The pythontex package

Geoffrey M. Poore
gpoore@gmail.com
github.com/gpoore/pythontex

v0.18 from 2021/06/06

Abstract

PythonTeX provides access to Python from within LaTeX documents. It allows Python code entered within a LaTeX document to be executed, and the results to be included within the original document. Python code may be adjacent to the figure or calculation it produces. The package also makes possible macro definitions that mix Python and LaTeX code. In addition, PythonTeX provides syntax highlighting for many programming languages via the Pygments syntax highlighter.

PythonTeX is fast and user-friendly. Python code is only executed when it has been modified, or when user-specified criteria are met. When code is executed, user-defined sessions automatically run in parallel. If Python code produces errors, the error message line numbers are synchronized with the LaTeX document line numbering, simplifying debugging. Dependencies may be specified so that code is automatically re-executed whenever they are modified.

Because documents that use PythonTeX mix LaTeX and Python code, they are less suitable than plain LaTeX documents for journal submission, sharing, and conversion to other formats. PythonTeX includes a `depythontex` utility that creates a copy of a document in which all PythonTeX content is replaced by its output.

While Python is the focus of PythonTeX, adding basic support for an additional language is usually as simple as creating a new class instance and a few templates, usually totaling less than 100 lines of code. The following languages already have built-in support: Ruby, Julia, Octave, Bash, Rust, R, Perl, Perl 6, and JavaScript.

Warning

PythonTeX makes possible some pretty amazing things. But that power brings with it a certain risk and responsibility. Compiling a document that uses PythonTeX involves executing Python code, and potentially other programs, on your computer. You should only compile PythonTeX documents from sources you trust. PythonTeX comes with NO WARRANTY.[1] The copyright holder and any additional authors will not be liable for any damages.

[1] All LaTeX code is licensed under the LaTeX Project Public License (LPPL) and all Python code is licensed under the BSD 3-Clause License.

Contents

1 Introduction

This introduction provides background and objectives for the PythonTeX package. To jump right in and get started, you may wish to consult the `pythontex_quickstart` and `pythontex_gallery` documents, as well as Sections 3 and 4, below. If you are primarily interested in using PythonTeX with a language other than Python, see Section 7.

LaTeX can do a lot,[2] but the programming required can sometimes be painful.[3] In spite of the many packages available for LaTeX, the libraries and packages of a general-purpose programming language are lacking. Furthermore, it can be convenient to include non-LaTeX code in a document to make it more reproducible. For these reasons, there have been multiple systems that allow other languages to be used within LaTeX documents.[4]

- PerlTeX allows the bodies of LaTeX macros to be written in Perl.
- SageTeX allows code for the Sage mathematics software to be executed from within a LaTeX document.
- Martin R. Ehmsen's `python.sty` provides a very basic method of executing Python code from within a LaTeX document.
- SympyTeX allows more sophisticated Python execution, and is largely based on a subset of SageTeX.
- LuaTeX extends the pdfTeX engine to provide Lua as an embedded scripting language, and as a result yields tight, low-level Lua integration.

PythonTeX attempts to fill a perceived gap in the current integrations of LaTeX with an additional language. It has a number of objectives, only some of which have been met by previous packages.

Execution speed

In the approaches mentioned above, all the non-LaTeX code is executed at every compilation of the LaTeX document (PerlTeX, LuaTeX, and `python.sty`), or all the non-LaTeX code is executed every time it is modified (SageTeX and SympyTeX). However, many tasks such as plotting and data analysis take a significant time to execute. We need a way to fine-tune code execution, so that independent blocks of slow code may be separated into their own sessions and are only executed when modified. If we are going to split code into multiple sessions, we might as well run these sessions in parallel, further increasing speed. A byproduct of this approach is that it now becomes much more feasible to include slower code, since we can still have fast compilations whenever the slow code isn't modified.

Compiling without executing

Even with all of these features to boost execution speed, there will be times

[2] TeX is a Turing-complete language.

[3] As I learned in creating this package.

[4] I am not including the various web and weave dialects in my discussion, since they typically involve a web or weave document from which the .tex source is generated, and thus weaker integration with LaTeX. Two sophisticated examples of this approach are Sweave and knitr, both of which combine LaTeX with the R language for tasks such as dynamic report generation.

when we have to run slow code. Thus, we need the execution of non-LaTeX code to be separated from compiling the LaTeX document. We need to be able to edit and compile a document containing unexecuted code. Unexecuted code should be invisible or be replaced by placeholders. SageTeX and SympyTeX have implemented such a separation of compiling and executing. In contrast, LuaTeX and PerlTeX execute all the code at each compilation—but that is appropriate given their goal of simplifying macro programming.

Error messages

Whenever code is saved from a LaTeX document to an external file and then executed, the line numbers for any error messages will not correspond to the line numbering of the original LaTeX document. At one extreme, `python.sty` doesn't attempt to deal with this issue, while at the other extreme, SageTeX uses an ingenous system of `Try`/`Except` statements on every chunk of code. We need a system that translates all error messages so that they correspond to the line numbering of the original LaTeX document, with minimal overhead when there are no errors.

Syntax highlighting

Once we begin using non-LaTeX code, sooner or later we will want to typeset some of it, which means we need syntax highlighting. A number of syntax highlighting packages currently exist for LaTeX; perhaps the most popular are `listings` and `minted`. `listings` uses pure LaTeX. It has not been updated since 2007, which makes it a less ideal solution in some circumstances. `minted` uses the Python-based syntax highlighter Pygments to perform highlighting. Pygments can provide superior syntax highlighting, but `minted` can be very slow because all code must be highlighted at each compilation and each instance of highlighting involves launching an external Python process. We need high-speed, user-friendly syntax highlighting via Pygments.[5]

Printing

It would be nice for the `print` statement/function,[6] or its equivalent, to automatically return its output within the LaTeX document. For example, using `python.sty` it is possible to generate some text while in Python, open a file, save the text to it, close the file, and then `\input` the file after returning to LaTeX. But it is much simpler to generate the text and `print` it, since the printed content is automatically included in the LaTeX document. This was one of the things that `python.sty` really got right.

Pure code

LaTeX has a number of special characters (`# $ % & ~ _ ^ \ { }`), which complicates the entry of non-LaTeX code since these same characters are common in many languages. SageTeX and SympyTeX delimit all inline code with curly braces (`{}`), but this approach fails in the (somewhat unlikely) event that code needs to contain an unmatched brace. More seriously, they

[5] The author recently started maintaining the `minted` package. In the near future, `minted` will inherit PythonTeX's speed enhancements, and the two packages will become more compatible.

[6] In Python, `print` was a statement until Python 3, when it became a function. The function form is available via import from `\textunderscore\textunderscore future\textunderscore\textunderscore` in Python 2.6 and later.

do not allow the percent symbol `%` (modular arithmetic and string formatting in Sage and Python) to be used within inline code. Rather, a `\percent` macro must be used instead. This means that code must (sometimes) be entered as a hybrid between LaTeX and the non-LaTeX language. LuaTeX is somewhat similar: "The main thing about Lua code in a TeX document is this: the code is expanded by TeX before Lua gets to it. This means that all the Lua code, even the comments, must be valid TeX!"[7] In the case of LuaTeX, though, there is the `luacode` package that allows for pure Lua.

This language hybridization is not terribly difficult to work around in the SageTeX and SympyTeX cases, and is actually a LuaTeX feature in many contexts. But if we are going to create a system for general-purpose access to a non-LaTeX language, we need **all** valid code to work correctly in **all** contexts, with no hybridization of any sort required. We should be able to copy and paste valid code into a LaTeX document, without having to worry about hybridizing it. Among other things, this means that inline code delimiters other than LaTeX's default curly braces `{}` must be available.

Hybrid code
: Although we need a system that allows input of pure non-LaTeX code, it would also be convenient to allow hybrid code, or code in which LaTeX macros may be present and are expanded before the code is executed. This allows LaTeX data to be easily passed to the non-LaTeX language, facilitating a tighter integration of the two languages and the use of the non-LaTeX language in macro definitions.

Math and science libraries
: The author decided to create PythonTeX after writing a physics dissertation using LaTeX and realizing how frustrating it can be to switch back and forth between a TeX editor and plotting software when fine-tuning figures. We need access to a non-LaTeX language like Python, MATLAB, or Mathematica that provides strong support for data analysis and visualization. To maintain broad appeal, this language should primarily involve open-source tools, should have strong cross-platform support, and should also be suitable for general-purpose programming.

Language-independent implementation
: It would be nice to have a system for executing non-LaTeX code that depends very little on the language of the code. We should not expect to escape all language dependence. But if the system is designed to be as general as possible, then it may be expanded in the future to support additional languages.

Python was chosen as the language to fulfill these objectives for several reasons.

- It is open-source and has good cross-platform support.
- It has a strong set of scientific, numeric, and visualization packages, including NumPy, SciPy, matplotlib, and SymPy. Much of the initial motivation for PythonTeX was the ability to create publication-quality plots and perform complex mathematical calculations without having to leave the TeX editor.

[7] http://wiki.contextgarden.net/Programming_in_LuaTeX

- We need a language that is suitable for scripting. Lua is already available via LuaTeX, and in any case lacks the math and science tools.[8] Perl is already available via PerlTeX, although PerlTeX's emphasis on Perl for macro creation makes it rather unsuitable for scientific work using the Perl Data Language (PDL) or for more general programming. Python is one logical choice for scripting.

Now at this point there will almost certainly be some reader, sooner or later, who wants to object, "But what about language *X*!" Well, yes, in some respects the choice to use Python did come down to personal preference. But you should give Python a try, if you haven't already. You may also wish to consider the many interfaces that are available between Python and other languages. If you still aren't satisfied, keep in mind PythonTeX's "language-independent" implementation! In many cases, adding support for additional languages is relatively simple (see Section 7).

2 Citing PythonTeX

If you use PythonTeX in your writing and research, please consider citing it in any resulting publications. The best and most recent paper is in *Computational Science & Discovery*.

- "PythonTeX: reproducible documents with LaTeX, Python, and more," Geoffrey M Poore. *Computational Science & Discovery* 8 (2015) 014010. Full text and BibTeX entry available at `http://stacks.iop.org/1749-4699/8/i=1/a=014010`.

- "Reproducible Documents with PythonTeX," Geoffrey M. Poore. *Proceedings of the 12th Python in Science Conference* (2013), pp. 73–79. Full text and BibTeX entry available at `http://conference.scipy.org/proceedings/scipy2013/poore.html`.

3 Installing and running

3.1 Installing PythonTeX

PythonTeX requires a TeX installation. It has been tested with TeX Live and MiKTeX, but should work with other distributions. The following LaTeX packages, with their dependencies, are required: `fancyvrb`, `fvextra`, `etoolbox`, `xstring`, `pgfopts`, `newfloat` (part of the `caption` bundle), `currfile`, and `color` or `xcolor`. A current TeX installation is recommended, since some features require recent versions of the packages. If you are creating and including graphics, you will also need `graphicx`. The `mdframed` package is recommended for enclosing typeset code in boxes with fancy borders and/or background colors; `tcolorbox` and `framed` are alternatives.

PythonTeX also requires a Python installation. Python 2.7 is recommended for the greatest compatibility with scientific tools, although many scientific packages are now compatible with Python 3. PythonTeX is compatible with Python

[8] One could use Lunatic Python, and some numeric packages for Lua are in development.

2.7 and 3.2+. The Python package Pygments must be installed for syntax highlighting to function. PythonTEX has been tested with Pygments 1.4 and later, but the latest version is recommended. For scientific work, or to compile `pythontex_gallery.tex`, the following are also recommended: NumPy, SciPy, matplotlib, and SymPy. When using PythonTEX with LyX, be aware that LyX may try to use its own version of Python; you may need to reconfigure LyX.

PythonTEX also provides support for other languages such as Ruby, so you will need to install any additional languags you plan to use. Typically, the most recent major version of these languages is supported.

PythonTEX consists of the following files:

- Installer file `pythontex.ins`
- Documented LaTeX source file `pythontex.dtx`, from which `pythontex.pdf` and `pythontex.sty` are generated
- Main script `pythontex.py`, which imports from `pythontex2.py` or `pythontex3.py`, based on the Python version
- Language definitions `pythontex_engines.py`
- Utilities class `pythontex_utils.py`
- `depythontex.py`, which imports from `depythontex2.py` or `depythontex3.py`, based on the Python version; used to remove PythonTEX dependence
- Synchronized Python Debugger `syncpdb.py`
- `README` (in rst style)
- `pythontex_gallery.tex` and `pythontex_gallery.pdf`
- `pythontex_quickstart.tex` and `pythontex_quickstart.pdf`
- Optional installation script `pythontex_install.py` for TEX Live and MiKTeX
- Optional batch file `pythontex.bat` for use in launching `pythontex.py` under Windows
- Optional conversion script `pythontex_2to3.py` for converting PythonTEX code written for Python 2 into a form compatible with Python 3

The style file `pythontex.sty` may be generated by running LaTeX on `pythontex.ins`. The documentation you are reading may be generated by running LaTeX on `pythontex.dtx`. Some code is provided in two forms, one for Python 2 and one for Python 3 (names ending in `2` and `3`). Whenever this is the case, a version-independent wrapper is supplied that automatically runs the correct code based on the Python version. For example, there are two main scripts, `pythontex2.py` and `pythontex3.py`, but you should actually run `pythontex.py`, which imports the correct code based on the Python version.[9]

[9] Unfortunately, it is not possible to provide full Unicode support for both Python 2 and 3 using a single script. Currently, all code is written for Python 2, and then the Python 3 version is automatically generated via the `pythontex\_2to3.py` script. This script comments out code that is only for Python 2, and un-comments code that is only for Python 3.

If you want the absolute latest version of PythonTeX, you should install it manually from github.com/gpoore/pythontex. A Python installation script is provided for use with TeX Live and MiKTeX. It has been tested with Windows, Linux, and OS X, but may need manual input or slight modifications depending on your system. The installation script performs the steps described below.

For a MiKTeX installation, you may need administrator privileges; running `pythontex_install.bat` as administrator may be simplest.

Note that for a typical TeX setup under Linux, you may need to run the script with elevated privileges, and may need to run it with the user's PATH. This can be necessary when you are using a Linux distribution that includes an outdated version of TeX Live, and have installed a new version manually. **If you are installing PythonTeX on a machine with multiple versions of TeX, make sure you install PythonTeX for the correct version.** For example, under Ubuntu Linux, you will probably need the following command if you have installed the latest version of TeX Live manually:

```
sudo env PATH=$PATH python pythontex_install.py
```

The installer creates the following files. It will offer to create the paths if they do not exist. If you are installing in TEXMFLOCAL, the paths will have an additional `local/` at the end.

- ⟨*TeX tree root*⟩`/doc/latex/pythontex/`
  - `pythontex.pdf`
  - `README`
  - `pythontex_quickstart.tex`
  - `pythontex_quickstart.pdf`
  - `pythontex_gallery.tex`
  - `pythontex_gallery.pdf`
- ⟨*TeX tree root*⟩`/scripts/pythontex/`
  - `pythontex.py`, `pythontex2.py` and `pythontex3.py`
  - `pythontex_engines.py`
  - `pythontex_utils.py`
  - `depythontex.py`, `depythontex2.py` and `depythontex3.py`
  - `syncpdb.py`
- ⟨*TeX tree root*⟩`/source/latex/pythontex/`
  - `pythontex.dtx`
  - `pythontex.ins`
- ⟨*TeX tree root*⟩`/tex/latex/pythontex/`
  - `pythontex.sty`

After the files are installed, the system must be made aware of their existence. The installer runs `mktexlsr` to do this. In order for `pythontex.py` and `depythontex.py` to be executable, a symlink (TeX Live under Linux), launching wrapper (TeX Live under Windows), or batch file (general Windows) should be created in the `bin/`⟨*system*⟩ directory. The installer attempts to create a symlink or launching wrapper automatically. For TeX Live under Windows, it copies `bin/win32/runscript.exe` to `bin/win32/pythontex.exe` to create the wrapper.[10]

3.2 Compiling documents using PythonTeX

Compiling a document with PythonTeX involves three steps: running a LaTeX-compatible TeX engine (binary executable), running `pythontex.py` (preferably via a symlink, wrapper, or batch file, as described above), and finally running the TeX engine again. The first TeX run saves code into an external file where PythonTeX can access it. The second TeX run pulls the PythonTeX output back into the document.

