PyThesis - A Python Framework for LaTeX

Henning Metzmacher

December 16, 2020

Abstract

PyThesis is a lightweight Python framework that lets you integrate Python code into LaTeXdocuments. The framework consists of two components: A web React application that can be used to quickly build and view the final PDF document and a Python backend that acts as a preprocessor for the LaTeXcode and embedded Python code. The framwork also contains classes that can be used to interact with a shared MathWorks MATLAB session. Specifically, datasets can be loaded using the Python CSV loader and then sent to MATLAB for further processing or to generate plots. The resulting graphics can be stored in a predefined directory in order to directly embed them as LaTeXfigures.

Contents

Chapter 1

Setup

1.1 Install LaTeX

In order to compile LaTeXfiles, a LaTeXdistribution needs to be installed. PyThesis executes pdflatex to generate a PDF file after transpiling the source files. For Windows, the proTeXt distribution is recommended: `https://www.tug.org/protext/`

1.2 Install Python Dependencies

The software was tested using Python 3.6. Installation using pip:

```
pip install numpy
pip install matplotlib
pip install jinja2
pip install flask
pip install flask_cors
```

In Linux use pip3 instead.

1.3 Install npm to modify the Browser App (Optional)

The prebuild app is now part of the repository. You can skip this unless you want to modify the app

The browser app is written using the React framework. Node.js needs to be installed to generate a build: `https://nodejs.org`

```
cd app
npm install
npm run build
```

This generates a static build that is served by the integrated Python Flask server.

1.4 Install the MATLAB Engine API for Python (Optional)

Follow the Mathworks tutorial: `https://de.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html`

1.5 Run PyThesis

In Windows you can click on the run_example.bat to start the example. The webapp should open in a new browser window. Using the command line, the software can be started as follows:

```
python start.py --project_root <project_root>
```

On Linux use python3 instead. <project_root> should be replaced with an absolute project path. In order to test the framework and the browser app point <project_root> to the example project. A new project should match the layout of the example.

1.6 Build

In order to **build** the document press **b** or click the play button in the web app. This executes all .py files and generates .tex files from them, merges them into one output document in the build directory and compiles it using the pdflatex (the exact order of build commands is defined in the build.config in the PyThesis root directory, each line is a single command). At some point, when the project grows bigger you might only want to build specific files that you are working on at the moment. In order to that you can do a **partial build** by pressing **p**.

1.6.1 _.always Files

During a **partial build** only .py files listed in an _.always file in that same directory are executed. All other .py files are ignored during a **partial build**. If, for example, the examples folder contains an _.always file with a single entry:

```
a_simple_text_generator
```

this means a_simple_text_generator.py is always executed during a **partial build** (during a full build it is executed no matter what). The .py extension is automatically appended. In order to add multiple entries just add them in new lines like so:

```
file1
file2
etc
```

In addition, you can do a **partial execute** by pressing **e** which is the same as a **partial build** just that the subsequent LaTeXbuild step is omitted. Hence, you won't see updates in the document.

Python output is always shown in the console (for example the command prompt in Windows).

Table 1.1 shows an overview over the build shortcuts in the web application:

Shortcut	Command
b	**Full build** (execute all .py files + compilation)
p	**Partial build** (execute only .py files listed in _.always files + compilation)
e	**Partial execute** (execute only .py files listed in _.always files)

Table 1.1: Overview over the build shortcuts in the web application

Chapter 2

Overview of the API

2.1 Folder Structure

PyThesis uses the following for a project:

```
build/
data/
src/
templates/
```

build stores the combined, transpiled tex file as well as LaTeXoutput files such as the generated PDF. **data** should hold CSV files that can be loaded using the Dataset class or _matlab. **src** should store all LaTeXand Python source files.

NOTE: DO NOT EDIT SOURCE FILES IN THE build FOLDER. THEY WILL GET OVERWRITTEN. ALSO .tex FILES IN _tex FOLDERS UNDER src ARE OVERWRITTEN.

