PythonTeX and LaTeX – Familiarize us with PythonTeX in order to build the CrypTool Book

Learning via Samples and Counter Examples

Version 1.01, 2022-03-08, by Bernhard Esslinger and students
esslinger@cryptool.org, `www.cryptool.org`

When writing LaTeX documents with formulas or results of mathematical operations it is sometimes cumbersome to do so, especially when things have to be changed often or have to be dynamic. There are packages to automate this by including **Python** code within a **LaTeX** document, execute the code, and access its output in the original document. An according package is **PythonTeX** which also supports the inclusion of other languages like Bash, Rust, or SageMath.

This is achieved by using PythonTeX macros by a TeX engine like `pdflatex`. Then `pythontex` is applied to the generated ".pytxcode" intermediate file to execute the Python code. Then the TeX engine is executed again – see Fig. 1.

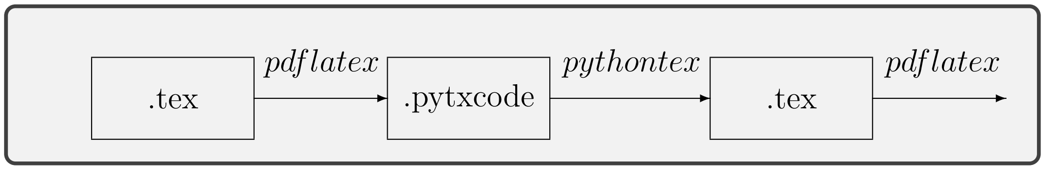


Figure 1: Needed program chain to operate on (intermediate) files

This document explains with many **examples** (like automatically create meaningful LaTeX tables) what can be achieved and what not. On the next page it lists some great **references** for further information. The **table of contents (TOC)** is at the end of this document.

1 Relevant resources used

[1] Karsten Brodmann. Dokumentenautomation mit PythonTeX. 10 page document in German. “Ultimately, anything that Python can do, can be used by PythonTeX for document creation.”, 2018. URL: https://punkt-akademie.de/wp-content/uploads/2018/10/PythonTeX.pdf [cited 2022-02-12].

[2] Andrew Mertz and William Slough. A Gentle Introduction to PythonTeX, 2013. URL: https://tug.org/tug2013/slides/Mertz-A_Gentle_Introduction_to_PythonTeX.pdf [cited 2022-02-12].

[3] B. Nettles and G. Poore. Using Python and pdflatex to Generate Customized Physics Problems. 18 slides, 2016. URL: https://www.aapt.org/docdirectory/meetingpresentations/WM16/AAPTpaper_CI07_Nettles.pdf [cited 2022-02-12].

[4] Günter Partosch. Wie kann ich die Möglichkeiten von Python für LaTeX nutzen? 148 slides, 2019. URL: https://github.com/GuenterPartosch/Vortraege-Kurse/blob/master/TeX+LaTeX/python+latex-alles.pdf [cited 2022-03-08].

[5] G. Poore. PythonTeX Quickstart. 3 pages. URL: http://tug.ctan.org/macros/latex/contrib/pythontex/pythontex_quickstart.pdf [cited 2022-02-12].

[6] G. Poore. The PythonTeX package. Last version from 2021-06-06. URL: https://ctan.org/pkg/pythontex [cited 2022-02-11].

[7] G. Poore. PythonTeX: Fast Access to Python from within LaTeX. 20:45 min presentation from SciPy2012, 2012. URL: https://www.youtube.com/watch?v=yIO4l0zHGjw [cited 2022-02-12].

[8] G. Poore. Advances in PythonTeX with an introduction to fvextra. 6 page document, TUGboat, Volume 37, No. 2, 2016. URL: https://tug.org/TUGboat/tb37-2/tb116poore.pdf [cited 2022-02-12].

[9] U. Ziegenhagen. Combining LaTeX with Python. 41 slides from TUG 2019. URL: https://tug.org/tug2019/slides/slides-ziegenhagen-python.pdf [cited 2022-02-12].

2 Command overview and some preliminary remarks

- Comments in LaTeX start with `%`, but in Python with `#`.
  To interchange these characters is a typical beginner's pitfall, especially if the tex file contains both sorts of code.

- The sources of this document are in the file `PythonTex-by-Examples.tex`, which includes LaTeX and PythonTeX code.
  To build the pdf from the document, you have to call `pdflatex, pythontex, pdflatex` (in this order). To make this easier from the command line, arara commands were inserted at the beginning of the file `PythonTex-by-Examples.tex`.
  Then the compilation is done by: `$ arara PythonTex-by-Examples.tex`
  Our environment is: Tex Live 2021, arara 6.1.1, and PythonTeX 0.18 under Ubuntu 20.04 with Python 3.8.10.

- PythonTeX writes a copy of the single Python commands to the file `PythonTex-by-Examples.pytxcode`.
  In the sub directory `pythontex-files-PythonTex-by-Examples`, further PythonTeX output can be found. Sometimes it is necessary for reruns or debugging to also delete the files in this sub directory. If PythonTeX gives "?? PythonTeX ??" or "??", that's an indication that it hasn't run, so no content has been generated.

- As debugging syntax mistakes in the PythonTeX code is not obvious, here the option `makestderr` for the `pythontex` package is set. This writes the Python stderr output to the file `py_default_default_0.stderr` which is in the sub directory mentioned above.

- Generally, PythonTeX offers the following 7 environments:
  - `pycode` Code executed, but not typeset
  - `pyblock` Code executed, but output only the prettyprinted code
    To show anything printed, use `\printpythontex` or `\stdoutpythontex`.
  - `pysub` Variable and expression substitution
  - `pyverbatim` Typesetting, but no code executed
    Same as `\begin{pygments}{python} ... \end{pygments}`

- `pygments` General code typesetting
- `pyconsole` Simulates an interactive Python console, i.e. executes the commands (like `pycode`) after `>>>`
- `pythontexcustomcode` Code executed like pycode, but also prints are not shown.

- Generally, PythonTeX offers 7 inline commands to be used within normal LaTeX running text:
  - `\py` Evaluates a Python expression and prints its value. Here also functions defined via `pycode` can be used.
  - `\pyc` Executes code, output goes to `stdout`, does not typeset it
  - `\pyb` Executes code, but outputs only the code prettyprinted. To show anything printed, use **`\printpythontex`** or **`\stdoutpythontex`**.
  - `\pys` Supports variable and expression substitution
  - `\pyv` Prettyprints code (nothing is executed).
    - Same as `\pygment{python}`
    - Other than the `pyverbatim` env no new paragraph is created.
  - `\pygment` General code typesetting
  - `\pycon` Accesses a variable from a previous `pyconsole` block

- PythonTeX provides many additional features, e.g. with the `mathescape` option and the `fvextra` package. See documentation.

- The `depythontex` utility creates a copy of a document in which all PythonTeX commands and environments have been replaced by their output. The resulting document is more suitable for journal submission and conversion to other document formats. This is necessary as many publishers will not accept LaTeX documents that require special packages or need special macros.

Many more details can be found in "The PythonTeX package" by G. Poore (`https://ctan.org/pkg/pythontex`).

Why was this document created? We familarized us with PythonTeX and SageTeX in order to automate building parts in the CrypTool Book. So we share some experiences we made. The most important samples for us have been those in Sections 3.8, 10, 11, and 14.

3 Execute Python code with `pycode` and evaluate Python expressions with `py` and `pyc`

Calculations can be done either in a `block` environment (like `pycode` or `pyblock`) or `inline` with macros like `py`.

The `\py` command sends code to Python, and Python returns a string representation of the code. Opening and closing delimiters must be either a pair of identical, non-space characters, or a pair of curly braces. If curly braces are used as delimiters, then curly braces may only be used within the code if they are paired. Thus, `\py{1+1}` sends the code "1+1" to Python, Python evaluates the string representation of this code, and the result "2" is returned to LaTeX. The commands `\py`#1+1# and `\py`@1+1@ would have the same effect. The command can also be used to access variable values. For example, if the code "a = 1" had been executed previously, then `\py{a}` simply brings the string representation of *a* back into the document as "1".

Assignment is not allowed using `\py`. For example, `\py{a = 1}` is not valid. This is because assignment cannot be converted to a string. The text returned by Python must be valid LaTeX code. Verbatim and other special content is allowed. The primary reason for using `\py` rather than print is that `\py` is more compact.

