pythonimmediate — Library to run Python code*

user202729

Released 2023/01/09

Abstract

Library to run Python code.

1 Motivation

Just like PerlTEX or PyLuaTEX (and unlike PythonTEX or lt3luabridge), this only requires a single run, and variables are persistent throughout the run.

Unlike PerlTEX or PyLuaTEX, there's no restriction on compiler or script required to run the code.

There's also debugging functionalities – TEX errors results in Python traceback, and Python error results in TEX traceback. Errors in code executed with the `pycode` environment gives the correct traceback point to the Python line of code in the TEX file.

For advanced users, this package allows the user to manipulate the TEX state directly from within Python, so you don't need to write a single line of TEX code.

2 Installation

In addition to the LATEX package, you need the Python pythonimmediate-tex package, installation instruction can be found at `https://pypi.org/project/pythonimmediate-tex/`.

Note that not all TEX package versions are compatible with all Python package versions. This TEX package is compatible with Python package version `0.1.2`.

Remember to enable unrestricted[1] shell-escape. (there's a guide on TEX.SE if necessary: `https://tex.stackexchange.com/q/598818/250119`)

2.1 Installation on Overleaf

At the point of writing, this package can be used on Overleaf.

Nevertheless, you cannot use `pip` to install Python packages on Overleaf directly, instead it's possible to download `.zip` file, include it in your Overleaf project, and specify where the package can be found to Python using `PYTHONPATH` environment variable.

Instruction:

- Download the following files and place it in the root folder of Overleaf:

*This file describes version 0.1.0, last revised 2023/01/09.

[1]There's little point in supporting restricted mode, since it's possible to execute arbitrary shell commands in Python anyway. If it's needed to execute untrusted TEX code, separate sandboxing should be used.

  - `saveenv.sty`
  - `precattl.sty`
  - `pythonimmediate.sty`
  - `pythonimmediate-tex-0.1.2.zip`

  The `0.1.2` part should be replaced with the desired version of the Python package.

  The `.sty` files can be downloaded from CTAN at `https://ctan.org/pkg/saveenv`, `https://ctan.org/pkg/precattl`, `https://ctan.org/pkg/pythonimmediate` respectively.

  The `.zip` file containing Python source code can be downloaded from PyPI: `https://pypi.org/project/pythonimmediate-tex/#files`.

- Write the following in the preamble:

  ```
  \usepackage[abspath]{currfile}
  \usepackage[python-executable={PYTHONPATH=pythonimmediate-tex-
  ↪   0.1.2.zip/pythonimmediate-tex-0.1.2/
  ↪   python3},args={--mode=unnamed-pipe}]{pythonimmediate}
  ```

  As above, replace both occurrences of `0.1.2` with the downloaded version specified in the zip file above.

  Refer to 4.1 for explanation of the `abspath` option.

For some unknown reason in the default mode on Overleaf (`\nonstopmode`), when there's an error the log file might be truncated, so in that case consider writing `\errorstopmode`.

Refer to 4.2 to read the error traceback in case of Python error.

Some construct inside the `pycode` block might make the code editor on Overleaf report an error, even though the code is valid. Refer to `https://www.overleaf.com/learn/how-to/Code_Check#Code_Check_Limitations`.

3 Usage

3.1 Package options

Normally no options are required. If you're not sure what they do, just use the default options.

`args=` Arguments to be passed to the Python component of the program. Run

```
python -m pythonimmediate.pytotex --help
```

on the command-line to view the available options.

The documentation is also available at `https://pythonimmediate.readthedocs.io/en/latest/pythonimmediate.html#module-pythonimmediate.pytotex`.

`python-executable=` The name/path to the Python executable. Default to `python3`.

Can also be used to provide environment variables to the Python process. An example how to do that is explained in 2.1.

`python-flags=` Flags to be passed to the Python interpreter. For example pass `-O` to disable assertions.

The documentations can be found by running `python --help` on the command-line, or visit https://docs.python.org/3/using/cmdline.html.

3.2 TEX interface

The interface mimics those in popular packages such as PythonTEX or PyLuaTEX.

3.2.1 Inline commands

`\py` Evaluate some Python expression, consider the result as a string, then execute the result as TEX command.

TEXhackers note: The command is not expandable, and the argument will be fully expanded with the active ~ set to `\relax`, `\set@display@protect` executed and `\escapechar=-1`, then the result passed to Python as a string.