If you plan to use code that contains non-ASCII characters such as Unicode, you should make sure that your document is properly configured:

- Under pdfLaTeX, your documents need `\usepackage[T1]{fontenc}` and `\usepackage[utf8]{inputenc}`, or a similar configuration.
- Under LuaLaTeX, your documents need `\usepackage{fontspec}`, or a similar configuration.
- Under XeLaTeX, your documents need `\usepackage{fontspec}` as well as `\defaultfontfeatures{Ligatures=TeX}`, or a similar configuration.

For an example of a PythonTeX document that will correctly compile under all three engines, see the `pythontex_gallery.tex` source.

If you use XeLaTeX, and your non-LaTeX code contains tabs, you **must** invoke XeLaTeX with the `-8bit` option so that tabs will be written to file as actual tab characters rather than as the character sequence `^^I`.[11]

`pythontex.py` requires a single command-line argument: the name of the .tex file to process. The filename can be passed with or without an extension; the script really only needs the `\jobname`, so any extension is stripped off.[12] The filename may include the path to the file; you do not have to be in the same directory as the file to run PythonTeX. If you are configuring your editor to run PythonTeX automatically via a shortcut, you may want to wrap the filename in double quotes `"` to allow for space characters.[13] For example, under Windows with TeX Live and Python 2.7 we would create the wrapper `pythontex.exe`. Then we could run PythonTeX on a file ⟨*file name*⟩.tex using the command `pythontex.exe "`⟨*file name*⟩`"`.

[10] See the output of `runscript -h` under Windows for additional details.

[11] See http://tex.stackexchange.com/questions/58732/how-to-output-a-tabulation-into-a-file for more on tabs with XeTeX.

[12] Thus, PythonTeX works happily with .tex, .ltx, .dtx, and any other extension.

[13] Using spaces in the names of .tex files is apparently frowned upon. But if you configure things to handle spaces whenever it doesn't take much extra work, then that's one less thing that can go wrong.

`pythontex.py` accepts the following optional command-line arguments. Some of these options duplicate package-level options, so that settings may be configured either within the document or at the command line. In the event that the command-line and package options conflict, the package options always override the command-line options. For variations on these options that are acceptable, run `pythontex.py -h`.

- `--encoding=`⟨*encoding*⟩ This sets the file encoding. Any encoding supported by Python's `codecs` module may be used. The encoding should match that of the LaTeX document. If an encoding is not specified, PythonTeX uses UTF-8. If support for characters beyond ASCII is required, then additional LaTeX packages are required; see the discussion of TeX engines above.

- `--error-exit-code={true,false}` By default, `pythontex.py` returns an exit code of 1 if there were any errors, and an exit code of 0 otherwise. This may be useful when PythonTeX is used in a scripting or command-line context, since the presence of errors may be easily detected. It is also useful with some TeX editors. For example, TeXworks automatically hides the output of external programs unless there are errors.

  In some contexts, returning a nonzero exit code can be redundant. For example, with the WinShell editor under Windows with TeX Live, the complete output of PythonTeX is always available in the "Output" view, so it is clear if errors have occurred. Having a nonzero exit code causes `runscript.exe` to return an additional, redundant error message in the "Output" view. In such situations, it may be desirable to disable the nonzero exit code.

- `--runall=[{true,false}]` This causes **all** code to be executed, regardless of modification or `rerun` settings. It is useful when code has not been modified, but a dependency such as a library or external data has changed. Note that the PythonTeX utilities class also provides a mechanism for automatically re-executing code that depends on external files when those external files are modified.

  There is an equivalent `runall` package option. The command-line option `--rerun=always` is essentially equivalent.

- `--rerun={never,modified,errors,warnings,always}` This sets the threshold for re-executing code. By default, PythonTeX will rerun code that has been modified or that produced errors on the last run. Sometimes, we may wish to have a more lenient setting (only rerun if modified) or a more stringent setting (rerun even for warnings, or just rerun everything). `never` never executes code; a warning is issued if there is modified code. `modified` only executes code that has been modified (or that has modified dependencies). `errors` executes all modified code as well as all code that produced errors on the last run; this is the default. `warnings` executes all modified code, as well as all code that produced errors or warnings. `always` executes all code always and is essentially equivalent to `--runall`.

  There is an equivalent `rerun` package option.

- `--hashdependencies=[{true,false}]` This determines whether dependencies (external files highlighted by Pygments, code dependencies specified via `pytex.add_dependencies()`, etc.) are checked for changes via their hashes

or modification times. By default, mtime is used, since it is faster. The package option `hashdependencies` is equivalent.

- `--jobs` This sets the maximum number of concurrent processes. By default, this will be Python's `multiprocessing.cpu_count()`, which is the number of CPUs in the system. It may be useful to set a smaller value when some jobs are particularly resource intensive or themselves use subprocesses.

- `--verbose` This gives more verbose output, including a list of all processes that are launched.

- `--interpreter` This allows the interpreter for a given language to be specified. The argument should be in the form

  ```
  --interpreter "<interpreter>:<command>, <interp>:<cmd>, ..."
  ```

  where `<interpreter>` is `python`, `ruby`, etc., and `<command>` is the command for invoking the desired interpreter. The argument to `--interpreter` may also be in the form of a Python dictionary. The argument need not be enclosed in quotation marks if it contains no spaces.

  For example, by default Python code is executed with whatever interpreter the `python` command invokes. But Python 3 could be specified using `--interpreter python:python3` (many Linux distributions) or `--interpreter "python:py -3"` (Windows, with Python 3.3 installed so that the `py` wrapper is available).

- `--interactive [<family>:<session>:<restart>]` This is used to run a single session in interactive mode. This allows user input. Code output is written to stdout. Interactive mode is particularly useful when working with debuggers (but also see the `--debug` option).

  `[<family>:<session>:<restart>]` is optional; if it is not provided, the default session is executed. For non-default sessions (or if there are multiple default sessions, due to the use of multiple families of commands), simply supplying the session name is usually sufficient (for example, `--debug session`). The full combination of `[<family>:<session>:<restart>]` (for example, `py:session:default`) is only necessary when the session name alone would be ambiguous.

  Note that when a session is run in interactive mode, it will *not* save printed content in a form that may be brought back into the document. You will have to run the session again in normal mode to complete document compilation.

  Code that requires user input will cause PythonTeX to "hang" when PythonTeX is not running in interactive mode. This is because the code will request user input, but no input is possible given the way that the code is being executed, so the code will wait for input forever. It is inefficient constantly to add and then delete interactive code as you switch between normal and interactive modes. To avoid this, you can conditionally invoke code that requires input. In interactive mode, the temporary script that is executed is given the command-line argument `--interactive`. You can check for the presence of this argument, and only invoke interactive code if

it is present. For example, under Python you could start the `pdb` debugger, only when the code is being executed in interactive mode, using commands such as the following.

```
import pdb
import sys
if '--interactive' in sys.argv[1:]:
    pdb.set_trace()
```

This option is currently not compatible with Python console commands and environments.

- `--debug [<family>:<session>:<restart>]` This is used to run a single session with the default debugger in interactive mode. Currently, only standard Python sessions are supported. (Python console commands and environments are not supported.) Support for other languages and support for customization will be added in the future.

  `[<family>:<session>:<restart>]` is optional; if it is not provided, the default session is executed. For non-default sessions (or if there are multiple default sessions, due to the use of multiple families of commands), simply supplying the session name is usually sufficient (for example, `--debug session`). The full combination of `[<family>:<session>:<restart>]` (for example, `py:session:default`) is only necessary when the session name alone would be ambiguous.

  Note that when a session is run in debug mode, it will *not* save printed content in a form that may be brought back into the document. You will have to run the session again in normal mode to complete document compilation.

  The default Python debugger is `syncpdb`, the Synchronized Python Debugger. It provides a wrapper around `pdb` that is aware of the connection between the code and the LaTeX document from which it was extracted. All `pdb` commands function normally. In addition, commands that take a line number or filename:lineno as an argument will also take these same values with a percent symbol `%` prefix. If the percent symbol is present, then `syncpdb` interprets the filename and line number as referring to the document, rather than to the code that is executed. It will translate the filename and line number to the corresponding code equivalents, and then pass these to the standard `pdb` internals. For example, the `pdb` command `list 50` would list the code that is being executed, centered around line 50. `syncpdb` allows the command `list %10`, which would list the code that is being executed, centered around the code that came from line 10 in the main LaTeX document. (If no file name is given, then the main LaTeX document is assumed.) If the code instead came from an inputed file `input.tex`, then `list %input.tex:10` could be used. Further details are provided at github.com/gpoore/syncpdb.

  The temporary script that is executed is given the command-line argument `--interactive` when run in debug mode. You can check for the presence of this argument if you wish to invoke code that requires user input conditionally. See the `--interactive` command-line option for more details.

PythonTeX attempts to check for a wide range of errors and return meaningful error messages. But due to the interaction of LaTeX and Python code, some strange errors are possible. If you cannot make sense of errors when using PythonTeX, the simplest thing to try is deleting all files created by PythonTeX, then recompiling. By default, these files are stored in a directory called `pythontex-files-`⟨*jobname*⟩, in the same directory as your .tex document. See Section 8 for more details regarding troubleshooting.

4 Usage

4.1 Package options

Package options may be set in the standard manner when the package is loaded:

`\usepackage[`⟨*options*⟩`]{pythontex}`

All options are described as follows. The option is listed, followed by its possible values. When a value is not required, ⟨*none*⟩ is listed as a possible value. In this case, the value to which ⟨*none*⟩ defaults is also given. Each option lists its default setting, if the option is not invoked when the package is loaded.

Some options have a command-line equivalent. Package options override command-line options.

All options related to printed content are provided in two forms for convenience: one based on the word `print` and one based on `stdout`.

`usefamily=`⟨*basename*⟩`/{`⟨*basename1, basename2, ...*⟩`}`

By default, only the `py`, `sympy`, and `pylab` families of commands and environments are defined, to prevent possible package conflicts.[14] This option defines preconfigured families for other available languages. It takes either a single language base name, or a list of comma-separated names enclosed in curly braces. For example, the Ruby families `rb` and `ruby`, the Julia families `jl` and `julia`, and the Octave family `octave` may be enabled. For a full list of supported languages, see Section 7.

`gobble=none/auto`
`default:none`

This option is still under development and may change somewhat in future releases. If that occurs, equivalent functionality will be provided.

This option determines how code indentation is handled. By default, indentation is left as-is; leading whitespace is significant. `auto` will dedent all code by gobbling the largest common leading whitespace, using Python's `textwrap.dedent()`.[15] Keep in mind that Python's dedent will not work correctly with mixed tabs and spaces.

[14] For example, a `\ruby` command for Ruby code, and the `\ruby` command defined by the Ruby package in the CJK package.

[15] It would be possible to do the dedent on the LaTeX side, as is done manually in the `fancyvrb` and `listings` packages with the `gobble` option and is done automatically in the `lstautogobble` package. This is not done for stability and security reasons. `lstautogobble` determines the dedent by extracting the leading whitespace from the first line of code, and then applying this dedent to each subsequent line. This is adequate for **typesetting** code, since the worst-case scenario is that a subsequent line with less indentation will be typeset with the first few characters missing. Such an approach is not acceptable when the code will be **executed**, since a few missing characters could in principle cause serious damage. Doing the dedent on the Python side ensures that no characters are discarded, even if that results in an indentation error.

The `gobble` option always works correctly with **executed** code. However, currently the option **only works with typeset code when Pygments is used**. The option is currently only available at the document level, but finer-grained control is planned in the future.

The `gobble` option is supported by `depythontex`.

`beta=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

This option provides compatibility with the beta releases from before the full v0.11 release, which introduced some changes in syntax and command names. This option should **only** be used with old PythonTEX documents that require it.

You are encouraged to update old documents, since this compatility option will only be provided for a few releases.

`runall=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

This option causes all code to be executed, regardless of whether it has been modified. This option is primarily useful when code depends on external files, and needs to be re-executed when those external files are modified, even though the code itself may not have changed. Note that the PythonTEX utilities class also provides a mechanism for automatically re-executing code that depends on external files when those external files are modified.

A command-line equivalent `--runall` exists for `pythontex.py`. The package option `rerun=always` is essentially equivelent.

`rerun=never/modified/errors/warnings/always`
`default:errors`

This option sets the threshold for re-executing code. By default, PythonTEX will rerun code that has been modified or that produced errors on the last run. Sometimes, we may wish to have a more lenient setting (only rerun if modified) or a more stringent setting (rerun even for warnings, or always rerun). `never` never executes code; a warning is issued if there is modified code. `modified` only executes code that has been modified. `errors` executes all modified code as well as all code that produced errors on the last run; this is the default. `warnings` executes all modified code, as well as all code that produced errors or warnings. `always` executes all code regardless of its condition.

A command-line equivalent `--rerun` exists for `pythontex.py`.

`hashdependencies=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

When external code files are highlighted with Pygments, or external dependencies are specified via the PythonTEX utilities class, they are checked for modification via their modification time (Python's `os.path.getmtime()`). Usually, this should be sufficient—and it offers superior performance, which is important if data sets are large enough that hashing takes a noticeable amount of time. However, occasionally hashing may be necessary or desirable, so this option is provided.

A command-line equivalent `--hashdependencies` exists for `pythontex.py`.

`autoprint=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`
`autostdout=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`

Whenever a `print` command/statement is used, the printed content will automatically be included in the document, unless the code doing the printing is being typeset.[16] In that case, the printed content must be included using the

[16] Note that `autoprint` only works within the body of the document. The `code` command and environment can be used in the preamble, but `autoprint` is disabled there. It is usually a not a good idea to print in the preamble, because nothing can be typeset; the only thing that could be validly printed is LaTeX commands that do not typeset content, such as macro definitions. Thus, it is appropriate that printed content is only brought in while in the preamble if it is explicitly requested via `\string\printpythontex`. This approach is also helpful for writing packages using

`\printpythontex` or `\stdoutpythontex` commands.

Printed content is pulled in directly from the external file in which it is saved, and is interpreted by LaTeX as LaTeX code. If you wish to avoid this, you should print appropriate LaTeX commands with your content to ensure that it is typeset as you desire. Alternatively, you may use `\printpythontex` or `\stdoutpythontex` to bring in printed content in verbatim form, using those commands' optional `verb` and `verbatim` options.

The `autoprint` (`autostdout`) option sets autoprint behavior for the entire document. This may be overridden within the document using the `\setpythontexautoprint` command.

`debug`

This option aids in debugging invalid LaTeX code that is brought in from Python. It disables the inclusion of printed content/content written to stdout. Since printed content should almost **always** be included, a warning is raised when this option is used.

Not including printed content is useful when the printed content contains LaTeX errors, and would cause document compilation to fail. When the document fails to compile, this can prevent modified Python code from being written to the code file, resulting in an inescapable loop unless printed content is disabled or the saved output is deleted.

Note that since commands like `\py` involve printing, they are also disabled.

`makestderr=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

This option determines whether the stderr produced by scripts is available for input by PythonTeX, via the `\stderrpythontex` macro. This will not be needed in most situations. It is intended for typeseting incorrect code next to the errors that it produces. This option is not `true` by default, because additional processing is required to synchronize stderr with the document.