3.1 Execute simple Python code with `pyblock`

This is a minimal working example (MWE) of a LaTeX file with PythonTeX source code included.

```
% Minimal working example (MWE) of a LaTeX file with
  PythonTeX source code
% arara: pdflatex
% arara: pythontex
% arara: pdflatex

\documentclass[12pt]{article}
\usepackage[makestderr]{pythontex}
\begin{document}

```

```latex
\begin{pyblock}[][numbers=left]
a,b = 2,3
print(a, b, 'Sum: ', a+b)
\end{pyblock}
\stdoutpythontex  % \printpythontex

\begin{pyblock}[][numbers=left]
a = 5
print('Changed in second block: a = ', a)
\end{pyblock}
\printpythontex  % \stderrpythontex

\vspace{1cm}\noindent The variable is $a=\py{a}$.

\end{document}
```

And here is the output of the code in the above listing. The code within the blocks is printed (with highlighting). The results (outputs of `print`) are shown by calling `stdoutpythontex` and `printpythontex` **after** the blocks.

```python
a,b = 2,3
print(a, b, 'Sum: ', a+b)
```

2 3 Sum: 5

```python
a = 5
print('Changed in second block: a = ', a)
```

Changed in second block: a = 5

The variable is $a = 5$.

3.2 Execute Python code with `pycode` and `numpy`

```
\begin{pycode}
print(r'\begin{center}')
print(r'\textit{A message from Python!}')
print(r'\end{center}')

print("[pycode] Hello \LaTeX")
import numpy as np
\end{pycode}

[After the pycode block] Random value via Python and np: \py{np.random.randint(10)}
```

A message from Python!

[pycode] Hello LaTeX

[After the pycode block] Random value via Python and np: 1

Remarks:
a) After beginpycode there is no comment allowed.
b) The code inside the block must begin in the 1st column.

3.3 Show difference between `\py` and `\pyc`

```
\py{print("[py~] Bye \LaTeX 2")}
```

[py] Bye LaTeX2 None

We get “None” at the end because **`\py`** additionally evaluates the expression. For `print` it’s better to use **`\pyc`** instead of **`\py`** which works like `begin{pycode}` and avoids “None”.

```
 \pyc{print("[pyc] Hello \LaTeX 3")}

 [py~] \py{2+5}
```

[pyc] Hello LaTeX3

[py] 7

For calculating `2+5` we need the evaluation, so `\py` was correct [using `\pyc` here causes an pdflatex error].

3.4 Use variable from Python block outside in LaTeX code

```
\begin{pycode}
myvar = "hello"
\end{pycode}
Here is the value outside the \texttt{pycode} block: \py{myvar}
```

Here is the value outside the `pycode` block: hello

3.5 Use a self-defined Python function

This self defined function “AddPointAsThousandsSeparator” inserts points in large numbers to separate thousands.

```latex
Via \pygment{latex}{\py}: $ 2^{100}=\py{2**100} $

Via \pygment{latex}{\py}: $ 2^{100}=\py{print("{:,}".format(2**100))} $

Via \pygment{latex}{\pyc}: $ 2^{100}=\pyc{print("{:,}".format(2**100))} $

Via \pygment{latex}{\pyc} and using a function for German number separation:
\[
2^{100}=\pyc{print(AddPointAsThousandsSeparator(2**100))}
\]

\begin{pythontexcustomcode}{py}
def AddPointAsThousandsSeparator(x):
    if x < 0:
        return'-' + AddPointAsThousandsSeparator(-x)
    result = ''
    while x >= 1000:
        x, r =divmod(x, 1000)
        result = ".%03d%s" % (r, result)
    return "%d%s" % (x, result)
\end{pythontexcustomcode}
```

Via `\py`: $2^{100} = 1267650600228229401496703205376$

Via `\py`: $2^{100} = 1,267,650,600,228,229,401,496,703,205,376None$

Via `\pyc`: $2^{100} = 1,267,650,600,228,229,401,496,703,205,376$

Via `\pyc` and using a function for German number separation:

$$2^{100} = 1.267.650.600.228.229.401.496.703.205.376$$

Remark 1: In the second line we get “None” at the end because `\py` additionally evaluates the expression. For `print` it’s better to use `\pyc` like in the third line (instead of `\py`), which avoids “None”.

Remark 2: This listing is just a sample. A better way to handle language-dependent separators is with the `siunitx` package and the command \`num`. . . .

3.6 Use variable and function inside the pycode block

```
\begin{pycode}
x = 3
f = x**2; g = 1 / f
print('f:', f, '~~~//~~~g:', g, '~~~//~~~')

def h(x):
    return(x**3)

print('h(3):', h(3))
\end{pycode}

% [latex and py]
[After the pycode block] The variable $x$ and the function
    $h$ are also known outside the pycode block: ~$x$ = \py{x
    }, ~~$h(4)$ = \py{h(4)}.
```

f: 9 // g: 0.1111111111111111 // h(3): 27

[After the pycode block] The variable x and the function h are also known outside the pycode block: $x = 3$, $h(4) = 64$.

3.7 Use \pyc within \newcommand

```
\newcommand{\setmystring}[1]{\pyc{str="#1"}}
\setmystring{123 hello 456}

The value of $str$ is:
\begin{pycode}
print(str)
\end{pycode}
```

The value of *str* is: 123 hello 456

3.8 Invert a string / exchange first and last letter / pow (LaTeX commands defined with \py)

This is a very nice possibility to manipulate strings with Python features instead of with LaTeX and TeX only.

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

\newcommand{\swapfirstlast}[1]{%
     #1:~~~\pyc{s = "#1"} \py{s[-1] + s[1:-1] + s[0]}}

Invert:   \reverse{a129z}\\
Exchange: \swapfirstlast{a129z}
%
\newcommand{\pow}[2]{\py{#1**#2}}
Power: \pow{2}{8}
```

```
Invert:    a129z:   z921a
Exchange:  a129z:   z129a

Power: 256
```

Some of these examples are from https://raw.githubusercontent.com/gpoore/pythontex/master/pythontex/pythontex.pdf, Section 6.

The article [8] (https://tug.org/TUGboat/tb37-2/tb116poore.pdf) by G. Poore from 2016 gives in Section 4 a good explanation why this interplay between PythonTeX and LaTeX works: “LaTeX handles all text before code is seen by Python for evaluation, and then the result of evaluation is brought in during the next compile.”

4 pyb: Just show the Python code with \pyb

\pyc and **\pyb** both execute the Python code. Then **\pyc** writes the output to stdout, but does not typeset the code. **\pyb** does it the other way around and outputs only the code prettyprinted.

```
{[}pyb:{]} \pyb{'1 Python says hi!\n'}
{[}pyb:{]} \pyb{print('2 Python says hi!\n')}
{[}pyc:{]} \pyc{print('3 Python says hi!\n')}
```

[pyb:] `'1 Python says hi!\n'`

[pyb:] `print('2 Python says hi!\n')`

[pyc:] 3 Python says hi!

5 Performing code at the Python console

```
\begin{pyconsole}
x = 87.27
x = x**2
x
\end{pyconsole}

The variable $x$ is also available outside \pygment{latex}{\
   pyconsole} (the Python terminal): $x=\pycon{x}$
```

```
>>> x = 87.27
>>> x = x**2
>>> x
7616.0529
```

The variable x is also available outside `pyconsole` (the Python terminal): $x =$ 7616.0529

Remark: By default, the results from the console is printed in grey. For the `pyconsole` block this can be turned to black by adding in the preamble:
    `\setpygmentspygopt[pyconsole]{style=bw}`
However this didn't effect how **\pycon** colored the output.

6 Samples with Sympy

6.1 Automated math formulas Source is executed and shown via `sympyblock`. Output is shown via `printpythontex`.

The following sample from PythonTeX Gallery shows how PythonTEX can do some amazing document automation, such as this derivative and integral table. With `latex()` used in `sympyblock` the mathematical terms are nicely formatted. Without `sympy` and only using Python, functions like `sin` are not accessible.

Here follows the whole source code using our `lstlisting`.

