PythonTeX Quickstart

github.com/gpoore/pythontex

Installing

PythonTeX requires Python 2.7 or 3.2+. When using PythonTeX with LyX, be aware that LyX may try to use its own version of Python; you may need to reconfigure LyX to use other Python installations.

PythonTeX is included in TeX Live and MiKTeX. It may be installed via the package manager.

A Python installation script is included with the package. It should be able to install the package in most situations. Depending on the configuration of your system, you may have to run the installation script with administrative priviliges.

Detailed installation information is available in the main documentation, `pythontex.pdf`.

Compiling

Compiling a document that uses PythonTeX involves three steps: run LaTeX, run `pythontex.py`, and finally run LaTeX again. You may wish to create a symlink or launching wrapper for `pythontex.py`, if one was not created during installation. PythonTeX is compatible with the pdfTeX, XeTeX, and LuaTeX engines, so you can use `latex`, `pdflatex`, `xelatex`, or `lualatex`.

The last two compile steps are *only* necessary when code needs to be executed or highlighted. Otherwise, the document may be compiled just like a normal LaTeX document; all output is cached.

PythonTeX is compatible with `latexmk`. Details for configuring `latexmk` are provided in the main documentation.

Basic commands

`\py` returns a string representation of its argument. For example, `\py{2 + 4**2}` produces "18", and `\py{'ABC'.lower()}` produces "abc". `\py`'s argument can be delimited by curly braces, or by a matched pair of other characters (just like `\verb`).

`\pyc` executes code. By default, anything that is printed is automatically included in the document (see `autoprint`/`autostdout` in the main documentation). For example, `\pyc{var = 2}` creates a variable, and then its value may be accessed later via `\py{var}`: 2.

`\pyb` executes and typesets code. For example, `\pyb{var = 2}` typesets `var = 2` in addition to creating the variable. If anything is printed, it is not automatically included, but can be accessed via `\printpythontex` or `\stdoutpythontex`.

`\pyv` only typesets code; nothing is executed. For example, `\pyv{var = 2}` produces `var = 2`.

`\pys` performs variable substitution or string interpolation on code. Substitution fields are denoted by `!{...}`; details about escaping are provided in the main documentation. For example, using the pre-existing variable `var`, `\pys{\verb|var = !{var}|}` yields `var = 2`.

Basic environments

There are `pycode`, `pyblock`, `pyverbatim`, and `pysub` environments, which are the environment equivalents of `\pyc`, `\pyb`, `\pyv`, and `\pys`. For example,

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

produces

A message from Python!

The `\begin` and `\end` of an environment should be on lines by themselves. Code in environments may be indented; see the `gobble` option in the main documentation for more details.

More commands/environments

All commands and environments described so far have names beginning with `py`. There are equivalent commands and environments that begin with `sympy`; these automatically include

```
from sympy import *
```

There are also equivalent commands and environments that begin with `pylab`; these automatically use matplotlib's `pylab` module via

```
from pylab import *
```

The `sympy` and `pylab` commands and environments execute code in separate sessions from the `py` commands and environments. This can make it easier to avoid namespace conflicts.

There is also a `pyconsole` environment that emulates a Python interactive console. For example,

```
\begin{pyconsole}
var = 1 + 1
var
\end{pyconsole}
```

yields

```
>>> var = 1 + 1
>>> var
2
```

Console variable values may be accessed inline via the `\pycon` command. More console information is available in the main documentation.

Working with Python 2

PythonTEX supports both Python 2 and 3. Under Python 2, imports from `__future__` will work so long as they are the first user-entered code in a given session. PythonTEX imports most things from `__future__` by default. To control what is automatically imported, see the `pyfuture` and `pyconfuture` package options in the main documentation.

Support for additional languages

PythonTEX also provides support for additional languages. Currently, Ruby, Julia, Octave, Sage, Bash, and Rust support is included. To enable commands and environments for these language, see the `usefamily` package option in the main documentation.

Language support is provided via a template system; in most cases, a new language can be added with about 100 lines of template code—and basic support can require less than 20 lines. If you would like support for a new language, please open an issue at GitHub. The main documentation also contains a summary of the process for adding languages.

Macro programming

PythonTEX commands can be used inside other commands in macro programming. They will usually work fine, but curly braces should be used as delimiters and special LaTeX characters such as `%` and `#` should be avoided in the Python code. These limitations can be removed by passing arguments verbatim or through catcode trickery. PythonTEX environments cannot normally be used inside LaTeX commands, due to the way LaTeX deals with verbatim content and catcodes.

Additional features

PythonTEX provides many additional features. The working and output directories can be specified via `\setpythontexworkingdir` and `\setpythontexoutputdir`. The user can determine when code is executed with the package option `rerun`, selecting factors such as modification and exit status. By default, all commands and environments with the same base name (`py`, `sympy`, `pylab`, etc.) run in a single session, providing continuity. Commands and environments accept an optional argument that specifies the session in which the code is executed; sessions run in parallel. PythonTEX provides a utilities class that is always imported into each session. The utilities class provides methods for tracking dependencies and automatically cleaning up created files. The utilities class also allows information such as page width to be passed from the TEX side to Python/other languages. See the main documentation for additional information.

PythonTEX also provides the `depythontex` utility, which 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, sharing, and conversion to other document formats.

Code may be run in interactive mode on the command line via the `--interactive` and `--debug` options. This is primarily useful for working with interactive debuggers.

Customizing typesetting

PythonTEX typesets code using the `fancyvrb` package and the `fvextra` package that extends `fancyvrb`. There is a `\setpythontexfv` command for setting PythonTEX-specific `fancyvrb` and `fvextra` options. The normal `\fvset` works as well for document-wide settings. PythonTEX environments take a second optional argument that consists of `fancyvrb` and `fvextra` settings. This can be used to customize automatic line breaking or line highlighting for a single environment.

Unicode support

PythonTEX supports Unicode under all LaTeX engines. For example, consider the following example from Python:

```
my_string = '¥ § ß Ğ Ð Ñ Ü þ ø'
```

This requires some engine-specific packages. Typical packages are listed below.

- pdfLaTeX:

  ```
  \usepackage[T1]{fontenc}
  \usepackage[utf8]{inputenc}
  ```

- LuaLaTeX:

  ```
  \usepackage{fontspec}
  ```

- XeLaTeX:

  ```
  \usepackage{fontspec}
  \defaultfontfeatures{Ligatures=TeX}
  ```

If you are using Python 2, you will also need to specify that you are using Unicode. You may want

```
from __future__ import unicode_literals
```

at the beginning of your Python code. Or you can just load the PythonTEX package with the option `pyfuture=all`, which will import `unicode_literals` automatically.