2 sub pythontex { return system("pythontex \"$_[0]\""); }
3 $pdflatex = 'pdflatex -synctex=1 --shell-escape %O %S';
4 $latex = 'latex -synctex=1 --shell-escape %O %S';
```

Listing 1: pythontex.rc

Der Aufruf, um mit Hilfe von `latexmk` ein PDF-Dokument zu erzeugen, sieht dann so aus:

```
$ latexmk -pdf -r pythontex.rc DokumentName
```

3 Anwendungsbeispiele mit PythonTEX

Ich möchte hier nicht die englischsprachige Dokumentation ins Deutsche übersetzen oder einzelne Optionen von PythonTEX im Detail besprechen. Ich möchte Ihnen vielmehr ein paar Beispiele präsentieren, die zeigen, wie einfach es ist, mit Hilfe von PythonTEX Dokumente mit dynamisch generierten Inhalten zu erstellen.

PythonTEX stellt verschiedene Umgebungen zur Verfügung, die je nach Aufgabe gewählt werden können. Die Umgebungen unterscheiden sich danach, ob sie Pythoncode ausführen oder ihn lediglich, im Sinne eines Listings anzeigen. Die Umgebungen, die Pythoncode ausführen, gliedern sich dann nochmals danach, ob sie den Pythoncode anzeigen, das Ergebnis anzeigen oder im späteren Dokument unsichtbar sind. Das sind die Umgebungen `pyverbatim`, `pycode`, `pyblock`, `pysub` und `pythontexcustomcode`. Mit Hilfe der letzten Umgebung lassen sich quasi Biblioteksroutinen erstellen, die dann von beliebiger Stelle im Dokument aufgerufen werden können. Mit der Umgebung `pyconsole` steht eine Emulation der Python-Shell zur Verfügung. In dieser Umgebung ausgeführte Python-Anweisungen erscheinen im Dokument so, wie die Python-Shell sie darstellt.

Zum Setzen von Quellcodes unterstützt PythonTEX Syntaxhervorhebung. Für große Dokumente mit viel und/oder aufwendigem Pythoncode, stellt PythonTEX Sessions bereit, so dass mit mehreren parallelen Prozessen gerechnet werden kann.

Für Formatierung wird vor allem das LATEX-Paket `fancyvrb` genutzt.

Neben den Umgebungen gibt es Entsprechungen, die innerhalb von Fließtext verwendet werden können, ähnlich wie wir das vom mathematischen Satz kennen. So können wir mit beispielsweise `\py` Python-Ausdrücke auswerten und ausgeben, mit `\pyc` Code ausführen und mit `\pyb` Code ausführen und den Code ausgeben.

Wie auch sonst bei LaTeX üblich, muss PythonTeX im Dokument bekannt gemacht werden, bevor dessen Fetures genutzt werden können. Das geschieht ganz klassich mittels `\usepackage`. Das einzubindende Paket heißt `pythontex`. Bei der Einbindung können optionale Einstellungen definiert werden, die aber jederzeit im Dokument wieder überschrieben werden können, wenn das erforderlich sein sollte. Die Standardeinstellungen werden, ohne Optionen, mit

```
\usepackage{pythontex}
```

in das LaTeX-Dokument übernommen.

Hinweis: Wenn das Paket `listings` verwendet wird, muss `pythontex` vor diesem in das LaTeX-Dokument eingebunden werden. Andernfalls führt dies zu Konfusionen, die eine erfolgreiche Übersetzung des Dokuments verhindern.

3.1 Einfache Kommandos

Am einfachsten ist es sicherlich, mit Python einen arithmetischen Ausdruck auszuwerten und das Ergebnis zurückgeben zu lassen, um ihn im Dokument zu verwenden.

Soll beispielsweise die Fakultät einer (wirklich) großen Zahl berechnet und im Text ausgegeben werden, ist nichts einfacher als das. Wir wollen 2^{100} berechnen. Das ist ganz einfach:

$$2^{100} = 1267650600228229401496703205376$$

Das Tag `\py` gibt eine Zeichenkettendarstellung seiner Argumente zurück. Der Quellcode für die obige Gleichung sieht so aus:

```
1 \[
2 2^{100}=\py{2**100}
3 \]
```

Das ist wesentlich einfacher und kürzer zu schreiben als das Ergebnis von 2^{100} manuell zu berechnen und hier einzusetzen. Innerhalb der Mathematikumgebung wird stattdessen innerhalb des `\py`-Kommandos eine Python-Anweisung zur Berechnung der Potenz ausgeführt. Alternativ hätten wir auch `\pyc` verwenden können. Dieses Kommando führt Pythonkommandos aus, gibt jedoch nichts zurück. Daher muss, für den aktuellen Fall, zusätzlich das `print()`-Kommando benutzt werden, um das Ergebnis ins Dokument zu bekommen. Das eröfffnet dann auch Formatierungsmöglichkeiten, um das Ergebnis der Berechnung mit Tausendertrennzeichen ausgeben.