The *templates* folder should contain reusable templates along with their default JSON files.

2.2 Globals

PyThesis provides the following global variables when executing the .py files in a project:

```
_matlab     Instance of the MatlabAdapter class
_templater  Instance of the Templater class
_project    Current project root path
_data       Current data path
_build      Current build path
_lib        Current lib path (for custom libraries)
_templates  Current templates path
_file       File path of the executing .py file
```

```
_dir        Containing directory of the executing .py file
_basename   Basename of the executing .py file
_name       Name without extension of the executing .py file
```

The following modules can be accessed without importing:

```
os          Python core library
sys         Python core library
matplotlib  matplotlib module
_plt        matplotlib.pyplot module
np          numpy module
Dataset     PyThesis Dataset class
_dataset    pythesis.dataset module
```

For an example using _dataset see the Matplotlib Example, Section 3.2.

2.3 Jinja

PyThesis uses the Python Jinja templating engine for subincludes and to execute Python code. Therefore Jinja API calls in .tex files have an effect. LaTeXand Jinja make extensive use of curly brackets so in order to avoid clashes, critical LaTeXsections can be escape using:

```
{% raw %}
I can write whatever I want here, nothing gets interpreted.
{% endraw %}
```

Jinja documentation: `https://jinja.palletsprojects.com/en/2.10.x/`

2.4 Subincludes

PyThesis uses jinja2 and provides a subinclude extension which allows for relative includes. This way LaTeX projects can be structured in much smaller files just like in software projects (there is some native LaTeXway of doing this but I found this way to be much more convenient). The following example shows a way to organize sub- and subsubsections in a nested fashion. It is of course possible to add further levels, for example, to organize figures and graphics code. The following command includes a tex file which in turn subincludes another tex file:

```
{% subinclude 'subsections/a-subsection' %}
```

The result looks as follows:

2.4.1 A Subsection

This is a subsection which in turn includes a subsubsection:

```
{% subinclude 'subsubsections/a-subsubsection' %}
```

which looks like this:

A Subsubsection

This is a subsubsection.

2.5 Python Imports

PyThesis uses the currently executed file as working directory. Modules on the same level can therefore be imported as:

```
import _import_me
```

Modules to be imported should be prepended with an underscore. "Underscored" modules are not compiled directly and do not generate their own .tex files. In order for the imported module to have access to all global variables it needs to be wrapped using the _wrap function. The _wrap function also ensures that the module is properly reloaded when the code is changed.

```
import _import_me
_wrap(_import_me)
```

Import example:

```
Hello Import. I am imported by imports.py
```

Chapter 3

Examples

3.1 A Simple Text Generator

The text below is generated by the following code:

```
sentence = 'Sphinx of black quartz, judge my vow'
for i in range(1, len(sentence) + 1):
    print(f'{sentence[:i]}\\\\')
```

S
Sp
Sph
Sphi
Sphin
Sphinx
Sphinx
Sphinx o
Sphinx of
Sphinx of
Sphinx of b
Sphinx of bl
Sphinx of bla
Sphinx of blac
Sphinx of black
Sphinx of black
Sphinx of black q
Sphinx of black qu
Sphinx of black qua
Sphinx of black quar
Sphinx of black quart
Sphinx of black quartz
Sphinx of black quartz,
Sphinx of black quartz,
Sphinx of black quartz, j
Sphinx of black quartz, ju

Sphinx of black quartz, jud
Sphinx of black quartz, judg
Sphinx of black quartz, judge
Sphinx of black quartz, judge
Sphinx of black quartz, judge m
Sphinx of black quartz, judge my
Sphinx of black quartz, judge my
Sphinx of black quartz, judge my v
Sphinx of black quartz, judge my vo
Sphinx of black quartz, judge my vow