```
\begin{sympyblock}[][numbers=left,frame=single,framesep=5mm,label=An Automated Derivative and Integral Table]

var('x')

# Create a list of functions to include in the table
funcs = ['sin(x)',    'cos(x)',    'tan(x)',
#         'sin(x)**2', 'cos(x)**2', 'tan(x)**2',
         'asin(x)',   'acos(x)',   'atan(x)',
         ]

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

for func in funcs:
    # Put in some vertical space when switching to arc
    #                           and hyperbolic funcs
    if func == 'asin(x)' or func == 'sinh(x)':
        print(r'&\\')
    myderiv = 'Derivative(' + func + ', x)'
    myint = 'Integral(' + func + ', x)'
    print(latex(eval(myderiv)) + '&=' +
            latex(eval(myderiv + '.doit()')) + r'\quad & \quad')
    print(latex(eval(myint)) + '&=' +
            latex(eval(myint+'.doit()')) + r'\\')
print(r'\end{align*}')

\end{sympyblock}
```

Here only the **source code** follows which is WITHIN the `sympy` block. This is just to show, that **sympyblock** prints the source code (independent of our listing). The alternative **sympycode** would not print the source code by itself.

An Automated Derivative and Integral Table

```python
var('x')

# Create a list of functions to include in the table
funcs = ['sin(x)',    'cos(x)',    'tan(x)',
#         'sin(x)**2', 'cos(x)**2', 'tan(x)**2',
         'asin(x)',   'acos(x)',   'atan(x)',
         ]

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

for func in funcs:
    # Put in some vertical space when switching to arc
    #                              and hyperbolic funcs
    if func == 'asin(x)' or func == 'sinh(x)':
        print(r'&\\')
    myderiv = 'Derivative(' + func + ', x)'
    myint = 'Integral(' + func + ', x)'
    print(latex(eval(myderiv)) + '&=' +
            latex(eval(myderiv + '.doit()')) + r'\quad & \quad')
    print(latex(eval(myint)) + '&=' +
            latex(eval(myint+'.doit()')) + r'\\')

print(r'\end{align*}')
```

This text here is just to emphasize that the **output** is printed only when calling `\printpythontex` in the document.

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

$$\frac{d}{dx} \operatorname{asin}{\left(x \right)} = \frac{1}{\sqrt{1 - x^{2}}} \qquad \int \operatorname{asin}{\left(x \right)}\, dx = x \operatorname{asin}{\left(x \right)} + \sqrt{1 - x^{2}}$$
$$\frac{d}{dx} \operatorname{acos}{\left(x \right)} = - \frac{1}{\sqrt{1 - x^{2}}} \qquad \int \operatorname{acos}{\left(x \right)}\, dx = x \operatorname{acos}{\left(x \right)} - \sqrt{1 - x^{2}}$$
$$\frac{d}{dx} \operatorname{atan}{\left(x \right)} = \frac{1}{x^{2} + 1} \qquad \int \operatorname{atan}{\left(x \right)}\, dx = x \operatorname{atan}{\left(x \right)} - \frac{\log{\left(x^{2} + 1 \right)}}{2}$$

`Remark:`

It is possible to show the value of a variable from the previous `sympy` block in the LaTeX environment like it was done with variables from the `pycode` block (see Sections 3.6 and 12.1). However, not via `\py` and only in math mode – thanks to Geoff Poore.

So the command `Show a variable value: \py{funcs[2]}`
causes the `NameError: name 'funcs' is not defined`;
and the command `Show a variable value: \sympy{funcs[2]}`
causes the `LaTeX Error: \mathtt allowed only in math mode`.

It works this way: `$\sympy{funcs[2]}$` : tan(x)

Explanation: `sympy` wraps its arguments with the latex() function to handle math, so it often needs math mode depending on the situation. Using `py` doesn't work because all commands with "py" vs "sympy" in the names execute code in separate sessions or interpreters. So there isn't shared data between code executed with the different sets of commands.

6.2 Integration of a term. Source is executed via `pycode` importing `sympy` functions

The integration using the sympy function `\int` worked well, but the output of the function and the presentation of the results differed:
- fraction versus decimal number (1a versus 1b)
- root sign version exponent as fraction (1a versus 1b)
- root sign versus others (1a versus 1b and 2b)

In addition, in the second sub section I tried to not write the function term twice.

6.2.1 All arguments explicitely given

```
\begin{pycode}
from sympy import *
x = symbols('x')
result1a = latex(integrate("(1+x)**(1/2)",x))
result2a = latex(integrate((1+x)**(1/2),x))
# %??? result3a = latex(integrate(str((1+x)**(1/2)),x))
\end{pycode}

1a Result of integrating $\int \sqrt{1+x} \, dx$ is: $\py{
 result1a}$\\
2a Result of integrating $\int \sqrt{1+x} \, dx$ is: $\py{
 result2a}$\\
% 3a Result of integrating $\int \sqrt{1+x} \, dx$ is: $\py
 {result3a}$\\
```

1a Result of integrating $\int \sqrt{1+x} \, dx$ is: $\frac{2(x+1)^{\frac{3}{2}}}{3}$
2a Result of integrating $\int \sqrt{1+x} \, dx$ is: $0.666666666666667 \left(x + 1\right)^{1.5}$

Remark: The results **result1a** and **result2a** are displayed differently: In the first case the integral is defined symbolically (and shows the quotients of integers), in the second case it's defined numerically (and shows decimal numbers). However, the display of the function itself doesn't look well.

6.2.2 Write down the math function f only once

The following sample tries to define the code for a function f only once. The aim was to have here the same output as in Section 6.2.1 in the first line (1a).

Here we have two attempts, the second one worked well:

a) Numerical definition of the integral Here again the function is defines as a string ($f0$) and in the normal Python way ($f1$).

```
\begin{pycode}
from sympy import *
x = symbols('x')
f0 = "(1+x)**(1/2)"; latf0 = latex(f0)
f1 = (1+x)**(1/2)  ; latf1 = latex(f1)
# f2 = str(f)      ; latf2 = latex(f2)  # PythonTeX:
  TypeError: 'str' object is not callable
result1b = latex(integrate(f0,x))
result2b = latex(integrate(f1,x))
# result3b = latex(integrate(f2,x))
\end{pycode}

1b Result of integrating $\int \py{latf0} \, dx$ is: $\py{
  result1b}$\\
2b Result of integrating $\int \py{latf1} \, dx$ is: $\py{
  result2b}$\\
% 3b Result of integrating $\int \py{latf2} \, dx$ is: $\py
  {result3b}$
```

1b Result of integrating $\int$ (1+x)**(1/2) dx is: $\frac{2(x+1)^{\frac{3}{2}}}{3}$
2b Result of integrating $\int (x+1)^{0.5}\, dx$ is: $0.666666666666667\,(x+1)^{1.5}$

Remark: The function f to be integrated appeared twice in our first code snippet in Section 6.2.1: once in the pycode block after `integrate`, and once in the LaTeX output after the word "integrating".

To avoid flaws we tried to write down the code defining f only once. In this attempt the terms are still not displayed in a consistent way (we wanted fractions of integers, no decimal numbers).

[Remark: Showing it with fractions would be the default with the SageTeX package (`https://doc.sagemath.org/html/en/tutorial/sagetex.html`). Also see the SageMath appendix of the CrypTool Book at `https://www.cryptool.org/en/documentation/ctbook/`.]

b) Symbolic definition of the integral Displaying it with fractions can be achieved here too by defining the integral and the function $f2$ symbolically. This doesn't allow to define function f as string and there is also no need to do so. So this overcomes the previous code in this section 6.2.2. Thanks to Geoff again.

```
\begin{pycode}
from sympy import *
x = symbols('x')
f1 = (1 + x)**(1/2)         ; integral1 = Integral(f1,x)
f2 = (1 + x)**Rational(1,2); integral2 = Integral(f2,x)
\end{pycode}

1c Result of integrating $\displaystyle\py{latex(integral1)} = \py{latex(integral1.doit())}$\\
2c Result of integrating $\displaystyle\py{latex(integral2)} = \py{latex(integral2.doit())}$
```

1c Result of integrating $\displaystyle\int (x+1)^{0.5}\, dx = 0.666666666666667\,(x+1)^{1.5}$

2c Result of integrating $\displaystyle\int \sqrt{x+1}\, dx = \frac{2\,(x+1)^{\frac{3}{2}}}{3}$

Function definition $f2$ and output **2c** is the consistent way intended.

6.2.3 Putting the integration into a function

The following code snippet bundles the code in the `pycode` block from Section 6.2.1 into a function [the code of this subsection is from the website https://www.12000.org/my_notes/python_in_latex/index.htm, Section 3 “Calling Python passing an argument to process it and getting the LaTeX back”, which reuses many samples from the references listed at Section 1 and offers for each sample a ready-to-run LaTeX document]. This could be further improved by defining it like in the above paragraph b) Symbolic definition of the integral.