`stderrfilename=full/session/genericfile/genericscript`
`default:full`

This option governs the file name that appears in `stderr`. Python errors begin with a line of the form

```
File "<file or source>", line <line>
```

By default (option `full`), `<file or source>` is the actual name of the script that was executed. The name will be in the form ⟨*family name*⟩`_`⟨*session*⟩`_`⟨*restart*⟩`.`⟨*extension*⟩. For example, an error produced by a `py` command or environment, in the session `mysession`, using the default restart (that is, the default `\restartpythontexsession` treatment), would be reported in `py_mysession_default.py`. The `session` option replaces the full file name with the name of the session, `mysession.py` in this example. The `genericfile` and `genericscript` options replace the file name with `<file>` and `<script>`, respectively.

`pyfuture=none/all/default`
`default:default`

Under Python 2, this determines what is automatically imported from `__future__` for all code. It does not apply to `console` content. `none` imports nothing from `__future__`. `all` imports everything available in Python 2.7 (`absolute_import`, `division`, `print_function`, and `unicode_literals`).

PythonTeX, since the author does not have to worry about any LaTeX commands printed by the package either not being included (if `autoprint` is relied upon, but the user turns it off) or being included twice (if `\string\printpythontex` is used and `autoprint` is enabled). Printing should only be used in the preamble with great care.

`default` imports a default set of features that should be compatible with almost all packages. Everything except `unicode_literals` is imported, since `unicode_literals` can occasionally cause conflicts. Note that imports from `__future__` are also allowed within sessions, so long as they are at the very beginning of the session, as they would have to be in a normal script.

This option has no effect under Python 3.

`pyconfuture=none/all/default`
`default:none`

This is the equivalent of `pyfuture` for Python `console` content. The two options are separate, because in the `console` context it may be desirable to show explicitly all code that is executed.

`upquote=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`

This option determines whether the `upquote` package is loaded. In general, the `upquote` package should be loaded, because it ensures that quotes within verbatim contexts are "upquotes," that is, ' rather than '.

Using `upquote` is important beyond mere presentation. It allows code to be copied directly from the compiled PDF and executed without any errors due to quotes ' being copied as acute accents ´.

`fixlr=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

This option removes "extra" spacing around `\left` and `\right` in math mode. This spacing is sometimes undesirable, especially when typesetting functions such as the trig functions. See the implementation for details. Similar functionality is provided by the `mleftright` package

`keeptemps=`⟨*none*⟩`/all/code/none`
`default:none` ⟨*none*⟩`=all`

When PythonTeX runs, it creates a number of temporary files. By default, none of these are kept. The `none` option keeps no temp files, the `code` option keeps only code temp files (these can be useful for debugging), and the `all` option keeps all temp files (code, stdout and stderr for each code file, etc.). Note that this option does not apply to any user-generated content, since PythonTeX knows very little about that; it only applies to files that PythonTeX automatically creates by itself.

`prettyprinter=pygments/fancyvrb`
`default:pygments`

This allows the user to determine at the document level whether code is typeset using Pygments or `fancyvrb`.

The package-level option can be overridden for individual command and environment families, using the `\setpythontexprettyprinter` command. Overriding is never automatic and should generally be avoided, since using Pygments to highlight only some content results in an inconsistent style. Keep in mind that Pygment's `text` lexer and/or `bw` style can be used when content needs little or no syntax highlighting.

`prettyprintinline=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`

This determines whether inline content is pretty printed. If it is turned off, inline content is typeset with `fancyvrb`.

`pygments=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`

This allows the user to determine at the document level whether code is typeset using Pygments rather than `fancyvrb`. It is an alias for `prettyprinter=pygments`.

`pyginline=`⟨*none*⟩`/true/false`
`default:true` ⟨*none*⟩`=true`

This option governs whether inline code, not just code in environments, is highlighted when Pygments highlighting is in use. When Pygments is in use, it will highlight everything by default.

It is an alias for `prettyprintinline`.

`pyglexer=`⟨*pygments lexer*⟩
`default:`⟨*none*⟩

This allows a Pygments lexer to be set at the document level. In general, this option should **not** be used. It overrides the default lexer for all commands and environments, for both PythonTeX and Pygments content, and this is usually not desirable. It should be useful primarily when all content uses the same lexer, and multiple lexers are compatible with the content.

`pygopt={`⟨*pygments options*⟩`}`
`default:`⟨*none*⟩

This allows Pygments options to be set at the document level. The options must be enclosed in curly braces `{}`. Currently, three options may be passed in this manner: `style=`⟨*style name*⟩, which sets the formatting style; `texcomments`, which allows LaTeX in code comments to be rendered; and `mathescape`, which allows LaTeX math mode (`$...$`) in comments. The `texcomments` and `mathescape` options may be used with an argument (for example, `texcomments=true/false`); if an argument is not supplied, `true` is assumed. Example: `pygopt={style=colorful, texcomments=true, mathescape=false}`.

Pygments options for individual command and environment families may be set with the `\setpythontexpygopt` macro; for Pygments content, there is `\setpygmentspygopt`. These individual settings are always overridden by the package option.

`fvextfile=`⟨*none*⟩`/`⟨*integer*⟩
`default:`∞ ⟨*none*⟩`=25`

This option speeds the typesetting of long blocks of code that are created on the Python side. This includes content highlighted using Pygments and the `console` environment. Typesetting speed is increased at the expense of creating additional external files (in the PythonTeX directory). The ⟨*integer*⟩ determines the number of lines of code at which the system starts using multiple external files, rather than a single external file. See the implementation for the technical details; basically, an external file is used rather than `fancyvrb`'s `SaveVerbatim`, which becomes increasingly inefficient as the length of the saved verbatim content grows. In most situations, this option should not be needed, or should be fine with the default value or similar "small" integers.

`pyconbanner=none/standard/default/pyversion`
`default:none`

This option governs the appearance (or disappearance) of a banner at the beginning of Python console environments. (A banner only appears in the first environment within each session.) The options `none` (no banner), `standard` (standard Python banner), `default` (default banner for Python's `code` module, standard banner plus interactive console class name), and `pyversion` (banner in the form `Python x.y.z`) are accepted.

`pyconfilename=stdin/console`
`default:stdin`

This governs the form of the filename that appears in error messages in Python console environments. Python errors messages have a form such as the following:

```
>>> z = 1 + 34 +
  File "<name>", line 1
    z = 1 + 34 +
                ^
SyntaxError: invalid syntax
```

The `stdin` option replaces `<name>` with `<stdin>`, as it appears in a standard Python interactive session. The `console` option uses `<console>` instead, which

is the default setting for the Python `code` module used by PythonTeX to create Python console environments.

`depythontex=`⟨*none*⟩`/true/false`
`default:false` ⟨*none*⟩`=true`

This option is used to create a version of the LaTeX document that does not require the PythonTeX package. When invoked, it creates an auxiliary file called `<filename>.depytx`. The script `depythontex.py` uses the original document and this auxiliary file to create a new document in which all PythonTeX commands and environments have been replaced by typeset code and code output. For additional information on `depythontex`, see Section 5.

4.2 Commands and environments

PythonTeX provides four types of commands for use with inline code and three environments for use with multiple lines of code, plus special commands and environments for `console` content. All commands and environments are named using a base name and a command- or environment-specific suffix. A complete set of commands and environments with the same base name constitutes a **command and environment family**. In what follows, the different commands and environments are described using the `py` base name (the `py` family) as an example.

Most commands and environments cannot be used in the preamble, because they typeset material and that is not possible in the preamble. The one exception is the `code` command and environment. These can be used to enter code, but need not typeset anything. This allows you to collect your PythonTeX code in the preamble, if you wish, or even use PythonTeX in package writing. Note that the package option `autoprint` is never active in the preamble, so even if a `code` command or environment did print in the preamble, printed content would never be inputted unless `\printpythontex` or `\stdoutpythontex` were used.

All commands and environments take a session name as an optional argument. The session name determines the session in which the code is executed. This allows code to be executed in multiple independent sessions, increasing speed (sessions run in parallel) and preventing naming conflicts. If a session is not specified, then the `default` session is used. Session names should use the characters a-z, A-Z, 0-9, the hyphen, and the underscore. All characters used **must** be valid in file names, since session names are used to create temporary files. The colon is also allowed, but it is replaced with a hyphen internally, so the sessions `code:1` and `code-1` are identical.

In addition, all environments take `fancyvrb` settings as a second, optional argument. See the fancyvrb documentation for an explanation of accepted settings. This second optional argument **must** be preceeded by the first optional argument (session name). If a named session is not desired, the optional argument can be left empty (`default` session), but the square brackets `[]` must be present so that the second optional argument may be correctly identified:

`\begin{`⟨*environment*⟩`}[][`⟨*fancyvrb settings*⟩`]`

4.2.1 Inline commands

Inline commands are suitable for single lines of code that need to be executed within the body of a paragraph or within a larger body of text. The commands use arbitrary code delimiters (like `\verb` does), which allows the code to contain

arbitrary characters. Note that this is only guaranteed to work properly when the inline commands are **not** inside other macros. If an inline command is used within another macro, the code will be read by the external macro before PythonTeX can read the special code characters (that is, LaTeX will tokenize the code). The inline commands can work properly within other macros, but it is best to stick with curly braces for delimiters in this case and you may have trouble with the hash `#` and percent `%` characters.

`\py[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command is used for including variable values or other content that can be converted to a string. It is an alternative to including content via the `print` statement/function within other commands/environments.

The `\py` command sends ⟨*code*⟩ to Python, and Python returns a string representation of ⟨*code*⟩. ⟨*opening delim*⟩ and ⟨*closing delim*⟩ 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 ⟨*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 is returned to LaTeX and included as `2`. 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 represantation 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.[17]

The text returned by Python must be valid LaTeX code. Verbatim and other special content is allowed. The primary reasons for using `\py` rather than `print` are (1) `\py` is more compact and (2) `print` requires an external file to be created for every command or environment in which it is used, while `\py` and equivalents for other families share a single external file. Thus, use of `\py` minimizes the creation of external files, which is a key design goal for PythonTeX.[18] The main reason for using `print` rather than `\py` is if you need to include a very large amount of material; `print`'s use of external files won't use up TeX's memory, and may give noticeably better performance once the material is sufficiently long.

`\pyc[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command is used for executing but not typesetting ⟨*code*⟩. The suffix `c` is an abbreviation of `code`. If the `print` statement/function is used within ⟨*code*⟩, printed content will be included automatically so long as the package `autoprint` option is set to true (the default setting).

`\pys[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

[17] It would be simple to allow any code within `\py`, including assignment, by using a `try/except` statement. In this way, the functionality of `\py` and `\pyc` could be merged. While that would be simpler to use, it also has serious drawbacks. If `\py` is not exclusively used to typeset string representations of ⟨*code*⟩, then it is no longer possible on the LaTeX side to determine whether a command should return a string. Thus, it is harder to determine, from within a TeX editor, whether `pythontex.py` needs to be run; warnings for missing Python content could not be issued, because the system wouldn't know (on the LaTeX side) whether content was indeed missing.

[18] For `\py`, the text returned by Python is stored in macros and thus must be valid LaTeX code, because LaTeX interprets the returned content. The use of macros for storing returned content means that an external file need not be created for each use of `\py`. Rather, all macros created by `\py` and equivalent commands from other families are stored in a single file that is inputted. Note that even though the content is stored in macros, verbatim content is allowed, through the use of special macro definitions combined with `\scantokens`.

This command performs variable and expression substitution, or string interpolation, on ⟨*code*⟩. Fields of the form `!{`⟨*expr*⟩`}` in ⟨*code*⟩ are replaced with the evaluated and printed output of ⟨*expr*⟩. Then the modified ⟨*code*⟩ is inserted into the document and interpreted as LaTeX. The suffix `s` is an abbreviation of `sub`, for "substitute."

This command is useful for inserting Python-generated content in contexts where the normal `\py` and `\pyc` would not function or would be inconvenient due to the restrictions imposed by LaTeX. Since Python processes ⟨*code*⟩ and performs substitutions *before* the result is passed to LaTeX, substitution fields may be anywhere, including within parts of ⟨*code*⟩ that will become LaTeX comments.

Literal exclamation points `!` in ⟨*code*⟩ only need to be escaped when they immediately precede an opening curly brace `{`, or when they precede exclamation points that precede a brace. Escaping is performed by doubling. Thus, `!!{` would indicate a literal exclamation point followed by a literal curly brace (`!{`), not the beginning of a substitution field. And `!!!{` would indicate a literal exclamation point (`!!`) followed by a substitution field (`!{...}`). Because curly braces `{}` only have the meaning of field delimiters when immediately following a non-escaped exclamation point, curly braces never need to be escaped.

The rules for delimiting ⟨*expr*⟩ depend on the number of curly braces used.

1 pair If ⟨*expr*⟩ is delimited by a single pair of braces, then ⟨*expr*⟩ may contain curly braces so long as the braces only appear in matched pairs and are nested no more than 5 levels deep. This is essentially the same as standard LaTeX tokenization rules except for the nesting limit.

2–6 pairs If ⟨*expr*⟩ is delimited by 2–6 immediately adjacent curly braces (`!{{...}}` to `!{{{{{{...}}}}}}`), then ⟨*expr*⟩ may contain any combination of braces, paired or unpaired, so long as the longest sequence of identical brace characters is shorter than the delimiters. Thus, `!{{...}}` can only contain single braces `{` and `}` (paired or unpaired); `!{{{...}}}` can contain `{`, `}`, `{{`, or `}}`; and so forth.

In both cases, anything more than five identical, immediately adjacent braces will always trigger an error. If a greater level of nesting is needed, then a function should be created within a `pycode` environment and afterward used to assemble the desired result.

Curly braces used for delimiting ⟨*expr*⟩ must not be immediately adjacent to braces that are part of ⟨*expr*⟩, because it would be impossible to distinguish them in the general case. If ⟨*expr*⟩ begins/ends with a literal curly brace, the brace should be preceded/followed by a space or other whitespace character; leading and trailing whitespace in ⟨*expr*⟩ is stripped, so this will not affect the output.

Besides braces, ⟨*expr*⟩ may contain any character except for literal newlines. In some cases, it may be appropriate to represent newlines in escaped form (`\n`). In other cases, it will be more appropriate to perform most calculations within a preceding `pycode` environment, and then access them via a variable or function call.

Because ⟨*expr*⟩ is evaluated and printed, it must be suitable for insertion in a `print()` function (or the equivalent, for languages besides Python). If string conversion as performed by `print()` is not desirable, then commands for explicit string conversion should be used.

`\pyv[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command is used for typesetting but not executing ⟨*code*⟩. The suffix `v` is an abbreviation for `verb`.

`\pyb[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command both executes and typesets ⟨*code*⟩. Since it is unlikely that the user would wish to typeset code and then **immediately** include any output of the code, printed content is **not** automatically included, even when the package `autoprint` option is set to true. Rather, any printed content is included at a user-designated location via the `\printpythontex` or `\stdoutpythontex` macros.

4.2.2 Environments

`pycode [`⟨*session*⟩`][`⟨*fancyvrb settings*⟩`]`

(*env.*) This environment encloses code that is executed but not typeset. The second optional argument ⟨*fancyvrb settings*⟩ is irrelevant since nothing is typeset, but it is accepted to maintain parallelism with the `verbatim` and `block` environments. If the `print` statement/function is used within the environment, printed content will be included automatically so long as the package `autoprint` option is set to true (the default setting).

`pysub [`⟨*session*⟩`][`⟨*fancyvrb settings*⟩`]`

(*env.*) This environment performs variable and expression substitution, or string interpolation, on the enclosed code. Fields of the form `!{`⟨*expr*⟩`}` in ⟨*code*⟩ are replaced with the evaluated and printed output of ⟨*expr*⟩. See the description of the `\pys` command for details about substitution and the substitution field syntax.

`pyverbatim [`⟨*session*⟩`][`⟨*fancyvrb settings*⟩`]`

(*env.*) This environment encloses code that is typeset but not executed.