Which, for the users who are not familiar with TEX terminology, roughly means the following:

- the value can only be used to typeset text, it must not be used to pass "values" to other LATEX commands.

  The following is legal:

  ```
  The value of $1+1$ is $\py{1+1}$.
  ```

  The following is illegal, as the result (2) can only be used to typeset text, not passed to another command that expect a "value":

  ```
  \setcounter{abc}{\py{1+1}}
  ```

  A possible workaround is:

  ```
  \py{ r'\\setcounter{abc}{' + str(1+1) + '}' }
  ```

  In this example it works without escaping the `{}` characters, but if the Python code has those unbalanced then you can escape them as mentioned below.

- Special characters can be "escaped" simply by prefixing the character with backslash.

  For example

  ```
  \pyc{assert len('\ \ ')==2}
  \pyc{assert ord('\\\\')==0x5c}
  \pyc{assert ord('\%')  ==0x25}
  ```

  In the examples above, Python "sees" (i.e. the Python code being executed is)

  ```
  assert len('  ')==2
  assert ord('\\')==0x5c
  assert ord('%') ==0x25
  ```

respectively.

- Macros will be expanded.

```
\def\mycode{1+1}
The value of $1+1$ is $\py{\mycode}$.
```

`\pyc` Execute some Python code provided as an argument (the argument will be interpreted as described above).

The command is not expandable – roughly speaking, you can only use this at "top level".

Any output (as described in 3.3.1) will be typesetted.

The difference between `\py` and `\pyc` is that the argument of `\py` should be a Python expression (suitable for passing into `eval()` Python function) while the argument of `\pyc` should be a Python statement (suitable for passing into `exec()` Python function).

Therefore,

- `\py{1+1}` will typeset 2.
- `\pyc{1+1}` is valid, but will do nothing just like `exec("1+1")`.
- `\py{x=1}` is invalid.
- `\pyc{x=1}` is valid and assigns the variable `x` to be 1.

`\pycq` Same as above, but output (3.3.1) will not be typesetted.

`\pyfile` Given an argument being the file name, execute that file.

`\pys` Performs "string interpolation", the same way as PythonTEX. (not yet implemented)

3.2.2 Environments

`pycode` Verbatim-like environment that executes the code inside as Python.

Example usage: The following will typeset `123`

```
\begin{pycode}
pythonimmediate.print("123")
\end{pycode}
```

Special note: white spaces at the end of lines are preserved.

Any output (as described in 3.3.1) will be typesetted.

`pycodeq` Same as above, but output will not be typesetted.

`pysub` Not yet implemented.

3.3 Python interface

The TEX interface is only used to call Python. Apart from that, all the work can be done on the Python side.

All functions in this section should be imported from `pythonimmediate` package, unless specified otherwise.

Currently, all the documentations are moved to the Python package documentation, see `https://pythonimmediate.readthedocs.io/`.

Documentation of a few functions are still kept here for convenience, but **they might be outdated**. Always refer to the online documentation.

3.3.1 Print to TEX

.print_TeX()
.file

These functions are used in `\pyc` command or `pycode` environment.

Unlike most other packages, using `print()` function in Python will print to the console (TEX standard output). In order to print TEX code to be executed, you can do one of

```
pythonimmediate.print_TeX(...)
print(..., file=pythonimmediate.file)
with contextlib.redirect_stdout(pythonimmediate.file):
    print(...)
```

Note that in quiet environments, `pythonimmediate.file` is None, the second variant using `print()` will print to stdout instead of suppress output. The third variant works as expected.

All output will be buffered until the whole Python code finishes executing. In order to typeset the text immediately use one of the advanced commands.

.newcommand()
.renewcommand()

Same as LATEX's `\newcommand` and `\renewcommand`. Can be used as follows:

```
from pythonimmediate import newcommand, renewcommand

@newcommand
def function():
    ...
# define |\function| in TeX

@newcommand("controlsequencename")
def function():
    ...
# define |\controlsequencename| in TeX

def function():
    ...
newcommand("controlsequencename", function)
```

.get_arg_str()
.get_optional_arg_str()
.get_verb_arg()
.get_multiline_verb_arg()
.peek_next_char()
.get_next_char()

There are those functions that is mostly understandable to an inexperienced LATEX user, and should be sufficient for a lot of programming works.

