

pythonimmediate — Library to run Python code*

user202729

Released 2022/12/23

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

The package should work out of the box with no additional Python package needed.

2.1 Installation on Overleaf

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

Instruction:

- Download the following files and place it in the root folder of Overleaf:
 - `saveenv.sty`
 - `precatatl.sty`
 - `pythonimmediate.sty`
 - `pythonimmediate_script_textopy.py`
 - `pythonimmediate_script_pytotex.py`
- Write the following in the preamble:

*This file describes version 0.0.0, last revised 2022/12/23.

```

1 \usepackage[abspath]{currfile}
2 \usepackage[mode=unnamed-pipe]{pythonimmediate}

```

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.

3 Usage

3.1 Package options

`outputdir=` Specify the output directory if it's not the default value.

T_EXhackers note: The value will be `x`-expanded.

Using the `outputdir` package, it's possible to determine the output directory automatically, subject to restrictions.

An usage example, if you invoked `pdflatex` with the command

```

1 pdflatex --output-directory=/tmp/ a.tex

```

then in the file `a.tex` you should have

```

1 \usepackage[outputdir=/tmp/]{pythonimmediate}

```

`mode=` Method to communicate between T_EX and Python. Usually the method will be automatically detected.

Possible values include `multiprocessing-network` and `unnamed-pipe`.

3.2 T_EX interface

The interface mimics those in popular packages such as PythonT_EX or PyLuaT_EX.

3.2.1 Inline commands

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

T_EXhackers 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 T_EX terminology, roughly means the following:

- the value can only be used to typeset text, it must not be used to pass “values” to other L^AT_EX commands.

The following is legal:

```
1 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”:

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

A possible workaround is:

```
1 \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

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

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

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

respectively.

- Macros will be expanded.

```
1 \def\mycode{1+1}
2 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.

<code>\pycq</code>	Same as above, but output (3.3.1) will not be typesetted.
<code>\pyfile</code>	Given an argument being the file name, execute that file.
<code>\pys</code>	Performs “string interpolation”, the same way as <code>PythonTeX</code> . (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

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

Special note: white spaces at the end of lines are preserved.
Any output (as described in 3.3.1) will be typesetted.

<code>pycodeq</code>	Same as above, but output will not be typesetted.
<code>pysub</code>	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.

3.3.1 Print to `TeX`

`.print()` These functions are used in `\pyc` command or `pycode` environment.
`.file` 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

```
1 pythonimmediate.print(...)
2 print(..., file=pythonimmediate.file)
3 with contextlib.redirect_stdout(pythonimmediate.file):
4     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()` Same as `LATeX`’s `\newcommand` and `\renewcommand`. Can be used as follows:
`.renewcommand()`

```
1 from pythonimmediate import newcommand, renewcommand
2
3 @newcommand
4 def function():
5     ...
```

```

6 # define |\function| in TeX
7
8 @newcommand("controlsequencename")
9 def function():
10     ...
11 # define |\controlsequencename| in TeX
12
13 def function():
14     ...
15 newcommand("controlsequencename", function)

```

`.get_arg_str()` There are those functions that is mostly understandable to an inexperienced L^AT_EX user, and should be sufficient for a lot of programming works.
`.get_optional_arg_str()`
`.get_verb_arg()` This is an example of how the functions could be used. The name should be mostly self-explanatory.
`.get_multiline_verb_arg()`
`.peek_next_char()`
`.get_next_char()`

```

1 \documentclass{article}
2 \usepackage{pythonimmediate}
3 \begin{document}
4 \begin{pycode}
5 from pythonimmediate import newcommand, peek_next_char, get_next_char,
6   ↪ get_arg_str, print
7 @newcommand
8 def innerproduct():
9     s = get_arg_str()    # in the example below this will have the value
10    ↪ '\mathbf{a},\mathbf{b}'
11    x, y = s.split(",")  # it's just a Python string, manipulate
12    ↪ normally (but be careful of comma inside braces, parse the
13    ↪ string yourself)
14    print(r"\left\langle" + x + r"\middle|" + y + r"\right\rangle")
15
16 @newcommand
17 def fx():
18     if peek_next_char() == "_":
19         get_next_char()
20         subscript = get_arg_str()
21         print("f_" + subscript + "(x)")
22     else:
23         print("f(x)")
24
25 @newcommand
26 def sumManyArgs():
27     s = 0
28     while peek_next_char() == "{":
29         i = get_arg_str()
30         s += int(i)
31     print(str(s))
32 \end{pycode}
33 $1+2+3 = \sumManyArgs{1}{2}{3}$
34

```

```

31 $\innerproduct{\mathbf{a},\mathbf{b}}=1$
32
33 $\fx = 1$, $\fx_i = 2$, $\fx_{ij} = 3$
34 \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()` 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()`)

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