3.2 A Matplotlib Plot

Python files should be created within a directory called **py** on the level of the including .tex file. The transpiler creates a folder named **_tex** on the same level as the py folder. This folder includes resulting .tex files on the level of the including .tex file. Hence, if on the level of this file there is a directory as follows:

`py/a_matplotlib_plot.py`

then the result can be embedded using:

`{% subinclude '_tex/a_matplotlib_plot' %}`

The .tex extension is added automatically. In order to execute Python code to plot or generate illustrations you can use global variables that point to respective project paths. In addition, you can use global objects and classes that can be used to perform common tasks like data set loading. The following is used to create Figure 3.1.

```
columns = [
    {'key': 'SECONDS'},
    {'key': 'T_01'},
    {'key': 'T_02'}
]
csv_file = f'{_data}/example.csv'
dataset = _dataset.load_dataset(csv_file, columns)
data = dataset.to_dict()
fig, ax = _plt.subplots(figsize=(6, 4))
ax.plot(data['SECONDS'], data['T_01'])
ax.plot(data['SECONDS'], data['T_02'])
ax.set(
    xlabel='Time [s]',
    ylabel='Temperature [\u00b0C]',
    title='A Matplotlib Plot',
    ylim=[30, 40]
)
ax.legend(['Temperature 1', 'Temperature 2'])
ax.grid()
_plt.tight_layout()
fig.savefig(f'{_build}/images/{_name}.pdf')
figure = _templater.render(
    f'{_templates}/latex/figure.tex',
    path=f'images/{_name}',
    label=f'fig:{_name}',
```

```
    caption='A Matplotlib Plot'
)
print(figure)
```

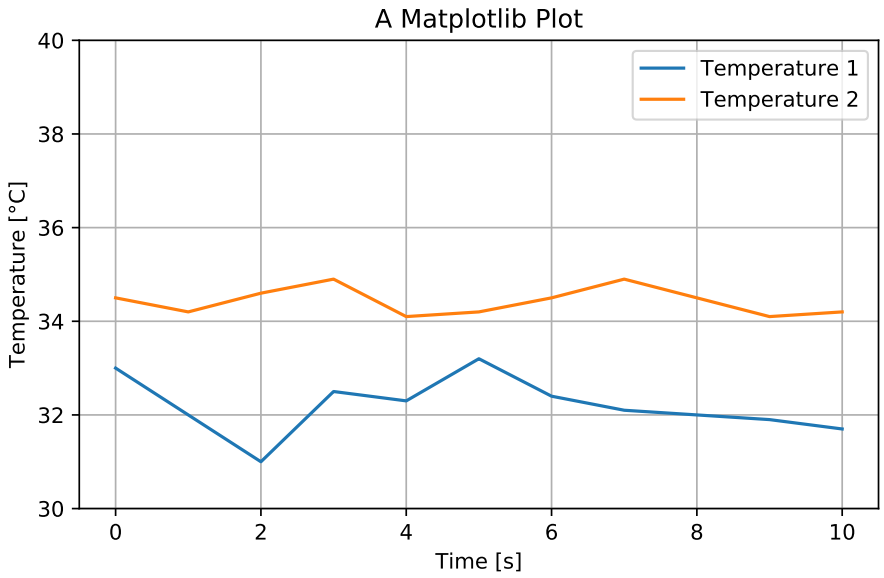

Figure 3.1: A Matplotlib Plot

3.3 A MATLAB Plot

In order to generate MATLAB plots PyThesis needs to be started with the MATLAB flag:

`python start.py [...] --matlab`

The MATLAB Python engine must be installed and MATLAB must be running. In order to share the MATLAB engine, enter the following command into the MATLAB terminal:

`matlab.engine.shareEngine`

The following uses the _matlab singleton, the _templater and _dataset to generate a MATLAB plot:

```
# Only execute if the _matlab singleton exists:
if not(_matlab == None):
    columns = [
        {'key': 'SECONDS'},
        {'key': 'T_01'},
        {'key': 'T_02'}
    ]
    _matlab.load_dataset(f'{_data}/example.csv', _name, columns)
    _matlab.eval_template(
        f'{_templates}/matlab/figure.m',
        height=300
```

```
    )
    _matlab.eval_template(
        './a_matlab_plot.m',
        name=_name
    )
    _matlab.eval_template(
        f'{_templates}/matlab/axis.m',
        xlabel='Time [s]',
        ylabel='Temperature [\u00b0C]',
        xmin=0,
        xmax=10,
        xtickminor=1,
        xtick=5,
        ymin=30,
        ymax=40,
        ytickminor=1,
        ytick=5
    )
    _matlab.eval_template(
        f'{_templates}/matlab/pdf.m',
        pdfpath=f'{_build}/images/{_name}.pdf'
    )
figure = _templater.render(
    f'{_templates}/latex/figure.tex',
    label=f'fig:{_name}',
    caption='A MATLAB Plot',
    path=f'images/{_name}'
)
print(figure)
```

The following shows the accompanying MATLAB template "a_matlab_plot.m". Note that the variable "name" within the double curly brackets is replaced with the parameter set in eval_template in the .py script above (for information on Jinja templates see Section 2.3).

```
hold on;
p1 = plot( ...
    {{name}}.SECONDS, ...
    {{name}}.T_01, ...
    'Color', 'k',  ...
    'LineWidth', 1 ...
);
p2 = plot( ...
    {{name}}.SECONDS, ...
    {{name}}.T_02, ...
    '--', ...
    'Color', 'k', ...
    'LineWidth', 1 ...
);
title('A MATLAB Plot');
legend([p1, p2], 'Temperature 1', 'Temperature 2');
hold off
```

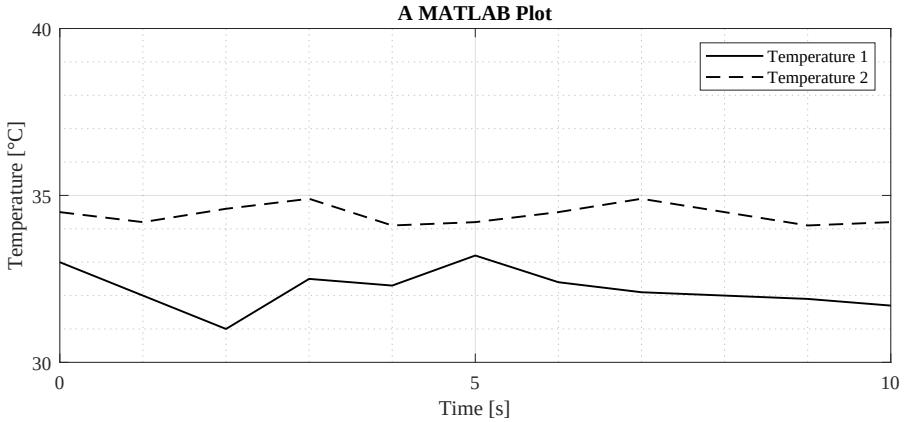

Figure 3.2: A MATLAB Plot

3.3.1 MATLAB Inline Evals

The example uses templates to execute MATLAB code. It is of course also possible to execute MATLAB code directly. For example:

```
_matlab.eval('x = [0.2 0.3 0.4];')
_matlab.eval('plot(x);')
```

For multiple lines of code you can do:

```
name = 'My Plot'
code = f'''
    hold on;
    x = [0.2 0.3 0.4];
    plot(x);
    legend(['x']);
    title('{name}');
'''
_matlab.eval(code)
```

Note that Python f-strings use **single** curly brackets for variable replacement while Jinja strings use **double** curly brackets (this can result in a fun character escaping mess).

3.4 A Table

This example shows how to load a CSV file into a dataset and display it as a table. In addition, the dataset metrics function can be used to calculate statistical metrics from the data and append it to the table.