```
1 \[
2 2^{100}=\pyc{print("{:,}".format(2**100))}
3 \]
```

Das ergibt dann die folgende Ausgabe:

$$2^{100} = 1,267,650,600,228,229,401,496,703,205,376$$

Als Deutsche mögen wir diese amerikanische Darstellung allerdings nicht so sehr. Wir bevorzugen Tausenderpunkte und wollen das Ergbnis eigentlich so dargstellt wissen:

$$2^{100} = 1.267.650.600.228.229.401.496.703.205.376$$

Wie das zu erreichen ist, wird in Unterabschnitt 3.2.3, auf Seite 7, behandelt.

3.2 Komplexere Anwendungen

Das kurze Beispiel im ersten Abschnitt mag genügen, um zu demonstrieren, wie kleine Berechnungen oder ähnliches schnell und einfach mit PythonTEX realisiert werden können. Interessanter ist es, komplexere Dinge zu tun, Dinge, die aufwendig und möglicherweise auch fehleranfällig sind. Darin liegt der Charme von PythonTEX.

3.2.1 Mathematik und Formelsatz

LATEX ist sicherlich unangefochtener Meister im Setzen von Formeln. Wer das öfter macht, weiß aber auch, wie viel Arbeit darin steckt und wie leicht man sich dabei verschreiben kann. Da erscheint es reizvoll, auf die Dienste eines CAS (Computer Algebra System) zurückgreifen zu können. Genau das können wir tun. Mit `sympy` steht uns ein solches für Python zur Verfügung. `sympy` kann alles, was ein CAS können muss. Insbesondere beherrscht `sympy` symbolische Mathematik, kann integrieren, differenzieren und so weiter. Außerdem hat es eine LATEX-Ausgabe. Dies ausnutzend, können wir den Satz mathematischer Formeln mit Hilfe von PythonTEX bedeutend optimieren.

Hier ein Beispiel, in dem Binome erzeugt und expandiert werden.

```
1  \begin{pycode}
2  from sympy import *
3  x, y = symbols("x y")
4  binome = []
5  for exponent in range(3, 6):
6    binome.append((x + y)**exponent)
7  print(r"\begin{align*}")
8  for expr in binome:
9    print(r"%s &= %s\\" % (latex(expr), latex(expand(expr))))
10 print(r"\end{align*}")
11 \end{pycode}
```

Die `pycode`-Umgebung führt den enthaltenen Code direkt an der Stelle im Dokument aus, an welcher er notiert ist. Die erforderlichen LATEX-Anweisungen werden mittels `print()` ausgegeben. Den Formelsatz übernimmt `sympy`. Hier das fertige Resultat:

$$
\begin{aligned}
&(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3\\
&(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4\\
&(x+y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5\\
&(x+y)^6 = x^6 + 6x^5y + 15x^4y^2 + 20x^3y^3 + 15x^2y^4 + 6xy^5 + y^6\\
&(x+y)^7 = x^7 + 7x^6y + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7xy^6 + y^7\\
&(x+y)^8 = x^8 + 8x^7y + 28x^6y^2 + 56x^5y^3 + 70x^4y^4 + 56x^3y^5 + 28x^2y^6 + 8xy^7 + y^8\\
&(x+y)^9 = x^9 + 9x^8y + 36x^7y^2 + 84x^6y^3 + 126x^5y^4 + 126x^4y^5 + 84x^3y^6 + 36x^2y^7 + 9xy^8 + y^9
\end{aligned}
$$

Einfacher lassen sich die obigen Formelns nicht setzen.

In der `pyconsole`-Umgebung können wir uns ansehen, wie das Skript arbeitet, wie die Ausgaben aussehen, die ins LaTeX-Dokument übernommen werden. Das Intervall der Schleife habe ich hier kurz geändert, damit die LaTeX-Ausgabe des Formalsatzes auf die Seite passt.