```

1 \documentclass{article}
2 \usepackage{tikz}
3 \usepackage{pythonimmediate}
4 \begin{document}
5
6 \begin{tikzpicture}
7 \begin{pycode}
8 from pythonimmediate import execute
9 execute(r'\draw (0, 0) to (1, 1);')
10 execute(r'\draw (2, 2) to (p);')
11 execute(r'\draw (3, 3) to (4, 4);')
12 \end{pycode}
13 \end{tikzpicture}
14
15 \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

```
1 \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 “T_EX 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 L^AT_EX3 have such a function).

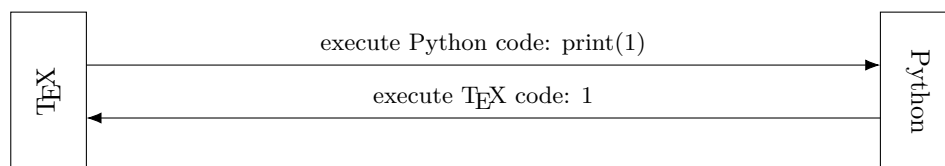
There are various methods for the 2 Python child processes to communicate with each other. After some initial bootstrapping to setup the communication, we can consider only the `textopy` script, the other merely serves as the bridge to send input to `TEX`.

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

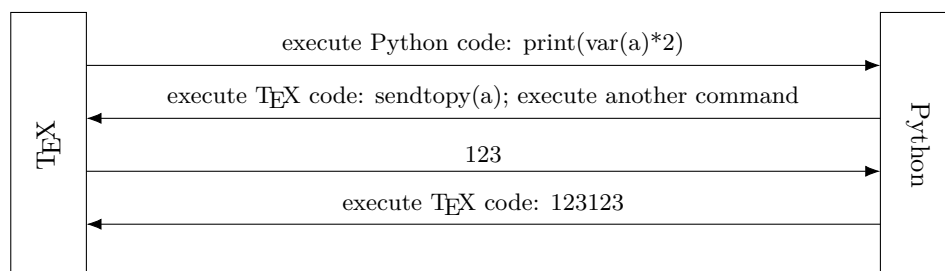
Besides, I believe it's not possible to make a “background listener” on the `TEX` side, so it must keep track of whether a command should be read from Python and executed.

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

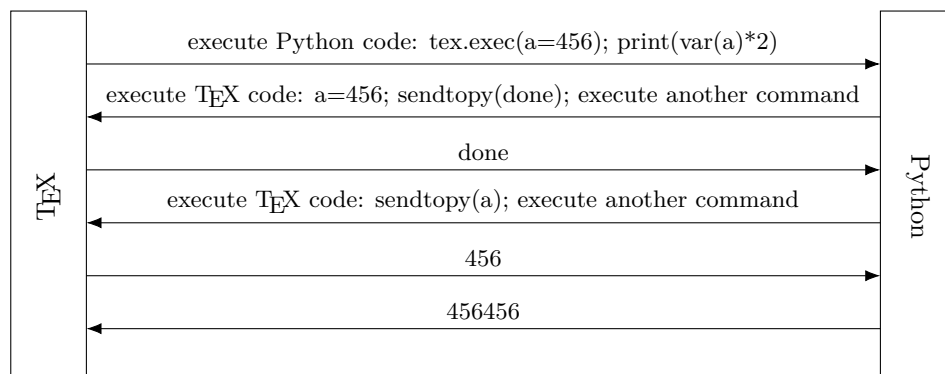
These are some examples of what could happen in the communication protocol.



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



Or:



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.

Symbols		E
<code>\.execute()</code>	<i>6</i>	environments:
<code>\.file</code>	<i>4</i>	<code>pycode</code>
<code>\.get_arg_estr()</code>	<i>6</i>	<code>pycodeq</code>
<code>\.get_arg_str()</code>	<i>5</i>	<code>pysub</code>
<code>\.get_multiline_verb_arg()</code>	<i>5</i>	
<code>\.get_next_char()</code>	<i>5</i>	P
<code>\.get_optional_arg_estr()</code>	<i>6</i>	<code>\py</code>
<code>\.get_optional_arg_str()</code>	<i>5</i>	<code>\pyc</code>
<code>\.get_verb_arg()</code>	<i>5</i>	<code>pycode</code> (environment)
<code>\.newcommand()</code>	<i>4</i>	<code>pycodeq</code> (environment)
<code>\.peek_next_char()</code>	<i>5</i>	<code>\pycq</code>
<code>\.print()</code>	<i>4</i>	<code>\pyfile</code>
<code>\.renewcommand()</code>	<i>4</i>	<code>\pys</code>
		<code>pysub</code> (environment)