Seconds/Metric	T_01	T_02
0.0	33.0	34.5
1.0	32.0	34.2
2.0	31.0	34.6
3.0	32.5	34.9
4.0	32.3	34.1
5.0	33.2	34.2
6.0	32.4	34.5
7.0	32.1	34.9
9.0	31.9	34.1
10.0	31.7	34.2
μ	32.21	34.42
σ	0.6	0.29
median	32.2	34.35
min	31.0	34.1
max	33.2	34.9

Table 3.1: A table example

Python code to generate the table:

```
rows = []
columns = [
    {'key': 'SECONDS'},
    {'key': 'T_01'},
    {'key': 'T_02'}
]
dataset = _dataset.load_dataset(
    f'{_data}/example.csv',
    columns=columns
)

# Extract a matrix and the metrics from the dataset:
matrix = dataset.to_matrix(['SECONDS', 'T_01', 'T_02'])
metrics = dataset.metrics(['T_01', 'T_02'])

# Create LaTeX table rows from values:
for row in matrix:
    rows.append(' & '.join(map(str, row)) + ' \\\\')

# Add a line between the values and the metrics:
rows.append('\\midrule')

# Map values to strings so we can join them:
means   = map(str, metrics['mean'])
stds    = map(str, metrics['std'])
medians = map(str, metrics['median'])
mins    = map(str, metrics['min'])
maxs    = map(str, metrics['max'])

# Create LaTex table rows from metrics:
rows.append('\mu & '    + ' & '.join(means)   + ' \\\\')
rows.append('\sigma & ' + ' & '.join(stds)    + ' \\\\')
rows.append('median & ' + ' & '.join(medians) + ' \\\\')
rows.append('min & '    + ' & '.join(mins)    + ' \\\\')
rows.append('max & '    + ' & '.join(maxs)    + ' \\\\')
```

```
36
37 # Render the table:
38 table = _templater.render(
39     f'./a_table.tex',
40     rows=' \n'.join(rows),
41     name=f'a_table'
42 )
43 print(table)
```

3.5 Custom Library

This example shows how to create a custom class in the lib directory and use it to create Matplotlib plots.