```
\begin{pycode}
from sympy import *
def inte(theIntegrand,var):
    var  = symbols(var)
    anti = integrate(theIntegrand,var)
    return latex(anti)
\end{pycode}

The result of integrating $\int \frac{1}{\sqrt{1+x}} \, dx$
is given by $\py{inte("1/(1+x)**(1/2)","x")}$

Here is a list of six integrations:
\begin{align*}
\int \frac{1}{\sqrt{1+x}} \, dx &=  \py{inte("1/(1+x)**(1/2)"
    ,"x")}\\
\int \sin x \, dx &=  \py{inte("sin(x)","x")}\\
\int x \sin x \, dx &=  \py{inte("x*sin(x)","x")}\\
\int x^2 \sin x \, dx &=  \py{inte("x**2 * sin(x)","x")}\\
\int x e^{2x} \, dx &=  \py{inte("x*exp(2*x)","x")}\\
\int \frac{1}{1+u} \, du &=  \py{inte("1/(1+u)","u")}\\
\end{align*}
```

The result of integrating $\int \frac{1}{\sqrt{1+x}} dx$ is given by $2\sqrt{x+1}$

Here is a list of six integrations:

$$\begin{aligned}
\int \frac{1}{\sqrt{1+x}}\,dx &= 2\sqrt{x+1} \\
\int \sin x\,dx &= -\cos{(x)} \\
\int x\sin x\,dx &= -x\cos{(x)} + \sin{(x)} \\
\int x^2\sin x\,dx &= -x^2\cos{(x)} + 2x\sin{(x)} + 2\cos{(x)} \\
\int xe^{2x}\,dx &= \frac{(2x-1)\,e^{2x}}{4} \\
\int \frac{1}{1+u}\,du &= \log{(u+1)}
\end{aligned}$$

7 Create and expand binoms with Sympy

This sample is from [1], Section 3.2.1 “Mathematik und Formelsatz”, written by Karsten Brodmann. Very similar code is in [4], 2018, slide 36 by G. Partosch, and in [2], 2013, slide 28 by Mertz/Slough.

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

The `latex` command from `sympy` does the setting of the formulas. The `print` command from `pycode` outputs the LaTeX code for this document.

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

8 Create and include a function plot with matplotlib

The paper [1] referred before in Section 7 has two more interesting samples:

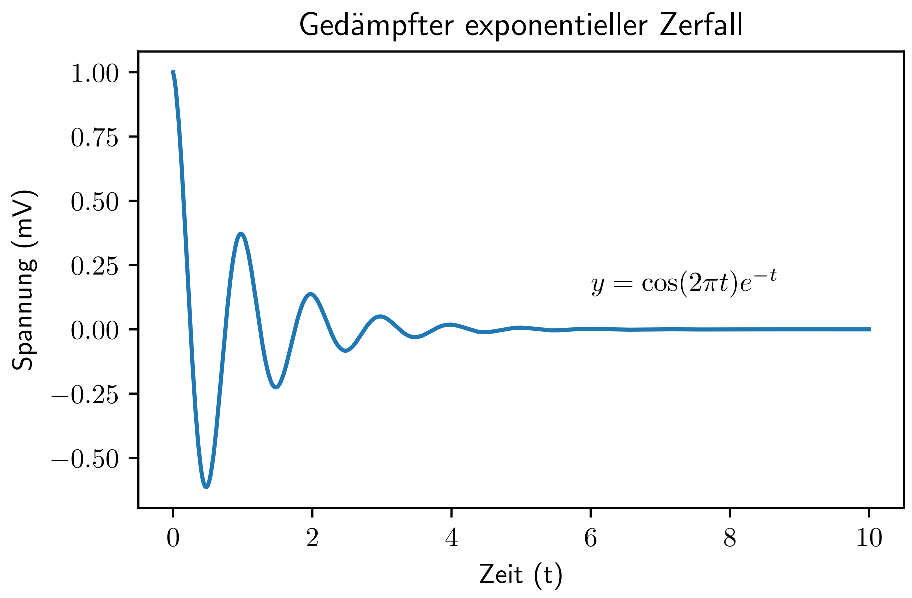
- Section 3.2.2 for getting data from a mysql database and
- Section 3.2.4 for creating a function plot with pylab instead of math-plotlib)

Below is the one for creating a function plot (very similar code is in [4], slide 78-82 (file pytex38.text) and in [2], slide 33) which seems to be the origin).

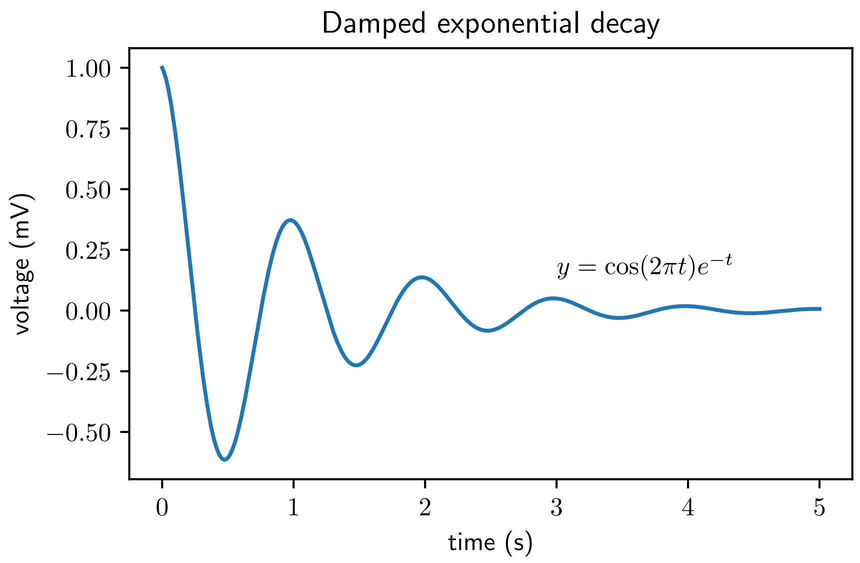
As `pylab` which was used in the cited literature, is depricated it was substituted here by the two imports (`matplotlib` and `numpy`) which pylab called internally.

```
\begin{pycode}
from matplotlib.pyplot import *
from numpy import *

def f(t):
    return cos(2 * pi * t) * exp(-t)

t = linspace(0, 10, 500)
y = f(t)
clf()
figure(figsize = (5, 3))
rc("text", usetex=True)
plot(t, y)
title(r'Ged\"ampfter exponentieller Zerfall') # Achtung:
   Umlaut!
text(6, 0.15, r"$y = \cos(2 \pi t) e^{-t}$")
xlabel("Zeit (t)")
ylabel("Spannung (mV)")
savefig("zerfall.pdf", bbox_inches="tight")
print(r"\begin{center}")
print(r"\includegraphics[width=0.9\textwidth]{zerfall.pdf}")
print(r"\end{center}")
\end{pycode}
```

The **pycode** sample above generates the following plot (with German labels) and stores it in the file “zerfall.pdf”.



And here the according version with English labels from [2]:



9 Sample: Python code with internet calls; result written in table

Here the local `includecomment` variable “CVInternetisavailable” is enabled.

Calling `pip install yahoo_fin` is needed before performing the code.

```
\pyc{from yahoo_fin import stock_info as si}

\begin{tabular}{lr} \toprule
Company & Latest quote \\ \midrule
& (usd)\\ \midrule
Apple & \py{round(si.get_live_price("aapl"),2)} \\
Amazon & \py{round(si.get_live_price("amzn"),2)} \\
Facebook/Meta & \py{round(si.get_live_price("fb"),2)} \\
Google/Alphabet & \py{round(si.get_live_price("goog"),2)} \\
Microsoft & \py{round(si.get_live_price("msft"),2)} \\
\midrule
& (euro)\\ \midrule
Microsoft & \py{round(si.get_live_price("msf.de"),2)} \\
\bottomrule
\end{tabular}
```

Via `\pyc` and `\py` the stock price at March 8, 2022 at 15:15:

Company	Latest quote
	(usd)
Apple	163.17
Amazon	2912.82
Facebook/Meta	200.06
Google/Alphabet	2642.44
Microsoft	289.86
	(euro)
Microsoft	263.65

10 Create a list with Python and show it with LaTeX

Creating lists in Python is much easier than creating them purely with LaTeX syntax. Here, two lists are defined in Python and then shown in LaTeX.