`pyblock [`⟨*session*⟩`][`⟨*fancyvrb settings*⟩`]`

(*env.*) This environment encloses code that is both executed and typeset. Since it is unlikely that the user would wish to typeset code and then **immediately** print any output of the code, printed content is **not** automatically included, even when the package `autoprint` option is set to true. Rather, any printed content is included at a user-designated location via the `\printpythontex` or `\stdoutpythontex` macros.

4.2.3 Console command and environment families

So far, we have considered the `py` command and environment family. PythonTEX also provides families for `console` content. These emulate the behavior of a Python interactive console. In what follows, the `pycon` family is described

The `pycon` family includes a `\pyconv` and `pyconverbatim` that typeset a console session pasted from an interpreter. It also includes a `\pyconc` and `pyconcode` that execute code but typeset nothing. These should be used with care, since it may often be advisable to show all executed code when working with an interactive console.

The `pycon` family also includes a special environment and command.

`pyconsole [`⟨*session*⟩`][`⟨*fancyvrb settings*⟩`]`

(*env.*) This environment treats its contents as a series of commands passed to an interactive Python console. Python's `code` module is used to intersperse the commands with their output, to emulate an interactive Python interpreter.

When a multi-line command is entered (for example, a function definition), a blank line after the last line of the command may be necessary.

For example,

```
a = 1
b = 2
a + b
```

produces

```
>>> a = 1
>>> b = 2
>>> a + b
3
```

`\pycon[`⟨*session*⟩`]`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command executes ⟨*code*⟩ using the emulated interpreter, and brings the output back into the document, **discarding the input**. The output is typeset verbatim (since it will not in general be valid LaTeX), with the same font used for the `pyconsole` environment.

For example, `\pycon{a + b}` would create `3`.

This command is primarily for use in referencing console variable values.

Notice that there is **not** a command or environment for `console` content that parallels the `block` command and environment. That is, there is not a command or environment that both typesets and executes code in the console, but does not show the output. This is intentional. In most cases, if you are going to use the console, you should use it consistently, showing input and output together.

4.2.4 Default families

By default, three command and environment families are defined, with three corresponding console families.

- Python
  - Base name `py`: `\py`, `\pyc`, `\pyv`, `\pyb`, `pycode`, `pyverbatim`, `pyblock`.
  - Base name `pycon`: `\pycon`, `\pyconc`, `\pyconv`, `pyconsole`, `pyconcode`, `pyconverbatim`.
  - Imports: None.
- Python + pylab (matplotlib module)
  - Base name `pylab`: `\pylab`, `\pylabc`, `\pylabv`, `\pylabb`, `pylabcode`, `pylabverbatim`, `pylabblock`.
  - Base name `pylabcon`: `\pylabcon`, `\pylabconc`, `\pylabconv`, `pylabconsole`, `pylabconcode`, `pylabconverbatim`.
  - Imports: matplotlib's `pylab` module, which provides access to much of matplotlib and NumPy within a single namespace. pylab content is brought in via `from pylab import *`.
  - Additional notes: matplotlib added a `pgf` backend in version 1.2. You will probably want to use this for creating most plots. However, this is not currently configured automatically because many users will want to customize font, TeX engine, and other settings. Using TeX to create plots also introduces a performance penalty.

- Python + SymPy
  - Base name `sympy`: `\sympy`, `\sympyc`, `\sympyv`, `\sympyb`, `sympycode`, `sympyverbatim`, `sympyblock`, `sympyconsole`.
  - Base name `sympycon`: `\sympycon`, `\sympyconc`, `\sympyconv`, `sympyconsole`, `sympyconcode`, `sympyconverbatim`.
  - Imports: SymPy via `from sympy import *`.
  - Additional notes: By default, content brought in via `\sympy` is formatted using a context-sensitive interface to SymPy's `LatexPrinter` class, described below.

Under Python 2.7, all non-`console` families import `absolute_import`, `division`, and `print_function` from `__future__` by default. This may be changed using the package option `pyfuture`. There is an equivalent `pyconfuture` for `console` families. Keep in mind that importing `unicode_literals` from `__future__` may break compatibility with some packages; this is why it is not imported by default. Imports from `__future__` are also possible without using the `pyfuture` option. You may use the `\pythontexcustomc` command or `pythontexcustomcode` environment (described below), or simply enter the import commands immediately at the beginning of a session.

4.2.5 Custom code

You may wish to customize the behavior of one or more families within a document by adding custom code to the beginning and end of each session. The custom code command and environment make this possible. While the custom code command and environment work with `console` content, most of the discussion below is geared toward the non-`console` case.

If you wish to share these customizations among several documents, you can create your own document class or package containing custom code commands and environments.

While custom code can be added anywhere in a document, it is probably best for organizational reasons to add it in the preamble or near the beginning of the document.

Note that custom code is executed, but never typeset. Only code that is actually entered within a `block` (or `verbatim`) command or environment is ever typeset. This means that you should be careful about how you use custom code. For example, if you are documenting code, you probably want to show absolutely all code that is executed, and in that case using custom code might not be appropriate. If you are using PythonTeX to create figures or automate text, are using many sessions, and require many imports, then custom code could save some typing by centralizing the imports.

Any errors or warnings due to custom code will be correctly synchronized with the document, just like normal errors and warnings. Any errors or warnings will be specifically identified as originating in custom code.

Custom code is not allowed to print or write to stdout. It would be pointless for custom code at the beginning of a session to print, because all printed content would be identical since custom code at the beginning comes before any regular code that might make the output session-specific. In addition, it is not obvious where printed content from custom code would be included, especially for custom

code at the end of a session. Furthermore, custom code may be in the preamble, where nothing can be typeset.

If custom code does attempt to print, a warning is raised and the printed content is included in the PythonTeX run summary. This gives you access to the printed content, while not including it in the document. This can be useful in cases where you cannot control whether content prints (for example, if a library automatically prints debugging information).

`\pythontexcustomc[`⟨*position*⟩`]{`⟨*family*⟩`}{`⟨*code*⟩`}`

This macro allows custom code to be added to all sessions within a command and environment family. ⟨*position*⟩ should be either `begin` or `end`; it determines whether the custom code is executed at the beginning or end of each session. By default, custom code is executed at the beginning. ⟨*code*⟩ should be a **single line** of code. For example, `\pythontexcustomc{py}{a=1; b=2}` would create the variables `a` and `b` within all sessions of the `py` family, by adding that line of code at the beginning of each session.

If you need to add more than a single line of custom code, you could use the command multiple times, but it will be more efficient to use the `pythontexcustomcode` environment.

⟨*code*⟩ may contain imports from `__future__`. These must be the first elements in any custom code command or environment, since `__future__` imports are only possible at the very beginning of a Python script and only the very beginning of custom code is checked for them. If imports from `__future__` are present at the beginning of both custom code and the user's code, all imports will work correctly; the presence of the imports in custom code, before user code, does not turn off checking for `__future__` imports at the very beginning of user code. However, it is probably best to keep all `__future__` imports in a single location.

`pythontexcustomcode[`⟨*position*⟩`]{`⟨*family*⟩`}`

(*env.*) This is the environment equivalent of `\pythontexcustomc`. It is used for adding multi-line custom code to a command and environment family. In general, the environment should be preferrred to the command unless only a very small amount of custom code is needed. The environment has the same properties as the command, including the ability to contain imports from `__future__`.

4.2.6 PythonTeX utilities class

All non-`console` families import `pythontex_utils.py`, and create an instance of the PythonTeX utilities class called `pytex`. This provides various utilities for interfacing with LaTeX and PythonTeX.

The utilities class has an attribute `context`. This is a dictionary that can contain contextual information, such as page dimensions, from the TeX side. Values may also be accessed as attributes rather than as dictionary keys. To determine what contextual information is available, and for additional details, see `\setpythontexcontext` under Section 4.5. For working with contextual data, the utilities class provides `pt_to_in()`, `pt_to_cm()`, `pt_to_mm()`, and `pt_to_bp()` methods for converting from TeX points to other units.

The utilities class provides an interface for determining how Python objects are converted into strings in commands such as `\py`. The `pytex.set_formatter(`⟨*formatter*⟩`)` method is used to set the conversion. Two formatters are provided:

- `'str'` converts Python objects to a string, using the `str()` function under Python 3 and the `unicode()` function under Python 2. (The use of

`unicode()` under Python 2 should not cause problems, even if you have not imported `unicode_literals` and are not using unicode strings. All encoding issues should be taken care of automatically by the utilities class.)

- `'sympy_latex'` uses SymPy's `LatexPrinter` class to return context-sensitive LaTeX representations of SymPy objects. Separate `LatexPrinter` settings may be created for the following contexts: `'display'` (displaystyle math), `'text'` (textstyle math), `'script'` (superscripts and subscripts), and `'scriptscript'` (superscripts and subscripts, of superscripts and subscripts). Settings are created via `pytex.set_sympy_latex(`⟨*context*⟩`,`⟨*settings*⟩`)`. For example, `pytex.set_sympy_latex('display', mul_symbol='times')` sets multiplication to use a multiplication symbol ×, but only when math is in displaystyle.[19] See the SymPy documentation for a list of possible settings for the `LatexPrinter` class.

  By default, `'sympy_latex'` only treats matrices differently based on context. Matrices in displaystyle are typeset using `pmatrix`, while those in all other styles are typeset via `smallmatrix` with parentheses.

  The context-sensitive interface to SymPy's `LatexPrinter` is always available via `pytex.sympy_latex()`.

The PythonTeX utilities formatter may be set to a custom function that returns strings, simply by reassigning the `pytex.formatter()` method. For example, define a formatter function `my_func()`, and then `pytex.formatter=my_func` within a `pycode` or `pythontexcustomcode` environment. Any subsequent uses of `\py` will then use `my_func()` to perform formatting.

The utilities class also provides methods for tracking dependencies and created files.

- `pytex.add_dependencies(`⟨*dependencies*⟩`)` This adds ⟨*dependencies*⟩ to a list. If any dependencies in the list change, code is re-executed, even if the code itself has not changed (unless `rerun=never`). Modified dependencies are determined via either modification time (default) or hash; see the package option `hashdependencies` for details. This method is useful for tracking changes in external data and similar files.

  ⟨*dependencies*⟩ should be one or more strings, separated by commas, that are the file names of dependencies. Dependencies should be given with relative paths from the current working directory, with absolute paths, or with paths based on the user's home directory (that is, starting with a tilde `~`). Paths can use a forward slash "`/`" even under Windows. Remember that by default, the working directory is the main document directory. This can be adjusted with `\setpythontexworkingdir`.

  It is possible that a dependency of one session might be modified by another session while PythonTeX runs. The first session might not be executed during the PythonTeX run because its dependency was unmodified at the beginning. A more serious case occurs when the first session does run, but we don't know whether it accessed the dependency before or after the dependency was updated (remember, sessions run in parallel). PythonTeX

[19] Internally, the `'sympy\_latex'` formatter uses the `\mathchoice` macro to return multiple representations of a SymPy object, if needed by the current settings. Then `\mathchoice` typesets the correct representation, based on context.

keeps track of the time at which it started. Any sessions with dependencies that were modified after that time are set to re-execute on the next run. A warning is also issued to indicate that this is the case.

- `pytex.add_created(`⟨*created files*⟩`)` This adds ⟨*created files*⟩ to a list of files created by the current session. Any time the code for the current session is executed, **all of these files will be deleted**. Since this method deletes files, it should be used with care. It is intended for automating cleanup when code is modified. For example, if a figure's name is changed, the old figure would be deleted if its name had been added to the list. By default, PythonTEX can only clean up the temporary files it creates; it knows nothing about user-created files. This method allows user-created files to be specified, and thus added to PythonTEX's automatic cleanup.

  ⟨*created files*⟩ should be one or more strings, separated by commas, that are the file names of created files. Paths should be the same as for `pytex.add_dependencies()`: relative to the working directory, absolute, or based on the user's home directory. Again, paths can use a forward slash "`/`" even under Windows.

  Depending on how you use PythonTEX, this method may not be very beneficial. If all of the output is contained in the default output directory, or a similar directory of your choosing, then manual cleanup may be simple enough that this method is not needed.

These two methods for tracking files may be used manually. However, that is prone to errors, since you will have to modify both a PythonTEX utilities command and an open or save command every time you change a file name or add or remove a dependency or created file. It may be better to redefine your open and save commands, or define new ones, so that a single command opens (or saves) and adds a dependency (or adds a created file). For this reason, the PythonTEX utilities class provides an `open()` method that automatically tracks dependencies and created files.

- `pytex.open(`⟨*file*⟩`,` ⟨*mode*⟩`,` ⟨*args*⟩`,` ⟨*kwargs*⟩`)` This method automatically tracks all files opened for reading (text or binary mode) as dependencies. It automatically tracks all files opened for writing (text or binary mode) as created files. Files opened for updating and appending will raise a warning, since it is not necessarily obvious how these files should be treated. The general form of the custom `open()` function is shown below.

```
def track_open(name, mode='r', *args, **kwargs):
    if mode in ('r', 'rt', 'rb'):
        pytex.add_dependencies(name)
    elif mode in ('w', 'wt', 'wb'):
        pytex.add_created(name)
    else:
        warnings.warn('Unsupported mode {0} for file tracking'.format(mode))
    return open(name, mode, *args, **kwargs)
```

  Unicode note for Python 2: By default, `pytex.open()` call the standard Python 2 `open()`. If more than 3 positional arguments are used, or

if the `encoding` keyword is used, then `io.open()` will be called instead. So if you are working with Unicode, make sure to specify an encoding in `pytex.open()` so that `io.open()` will be used, or manually encode/decode everything.

The utilities class provides a pair of methods, `before()` and `after()`, that are called immediately before and after each chunk of user code. These may be redefined to customize the output of user code. For example, LaTeX commands could be printed at the beginning and end of each command or environment, wrapping any content printed by the user. Or any matplotlib figures that were created in the chunk of code could be detected and saved, and LaTeX commands to include them in the document could be printed. Or stdout could be redirected to a StringIO stream in `before()`, then processed in `after()` before being sent to the original stdout.

The `before()` and `after()` methods may be redefined in any code or block command or environment, using the techniques described below. Once they have been redefined, the new methods will be called for all subsequent commands and environments. When redefining these methods, it is important to realize that the order of redefinition may be important. For example, if the new `before()` and `after()` depend upon one another, then you should call the old `after()` (if it does anything), then redefine the methods, and finally call the new `before()`. This is necessary because `after()` will be called after the end of the command or environment in which redefinition takes place. If `after()` has been redefined so that it depends on the new `before()`, and the new `before()` has not yet been called, errors will likely result. Other methods of dealing with this scenario, involving disabling `before()` and `after()` for a given command or environment, are being considered as potential features for a future release.

When redefining `before()` and `after()`, you may wish to have behavior that is command- or environment-specific. Information about the current command or environment is available in `pytex.command`. The string `i` corresponds to an inline command such as `\py`; `b`, to an inline block such as `\pyb`; `c`, to inline code such as `\pyc`. Similarly, `code` corresponds to a code environment and `block` to a block environment.

You may redefine `before()` and `after()` at the class level. For example,

```
def open(self):
    <body>
PythontexUtils.open = open
```

Or you may redefine these methods as instance attributes that happen to be functions (rather than bound methods). Notice that in this case `self` is not allowed.

```
def open():
    <body>
pytex.open = open
```

Finally, you may redefine these methods as bound methods for the `pytex` instance.

```
def open(self):
    <body>
```

```
import types
pytex.open = types.MethodType(open, pytex)
```

The first and third approaches are necessary if you want to be able to use `self` (for example, to access instance attributes). Notice that `before()` and `after()` take no arguments (except `self` where applicable).