```
>>> from sympy import *
>>> x, y = symbols("x y")
>>> binome = []
>>> for exponent in range(1, 3):
...     binome.append((x + y)**exponent)
...
>>> print(r"\begin{align*}")
\begin{align*}
>>> for expr in binome:
...     print(r"%s &= %s\\" % (latex(expr), latex(expand(expr))))
...
x + y &= x + y\\
\left(x + y\right)^{2} &= x^{2} + 2 x y + y^{2}\\
>>> print(r"\end{align*}")
\end{align*}
```

Auch die Berechnung und Darstellung von Ableitungen ist Dank `sympy` kein Problem:

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

Der folgende Quellcode sorgte für die obige Ausgabe:

```
\begin{pycode}
funktionen = [sin(x), cos(x), tan(x)]
print(r"\begin{align*}")
for f in funktionen:
  ableitung= Derivative(f, x)
  print(latex(ableitung) + "&=" + latex(ableitung.doit()) + r"\\")
print(r"\end{align*}")
\end{pycode}
```

3.2.2 Datenbankabfragen

Der Satz von Formeln ist sicherlich eher für den wissenschaftlich orientierten Anwender interessant. Es lassen sich aber auch ganz profane Aufgaben mit PythonTEX realisieren. Denken Sie an ein Textdokument, beispielsweise einen Serienbrief mit vielen Veränderlichen, den sie tausendfach versenden wollen. Auch Statistiken, Controllingberichte und ähnliche Dinge mehr, sind typische Beispiele für Dokumente, in denen veränderliche Daten verarbeitet und formatiert aufbereitet werden müssen. Massendaten liegen üblicherweise in einer Datenbank. Auch auf die kann mit Python aus LATEX zugegriffen werden. Die folgende Tabelle wurde auf diese Weise erstellt.

Id	**Produkt**	**Preis**
1	Hocker	10.00
2	Stuhl	25.50
3	Sessel	75.90
4	Sofa	250.50
5	Tisch	110.00
6	Schrank	333.33
7	Beistelltisch	49.90

Durchschnittspreis: 122.16

Die Beispiel-Tabelle enthält nur wenige Daten und die vorgenommene Auswertung ist trivial. Das Prinzip wird jedoch deutlich. Es ist sehr einfach und effizient.

```
\begin{pythontexcustomcode}{py}
import mysql.connector

def mysqlDemo(minpreis=0):
  con = mysql.connector.connect(user='root',        # Loginname
                                db='seminar',       # Datenbank
                                passwd='12345678') # geheimes Passwort
  cursor = con.cursor()

  query = ("SELECT * FROM produkte where preis >= " + str(minpreis))

  cursor.execute(query)
  durchschnitt = 0.0
  anzahl = 0
  print(r"\begin{tabular}{r|lr}")
```

```
16    print(r"\textbf{Id} & \textbf{Produkt} & \textbf{Preis} \\ \hline")
17    for (produktid, produkt, preis) in cursor:
18        anzahl += 1
19        durchschnitt += float(preis)
20        print(r"{:d} & {} & {:8.2f}\\".format(produktid, produkt, preis))
21    print(r"\end{tabular}")
22    durchschnitt /= anzahl
23    print("")
24    print(r"Durchschnittspreis: {:.2f}".format(durchschnitt))
25
26    cursor.close()
27    con.close()
28  \end{pythontexcustomcode}
```

In einer `pythontexcustumcode`-Umgebung werden eine oder mehrere Funktionen definiert. Diese können dann, von beliebiger Stelle, aufgerufen werden. Die Ausgabe der oben gezeigten Tabelle wurde mit folgendem Aufruf erstellt.

```
1  \begin{center}
2    \pyc{mysqlDemo()}
3  \end{center}
```

Weil der Code in einer `pythontexcustumcode`-Umgebung wie eine Python-Bibliothek im Dokument zur Verfügung steht, können die dort implementierten Routinen mehrfach aufgerufen unf genutzt werden, ohne den Code zu duplizieren. Die obige Tabelle kann also beliebig oft ausgegeben werden. Dazu muss nur `mysqldemo()` erneut aufgerufen werden. Hier habe ich den Parameter `minpreis` genutzt, um nur einen Teil der Daten zu filtern.