```
\begin{pycode}
my_list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print("my\_list1: ", my_list1, "\n")
result = filter(lambda x: x % 2 == 0, my_list1)
resultL = list(result) # Assign to a variable to have access
    more than once.

print("resultL: ", resultL, "~~~~~(elements of my\_list1
    which can be divided by 2)\n")

print("Second value in my\_list1:", my_list1[1], "\n")  #
    escape necessary
print("Second value in resultL:", resultL[1], "\n")
\end{pycode}

After the \texttt{pycode} block a variable of the block can
    be accessed inline the LaTeX text:\\
Second value in \texttt{my\_list1}: \py{my_list1[1]}\\
Second value in \texttt{resultL}:  \py{resultL[1]}
```

my_list1: [1, 2, 3, 4, 5, 6, 7, 8, 9]

resultL: [2, 4, 6, 8]   (elements of my_list1 which can be divided by 2)

Second value in my_list1: 2

Second value in resultL: 4

After the `pycode` block a variable defined in the block can be accessed in the LaTeX text via \py:

  Second value in `my_list1`: 2
  Second value in `resultL`: 4

Here is another `pycode` block, now with strings instead of numbers in the list:

```
\begin{pycode}
my_list2 = ["alpha", "beta", "gamma"]
print("my\_list2: ", my_list2, "\n")
\end{pycode}
```

my_list2: ['alpha', 'beta', 'gamma']

```
Again access the \texttt{pycode} block variable \texttt{after
   } the block:\\
\indentation Second element in \texttt{my\_list2}: \py{
   my_list2[1]}
```

Again access the `pycode` block variable `after` the block:
Second element in `my_list2`: beta

11 Dynamic creation of a tabular environment using a Python loop

LaTeX tables can be created dynamically using `pyblock`. The loop variable can be passed over and used as index for arrays defined in another `pyblock`.

What is not involved here is passing the loop variable as parameter to a Python function.

11.1 Print the code of a `pyblock` environment and print the code's result via `printpythontex`

The Python code generates the LaTeX commands to build the table. The LaTeX code is shown via `printpythontex`. And these LaTeX commands are then processed with the TeX engine to show the resulting table.
Remark: Instead of `pyblock` and `printpythontex`, we could have also used `lstlisting` and `pycode` (like we did before).

```
anfang, ende = 1, 5
print(r"\begin{tabular}{r|r}")  # raw string r"..." saves burdon
print(r"$m$ & $2^m$ \\ \hline") #     of escaping backslash.
for m in range(anfang, ende + 1):
    print(m, "&", 2**m, r"\\")
print(r"\end{tabular}")
```

m	2^m
1	2
2	4
3	8
4	16
5	32

Show the value of a variable defined in the previous `pyblock`: ende = 5.

Explanation: What and how many statements can be put between the end of the `pyblock` and the call of `\printpythontex`?

`\printpythontex` refers back to the most recent PythonTeX environment/-command. So you can have things between pyblock and `\printpythontex`, just nothing that involves PythonTeX.

11.2 Create and print a table via `pycode` in a loop

Here again, the PythonTeX code generates the LaTeX commands to build the table.

Again `pyblock` and `printpythontex` are used like in Section 11.1: `pyblock` generates the LaTeX code and prints the code within the block; `printpythontex` is needed to print the results.

The table created has 2 columns: The first col is the running index of the loop; the second column contains the array elements of `my_list2`, defined in Section 10.

The index of the for loop is considered to be a Python variable. It is used to also act as index in the Python array `my_list2`.

```
print(r"\begin{tabular}{l|r} \toprule")
print(r"Number & String \\ \midrule")
for i in range(0, len(my_list2)):
    print(i+1, "&", my_list2[i], r"\\")
print(r"\end{tabular}")
```

Number	String
1	alpha
2	beta
3	gamma

12 Passing variables – Subsections here show what works well

Current STATUS: Using variables from `LaTeX` and `pycode` being passed to a PythonTeX command like `\py` works well.

However it doesn't work, if `LaTeX` variables are passed as a parameter to a function in `pycode` or in `pythontexcustomcode` [such self-defined functions are like Primzahlen1() and Primzahlen2() below]. Compare Section 15.5 on page 49 to see in more detail what does **not** work.

12.1 Defining PythonTeX variables within pycode and using them outside in a LaTeX context via py

`\py` can be used to insert the value of a variable previously defined in a `pycode` environment or within a command like `\pyc{x = 2**8}`.

1) Passing a PythonTeX variable defined in `pyc{}` to another `pyc{}` works well, also when passed to a self-defined PythonTeX function (here `Primzahlen1()` from 13.1 on page 37).

```
This is LaTeX code including PythonTeX calls:\\
\pyc{mypyvar = 6}   % defining a PythonTeX variable
aa \pyc{mypyvar}\\
aa \pyc{print(mypyvar)}\\
bb \py{mypyvar}\\
cc \py{Primzahlen1(mypyvar)}\\  % prints result plus "None"
cc \pyc{Primzahlen1(mypyvar)}   % get the first 6 primes
```

This is LaTeX code including PythonTeX calls:
aa
aa 6
bb 6
cc 2 3 5 7 11 and 13 None
cc 2 3 5 7 11 and 13

Remark 1: What doesn't work is passing `\py{mypyvar}` or `\pyc{mypyvar}` as function argument instead of `mypyvar`:

```
\begin{pyverbatim}
  \pyc{Primzahlen1(\py{mypyvar})}.  % PythonTeX error
  \pyc{Primzahlen1(\pyc{mypyvar})}.  % PythonTeX error
\end{pyverbatim}
```

Remark 2: A more nice output can be displayed via a `pyverbatim` block, but this requires to extra copy the result.

```
\begin{pyverbatim}
  aa 6
  bb 6
  cc 2 3 5 7 11 and 13
\end{pyverbatim}
```

```
aa 6
bb 6
cc 2 3 5 7 11 and 13
```

2) Further example: Given the following code:

```
  \begin{pycode}
  yvalues = [x**4-13*x**2+36 for x in range(-5,5)]
  sols = [x for x in range(-5,5) if (x**4-13*x**2+36==0)]
  \end{pycode}
```

```
yvalues: [336, 84, 0, 0, 24, 36, 24, 0, 0, 84]

sols: [-3, -2, 2, 3]
```

Remark: Indentation and vertical parskip are tricky, if the print calls are within `\pycode`. It's better to do the prints not in the `pycode` block but afterwards in LaTeX and with inline calls to `\py`.

AFTER finishing `pycode` an inline call in LaTeX text is made with `\py{sols}` and `\py{sols[1]}`:

```
~~~ sols: \py{sols}\\
~~~ Second value in sols: \py{sols[1]}
```

```
    sols: [-3, -2, 2, 3]
Second value in sols: -2
```

12.2 Passing LaTeX variables as a parameter to a PythonTeX command

In the next subsections, there are working samples, how to pass expanded LaTeX parameters to a PythonTeX command [but still no passing of a param to a self-defined `pycode` function which is called within a `pycode` block or within a `\py` command. However, passing to the default print() function works!].

12.2.1 Case 1: Passing a self-defined LaTeX variable to `pyc`

The following samples show the syntax how to self-define LaTeX variables like `\def\a{1}` or `\def\a{\b}`.

```
\def\a{1}
\def\b{3}
a1: \a \\
b1: \b \\
\def\a{7}
a2: \a \\
\def\a{\b}
a3: \a \\
\def\a{3*\b}
a4: \a \\
\def\az{\b}
az1: \az \\
\def\az{\b+\a}
az2: \az
```

And here is their output:
a1: 1
b1: 3
a2: 7
a3: 3
a4: 3*3
az1: 3
az2: 3+3*3

Remark: Self reflection of a LaTeX variable via `\def\a{3*\a}` doesn't work (TeX capacity exceeded).

12.2.2 Case 2: Passing a PythonTeX context variable to `pyc`

Here, `\setpythontexcontext{xxx=<<<var>>>}` is used to define the PythonTeX context variable `xxx`. Context variables must be defined in the **preamble**. Initially it was set to 4.

```
\def\x{2}
\def\y{3}
\setpythontexcontext{xxx=4, yyy=\y}
...
\indentation \pyc{print('xxx: ', pytex.context.xxx)}\\
\indentation \pyc{print('yyy: ', pytex.context.yyy)}
```

```
    xxx: 4
    yyy: 3
```

With `\pyc{print('xxx: ', pytex.context.xxx)}` we get the initial value of the context variables `xxx` and `yyy`.

Now with `\def\x{7}`, a new value is assigned to the self-defined LaTeX variable `x`. This doesn't change the context variable `xxx`.

```
\def\x{7}
\indentation \pyc{print('xxx: ', pytex.context.xxx)}
\indentation x = \x
```

```
    xxx: 4      x = 7
```

Here with `\def\x{\a}`:

```
\def\x{\a}
\indentation x = \x
```

```
    x = 3*3
```

12.2.3 Expand `thesection` before using it with `py`

a) Here, pure LaTeX methods are used to get the current section number.