An example of using the `after()` method to automatically save and include all matplotlib figures created in a command or environment is shown below. This example is designed for the `pylab` family of commands, or when `from pylab import *` is used. If `pyplot` is imported as `plt` instead, then `plt.get_fignums()`, `plt.figure()`, `plt.savefig()`, `plt.close()`, etc., would be needed.

```
# Basename for figures that will be created
pytex.basename = '_'.join([pytex.family, pytex.session, pytex.restart])

# Need to keep track of total number of figures in each session
pytex.fignum = 0

# The figure could be included in more sophisticated ways
# For example, a ``figure`` environment could be used
def after():
    for num in get_fignums():
        fname = pytex.basename + '_fig' + str(pytex.fignum) + '.pdf'
        pytex.fignum += 1
        figure(num)
        savefig(fname)
        pytex.add_created(fname)
        close(num)
        print(r'\includegraphics{' + fname + '}')

# In this case, I'm taking the easy approach to redefine ``open()``
pytex.after = after
```

4.2.7 Formatting of typeset code

`\setpythontexfv[`⟨*family*⟩`]{`⟨*fancyvrb settings*⟩`}`

This command sets the `fancyvrb` settings for all command and environment families. Alternatively, if an optional argument ⟨*family*⟩ is supplied, the settings only apply to the family with that base name. The general command will override family-specific settings.

Each time the command is used, it completely overwrites the previous settings. If you only need to change the settings for a few pieces of code, you should use the second optional argument in `block` and `verb` environments.

Note that `\setpythontexfv` and `\setpygmentsfv` are equivalent when they are used without an optional argument; in that case, either may be used to determine the document-wide `fancyvrb` settings, because both use the same underlying macro.

`\setpythontexprettyprinter[`⟨*family*⟩`]{`⟨*printer*⟩`}`

This should generally not be needed. It sets the pretty printing used by the document, or by ⟨*family*⟩ if given. Valid options for ⟨*printer*⟩ are `fancyvrb` and

pygments. The option auto may be given for ⟨*family*⟩, in which case the formatter is inherited from the document-level settings. Using either of the other two options will force ⟨*family*⟩ to use that printer, regardless of the document-level settings. By default, families use auto.

Remember that Pygments has a text lexer and a bw style. These are an alternative to setting the formatter to fancyvrb.

`\setpythontexpyglexer[⟨family⟩]{⟨pygments lexer⟩}`

This allows the Pygments lexer to be set for ⟨*family*⟩. ⟨*pygments lexer*⟩ should use a form of the lexer name that does not involve any special characters. For example, you would want to use the lexer name csharp rather than C#. This will be a consideration primarily when using the Pygments commands and environments to typeset code of an arbitrary language.

If a ⟨*family*⟩ is not specified, the lexer is set for the entire document.

`\setpythontexpygopt[⟨family⟩]{⟨pygments options⟩}`

This allows the Pygments options for ⟨*family*⟩ to be redefined. Note that any previous options are overwritten. The same Pygments options may be passed here as are available via the package pygopt option. Note that for each available option, individual family settings will be overridden by the package-level pygopt settings, if any are given.

If a ⟨*family*⟩ is not specified, the options are set for the entire document.

4.2.8 Access to printed content (stdout) and error messages (stderr)

The macros that allow access to printed content and any additional content written to stdout are provided in two identical forms: one based off of the word print and one based off of stdout. Macro choice depends on user preference. The stdout form provides parallelism with the macros that provide accesss to stderr.

`\printpythontex[⟨mode⟩][⟨options⟩]`
`\stdoutpythontex[⟨mode⟩][⟨options⟩]`

Unless the package option autoprint is true, printed content from code commands and environments will not be automatically included. Even when the autoprint option is turned on, block commands and environments do not automatically include printed content, since we will generally not want printed content immediately after typeset code. This macro brings in any printed content from the **last** command or environment. It is reset after each command/environment, so its scope for accessing particular printed content is very limited. It will return an error if no printed content exists.

⟨*mode*⟩ determines how printed content is handled. It may be raw (interpreted as LaTeX), verb (inline verbatim), or verbatim; raw is the default. Verbatim content is brought in via fancyvrb. ⟨*options*⟩ consists of fancyvrb settings.

`\saveprintpythontex{⟨name⟩}`
`\savestdoutpythontex{⟨name⟩}`
`\useprintpythontex[⟨verbatim options⟩][⟨fancyvrb options⟩]{⟨name⟩}`
`\usestdoutpythontex[⟨verbatim options⟩][⟨fancyvrb options⟩]{⟨name⟩}`

We may wish to be able to access the printed content from a command or environment at any point after the code that prints it, not just before any additional commands or environments are used. In that case, we may save access to

the content under ⟨*name*⟩, and access it later via `\useprintpythontex{`⟨*name*⟩`}`. ⟨*mode*⟩ must be `raw`, `verb`, or `verbatim`. If content is brought in verbatim, then ⟨*fancyvrb options*⟩ are applied.

`\stderrpythontex[`⟨*mode*⟩`][`⟨*fancyvrb options*⟩`]`

This brings in the stderr produced by the last command or environment. It is intended for typesetting incorrect code next to the errors that it produces. By default, stderr is brought in verbatim. ⟨*mode*⟩ may be set to `raw`, `verb`, or `verbatim`. In general, bringing in stderr `raw` should be avoided, since stderr will typically include special characters that will make TEX unhappy.

The line number given in the `stderr` message will correctly align with the line numbering of the typeset code. Note that this only applies to `code` and `block` environments. Inline commands do not have line numbers, and as a result, they **do not** produce stderr content.

By default, the file name given in the message will be in the form

⟨*family name*⟩`_`⟨*session*⟩`_`⟨*group*⟩`.`⟨*extension*⟩

For example, an error produced by a `pycode` environment, in the session `mysession`, using the default group (that is, the default `\restartpythontexsession` treatment), would be reported in `py_mysession_default.py`. The package option `stderrfilename` may be used to change the reported name to the following forms: `mysession.py`, `<file>`, `<script>`.

`\savestderrpythontex{`⟨*name*⟩`}`
`\usestderrpythontex[`⟨*mode*⟩`][`⟨*fancyvrb options*⟩`]{`⟨*name*⟩`}`

Content written to `stderr` may be saved and accessed anywhere later in the document, just as `stdout` content may be. These commands should be used with care. Using Python-generated content at multiple locations within a document may often be appropriate. But an error message will usually be most meaningful in its context, next to the code that produced it.

`\setpythontexautoprint{`⟨*boolean*⟩`}`
`\setpythontexautostdout{`⟨*boolean*⟩`}`

This allows autoprint behavior to be modified at various points within the document. The package-level `autoprint` option is also available for setting autoprint at the document level, but it is overridden by `\setpythontexautoprint`. ⟨*boolean*⟩ should be `true` or `false`.

4.3 Pygments commands and environments

Although PythonTEX's goal is primarily the execution and typesetting of Python code from within LATEX, it also provides access to syntax highlighting for any language supported by Pygments.

`\pygment{`⟨*lexer*⟩`}`⟨*opening delim*⟩⟨*code*⟩⟨*closing delim*⟩

This command typesets ⟨*code*⟩ in a suitable form for inline use within a paragraph, using the specified Pygments ⟨*lexer*⟩. Internally, it uses the same macros as the PythonTEX inline commands. ⟨*opening delim*⟩ and ⟨*closing delim*⟩ may be a pair of any characters except for the space character, or a matched set of curly braces `{}`.

As with the inline commands for code typesetting and execution, there is not an optional argument for `fancyvrb` settings, since almost all of them are not relevant

for inline usage, and the few that might be should probably be used document-wide if at all.

`pygments [⟨fancyvrb settings⟩]{⟨lexer⟩}`
(*env.*) This environment typesets its contents using the specified Pygments ⟨*lexer*⟩ and applying the ⟨*fancyvrb settings*⟩.

`\inputpygments[⟨fancyvrb settings⟩]{⟨lexer⟩}{⟨external file⟩}`
This command brings in the contents of ⟨*external file*⟩, highlights it using ⟨*lexer*⟩, and typesets it using ⟨*fancyvrb settings*⟩.

`\setpygmentsfv[⟨lexer⟩]{⟨fancyvrb settings⟩}`
This command sets the ⟨*fancyvrb settings*⟩ for ⟨*lexer*⟩. If no ⟨*lexer*⟩ is supplied, then it sets document-wide ⟨*fancyvrb settings*⟩. In that case, it is equivalent to `\setpythontexfv{⟨fancyvrb settings⟩}`.

`\setpygmentspygopt[⟨lexer⟩]{⟨pygments options⟩}`
This sets ⟨*lexer*⟩ to use ⟨*pygments options*⟩. If there is any overlap between ⟨*pygments options*⟩ and the package-level `pygopt`, the package-level options override the lexer-specific options.

If ⟨*lexer*⟩ is not given, options are set for the entire document.

`\setpygmentsprettyprinter{⟨printer⟩}`
This usually should not be needed. It allows the pretty printer for the document to be set; it is equivalent to using `\setpythontexprettyprinter` without an optional argument. Valid options for ⟨*printer*⟩ are `fancyvrb` and `pygments`.

Remember that Pygments has a `text` lexer and a `bw` style. These are an alternative to setting the formatter to use `fancyvrb`.

4.4 General code typesetting

4.4.1 Listings float

`listing` (*env.*)
PythonTEX will create a float environment `listing` for code listings, unless an environment with that name already exists. The `listing` environment is created using the `newfloat` package. Customization is possible through `newfloat`'s `\SetupFloatingEnvironment` command.

`\setpythontexlistingenv{⟨alternate listing environment name⟩}`
In the event that an environment named `listing` already exists for some other purpose, PythonTEX will not override it. Instead, you may set an alternate name for PythonTEX's `listing` environment, via `\setpythontexlistingenv`.

4.4.2 Background colors

PythonTEX uses `fancyvrb` internally to typeset all code. Even code that is highlighted with Pygments is typeset afterwards with `fancyvrb`. Using `fancyvrb`, it is possible to set background colors for individual lines of code, but not for entire blocks of code, using `\FancyVerbFormatLine` (you may also wish to consider the `formatcom` option). For example, the following command puts a green background behind all the characters in each line of code:

```
\renewcommand{\FancyVerbFormatLine}[1]{\colorbox{green}{#1}}
```

If you need a completely solid colored background for an environment, or a highly customizable background, you should consider the `mdframed` package. Wrapping PythonTEX environments with `mdframed` frames works quite well. You can even automatically add a particular style of frame to all instances of an environment using the command

`\surroundwithmdframed[`⟨*frame options*⟩`]{`⟨*environment*⟩`}`

Or you could consider using `etoolbox` to do the same thing with `mdframed` or another framing package of your choice, via `etoolbox`'s `\BeforeBeginEnvironment` and `\AfterEndEnvironment` macros.

4.4.3 Referencing code by line number

It is possible to reference individual lines of code, by line number. If code is typeset using pure `fancyvrb`, then LaTeX labels can be included within comments. The labels will only operate correctly (that is, be treated as LaTeX rather than verbatim content) if `fancyvrb`'s `commandchars` option is used. For example, `commandchars=\\\{\}` makes the backslash and the curly braces function normally **within** `fancyvrb` environments, allowing LaTeX macros to work, including label definitions. Once a label is defined within a code comment, then referencing it will return the code line number.

The disadvantage of the pure `fancyvrb` approach is that by making the backslash and curly braces command characters, we can produce conflicts if the code we are typesetting contains these characters for non-LaTeX purposes. In such a case, it might be possible to make alternate characters command characters, but it would probably be better to use Pygments.

If code is typeset using Pygments (which also ties into `fancyvrb`), then this problem is avoided. The Pygments option `texcomments=true` has Pygments look for LaTeX code only within comments. Possible command character conflicts with the language being typeset are thus eliminated.

Note that when labels are created within comments, the labes themselves will be invisible within the final document but the comment character(s) and any other text within comments will still be visible. For example, the following

```
abc = 123  # An important line of code!\label{lst:important}
```

would appear as

```
abc = 123  # An important line of code!
```

If a comment only contains the `\label` command, then only the comment character `#` would actually be visible in the typeset code. If you are typesetting code for instructional purposes, this may be less than ideal. Unfortunately, Pygments currently does not allow escaping to LaTeX outside of comments (though this feature has been requested). At the same time, by only allowing references within comments, Pygments does force us to create code that would actually run. And in many cases, if a line is important enough to label, it is also important enough for a brief comment.

4.4.4 Beamer compatibility

PythonTEX is compatible with Beamer. Since PythonTEX typesets code as verbatim content, Beamer's `fragile` option must be used for any frame that contains typeset code. Beamer's `fragile` option involves saving frame contents to an external file and bringing it back in. This use of an external file breaks PythonTEX's error line number synchronization, since the error line numbers will correspond to the temporary external file rather than to the actual document.

If you need to typeset code with Beamer, but don't need to use overlays on the slides containing code, you should use the `fragile=singleslide` option. This allows verbatim content to be typeset without using an external file, so PythonTEX's error line syncronization will work correctly.

4.5 Advanced PythonTEX usage

`\setpythontexcontext{`⟨*key-value pairs*⟩`}`

This macro is used for passing contextual information such as page dimensions from the TEX side to the Python/other language side. ⟨*key-value pairs*⟩ is a set of comma-delimited key-value pairs. An evaluated version of ⟨*key-value pairs*⟩ is passed to the programming language, wrapped in quotation marks to become a string. Thus, ⟨*key-value pairs*⟩ should *not* contain quotation marks, backslashes, or other characters that would prevent the evaluated contents from being the body of a normal, quoted string.

As an example, the following would pass the values of `\textwidth` and `\textheight` to the Python side.

```
\setpythontexcontext{textwidth=\the\textwidth, textheight=\the\textheight}
```

Python would receive a string something like `"textwidth=390pt, textheight=592pt"`. This string would be parsed into key-value pairs, and the results stored in the `pytex.context` dictionary. For Python, the keys also become the names of attributes of `pytex.context`. Thus, the values may be accessed on the Python side via `pytex.context['textwidth']`, `pytex.context.textwidth`, etc.

All contextual data is available as strings on the Python/other language side. For convenience, the utilities class provides unit conversion methods for converting from TEX points to inches, centimeters, millimeters, and big (DTP or PostScript) points. These methods take integers, floats, or strings that consist of digits (optionally ending in "pt"), and return floats. For example, `pytex.pt_to_in()`, `pytex.pt_to_cm()`, `pytex.pt_to_mm()`, `pytex.pt_to_bp()`. Keep in mind that the units of TEX points are 1/72.27 of an inch, *not* 1/72 of an inch (which is a bp).

There is also a type system for Python that allows the types of ⟨*values*⟩ to be specified. Any ⟨*value*⟩ beginning with `!!int` will become an integer; with `!!float`, a float; with `!!str`, a string. This notation is borrowed from YAML. For example,

```
\setpythontexcontext{a=!!int 42, b=!!float 42, c=!!str 42}
```

This type system is still under development and is subject to change in the future. Once the system stabilizes, it will be extended to non-Python languages. Comments on the type system are welcome.

The context may only be set in the preamble.

Technical note: Contextual data is cached, so the dictionary (and its attributes, if applicable) is only updated when contextual data changes. This largely eliminates any potential overhead from contextual data.

`\restartpythontexsession{`⟨*counter value(s)*⟩`}`

This macro determines when or if sessions are restarted (or "subdivided"). Whenever ⟨*counter value(s)*⟩ change, the session will be restarted.

By default, each session corresponds to a single code file that is executed. But sometimes it might be convenient if the code from each chapter or section or subsection were to run within its own file, as its own session. For example, we might want each chapter to execute separately, so that changing code within one chapter won't require that all the code from all the other chapters be executed. But we might not want to have to go to the bother and extra typing of defining a new session for every chapter (like `\py[ch1]{`⟨*code*⟩`}`). To do that, we could use `\restartpythontexsession{\thechapter}`. This would cause all sessions to restart whenever the chapter counter changes. If we wanted sessions to restart at each section within a chapter, we would use `\restartpythontexsession{\thechapter`⟨*delim*⟩`\thesection}`. ⟨*delim*⟩ is needed to separate the counter values so that they are not ambiguous (for example, we need to distinguish chapter 11-1 from chapter 1-11). Usually ⟨*delim*⟩ should be a hyphen or an underscore; it must be a character that is valid in file names.