```
1  \begin{center}
2    \pyc{mysqlDemo(200)}
3  \end{center}
```

Id	**Produkt**	**Preis**
4	Sofa	250.50
6	Schrank	333.33

Durchschnittspreis: 291.91

3.2.3 Deutsche Zahlendarstellung

Im ersten Beispiel haperte es an den deutschen Tausenderpunkten. Eiegntlich sollte es mit Hilfe von `locale` möglich sein, eine deutsche Zahlenformatierung zu realisieren.

Auf der Website *Python-Tutorial.de* findet sich auch ein schönes Beispiel[1] dazu. Bedauerlicherweise funktioniert es auf meinem Rechner nicht. Ich verwende macOS 10.14 mit Python 3.7.0.

So sollte es laut Python-Dokumentation eigentlich aussehen, mittels `local.format` eine Zahl landesspezifisch zu formatieren:

[1] https://py-tutorial-de.readthedocs.io/de/latest/stdlib2.html

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1.234.567'
```

Und so sieht das aus, wenn ich es in IDLE oder hier in der `pyconsole`-Umgebung ausführe:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> x = 1234567
>>> locale.format("%d", x, grouping=True)
'1234567'
```

Es fehlen ganz augenscheinlich die Tausenderpunkte. Die Option `grouping=True` wird anscheinend ignoriert. Der Tausenderpunkt ist als Tausendertrennzeichen in den Lokaleinstellungen angegeben. Ich habe mich vergewissert. Warum er nicht genutzt wird, weiß ich jedoch im Moment nicht.

Die Umgebung `pythontexcustomcode` liefert jedoch den Schlüssel zur Lösung des Problems. Wir schreiben, so wie wir das schon für die Datenbankabfrage getan haben, einfach eine entsprechende Funktion, die uns die gewünschte Formatierung liefert. Sie soll `mitTausenderPunkten` heißen und wie folgt genutzt werden können.

```
\[
2^{100}=\pyc{print(intMitTausenderpunkten(2**100))}
\]
```

Die Funktion selbst ist denkbar einfach.

```
1  \begin{pythontexcustomcode}{py}
2  def intMitTausenderpunkten(x):
3      if x < 0:
4          return '-' + intMitTausenderpunkten(-x)
5      ergebnis = ''
6      while x >= 1000:
7          x, r = divmod(x, 1000)
8          ergebnis = ".%03d%s" % (r, ergebnis)
9      return "%d%s" % (x, ergebnis)
10 \end{pythontexcustomcode}
```

Auch wenn die vorgestellte Funktion eher trivial ist, macht Sie doch deutlich, wie einfach wir uns eigene Formatierungen und ähnliches basteln und konsistent in unseren Dokumenten verwenden können.

Allgemein und mehrfach nutzbare Funktionen in einem Dokument unterbringen zu können, ist eine fantastische Sache!

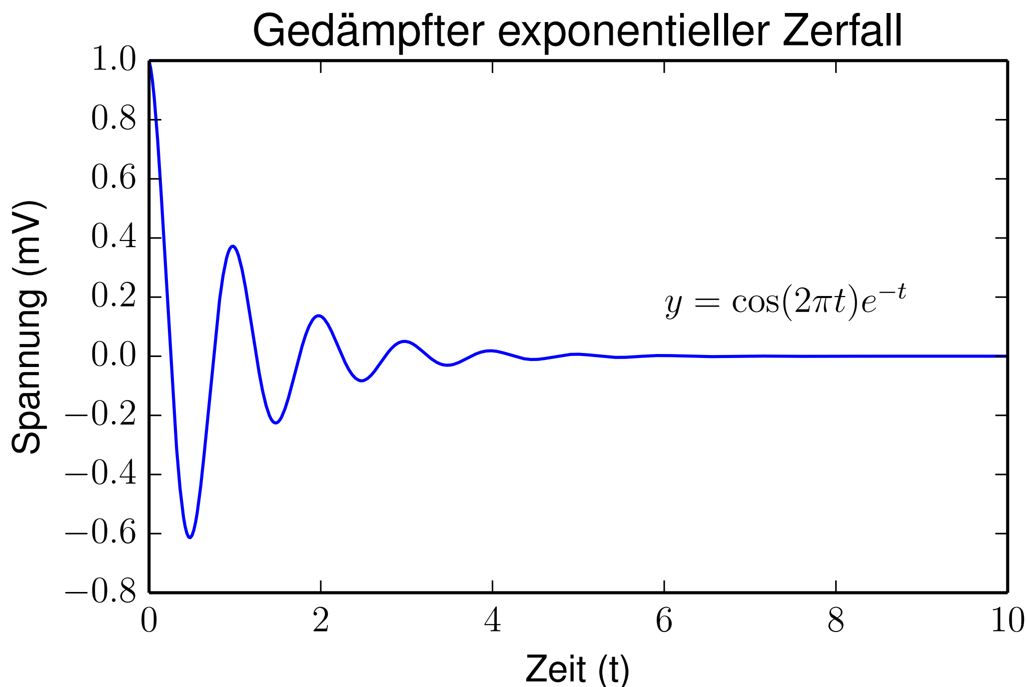
3.2.4 Funktionsplots

Plots von Funktionen sind ein weiterer Anwendungsbereich, für den Python innerhalb von LaTeX hervorragend geeignet ist. Freilich sind *gnuplot* oder auch *PGF/TikZ* tolle Werkzeuge. Mit `pylab` geht es aber, wie ich meine, einfacher und schneller anspruchsvolle Plots in ein Dokument zu integrieren.

Wir wollen hier eine komplexere, physikalische Funktion plotten, wozu wir hier das genannte `pylab` verwenden wollen.

Hinweis: Bei der Verwendung von `pylab` sollte auf Nicht-ASCII-Zeichen verzichtet werden. `pylab` kann mit UTF-8 codierten Zeichen, wie zum Beispiel deutschen Umlauten, nicht korrekt umgehen.

Zuerst möchte ich den zu erstellenden Funktionsgraphen zeigen. Der Quellcode folgt dahinter.



(vergleiche TU München: `https://bit.ly/2C5eEAF`)

`pylab` hat ein Problem mit Zeichen, die nicht dem 7-Bit-ASCII-Code entsprechen. Daher muss auf die „gute, alte LaTeX-Methode“, Umlaute einzugeben, zurückgegriffen werden. Ich persönlich hatte mich davon eigentlich schon mehr oder minder verabschiedet. Beachten Sie deshalb den Raw-String in Zeile 12 des Quellcodes.