```
  \newcommand{\getcurrentref}[1]{%
    \ifnum\value{#1}=0 ??\else\csname the#1\endcsname\fi
  }

  \indentation a Section: \getcurrentref{section}\\
  \indentation a Subsection: \getcurrentref{subsection}\\
  \indentation a Subsubsection: \getcurrentref{subsubsection}
```

```
    a Section: 12
    a Subsection: 12.2
    a Subsubsection: 12.2.3
```

b) PythonTeX commands (like `\pyc`) are designed to send code verbatim to Python. Here are two ways to access the `thesection` LaTeX variables.

b1) Here, the trick here is to fully expand `\thesection` or `\thesubsection` before storing it in a Python variable. This requires to define first a macro for each variable.

```
% In preamble
\makeatletter
\newcommand{\setpyvar}[1]{%
  \edef\tmp{#1 = "\csname the#1\endcsname"}%
  \expandafter\pyc\expandafter{\tmp}}
\makeatother

\setpyvar{section}
\indentation b1 Section: \py{section}\\
\setpyvar{subsection}
\indentation b1 Subsection: \py{subsection}\\
\setpyvar{subsubsection}
\indentation b1 Subsubsection: \py{subsubsection}
```

```
b1 Section: 12
b1 Subsection: 12.2
b1 Subsubsection: 12.2.3
```

b2) Another way is to make an assignment with `setpythontexcontext` in the preamble and then call `pytex.context.<<<var>>>`. And this internal LaTeX variable `\y` always seems to be expanded!

```
\setpythontexcontext{xxx=4, yyy=\y}   % was defined in the
 preamble
...
\def\y{\thesection}  % assign a new value to the self-
 defined variable x
\indentation b2 Section: \pyc{print(pytex.context.yyy)}\\

\def\y{\thesubsection}  % overwrite x again
\indentation b2 Subsection: \pyc{print(pytex.context.yyy)
 }\\

\def\y{\thesubsubsection}  % overwrite x again
\indentation b2 Subsubsection: \pyc{print(pytex.context.yyy
 )}
```

```
b2 Section: 12
b2 Subsection: 12.2
b2 Subsubsection: 12.2.3
```

12.2.4 Expand `lipsum` before using it with `py`

While Sections 3.6 and 12.1 used a PythonTeX variable outside of the `pycode` block, here again it is the other way around: A variable defined in the LaTeX environment is passed to a PythonTeX command.

The PythonTeX environments (pyblock and pycode) do not support LaTeX macros. However, the commands (`\pyb` and `\pyc`) do so !

Here we use a macro defined at https://tex.stackexchange.com/questions/48403.

PythonTeX cannot only handle `\lipsum`, but also macros like this one (`loremlines`) around `\lipsum`, which allows to determine the number of lines (here 2):

```
\newbox\one
\newbox\two
\long\def\loremlines#1{%
    \setbox\one=\vbox {%
        \lipsum
     }
   \setbox\two=\vsplit\one to #1\baselineskip
   \unvbox\two}

\pyc{mytext = """\loremlines{2}"""} % This macro shortens
   lipsum #16 to 2 lines
\py{mytext}
```

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida

Remark: LaTeX problem: `mdframed` only builds the frame for one line, even if `loremlines` prints more. `parbox` around the `\py` commands created an error. So here I gave up painting the frame.

13 Own function with parameters defined in a pycode block

For the definition of these own functions sympy is used. The functions are defined either in a `pycode` block or in a `pythontexcustomcode` block.

These functions will later be used for the experiments / tries in Section 15.

13.1 prime() from sympy used in own function to generate primes in a loop. Pass function parameter as number

1) The first two self-defined functions `Primzahlen1(n)` and `Primzahlen2(n)` output all primes $2, ..., n$ via print commands. These functions are identical. The only difference is, that one is defined in a `pycode` block, the other one in a `pythontexcustomcode` block.

```
\begin{pythontexcustomcode}{py}
from sympy import prime          # symbolic maths, here prime numbers
def Primzahlen1(n):              # definition of a Python function
    for i in range(1, n):
        print(prime(i), " ")     # next prime number
    print("and ", prime(n))      # last prime
\end{pythontexcustomcode}

[pythontexcustomcode] The first 8 primes are \pyc{Primzahlen1(8)}.

\begin{pycode}
from sympy import prime          # symbolic maths, here prime numbers
def Primzahlen2(n):              # definition of a Python function
    for i in range(1, n):
        print(prime(i), " ")     # next prime number
    print("and ", prime(n))      # last prime
\end{pycode}
```

```

[pycode] The first 9 primes are \pyc{Primzahlen2(9)}.
```

```
[pythontexcustomcode] The first 8 primes are 2 3 5 7 11 13 17 and 19 .

[pycode] The first 9 primes are 2 3 5 7 11 13 17 19 and 23 .
```

2) The third self-defined function `GenPrimeList(n)` stores all primes $2, ..., n$ in a list and returns it (the only difference to the first two functions is that the first two do the printing themselves instead of returning a value).

```python
from sympy import prime         # symbolic maths, here prime numbers
def GenPrimeList(n):            # definition of a Python function
    list = []
    for i in range(1, n+1):
        list.append(prime(i))   # next prime number
    return list
```

```
 [pythontexcustomcode] The first 7 primes are \py{
  GenPrimeList(7)}.
```

```
[pythontexcustomcode] The first 7 primes are [2, 3, 5, 7, 11, 13, 17].
```

13.2 Pass a variable as input to a self-defined `pycode` function in a loop

While in Sections 13.1 and 13.1 each of the parameters “9” and “7” had to be written twice (in the LaTeX text after “first” and as parameter for the function `Primzahlen<1,2>`),

```
[pythontexcustomcode] The first 7 primes are \py{
 GenPrimeList(7)}.
```

we like to pass the value of the parameters only once, and we want to use a variable for this (makes it easier to keep documents consistent and flexible).

So there are two sub **goals**:

1) Within a loop: We’d like to pass the loop index (variable) as input parameter. This works. See Section 14.

2) Passing the PythonTeX variable `mylavar` to `\py` also works well. When passing the variable to the self-defined function `Primzahlen1()` within `\pyc`, mind the difference that it doesn’t work, when being passed via another `\py` statement, but it works when just passed directly via `mylarvar`:

```
\pyc{mylavar = 11}  % defining a LaTeX variable
The first \py{mylavar}~primes          % ok.
are \pyc{Primzahlen1(mylavar)}.        % ok.
```

The first 11 primes are 2 3 5 7 11 13 17 19 23 29 and 31 .

Remark: The `general problem` is how to extend the parameter, BEFORE it is used in the called self-defined function!

3) Another alternative is to put the whole thing in another PythonTeX function. Because of problems with the `None` output, I had to split it up in several print calls.

```
\begin{pythontexcustomcode}{py}
def PrintWholeSentence(n):
    print("The first %d primes are " % ( n ) )
    Primzahlen1(n)
    print(".")
\end{pythontexcustomcode}

\pyc{PrintWholeSentence(14)}
```

The first 14 primes are 2 3 5 7 11 13 17 19 23 29 31 37 41 and 43 .

4) Another alternative is to use a LaTeX makro surrounding the text.

However, passing a variable instead of the fixed number "14" first didn't work in the step where it is passed to the self-defined PythonTeX function `Primzahlen1` within the LaTeX macro (we tried with `mylavar`, with a PythonTeX context variable and with a LaTeX variable `\y`).

What's needed is to define LaTeX variables first and then insert their expansion into PythonTeX commands. The **solution** can be found in Section 15.8.

So here is the first try of a LaTeX macro which only works, if no variables are passed.