Note that **counter values**, and not counters themselves, must be supplied as the argument. Also note that the command applies to **all** sessions. If it did not, then we would have to keep track of which sessions restarted when, and the lack of uniformity could easily result in errors on the part of the user.

Keep in mind that when a session is restarted, all continuity is lost. It is best not to restart sessions if you need continuity. If you must restart a session, but also need to keep some data, you could save the data before restarting the session and then load the saved data after the restart. This approach should be used with **extreme** caution, since it can result in unanticipated errors due to sessions not staying synchronized.[20]

This command can only be used in the preamble.

`\setpythontexoutputdir{`⟨*output directory*⟩`}`

By default, PythonTeX saves all temporary files and automatically generated content in a directory called `pythontex-files-`⟨*sanitized jobname*⟩, where ⟨*sanitized jobname*⟩ is just `\jobname` with any space characters or asterisks replaced with hyphens. This directory will be created by `pythontex.py`. If we wish to specify another directory (for example, if `\jobname` is long and complex, and there is no danger of two files trying to use the same directory), then we can use the `\setpythontexoutputdir` macro to redefine the output directory.[21]

[20] For example, suppose sessions are restarted based on chapter. `session-ch1` saves a data file, and `session-ch2` loads it and uses it. You write the code, and run PythonTeX. Then you realize that `session-ch1` needs to be modified and make some changes. The next time PythonTeX runs, it will only execute `session-ch1`, since it detects no code changes in `session-ch2`. This means that `session-ch2` is not updated, at least to the extent that it depends on the data from `session-ch1`. Again, saving and loading data between restarted sessions, or just between sessions in general, can produce unexpected behavior. This can be avoided by using the `pytex.add_dependencies()` method for all data that is loaded. It will ensure that all sessions stay in sync.

[21] In the rare event that both `\string\setpythontexoutputdir` is used and `\string\printpythontex` is needed in the preamble, `\string\setpythontexoutputdir` must be

Any slashes in ⟨*output directory*⟩ should be forward slashes "`/`" (even under Windows). Tildes `~` may be used to refer to the user's home directory, including under Windows.

`\setpythontexworkingdir{`⟨*working directory*⟩`}`

The PythonTEX working directory is the current working directory for PythonTEX scripts. This is the directory in which any open or save operations will take place, unless a path is explicitly specified. By default, the working directory is the same as the main document directory. For example, if you are writing `my_file.tex` and save a matplotlib figure with `savefig('my_figure.pdf')`, then `my_figure.pdf` will be created in the same directory as `my_file.tex`. But maybe you have a directory called `plots` in your document root directory. In that case, you could leave the working directory unchanged, and simply specify the relative path to `plots` when saving. Or you could set the working directory to `plots` using `\setpythontexworkingdir{plots}`, so that all content would automatically be saved there.

Any slashes in ⟨*working directory*⟩ should be forward slashes "`/`" (even under Windows). Tildes `~` may be used to refer to the user's home directory, including under Windows.

The working directory is automatically added to Python's `sys.path`, so that code in the working directory there may be imported without a path being specified.

Note that in many use cases, you may be able to use the output directory as the working directory. The `graphicx` package will automatically look for images and figures in the output directory when it is used as the working directory, so long as you do not use the `\graphicspath` command outside the preamble.[22] To use the output directory as the working directory, you may enter the full name of the output directory manually, or use the text "`<outputdir>`" as a shortcut:

```
\setpythontexworkingdir{<outputdir>}
```

It is also possible to change the working directory from within Python code, via `os.chdir()`.

4.6 Working with other programs

4.6.1 `latexmk`

PythonTEX is compatible with `latexmk`. How you configure `latexmk` largely depends on how you are using PythonTEX.

If you are compiling in the same location as the document source (if you are *not* using `-auxdir`, `-outdir`, or `$out_dir`, or alternatively `TEXINPUTS`), and are not using PythonTEX's dependency tracking, then adding a simple rule such as the following to your `.latexmkrc` should usually be sufficient.

```
add_cus_dep('pytxcode', 'tex', 0, 'pythontex');
```

used first, so that `\string\printpythontex` will know where to look for output.

[22] `graphicx` looks for graphics in the document root directory and in the most recent graphics path defined by `\graphicspath`. `\graphicspath` stores the graphics path in `\Ginput@path`, overwriting any previous value. At the end of the preamble, PythonTEX appends the output directory to `\Ginput@path` if the output directory is being used as the working directory. Thus, that directory will always be checked for graphics, so long as `\Ginput@path` is not overwritten by a subsequent use of `\graphicspath`. If you need to use `\graphicspath` within the document, you could consider creating a custom version that redefines `\Ginput@path` with the PythonTEX output directory automatically appended.

```
sub pythontex { return system("pythontex \"$_[0]\""); }
```

This tells `latexmk` that the document (`tex`) depends on the file of code extracted from the document (`pytxcode`).[23] Whenever the document is compiled, the file of code is updated. If `latexmk` detects that the code changed, then it will run PythonTeX. When PythonTeX runs, it will modify at least one file that is brought into the document. `latexmk` will detect this modification, and automatically recompile the document.

If you are compiling to a different directory (using `-auxdir`, `-outdir`, or `$out_dir`, or alternatively `TEXINPUTS`), then the preceding rule may fail due to the different directory configuration. In that case, you should use `\setpythontexoutputdir{.}` so that PythonTeX will store its output in the current default location, rather than in a subdirectory, to ensure that `latexmk` will locate the output files. Since the `tex` source is no longer in the location of the compiled output, you also need a different dependency specification. It is probably simplest to use the `pytxmcr` file that PythonTeX always generates.

```
add_cus_dep('pytxcode','pytxmcr',0,'pythontex');
sub pythontex { return system("pythontex.py \"$_[0]\""); }
```

Note that this configuration should *always* work, but has the disadvantage of requiring that PythonTeX not use a subdirectory to isolate the files it automatically generates.

If you are using PythonTeX's dependency tracking, then you should run PythonTeX once during *every* compile cycle (unless you simply wish to run it manually, as needed). Checking the `pytxcode` for modification is not sufficient, because it does not reflect the state of dependencies. If you are testing for dependency modification using modification time (the default) rather than hashing, this should typically add very little overhead. If PythonTeX detects modified dependencies and actually does execute code, then the `pytxmcr` file will be updated, which will trigger another compile. It is possible to have PythonTeX run after each individual LaTeX run by modifying `latexmk`'s `-latex`, `-pdflatex`, or `-xelatex` options. Ideally, however, PythonTeX would only run once per compile cycle.

The situation is similar if you are using the `rerun=always` setting. The above rules will fail to run PythonTeX on each and every compile; in that situation, you should configure your `.latexmkrc` so that PythonTeX always runs at least once during *every* compile cycle.

5 depythontex

PythonTeX can greatly simplify the creation of documents. At the same time, by introducing dependence on non-LaTeX external tools, it can constrain how these documents are used. For example, many publishers will not accept LaTeX documents that require special packages or need special macros. To address this issue, the package includes a feature called `depythontex` that can convert a PythonTeX document into a plain LaTeX document.

[23] This is a slightly atypical use, if not a "misuse," of `add_cus_dep()`. In the standard usage, the first argument is the extension of a file that is used to create another file with the extension given in the second argument, via the rule named in the fourth argument. In this case, we just want to run the rule whenever files with the first extension are modified. The extension given in the second argument is irrelevant, so long as a file with the document name and that extension exists. Since the `tex` file itself will exist, its extension is a logical choice for the second argument.

5.1 Preparing a document that will be converted

The conversion process should work flawlessly in most cases, with no special formatting required.

For best results, keep the following in mind.

- The PythonTeX package should have its own `\usepackage`.
- Currently, `depythontex` only supports the standard PythonTeX commands and environments. Support for user-defined commands and environments that incorporate PythonTeX is planned for a future release.
- If you need to insert content from Python in inline contexts, it is best to use `\py` or an equivalent command. If you use `print`, either directly (for example, from within `\pyc`) or via `\printpythontex`, make sure that the spacing following the printed content is correct. You may need to print an `\endinput` or `%` at the end of your content to prevent an extra trailing space. `depythontex` will attempt to reproduce the spacing of the original document, even if it is not ideal. See Section 5.3 for additional details.
- Some LaTeX environments, such as the `verbatim` environment from the `verbatim` package and the `Verbatim` environment from `fancyvrb`, do not allow text to follow the `\end{`⟨*environment*⟩`}`. If you bring Python-generated content that ends with one of these environments into your document, using `print` or `\py`, make sure that the end-of-environment command is followed by a newline. For example, if you are assembling a `Verbatim` environment to bring in, the last line should be the string

  ```
  '\\end{Verbatim}\n'
  ```

  Even if you neglect a final newline, `depythontex` will still function correctly in most cases. Whenever Python-generated content does not end with a newline, `depythontex` usually inserts one and gobbles spaces that follow the environment. This preserves the correct spacing while avoiding any issues produced by an end-of-environment command. But in some cases, `depythontex` cannot do this. For example, if `\py` is used to bring in a `Verbatim` environment, and there is text immediately after the `\py`, without any intervening space, `depythontex` cannot substitute a newline for spaces, because there are none. Because of the way that `print` and `\py` content is brought in, everything may still work correctly in the original PythonTeX document. But it would fail in the `depythontex` output.
- Do not create PythonTeX commands or environments on the Python side and `print` or otherwise bring them in. That is too many levels of complexity!
- `depythontex` is only designed to replace PythonTeX commands and environments that are actually in the main document file. Do not bring in anything that contains PythonTeX commands or environments via `\input`, `\include`, or `\usepackage`. The only exception is PythonTeX commands and environments that do not typeset anything (for example, `code` environments that don't print). If these are brought in via a package or external file, the command `\DepythontexOff` must come before them, and they must be

followed by the command `\DepythontexOn`. Basically, `depythontex` must be disabled for commands and environments brought in via external files. This works so long as the commands and environments only provide code and settings, rather than any typeset content.

Tools for automatically removing the `\usepackage` for packages that contain PythonTeX commands will be added soon; for now, these `\usepackage`'s must be removed manually in the `depythontex` output.

- Keep in mind that the file produced by `depythontex` will need to include any graphics that you create with PythonTeX. Make sure any graphics are saved in a location where they are easily accessible.

5.2 Removing PythonTeX dependence

Converting a document requires three steps.

1. Turn on the package option `depythontex`. Then compile the document, run `pythontex.py`, and compile the document again. Depending on the document, additional compiles may be necessary (for example, to resolve references). Any syntax highlighting will be turned off automatically during this process, to remove dependence on Pygments.

   During compilation, an auxiliary file called ⟨*jobname*⟩`.depytx` is created. This file contains information about the location of the PythonTeX commands and environments that need to be replaced, and about the content with which they are to be replaced.

2. Run the `depythontex.py` script. This takes the following arguments.

   - `--encoding` This is the encoding of the LaTeX file and all related files. If an encoding is not specified, UTF-8 is assumed.
   - `--overwrite` This turns off the user prompt in the event that a file already exists with the output name, making overwriting automatic.
   - `--listing` This option specifies the commands and environments that are used for any typeset code. This can be `verbatim`, `fancyvrb`, `listings`, `minted`, or `pythontex`.[24] `verbatim` is used by default. An appropriate `\usepackage` command is automatically added to the output document's preamble.

     When code is typeset with any option other than `verbatim`, listing line numbering from the original document will be preserved. When code is typeset with any option other than `verbatim` and `fancyvrb`, syntax highlighting will also be preserved. The only exception is when `listings` is used, and `listings`'s language name does not correspond to Pygments' lexer name. In this case, you should use the `--lexer-dict` option to specify how the Pygments lexer is to be translated into a `listings` language.

[24] The `pythontex` option is included for completeness. In most cases, you would probably use `depythontex` to remove all dependence on PythonTeX. But sometimes it might be useful to remove all Python code while still using PythonTeX for syntax highlighting.

- `--lexer-dict` This option is used to specify how Pygments lexers are converted to `listings` languages, when the two do not have the same name. It takes a comma-separated list of the form

  ```
  "<Pygments lexer>:<listings language>, ..., ... "
  ```

  A Python-style dict will also be accepted.
- `--preamble` This option allows additional commands to be added to the output document's preamble. This is useful when you want the output document to load a package that was automatically loaded by PythonTeX, such as `upquote`.
- `--graphicspath` This option adds the `outputdir` to any existing graphics path defined by `\graphicspath`, or adds a `\graphicspath` command if one does not already exist. This causes the `depythontex` document to automatically look in the `outputdir` for graphics. Only use this option if you want to continue using the `outputdir` with the `depythontex` document. Graphics are further discussed below.
- `-o --output` The name of the output file. If no name is given, the converted file is written to `stdout`.
- `TEXNAME` The name of the LaTeX file whose PythonTeX dependence is to be removed.

3. Compile the `depythontex` file, and compare it to the original.

   The original and `depythontex` files should be nearly identical. All Python-generated content is substituted directly, so it should be unchanged. Usually, any differences will be due to changes in the way that code is typeset. For example, by default all code in the `depythontex` file is typeset with `\verb` and `verbatim`. But `\verb` is more fragile than the inline PythonTeX commands (it isn't allowed inside other commands), and `verbatim` does not support line numbering or syntax highlighting.

   Remember that the `depythontex` file will need to include any graphics created by PythonTeX. By default, these are saved in the document root directory. They may be in other locations if you have set a non-default `workingdir` or have specified a path when saving graphics. Depending on your needs and configuration, you may wish to copy the graphics into a new location or specify their location via `\graphicspath`. If you are using the `outputdir` as the `workingdir`, you can run `depythontex` with the `--graphicspath` option, which will add the `outputdir` to any existing usage of `\graphicspath`, or add a `\graphicspath` command if one does not already exist.[25]

   Depending on your needs, you may wish to customize `depythontex.py`. The actual substitutions are performed in a few functions that are defined at the beginning of the script.

[25] Keep in mind that any time `\graphicspath` is used, it overwrites any previously specified path. If your document is using `\graphicspath` at multiple points in the preamble, or using it anywhere outside the preamble, then the `--graphicspath` option will fail due to the path being overwritten.

5.3 Technical details

The `depythontex` process should go smoothly under most circumstances, and the document produced usually should not need manual tweaking. There are a few technical details that may be of interest.

- Content that is printed (actually printed, not from a command like `\py`) is always followed by a space when included as LaTeX code rather than as verbatim. Usually this is only noticeable when the content is used inline, adjacent to other text. In such cases, you need to make sure that the spacing is correct in your original document, and need to be aware of how `depythontex` handles the conversion.

  This spacing behavior is due to LaTeX's `\input`. When the file of printed content is brought in via `\input`, LaTeX removes any newline characters (`\n`, `\r`, or `\r\n`) at the end of each line, and adds a space at the end of each line (even if there wasn't a newline character). Thus, when the printed content is brought in, a space is added to its end. Since this space is within the `\input`'s curly braces `{}`, it is not combined with any following spaces in the LaTeX document to make a single space. Rather, if the printed content is followed by one or more spaces, two spaces will result; and if it is followed immediately by text, there will be a single space before the text.

  The space added by `\input` is often invisible, and even when it is not, it is sometimes desirable.[26] But this space can be an issue in some inline contexts. The simplest solution is to use a command like `\py` to bring in content inline.

  If a command like `\py` is not practical for some reason, there are at least three ways to deal with the space introduced by `\input`: by printing `\endinput` at the end of the printed content (ending the content before the final space), by printing `%` at the end of the printed content (commenting out the final space), or by using `\unskip` after the printed content (eating preceding spaces). `depythontex` will work with all three approaches, but only under a limited range of circumstances. In summary, `depythontex` works with `\endinput` and `%` only if they are the very last thing printed (before a final newline), and works with a following `\unskip`.[27]

  - `\endinput` cannot be left in the printed content that is substituted into the new document, because it would cause the new document to end immediately. `depythontex` checks the very end of printed content for `\endinput`, and removes it if it is there before substituting the content. The terminating `\endinput` is only removed if it is not a string, `\string\endinput`.