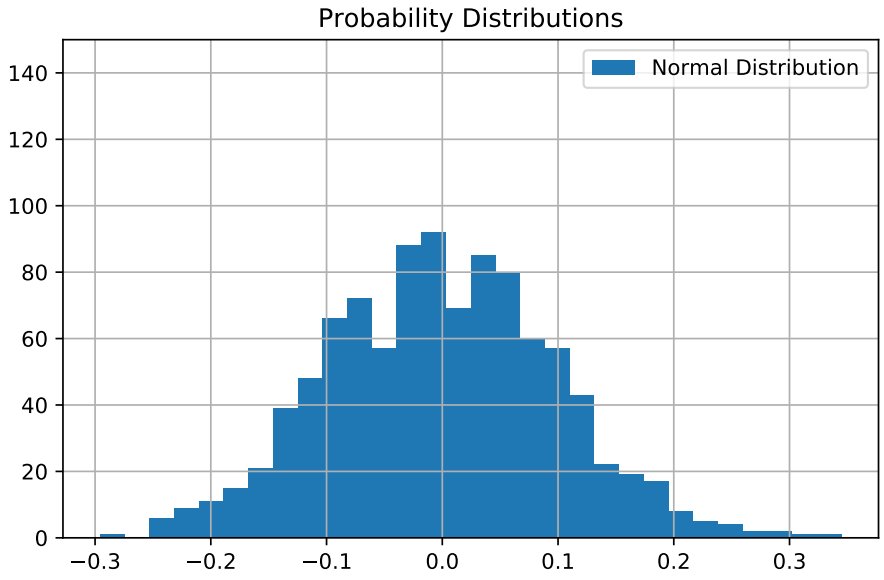

Figure 3.3: Normal Distribution

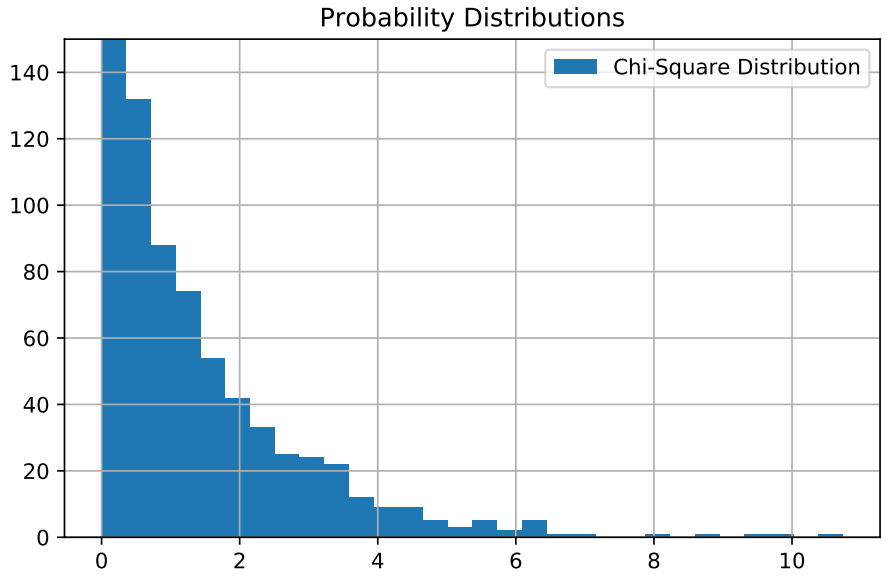

Figure 3.4: Chi-Square Distribution

Distribution classes used in the example (in module lib/_distributions):

```
class NormalDistribution:
    def __init__(self, mu=0, sigma=0.1):
        self.mu = mu
        self.sigma = sigma

    def generate_samples(self, num_samples=1000):
        return _np.random.normal(self.mu, self.sigma, num_samples)

class ChiSquareDistribution:
    def __init__(self, df=1):
        self.df = df

    def generate_samples(self, num_samples=1000):
        return _np.random.chisquare(self.df, num_samples)
```

Plot code:

```
sys.path.append(_lib) # Manually add _lib to system path
# The leading underscore of the _distribution module is only
# by convention and not technically necessary. However, it
# helps to keep the code clutter free especially when using
# short, precise module names that might also be used as
# variable names (without the leading underscore):
import _distributions
# Reload the module manually to include latest changes:
importlib.reload(_distributions)
# In order for lib modules to have access to project globals
# they also need to be wrapped:
_wrap(_distributions)

def plot_histogram(samples, label, name):
    fig, ax = _plt.subplots(figsize=(6, 4))
```

```
    ax.hist(samples, bins=30)
    ax.set(title='Probability Distributions', ylim=[0, 150])
    ax.legend([label])
    ax.grid()
    _plt.tight_layout()
    fig.savefig(f'{_build}/images/{name}.pdf')
    figure = _templater.render(
        f'{_templates}/latex/figure.tex',
        path=f'images/{name}',
        label=f'fig:{name}',
        caption=label
    )
    print(figure)

# Instantiate the distribution classes:
normal_dist         = _distributions.NormalDistribution()
chisquared_dist     = _distributions.ChiSquareDistribution()
# Generate samples:
normal_samples      = normal_dist.generate_samples(1000)
chisquare_samples   = chisquared_dist.generate_samples(1000)
# Set names:
normal_name         = 'Normal Distribution'
chisquare_name      = 'Chi-Square Distribution'
# Plot samples:
plot_histogram(normal_samples, normal_name, 'normal')
plot_histogram(chisquare_samples, chisquare_name, 'chisquare')
```