```
1  \begin{pycode}
2  from pylab import *
3  def f(t):
4    return cos(2 * pi * t) * exp(-t)
5
6  t = linspace(0, 10, 500)
7  y = f(t)
8  clf()
9  figure(figsize=(5, 3))
10 rc("text", usetex=True)
11 plot(t, y)
12 title(r'Ged\"ampfter exponentieller Zerfall')  # ACHTUNG: Umlaut!
13 text(6, 0.15, r"$y = \cos(2 \pi t) e^{-t}$")
14 xlabel("Zeit (t)")
15 ylabel("Spannung (mV)")
16 savefig("zerfall.pdf", bbox_inches="tight")
17 print(r"\begin{center}")
18 print(r"\includegraphics[width=0.9\textwidth]{zerfall.pdf}")
19 print(r"\end{center}")
20 \end{pycode}
```

Die augenscheinliche fehlende Unterstützung von Sonderzeichen ist ein wenig ärgerlich. Sie ist jedoch kein Manko von PythonTEX und noch weniger ist sie ein ernsthaftes Hindernis. – So einfach hätte ich mit anderen Mitteln den obigen Plot nicht erstellen können. Wer lieber mit `matplotlib` oder anderen Bibliotheken arbeitet, kann natülich auch diese nutzen. PythonTEX legt einem diesbezüglich keinerlei Beschränkung auf.

4 Fazit

GEOFFREY M. POORE hat mit PythonTEX eine ganz fantastische Arbeit geleistet. Die von mir, in diesem Beitrag, verwendete Version trägt die Versionsnummer 0.16. Das ist eine ziemlich kleine Versionsnummer, weit entfernt von einem *Production Release*. Dafür leistet PythonTEX aber wirklich erstaunlich viel und zuverlässig. Ich bin bei meinen Experimenten auf keinen Fehler gestoßen.

Kurz und gut: PythonTEX ist mehr als nur einen kurzen Blick wert. Damit eröffnet sich quasi die gesamte Python-Welt für LATEX. PythonTEX ist ganz besonders gut für die Erstellung mathematischer und wissenschaftlicher Dokumente geeignet. Studenten und wissenschaftliche Mitarbeiter an Hochschulen werden also ihre Freude daran haben. Berechnungen und Formelsatz werden massiv vereinfacht und beschleunigt. Aber auch der nicht-wissenschaftliche Anwender kann von den Fähigkeiten von PythonTEX in hohem Maße profitieren. Dokumente lassen sich auf extrem einfache Art und Weise dynamisieren und automatisieren, um Controllingberichte, statistische Analysen und ähnliches mehr zu erstellen. Besonders reizvoll ist dabei sicherlich die Verknüpfung von Dokumenten mit Datenbankabfragen.

PythonTEX kann noch weit mehr, als ich hier vorgestellt habe. Letztlich kann alles, was Python kann, für die Dokumenterstellung genutzt werden. – Eine wirklich tolle Sache!

Karsten Brodmann