    If `\endinput` is anywhere else in the printed content, and it is not immediately preceded by `\string`, `depythontex` issues a warning.

[26] For example, `\string\printpythontex` behaves as a normal command, and gobbles following spaces, but the space from `\string\input` puts a space back. So you often get the space you want in inline contexts.

[27] It would be possible to make `depythontex` work with `\endinput` and `%` anywhere, not just at the very end of printed content. But doing so would require a lot of additional parsing, especially for `\endinput`, to be absolutely sure that we found an actual command rather than a string. Furthermore, there is no reason that there should be any content after an `\endinput` or `%`, since such content would never be included in the document. Indeed, the current approach prevents any printed content from accidentally being eliminated in this manner.

- A terminating `%` cannot be left in the printed content that is substituted into the new document, because it would comment out any text in the remainder of the line into which it is substituted (in `\input`, its effect is limited to the print file). `depythontex` checks the very end of printed content for `%`, and removes it if it is there before substituting the content. `depythontex` only removes the terminating `%` if it is not a literal character `\%` or `\string%`.

  `depythontex` checks the last line of printed content for other `%` characters, and issues a warning if there are any `%` characters that are not part of `\%` or `\string%`.

- A following `\unskip` could be left in the new document, since it would not produce incorrect spacing. But it would be undesirable, since it was only there in the first place because of the way that PythonTeX was used. `depythontex` checks for `\unskip`, and if it is found, attempts to correct the spacing and remove the `\unskip`. This removal process is only possible if `\unskip` immediately follows a command (otherwise, it wouldn't work anyway) or is on the line immediately after the end of an environment.

  If `depythontex` finds `\unskip` following printed content, but cannot replace it (it doesn't immediately follow a command, or isn't on the line immediately after the end of an environment), a warning is issued. It is possible that the `\unskip` is not correctly positioned, and even if it produces the correct spacing, the user should know that due to its location it will survive in the converted document.

If one of the above approaches is not used to eliminate the space introduced by the final newline in printed content, `depythontex` still makes sure that the spacing in the new document matches that of the original document, even if that means forcing a double space. In the majority of cases, `depythontex` can create the correct spacing using actual spaces and newlines. But in a few instances, it will include a `\space{}` to ensure a double space that matches the original document. In those situations, a warning is issued in case the spacing was not intentional.

- Strings such as `\\}`, `\\{`, and `\string` can occur in PythonTeX content that is being replaced. It is possible that they might decrease performance somewhat in larger or more complex documents.

  PythonTeX commands for entering code allow the code to be delimited with either matched braces `{}` or with a repeated character such as `#` (as in `\verb`). Any verbatim code delimited by braces cannot **contain** any braces **unless** they are paired. So it is easy for `depythontex.py` to find the end of the delimited code.

  However, `depythontex.py` must also replace PythonTeX commands that take a normal, non-verbatim argument delimited by braces (for example, the various `\setpythontex...` commands). Finding the closing brace for these commands is usually straightforward, but it can be tricky because the argument might contain a literal brace such as `\}` or `\string}`. `depythontex.py` automatically accounts for `\}`. If it detects `\string`, it also accounts for it, but doing so requires more intense parsing. Similarly, `\\}` requires extra

parsing, because depending on what comes before it, the first backslash \ could be literal (for example, if preceded by `\string`), or the two backslashes \\ could go together to indicate a new line.

6 LaTeX programming with PythonTeX

This section will be expanded in the future. For now, it offers a brief summary.

6.1 Macro programming with PythonTeX

In many situations, you can use PythonTeX commands inside macro definitions without any special consideration. For example, consider the following macro, for calculating powers.

```
\newcommand{\pow}[2]{\py{#1**#2}}
```

Once this is defined, we can calculate `2**8` via `\pow{2}{8}`: ??. Similarly, we can reverse a string.

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

Now we can use `\reverse{``This is some text!''}`: ??.

Such approaches will break down when some special LaTeX characters such as percent % and hash # must be passed as arguments. In such cases, the arguments need to be captured verbatim. The `xparse` and `newverbs` packages provide commands for creating macros that capture verbatim arguments. You could also consult the PythonTeX implementation, particularly the implementation of the inline commands. In either case, you may need to learn about TeX's catcodes and tokenization, if you aren't already familiar with them.

Of course, there are many cases where macros don't need arguments. Here is code for creating a macro that generates random polynomials.

```
\begin{sympycode}
from sympy.stats import DiscreteUniform, sample
x = Symbol('x')
a = DiscreteUniform('a', range(-10, 11))
b = DiscreteUniform('b', range(-10, 11))
c = DiscreteUniform('c', range(-10, 11))
def randquad():
    return Eq(sample(a)*x**2 + sample(b)*x + sample(c))
\end{sympycode}
\newcommand\randquad{\sympy{randquad()}}
```

If you are considering writing macros that involve PythonTeX, you should keep a few things in mind.

- Do you really need to use PythonTeX? If another package already provides the functionality you need, it may be simpler to use an existing tool, particularly if you are working with special characters and thus need to capture verbatim arguments.

- A feature called `depythontex` has recently been added. It creates a copy of the original LaTeX document in which all PythonTeX commands and environments are replaced by their output, so that the new document does not depend on PythonTeX at all. This is primarily of interest for publication, since publishers tend not to like special packages or macros. `depythontex` does not yet support custom user commands. So if you decide to create custom macros now, and expect to need `depythontex`, you should expect to have to edit your macros before they will work with `depythontex`.

6.2 Package writing with PythonTeX

As of v0.10beta, the custom code command and environment, and the regular code command and environment, work in the preamble. This means that it is now possible to write packages that incorporate PythonTeX! At this point, packages are probably a good way to keep track of custom code that you use frequently, and maybe some macros that use PythonTeX.

However, you are encouraged not to develop a huge mathematical or scientific package for LaTeX using PythonTeX. At least not yet! As discussed above, `depythontex` will bring changes to macro programming involving PythonTeX. So have fun writing packages if you want—but keep in mind that PythonTeX will keep changing, and some things that are difficult now may be very simple in the future.

7 Support for additional languages

Details about adding support for additional languages are in Section 7.10. This section begins with a brief overview of supported languages and available features.

Languages beyond Python are typically not be enabled by default to prevent potential macro naming conflicts with other packages. Languages are enabled via the `usefamily` package option (Section 4). For example,

```
\usepackage[usefamily=ruby]{pythontex}
```

Usually at least two possible base names for commands and environments will be provided for each language. Typically these will be the name of the language and the language's file extension. For example, Ruby has the `ruby` and `rb` base names. You can choose which base name to use for creating a family of commands and environments based on personal preference and potential naming conflicts.

7.1 Ruby

Support for Ruby was added in v0.12. Ruby support should be almost at the same level as that for Python.

The utilities class is called `RubyTeXUtils`, and the class instance is `rbtex`. The variables and methods are the same as those for Python (Section 4.2.6), except that there is not currently a `set_formatter()` or an `open()` method. (The Python utilities class has the special SymPy formatter, but there aren't yet any specialized formatters for Ruby.)

A family of commands and environments for Ruby is not created by default. Two base names are provided for families: `ruby` and `rb`. Preconfigured families

for these names may be created via the `usefamily` package option. Keep in mind that a `ruby` command is defined as part of the Ruby package in the CJK package. I am unaware of a package that provides an `\rb` command.

Ruby exceptions are synchronized with the document, but the line numbering does not always correspond to the Python equivalent. For example, suppose that `\pyc{1+}` is on line 10 of a document. The SyntaxError will then be synchronized with line 10. If `\rubyc{1+}` were on the same line, the resulting error would be synchronized with line 11. This is because Ruby allows addition to continue on subsequent lines of code, so an error is only raised when the next line of code that is executed does not contain a number (there is always template code after user code).

7.2 Julia

Support for Julia was added in v0.12. Julia support should be at almost the same level as that for Python. The format of Julia stderr is somethat different from that of Python and Ruby. This required a modified parsing and synchronization algorithm. The current system is functional but will likely change somewhat in the future.

The utilities "class" is called `JuliaTeXUtils` (it is actually a composite type, very similar to a struct), and the "class" instance is `jltex`. The variables and methods are the same as those for Python (Section 4.2.6), except that there is not currently a `set_formatter()` method or an `open()` method. (The Python utilities class has the special SymPy formatter, but there aren't yet any specialized formatters for Julia.)

A family of commands and environments for Julia is not created by default. Two base names are provided for families: `julia` and `jl`. Preconfigured families for these names may be created via the `usefamily` package option. Keep in mind that Pygments only added Julia support in version 1.6, so you may need to update your Pygments installation, or just change the default lexer.

Julia exceptions are synchronized with the document, but the line numbering does not always correspond to the Python equivalent. This is because Julia allows expressions to be continued on subsequent lines in ways that Python does not.

Console

Julia console support was added in v0.16. It may be enabled by loading PythonTeX with `usefamily=juliacon`. The `juliaconsole` environment uses Weave.jl internally to evaluate code. There is also a `juliaconcode` environment that executes code but typesets nothing.

7.3 Octave

Support for Octave was added in v0.13. Octave support should be at almost the same level as that for Python. Parsing of stderr for synchronization is successful in most cases but not ideal; this will be improved in a future release by a rewrite of the stderr parser.

Octave does not have a genuine utilities class, since it only supports `@CLASS` classes and does not yet support newer MATLAB-style `classdef`. As a result of this limited support for classes, there is a struct `octavetex` rather than a utilities

class instance `octavetex`. What would have been attributes of a utilities class instance are instead fields of the struct. What would have been methods of a class are instead anonymous functions. This allows `octavetex` to be used in most respects as if it were a class instance, especially insofar as syntax is concerned.

There are no `set_formatter()` or `open()` methods.

If any "methods" need to be overwritten, the simplest approach is probably to define a function and then set the appropriate struct field to an anonymous function that will call that function. For example, to replace the default `octavetex.before()`, we might define a function `before()`, and then use the command `octavetex.before = @() before();`. Of course, if the function is sufficiently short, it will be simpler just to put everything in the anonymous function: `octavetex.before = @() <expression>;`

A family of commands and environments for Octave is not created by default; the base name `octave` is provided.

When `\setpythontexcontext` is used, it must be accessed as struct fields, of the form `octavetex.context.<name>`.

7.4 bash

Support for bash was added in v0.15. Support for bash is very basic. Bash commands may be executed, and their output (stdout and stderr) may be typeset. As with other languages, all commands are executed in a single session unless the user specifies otherwise. There is not a utilities class or any related features.

Bash will work with Windows if it is installed.

7.5 Rust

Support for Rust was added in v0.15, with the command/environment base names `rs` and `rust`. Complete support is provided, except that the utilities struct `rstex` does not have an `open()` method. Also `rstex.formatter()`, `rstex.before()`, and `rstex.after()` may need additional refinement in the future to make them more convenient to work with. All user code is inserted within a template-generated `main()` function; `main()` should not be defined explicitly. Future refinements of PythonTeX's template system may allow user code outside of `main()`.

Because Rust typically gives a long sequence of errors, PythonTeX processing and synchronization of `stderr` is currently verbose and may need to be improved in the future as well. There is no support for encodings other than UTF-8. Currently, executables always use the `.exe` extension, even under non-Windows systems.

Due to the way `rstex` is used in template-generated code, it needs to remain a mutable local variable. This means that, while there should be no problem using it through either shared or mutable references, taking it by value requires that the "altered" copy is reassigned to a new variable that shadows the old one. That is, code that needs to work with `rstex` by value should look like

`let mut rstex = ⟨code⟩;`

Additionally, when using `\rust` and `\rs`, keep in mind that these wrap code in a block, so you *cannot* use `rstex` by value in these contexts (both shared and mutable references are still fine, though).

7.6 R

Support for R was added in v0.17.

Loading PythonTEX with `usefamily=R` enables the `R` family of commands and environments (`\R`, `\Rc`, `Rcode`, ...). These execute code with `Rscript`. The `methods` library is loaded automatically as part of the template code. Expressions passed to the `\R` command are converted into strings via `toString()`. There is currently no utilities class or related features. A null graphics device, `pdf(file=NULL)`, is created by default to avoid the automatic, unintentional creation of plot files with default names. Plots that are to be saved require explicit graphics commands.

Console

Loading PythonTEX with `usefamily=Rcon` enables the `Rconsole` environment, which executes code to emulate an interactive R session. There is also an `Rconcode` environment that executes code but typesets nothing. Code is executed with `Rscript`. The `methods` library is loaded automatically as part of the template code. The option `echo=TRUE` is used to intersperse code with output, while `error=function(){}` is used to avoid halting on errors. A null graphics device, `pdf(file=NULL)`, is created by default to avoid the automatic, unintentional creation of plot files with default names. Plots that are to be saved require explicit graphics commands.

7.7 Perl

Support for Perl was added in v0.17.

Loading PythonTEX with `usefamily=perl` enables the `perl` family of commands and environments. Alternatively, `usefamily=pl` may be used to enable the `pl` family. There is currently no utilities class or related features.

7.8 Perl 6

Support for Perl 6 was added in v0.17.

Loading PythonTEX with `usefamily=perlsix` enables the `perlsix` family of commands and environments. Alternatively, `usefamily=psix` may be used to enable the `psix` family. There is currently no utilities class or related features.

7.9 JavaScript

Support for JavaScript was added in v0.17.

Loading PythonTEX with `usefamily=javascript` enables the `javascript` family of commands and environments. Alternatively, `usefamily=js` may be used to enable the `js` family. There is a utilities object `jstex`.

7.10 Adding support for a new language

Adding support for an additional language involves creating two templates, creating a new instance of a class, and using a PythonTEX macro. In some cases, additional changes may be necessary for full support. The information below

does not deal with creating `console` families; additional support for user-defined `console` families will be added in the future.

The system for adding languages should be relatively stable, but is subject to change as additional languages with additional requirements are added. The current system is sufficient for Python and similar languages. Languages with less regular `stderr` may require additional features and may not be able to have full synchronization between `stderr` and the LaTeX document. Keep in mind that if PythonTeX is unable to classify exceptions as errors or warnings, it treats them as errors or warnings based on the script exit status.

It may be helpful to refer to `pythontex_engines.py`, specifically the templates and utilities classes, while reading the section below.

7.10.1 Template

PythonTeX executes user code by inserting it in a script template. Replacement fields in the template are indicated by curly braces: `{`⟨*field*⟩`}`.[28] Space between ⟨*field*⟩ and the braces is not allowed. Replacement fields (**including** the braces) should be surrounded by quotation marks or equivalent when the replacement is to be a string rather than literal code.

The template should perform the following tasks.

- Set the script encoding. The `{encoding}` field will be replaced with a user-specified encoding or the default `UTF-8`. If you are not using anything beyond ASCII, this is not strictly necessary.
- Python templates should have a `{future}` field at the beginning, for compability with Python 2 and the package option `pyfuture`.[29]
- Set the `stdout` and `stderr` encoding, again using `{encoding}`. As before, this is not strictly necessary when only ASCII support is needed.
- Create a language-specific equivalent of the PythonTeX utilities class.[30] Create an instance of this class. It is recommended that the class be called ⟨*language name*⟩`TeXUtils` and the instance ⟨*language extension*⟩`tex`, by analogy with the Python case.[31] When the ⟨*language extension*⟩ is only a single character or is shared by multiple languages, it may be better to use the full ⟨*language name*⟩ or an abbreviation in the name of the class instance.

  For full PythonTeX support, the utilities class should provide the following methods:

[28] This follows Python's format string syntax. Literal curly braces are obtained by doubling.

[29] The beginning of user code is parsed for imports from `__future__`. Any imports are collected and inserted into the `{{future}}` field.

[30] Python templates can import the PythonTeX utilities class. In that case, `sys.path.append('{{utilspath}}')` is needed before the import, so that the location of the utilities class is known.