```
\newcommand\LaTeXCmdWholeSentencePlusFctCall[1]{%
The first #1 primes are \pyc{Primzahlen1(#1)}.
}

\LaTeXCmdWholeSentencePlusFctCall{14}
```

The first 14 primes are 2 3 5 7 11 13 17 19 23 29 31 37 41 and 43 .

14 Using the loop counter as parameter for a self-defined PythonTeX function

1) Calling a self-defined function (`Primzahlen1()`) within a PythonTeX loop: Here the function can take a variable as parameter, as there is no mix between LaTeX and PythonTeX parts:

```
\begin{pycode}
for i in range(15, 18):
    print(i, " / ")
    Primzahlen1(i)
    print("\n")
    i+=1
\end{pycode}
```

```
15 / 2 3 5 7 11 13 17 19 23 29 31 37 41 43 and 47

16 / 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 and 53

17 / 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 and 59
```

2) Creating a table using a self-defined function (`GenPrimeList()`) within a PythonTeX loop:

```
\begin{pycode}
count = 6
print(r"\begin{tabular}{l|r} \toprule")
print(r"Count & Primes list \\ \midrule")
for i in range(1, count+1):
    primelist = GenPrimeList(i)
    print(i, "&", primelist, r"\\")
    # print(i, "&", str(GenPrimeList(i)), r"\\")  #
 Alternative
print(r"\end{tabular}")
\end{pycode}
```

Count	Primes list
1	[2]
2	[2, 3]
3	[2, 3, 5]
4	[2, 3, 5, 7]
5	[2, 3, 5, 7, 11]
6	[2, 3, 5, 7, 11, 13]

3) Bundling the creation of the table in one PythonTeX function:

```
\begin{pycode}
print('TEST2') # Within pycode block prints are shown.
def GenPrimeTable(start, end):
    print(r"\begin{tabular}{l|r} \toprule")
    print(r"Count & Primes list \\ \midrule")
    for i in range(start, end+1):
        primelist = GenPrimeList(i)
        print(i, "&", primelist, r"\\")
    print(r"\end{tabular}")
\end{pycode}

Calling \verb!GenPrimeTable(4,6)! generates a table with 3
   lists with
the first 4, the first 5, and first 6 prime numbers:

\pyc{GenPrimeTable(4,6)}
```

TEST2

Calling `GenPrimeTable(4,6)` generates a table with 3 lists containing the first 4, the first 5, and first 6 prime numbers:

Count	Primes list
4	[2, 3, 5, 7]
5	[2, 3, 5, 7, 11]
6	[2, 3, 5, 7, 11, 13]

4) Compare the output via `\py{GenPrimeList(7)}` to the one via `\py{Primzahlen1(7)}`. Both print a sentence (no table, no loop here):

```
The first 7 primes are \pyc{Primzahlen1(7)}.

The first 7 primes are \py{GenPrimeList(7)}.

Get rid of the brackets of the list:\\
The first 7 primes are \pyc{print(*GenPrimeList(7), sep=",
 ")}.

Get rid of the brackets of the list and of the commas:\\
The first 7 primes are \pyc{print(*GenPrimeList(7))}.
```

The first 7 primes are 2 3 5 7 11 13 and 17 .

The first 7 primes are [2, 3, 5, 7, 11, 13, 17].

Get rid of the brackets of the list:
The first 7 primes are 2, 3, 5, 7, 11, 13, 17 .

Get rid of the brackets of the list and of the commas:
The first 7 primes are 2 3 5 7 11 13 17 .

Using functions returning a list is more flexible than including prints in the called function itself (like done in `Primzahlen1()`).

15 Passing variables – a major issue? No.

Here are some experiments when several students tried to pass LaTeX (or PythonTeX) variables as parameters to a self-defined `pycode` function which is called within a `pycode` block or within a `\py` command. These tries were done to make the **goal** described in Section 13.2 work. Despite they failed in the first run, these tries are listed for didactical purposes.

The **solution** can be found in Section 15.8. PythonTeX commands/environments need pure Python, so all LaTeX stuff has to be expanded before.

15.1 Try 1: Passing a PythonTeX variable to pyc{}

In the opposite to passing PythonTeX variables, passing LaTeX variables to `\py` doesn't work.

```
\newcommand{\myLvar}{6} % defining a LaTeX variable

[myLvar] \myLvar  % output the value of the LaTeX variable
% \pyc{Primzahlen1(\myLvar)}  % would cause PythonTeX error
% \pyc{Primzahlen1(\py{myLvar})}  % PythonTeX error
```

```
[myLvar] 6
```

15.2 Try 2: Passing a LaTeX variable defined with `newcommand` or `newcounter` to py does not work

What worked in Section 12.2.2 was passing a LaTeX variable to `\pyc` via the Python context.

What also works is passing fixed values (like below).

What doesn't work is passing a variable defined with **`\newcommand`** or **`\newcounter`** to **`\py`**:

```
\newcommand{\varname}{88}
\varname
% ~~~\py{88+3}  % ok.
% ~~~\py{\varname} % PythonTeX error

\newcounter{mycounter}{"my\_string"}
% ~~~\py{88+5}  % ok.
% \py{\mycounter}  % PythonTeX error
% \py{\themycounter}  % PythonTeX error

\newcommand{\pyvarname}[1]{\py{#1}}
\pyvarname{88+7}  % ok.
% ~~~\pyvarname{\varname}  % PythonTeX error
```

```
88

95
```

15.3 Try 3: Passing a LaTeX variable to pyc{`OwnFunction(...)`}

a) The following approach using `newcommand` doesn't work:

```
\newcommand{\mylavar}{6}  % defining a LaTeX variable

mylavar: \mylavar
% \pyc{Primzahlen1(\py{mylavar})}  % PythonTeX error
```

```
mylavar: 6
```

b) The following approach using a PythonTeX context variable to pass a variable to `Primzahlen1` doesn't work:

```
\verb!pytex.context.xxx!: \pyc{print('xxx: ' , pytex.
  context.xxx)}    OK

% \pyc{Primzahlen1(\pyc{pytex.context.xxx})}  % PythonTeX
  error
% \pyc{Primzahlen1(\py{pytex.context.xxx})}  % PythonTeX
  error
% \pyc{Primzahlen1(pytex.context.xxx)}  % PythonTeX error
```

pytex.context.xxx: xxx: 4 OK

c) The following approach using `def` to pass a variable to `Primzahlen1` doesn't work:

```
\def\x{12}
x: \x

% x: \pyc{print('x: ', \x)}  % PythonTeX error
% x: \py{print('x: ', \x)}  % PythonTeX error
% \pyc{Primzahlen1(\pyc{\x})}  % PythonTeX error
```

x: 12

d) The approach with `edef` instead of `def` doesn't work:

```
\edef\avar{13}  % defining a LaTeX variable
avar: \avar

% \pyc{Primzahlen1(\avar)}  % PythonTeX error
% \pyc{Primzahlen1(\pyc{\avar)}}  % PythonTeX error
```

avar: 13

15.4 Try 4: Passing a LaTeX variable via `pysub`

Section 4 of [8] elaborates about about PythonTeX using `Verbatim`. According to this section the command **`\pys`** and the environment **`\pysub`** can be used for “variable substitution” and “string interpolation”. It says it is applicable for `verb` and `tikzpicture`. Here we show both ways (**`\pys`** and **`\pysub`**), which are equivalent.

The content of the environment is passed verbatim to Python. Substitution fields take the form `!<expression>`. After `<expression>` is evaluated, a string representation of the result is returned to LaTeX. If `<expression>` is simply a variable name, then it is replaced with a string representation of the variable’s value.

Sadly, this approach didn’t work in our experiment when using variables.

a) Passing a LaTeX variable to the pysub expression

```
\newcommand{\intlatopy}[2]{\expandafter\intlatopytwo\
   expandafter{#2}{#1}}
\newcommand{\intlatopytwo}[2]{\pyc{#2=#1}}

\newcommand\xpp{30}
[From latex: xpp]: \xpp

\intlatopy{xpp}{\xpp}

\begin{pysub}
\begin{Verbatim}
  [From in pysub]: x = !{2**16}  % Note, this does not assign
    to a variable x !
  [From in pysub]: $y = !{2**8}$
  [From in pysub]: xpp = !{xpp}
  # [From pysub]: \xpp * 5  % This doesn't do any calculation
  # [From pysub]: !{\xpp * 5}  % PythonTeX error
\end{Verbatim}
\end{pysub}

[From latex: \textbackslash{}xpp*5]: $\xpp * 5$ \\
[From pys: value]: \pys{\verb|calculated value = !{2**7}|}\\
[From pys: xpp]: \pys{\verb|value = !{xpp}|}
[From pys: 5*xpp]: \pys{\verb|calculated value = !{5*xpp}|}
```

```
[From latex: xpp]: 30

[From in pysub]: x = 65536  % No assignment to variable x !
[From in pysub]: $y = 256$
[From in pysub]: xpp = 30

[From latex: \xpp*5]: 30 * 5
[From pys: value]: calculated value = 128
[From pys: xpp]: value = 30
[From pys: 5*xpp]: calculated value = 150
```

⇒ A LaTeX variable `\xpp` cannot be used in a pysub `!<expression>`, but its expanded PythonTeX equivalent `xpp` can. I guess, this is what G. Poore called "variable substitution".

b) Passing a LaTeX variable in a pysub expression as function argument

```
\renewcommand\xpp{7}
[From latex: \textbackslash{}xpp]: \xpp
\intlatopy{xpp}{\xpp}  % Without xpp stays on 30

\begin{pysub}
\begin{Verbatim}
  From in pysub: \pyc{Primzahlen1(\pys{\verb|!{1+2}|})}
  From in pysub: \pyc{Primzahlen1(\pys{\verb|!{xpp}|})}
  # From in pysub: \pyc{Primzahlen1(\pys{\verb|!{\xpp}|})}  %
    PythonTeX error
\end{Verbatim}
\end{pysub}