This is an example of how the functions could be used. The name should be mostly self-explanatory.

```
\documentclass{article}
\usepackage{pythonimmediate}
\begin{document}
\begin{pycode}
from pythonimmediate import newcommand, peek_next_char, get_next_char,
↪  get_arg_str
from pythonimmediate import print_TeX as print
@newcommand
def innerproduct():
    s = get_arg_str()    # in the example below this will have the value
    ↪  '\mathbf{a},\mathbf{b}'
    x, y = s.split(",")  # it's just a Python string, manipulate
    ↪  normally (but be careful of comma inside braces, parse the
    ↪  string yourself)
    print(r"\left\langle" + x + r"\middle|" + y + r"\right\rangle")

@newcommand
def fx():
    if peek_next_char() == "_":
        get_next_char()
        subscript = get_arg_str()
        print("f_{" + subscript + "}(x)")
    else:
        print("f(x)")

@newcommand
def sumManyArgs():
    s = 0
    while peek_next_char() == "{":
        i = get_arg_str()
        s += int(i)
    print(str(s))
\end{pycode}
$1+2+3 = \sumManyArgs{1}{2}{3}$

$\innerproduct{\mathbf{a},\mathbf{b}}=1$

$\fx = 1$, $\fx_i = 2$, $\fx_{ij} = 3$
\end{document}
```

It will typeset:

$1 + 2 + 3 = 6$

$\langle \mathbf{a} | \mathbf{b} \rangle = 1$

$f(x) = 1$, $f_i(x) = 2$, $f_{ij}(x) = 3$

`.get_arg_estr()`
`.get_optional_arg_estr()`
Similar to some functions above, except that the argument is fully expanded and "escapes" of common characters are handled correctly, similar to how `\py` command (3.2.1) reads its arguments.

`.execute()`
Takes a string and execute it immediately. (so that any `.execute()` will be executed before any `.print_TeX()`)

Assuming TeX is in errorstopmode (i.e. errors halt TeX execution), any error in TeX will create an error in Python and the traceback should point to the correct line of code.

For example, in the following code

```
\documentclass{article}
\usepackage{tikz}
```

```
\usepackage{pythonimmediate}
\begin{document}

\begin{tikzpicture}
\begin{pycode}
from pythonimmediate import execute
execute(r'\draw (0, 0) to (1, 1);')
execute(r'\draw (2, 2) to (p);')
execute(r'\draw (3, 3) to (4, 4);')
\end{pycode}
\end{tikzpicture}

\end{document}
```

each `\draw` command will be executed immediately when the Python `.execute()` function is executed, and as the second line throws an error, the Python traceback will point to that line.

4 Troubleshooting

4.1 "Source file not found!" error message

In order to obtain the exact code with trailing spaces and produce error traceback point to the correct TEX file, the Python code need to know the full path to the current TEX file for the `pycode` environment.

Nevertheless, this is difficult and does not always work (refer to the documentation of currfile for details), so this message is issued when the file cannot be found.

In that case try the following fixes:

- Include `\usepackage[abspath]{currfile}` at the start of the document, after the `\documentclass` line. (this option is not included by default because it's easy to get package clash, and usually currfile without the `abspath` option works fine – unless custom `jobname` is used)
- Explicitly override `currfilename` or `currfileabspath` – for example

  ```
  \def\currfilename{main.tex}
  ```

  Technically this is an abuse of the currfile package API, but it usually works regardless.

4.2 "Python error" error message

In case of Python error, the Python traceback is included in the terminal and TEX log file.

Search for "Python error traceback" before the error line in the log file.

On Overleaf, you can either view the log file ("Raw logs" section) or the traceback on stderr (download `output.stderr` file)

4.3 "TEX error" error message

If an error occur in TEX, traceback cannot be included in the log file.

Besides, this can only be detected in `\errorstopmode`. Such an error will always halt TEX, and Python will be force-exited after printing the error traceback.

On Overleaf, download `output.stderr` file to read the traceback.

5 Implementation note

Communication between TEX and Python are done by opening two pseudo-files from the output of a Python process `textopy` (similar to `\ior_shell_open:Nn`) and to the input of another Python process `pytotex` (this would be `\iow_shell_open:Nn`, if LATEX3 have such a function).