[From latex: \textbackslash{}xpp*5]: $\xpp * 5$ \\
```

```
[From latex: \xpp]: 7

From in pysub: \pyc{Primzahlen1(\pys{\verb|3|})}
From in pysub: \pyc{Primzahlen1(\pys{\verb|7|})}

[From latex: \xpp*5]: 7 * 5
```

⇒ It works like it should: The expression `!{1+2}` is correctly evaluated. The function itself is not called. The LaTeX variable was expanded successfully and worked as function parameter too.

Summary:

- Avoid nested PythonTeX commands.
- If you need LaTeX macro expansion, you must write your own commands to handle that and then assemble the PythonTeX commands you want.

The following sample uses `pys` instead of `pysub` (and no variable is involved):

```
1: \pys{\verb|calculated value = !{2**7}|}\\    % Ok
2: \pys{\verb|calculated value = !{(1+2)}|}\\    % Ok
% x: \pys{\verb|!{Primzahlen1(1+3)}|}     % Nok: LaTeX error
  : \verb ended by end of line.
3: \pys{\verb|Primzahlen1(!{1+2})|}\\     % Ok: just prints
  "Primzahlen1(3)"
4: Primzahlen1(\pys{\verb|!{1+2}|})\\     % Ok: just prints
  "Primzahlen1(3)"
```

```
1: calculated value = 128
2: calculated value = 3
3: Primzahlen1(3)
4: Primzahlen1(3)
```

15.5 Try 5: Passing a LaTeX variable via `pgf,num,expandafter,...`

a) Within `pgfmath` LaTeX variables (here `avarx`) can be handled like this.

```
\newcommand{\avarx}{10}
[latex-pgf]: 87 divided by \avarx~is approximately \
  pgfmathparse{int(round(87/\avarx))}\pgfmathresult.
```

[latex-pgf]: 87 divided by 10 is approximately 9.

This approach (`pgfmath` is called within the parameter for `\pyc\{OwnFunction(...)\}`) can also only work when the LaTeX variable has been expanded before.

```
\pyc{Primzahlen1(\pgfmathparse{\avarx}\pgfmathresult)}  %
 PythonTeX error
\pyc{Primzahlen1(\pgfmathparse{int(\avarx)}\pgfmathresult)}
   % PythonTeX error
```

b) This approach (with `counters`) can also only work when the LaTeX variable has been expanded before.

```
\newcounter{mycounter}  % Initializing the value to 0
\setcounter{mycounter}{12}  %  Changing this value to 12
\pyc{Primzahlen1(\value{mycounter})}  % PythonTeX error
\pyc{Primzahlen1(\themycounter)}  % PythonTeX error
```

c) Further unavailing tries to expand the LaTeX variable as parameter for a PythonTeX function – again it works with the **solution** described in Section 15.8.

```
\newcommand{\avar}{11}
\pyc{Primzahlen1(\avar)}  % PythonTeX error
\pyc{Primzahlen1(\num{\avar})}  % PythonTeX error

\expandafter\pyc\expandafter{Primzahlen1(\avar)}  %
 PythonTeX error
\pyc{Primzahlen1(\expandafter\avar\expandafter)}  %
 PythonTeX error
\pyc{\expandafter{Primzahlen1(\avar)}}  % PythonTeX error
\pyc{\expandafter Primzahlen1(\avar)}  % PythonTeX error

\def\anawkwardname{111}
[anawkwardname] \anawkwardname \\
\expandafter\def\csname anawkwardname\endcsname
[csname] \csname{122}  % Did not work, not further
 investigated
```

15.6 Try 6: Calling a LaTeX macro within pycode does not work

Just a note, what was tried too: It is clear that the following code doesn't work as LaTeX macros like `LaTeXCmdWholeSentencePlusFctCall()` (defined in Section 13.2) are not known within the pycode block.

```
\begin{pycode}
print(3, r"\\")
print(\LaTeXCmdWholeSentencePlusFctCall{15}, r"\\")  #
 PythonTeX error
\end{pycode}
```

15.7 Try 7: Passing the LaTeX loop counter to py within a native LaTeX loop

What worked was passing a LaTeX variable to `\py` via the Python context (see Section 12.2.2).

Using the counter in a LaTeX loop works if it gets converted to a PythonTeX variable BEFORE calling `\py`.

1) Sample 1 with `\foreach`:

```
\newcommand{\cmd}{-x-}  % just a string
\foreach \n in {0,...,2}{\cmd{}\\}
\\
\foreach \n in {0,...,2}{Term: \n*\n; calculation: \py
 {3*5}\\}
\\
\foreach \n in {0,...,2}{Loop index: \n : squared: \
 intlatopy{n}{\n}\py{n*n}\\}
\\
% \foreach \n in {0,...,2}{\py{\n*\n}\\} % PythonTeX error.
  \n not expanded.
\\
% \foreach \n in {0,...,2}{\py{\n{}*\n{}}\\} % PythonTeX
 error. \n not expanded.
```

```
-x-
-x-
-x-

Term: 0*0; calculation: 15
Term: 1*1; calculation: 15
Term: 2*2; calculation: 15

Loop index: 0: squared: 0
Loop index: 1: squared: 1
Loop index: 2: squared: 4
```

2) Sample 2 with `\forloop`:

The counter variable `\thect` works as argument of `\pgfmathparse`. However, `\pgfmathparse` can only be used within an argument for the self-defined function `LaTeXCmdWholeSentencePlusFctCall` (which internally uses `\pyc`), if it is expanded before.

```
\newcounter{c}
\forloop{c}{19}{\value{c} < 21}%
{%
    Loop \# \arabic{c} / \thect:\\
    \qquad \pgfmathparse{87}\pgfmathresult
    \qquad \py{3*7}
    % \qquad \pyc{\thec}  % Nok \thec not expanded in \pyc
    %
    % \qquad \pgfmathparse{c}\pgfmathresult   % pdflatex
   error
    \qquad \pgfmathparse{\value{c}}\pgfmathresult
    \qquad \pgfmathparse{\thec}\pgfmathresult
    \qquad \pgfmathparse{int(\thec * \thec)}\pgfmathresult
    \qquad \pgfmathparse{int(\value{c} * \value{c})}\
   pgfmathresult\\
    \\
}
```

```
Loop # 19 / 19:
87     42     –     19     19     361     361

Loop # 20 / 20:
87     42     –     20     20     400     400
```

15.8 Conclusion / Solution for passing LaTeX variables

If you want to pass LaTeX commands to Python, you need to define a custom command to work around LaTeX tokenization. Everything inside PythonTeX commands/environments is interpreted as pure Python (no LaTeX interspersed), so you need to create new commands that will expand macros and then insert the expansion into PythonTeX commands.[1] Exactly what you need depends on the macros you want to pass to Python. For simple macros that only require one level of expansion, you could do something like this:

```latex
\def\varint{123}
\def\varstring{string}

\newcommand{\inttopy}[2]{\expandafter\inttopytwo\
  expandafter{#2}{#1}}
\newcommand{\inttopytwo}[2]{\pyc{#2=#1}}
\newcommand{\strtopy}[2]{\expandafter\strtopytwo\
  expandafter{#2}{#1}}
\newcommand{\strtopytwo}[2]{\pyc{#2="#1"}}

\inttopy{varint}{\varint}
\strtopy{varstring}{\varstring}
\verb|\varint| is \py{varint} and \verb|\varstring| is \py{
  varstring}.
```

`\varint` is 123 and `\varstring` is string.

Applying this to the LaTeX command `LaTeXCmdWholeSentencePlusFctCall` first means to define a LaTeX variable and then derive a PythonTeX variable from it:

```latex
\def\LATvar{15}
\inttopy{PYvar}{\LATvar}
```

Passing `#1` to the PythonTeX function worked fine, but the first usage of `#1` just showed the variable name and not its value. So the definition had to be

[1] Thanks to Geoff Poore for his great support.

slightly reworked to always make PythonTeX handle the parameter `#1`. This was the change: `The first #1 primes  ==>  The first \py{#1} primes`

```
\newcommand\wLaTeXCmdWholeSentencePlusFctCall[1]{%
The first \py{#1} primes are \pyc{Primzahlen1(#1)}.
}
```

```
The first 14 primes are 2 3 5 7 11 13 17 19 23 29 31 37 41 and 43 .
The first 15 primes are 2 3 5 7 11 13 17 19 23 29 31 37 41 43 and 47 .
```

Contents