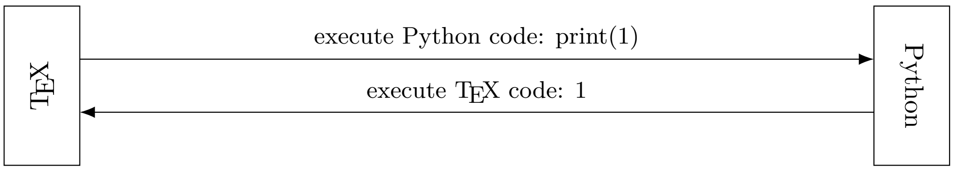
There are various methods for the 2 Python child processes to communicate with each other. After some initial bootstrapping to setup the communication, we can consider only the `textopy` script, the other merely serves as the bridge to send input to TEX.

The communication protocol is a little complicated, since it must support nesting bidirectional execution of TEX and Python.

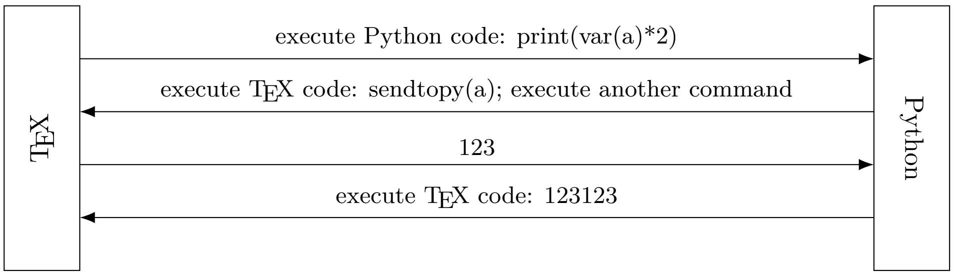
Besides, I believe it's not possible to make a "background listener" on the TEX side, so it must keep track of whether a command should be read from Python and executed.

Currently, exception handling (throwing a Python exception in a nested Python function, catch it in the outer Python function) is not supported.

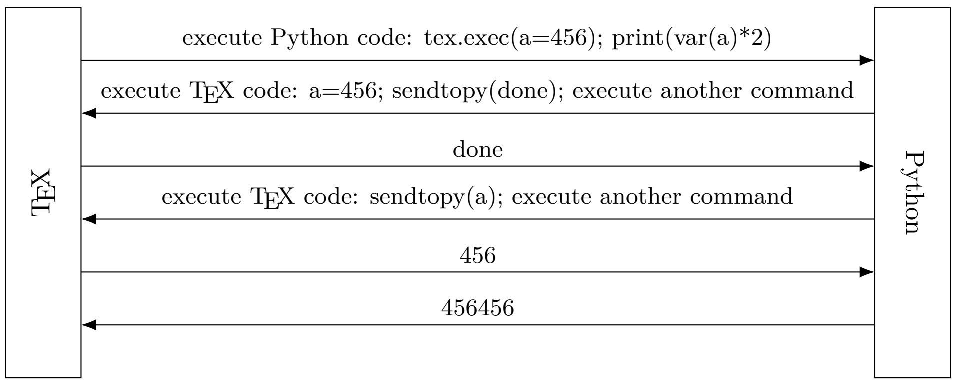
These are some examples of what could happen in the communication protocol.



Nevertheless, there may be more complicated cases where the Python code itself may call TEX code before actually returns:


TEX
Python
execute Python code: print(var(a)*2)
execute TEX code: sendtopy(a); execute another command
123
execute TEX code: 123123

Or:


TeX
execute Python code: tex.exec(a=456); print(var(a)*2)
execute TeX code: a=456; sendtopy(done); execute another command
done
execute TeX code: sendtopy(a); execute another command
456
456456
Python

The Python side must not just listen for "done" command back, but must potentially call a nested loop.

The exact protocol is:

- "execute Python code" sends from TeX to Python has a single line "`i`⟨*handler name*⟩", followed by any number of arguments (depends on the handler).

  Refer to the `define_TeX_call_Python` internal function for details.

- "done" sends from TeX to Python has the format "`r`⟨*optional return value as a string in a single line*⟩".

  This is sent by executing TeX command `\pythonimmediatecontinue`, which takes a single argument to be e-expanded using `\write` as the "return value".

- "execute TeX code" sends from Python to TeX must only be sent when the TeX side listens for a command. It consist of a single line specify the "command name", which TeX will execute the command named `\__run_`⟨*command name*⟩`:` which must already be defined on the TeX side.

  The command itself might contain additional code to execute more code, e.g. by reading more lines from Python.

  Refer to the `define_Python_call_TeX` internal function for details.

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.