[31] The class could be called ⟨*language name*⟩`TeX`. In that case, the class and the instance would have the very same name (except for capitalization) in cases where the language name and extension are the same (for example, Lua). That is probably not desirable, and besides, `Utils` adds additional clarity. The instance name ⟨*language extension*⟩`tex` is recommended because it will be short and easily remembered. Plain `tex` could be used instead, but that would be less descriptive (it lacks the interface connotations) and would not remind the user of the language currently in use (which could be beneficial in a document combining multiple languages, each with its own slightly different utilities class).

- `formatter()`: For formatting content for inline commands equivalent to `\py`. This should take a single argument of any type. By default, it should return a standard string representation of its argument.
- `before()` and `after()`: Initially, these should do nothing; they are provided to be redefined by the user. They should take no arguments.
- `add_dependencies()` and `add_created()`: These should accept an arbitrary number of comma-separated strings (if supported by the language). Each method should append its arguments to a list or equivalent data structure, for later use.
- `cleanup()`: This prints a dependencies delimiter string `{dependencies_delim}` to `stdout`, then prints all dependencies (one per line), then prints a created files delimiter string `{created_delim}`, then prints the names of all created files (one per line). The delimiters should be printed even if there are no dependencies or created files. The delimiters contain no backslashes or quotes.

The utilities class should also provide several variables, as described below.

- Attempt to change to the working directory `{workingdir}`. Raise an error and exit if this is not possible. For convenience, the script should check for a `--manual` command line argument. If this argument is present, the script should proceed even if the working directory cannot be found. This allows the user to manually invoke the script for debugging (the script can be saved via `keeptemps`).

  The working directory should be added to the module search path (Python `sys.path`, Ruby `$:` or `$LOAD_PATH`, etc.), unless it is the same as the document root directory or is otherwise already on the module search path.

- For full compatibility, the template should have an `{extend}` field where additional module imports or other code may be inserted. This allows a basic template to be a created for each language. The basic template may then be customized for specific purposes. The `{extend}` field should be after the utilities class instance has been created, so that the workings of the utilities class (`formatter()`, `before()`, `after()`, etc.) may be customized by it.

- LaTeX-related variables of the utilities class instance that do not change should be set. These use the fields `{family}`, `{session}`, and `{restart}`; all should be strings. These variables should be named after the fields if possible (for example, `pytex.family`). These variables are not strictly necessary, but they allow user code to access information about its origin on the TeX side.

- There should be a `{body}` field where the body of the script is inserted.

- The script should end by calling the `cleanup()` method.

7.10.2 Wrapper

Each chunk of user code is inserted into a wrapper template. This performs the following tasks.

- Set additional LaTeX-related utilities variables: `{command}`, `{context}`, `{args}`, `{instance}`, `{line}`. They are not required, but make possible closer LaTeX integration. `{args}` is not yet supported on the LaTeX side, but will allow arguments from LaTeX commands to be passed to user code.

  All utilities variables should be stored as strings, except for `context` and `args`. If possible, these should be dictionaries or equivalent associative arrays of string keys that point to string values. The dictionaries should be created by processing `{context}` and `{args}` into comma-separated lists of key-value pairs. For example, if `{context}` is the string "`k1=v1, k2=v2`", then `pytex.context` should be a dictionary, and `pytex.context['k1']` should yield the string "`v1`". The key-value pairs may optionally be accessed as attributes, when this is possible with a given language. For example, `pytex.context.k1` could yield the string "`v1`".

- Write a delimiter `{stdoutdelim}` to `stdout` and a delimiter `{stderrdelim}` to `stderr`. Both delimiters should be strings. Both should be written in such a way that the delimiter is followed by a newline; the delimiters that are inserted in the wrapper template **do not** contain a newline.[32] For example, something like `"{stderrdelim}\n"` might be necessary. The delimiters contain no backslashes or quotation marks.

- Call `before()`.

- Have a `{code}` field into which the current chunk of user code is inserted.

- Call `after()`.

7.10.3 The `CodeEngine` class

The final step in adding support for a language is creating a new instance of the `CodeEngine` class. The `CodeEngine` class manages the process of inserting user code into code templates and creates the records needed for synchronizing `stderr` with the document.

A new `CodeEngine` instance is initialized with the following arguments. All arguments are strings unless noted otherwise.

- The instance name. This will be the base name for commands and environments that use the instance. For example, `\py`, `\pyc`, `pycode`, etc., rely on the `py` instance of the `CodeEngine` class.

- The name of the language. In some cases, this may be the same as the instance name.

- The filename extension for scripts (with or without a period).

- The command for running scripts. The script that is executed should be referred to as "`{file}.`⟨*extension*⟩" (without the quotes).[33] The interpreter may be hardcoded (`python {file}.py`), but it is best to leave it as a substitution field (`{python} {file}.py`) so that the `--interpreter` command-line option can be used to provide a specific interpreter.

[32] This way, we don't have to assume that all languages will use `\n` for the newline character.

[33] It might seem that the extension is redundant, since it is specified separately. The command is specified in this form to simplify cases where there may be intermediary files in the execution process.

- The script template.
- The wrapper template.
- A template that specifies how code from commands like `\py` should be inserted into a call to the `formatter()` method. The user code is specified by `{code}`. The output of the `formatter()` method should be written to `stdout`, so something like `'print(pytex.formatter({code}))'` is needed.
- An optional list of strings (or an individual string) that gives patterns for identifying error messages.
- An optional list of strings (or an individual string) that gives patterns for identifying warning messages.
- An optional list of strings (or an individual string) that gives patterns for identifying code line numbers in `stderr`. These patterns use the field `{number}`. These patterns are searched for in any line of `stderr` that contains the name of the script that was executed.
- An optional boolean that specifies whether the engine emulates an interactive console. Currently, user-defined engines that emulate consoles are not supported.
- An optional string of startup commands for engines that emulate consoles.
- An optional list of strings (or an individual string) that specifies any files created during execution, beyond the script `{file}.`⟨*extension*⟩. The field `{file}` may be used in file names, if files are created with the same base name as the script; this could be useful with compiled languages, which might have a `{file}.`⟨*extension*⟩ that ultimately results in a `{file}.out`, `{file}.exe`, etc.

An example of creating the `py` engine is shown below. The `python_template` and `python_wrapper` are long enough that they are defined separately.

```
CodeEngine('py', 'python', '.py', 'python {file}.py',
           python_template, python_wrapper,
           'print(pytex.formatter({code}))',
           'Error:', 'Warning:', ['line {number}', ':{number}:'])
```

The script template and wrapper templates may be defined with Python's triple-quoted strings. All content within such a string may be indented for clarity, as can be seen in `pythontex_engines.py`. Strings are automatically dedented when `CodeEngine` instances are created.

In addition to the `CodeEngine` class, there is also a `SubCodeEngine` class. It allows a new engine to be created based on an existing engine. It requires the name of the engine from which to inherit and the name of the new engine. All of the other arguments listed above are optional; if any are provided, they overwrite the inherited arguments. The class also takes one additional optional argument, `extend`. This is a string that specifies additional code to be entered in the inherited template, in the `{extend}` field. Subengines of subengines may be created; in that case, any `extend`s are cumulative.

7.10.4 Creating the LaTeX interface

Once a new engine has been created, access from the LaTeX side must be provided. PythonTeX provides a macro for this purpose.

`\makepythontexfamily[`⟨*options*⟩`]{`⟨*engine*⟩`}`

This command creates a non-`console` family of commands and environments for ⟨*engine*⟩: `code`, `block`, and `verbatim` commands and environments, and an inline command like `\py`.

This command is appropriate for user-defined languages, but it is preferable (and more convenient) to use the package option `usefamily` when using an engine that is included with PythonTeX. The package option will create a preconfigured family in which things such as the appropriate Pygments lexer have already been set.

⟨*options*⟩ allows `prettyprinter`, `pyglexer`, and `pygopt` to be specified for the family.

8 Troubleshooting

- If a PythonTeX document will not compile, you may want to delete the directory in which PythonTeX content is stored and try compiling from scratch. It is possible for PythonTeX to become stuck in an unrecoverable loop. Suppose you tell Python to print some LaTeX code back to your LaTeX document, but make a fatal LaTeX syntax error in the printed content. This syntax error prevents LaTeX from compiling. Now suppose you realize what happened and correct the syntax error. The problem is that the corrected code cannot be executed until LaTeX correctly compiles and saves the code externally, but LaTeX cannot compile until the corrected code has already been executed. One solution in such cases is to correct the code, delete all files in the PythonTeX directory, compile the LaTeX document, and then run PythonTeX from scratch. You can also disable the inclusion of printed content using the `debug` package option.

  You may also run PythonTeX with the `--debug` option to launch the default debugger, or use the debugger of your choice by adding code that launches a debugger and then running PythonTeX with the `--interactive` option. See Section 3.2 for more details.

- Dollar signs $ may appear as £ in italic code comments typeset by Pygments. This is a font-related issue. One fix is to `\usepackage[T1]{fontenc}`.

- The `tabular` environment can conflict with PythonTeX under some circumstances, due to how `tabular` functions. Among other things, printing within a `tabular` environment can cause errors, because printing involves bringing in external content via `\InputIfFileExists`, but that macro is not expandable.[34] There are a few different ways to work around the limitations of `tabular`.

  - Put the printed content in a macro definition, and use the macro in `tabular`. You will have to create a dummy version of the macro, to avoid errors before the macro is defined by PythonTeX. An example is

[34] For more information, see this, this, and this.

given below. The `\global\def` is needed so that the macro is defined outside of the `pycode` environment.

```
\let\row\relax
\begin{pycode}
print("\\global\\def\\row{a & b & c & d & e \\\\}")
\end{pycode}

\begin{tabular}{|c|c|c|c|c|}
\row
\end{tabular}
```

  - Use `\py`. The end-of-row `\\` must be outside of the command. Example:

```
\begin{tabular}{|c|c|c|c|c|}
\py{"a & b & c & d & e"} \\
\end{tabular}
```

- PythonTEX commands like `\py` won't work inside `siunitx` macros, because `\py` and company aren't fully expandable.[35] There are different ways to work around this; some examples are shown below.

```
\documentclass{article}
\usepackage{siunitx}
\usepackage{pythontex}

\begin{pycode}
def SI(var, unit):
    return '\\SI{' + str(var) + '}{' + unit + '}'
\end{pycode}

\newcommand{\pySI}[2]{\py{'\\SI{' + str(#1) + '}{#2}'}}

\begin{document}
\pyc{y = 4}

The value of y is \py{SI(y, r'\metre')}.

The value of y is \pySI{y}{\metre}.

\end{document}
```

  Another example, this time using SymPy:

```
\newcommand{\sympySI}[2]{\sympy{SI(#1,r"#2")}}
\begin{sympycode}
def SI(var, unit):
    return '\\SI{{{0}}}{{{1}}}'.format(N(var, 4), unit)
\end{sympycode}
```

9 The future of PythonTEX

This section consists of a To Do list for future development. The To Do list is primarily for the benefit of the author, but also gives users a sense of what changes

[35] For more details, see this, this, and this.

are in progress or under consideration.

9.1 To Do

9.1.1 Modifications to make

- Add support for `depythontex` to remove the `\usepackage` for a package that contains PythonTeX commands and environments.
- Add better support for macro programming, including `depythontex` support for user-defined commands and environments.
- Add Pygments commands and environments that are compatible with basic `listings` and `minted` syntax. This will make it easier to work with documents converted to LaTeX from another format, for example via Pandoc.
- User-defined custom commands and environments for general Pygments typesetting.
- Additional documentation for the Python code (Sphinx?).
- Improved testing framework.
- It might nice to include some methods in the PythonTeX utilities for formatting numbers (especially with SymPy and Pylab).
- Test the behavior of files brought in via `\input` and `\include` that contain PythonTeX content.
- Continue adding support for additional languages. Under consideration: Perl, Lua, MATLAB, Mathematica, Sage, R, Octave.

9.1.2 Modifications to consider

- Consider fixing error line number synchronization with Beamer (and other situations involving error lines in externalized files). The `filehook` and `currfile` packages may be useful in this. One approach may be to patch the macros associated with `\beamer@doframeinput` in `beamerbaseframe.sty`. Note: Beamer's `fragile=singleslide` option makes this much less of an issue. This is low priority.
- Allow LaTeX in code, and expand LaTeX macros before passing code to `pythontex.py`. Maybe create an additional set of inline commands with additional `exp` suffix for `expanded`? This can already be done by creating a macro that contains a PythonTeX macro, though.
- Built-in support for background colors for blocks and verbatim, via `mdframed` or a similar package?
- Support for executing external scripts, not just internal code? It would be nice to be able to typeset an external file, as well as execute it by passing command-line arguments and then pull in its output.

- Is there any reason that saved printed content should be allowed to be brought in before the code that caused it has been typeset? Are there any cases in which the output should be typeset **before** the code that created it? That would require some type of external file for bringing in saved definitions.
- Consider some type of primitive line-breaking algorithm for use with Pygments. Could break at closest space, indent 8 spaces further than parent line (assuming 4-space indents; could auto-detect the correct size), and use LaTeX counter commands to keep the line numbering from being incorrectly incremented. Such an approach might not be hard and might have some real promise.
- Consider allowing names of files into which scripts are saved to be specified. This could allow PythonTeX to be used for literate programming, general code documentation, etc. Also, it could allow writing a document that describes code and also produces the code files, for user modification (see the `bashful` package for the general idea). Doing something like this would probably require a new, slightly modified interface to preexisting macros.

Acknowledgements

Thanks to Nicholas Lu Chee Seng for help testing the earliest versions.

Thanks to Øystein Bjørndal for many suggestions and for help with OS X compatibility.

Thanks to Alexander Altman for suggesting Rust support and providing template code.

Thanks to Nathan Carter for suggesting JavaScript support and providing template code.

Version History

v0.18 (2021/06/06)

- `\inputpygments` now checks inputted files for modification, so that typeset code will correctly update when the source is changed (#162).
- Julia now uses project flag "`--project=@.`" (#157, #158).
- Fixed bug in processing Pygments options (`pygopt`) when a key is used without a value (#181).
- Some error handling for Windows was incompatible with other operating systems: replaced checks for `WindowsError` with checks for `OSError` (#177).
- Rust support is now compatible with document and working directory paths that contain spaces (#167).

v0.17 (2019/09/22)

- Pygments syntax highlighting for the Python console (`pycon` lexer) now uses the `python3` option, and the default Python lexer is now `python3` (#156).
- Added support for JavaScript (#147; thanks to Nathan Carter).
- Updated Julia support for Julia versions 0.6 (#107), and 0.7 and 1.0 (#126, #130).
- There are now meaningful error messages for the Julia console when Weave.jl is not installed or raises errors (#131).
- `pythontexcustomcode` and `\pythontexcustomc` now set `pytex.context` (#65).
- Added support for R. The `R` family of commands and environments (`\R`, `\Rc`, `Rcode`, ...) executes code as a script. There is currently no utilities class or equivalent. The `Rcon` family (`Rconsole`) executes code to emulate an interactive R session (#121).
- `fancyvrb` settings from `\setpythontexfv` and console environments now work with Julia and R consoles.
- `pythontexcustomcode` now works with `juliacon`. There are now proper `juliaconcode` and `Rconcode` environments that execute code but typeset nothing, to parallel `pyconcode` (#134).
- Added support for Perl with the `perl` and `pl` families of commands and environments. There is currently no utilities class or equivalent.
- Added support for Perl 6 with the `perlsix` and `psix` families of commands and environments (#104). There is currently no utilities class or equivalent.
- Updated Rust support by using `dyn` with traits in utilities object.
- Under Windows, capitalization of script paths in `stderr` is now preserved.
- Fixed a bug that prevented the `sub` environment from working with `depythontex` (#155).
- Fixed a bug in checking mtime of dependencies to see if they have been modified while `pythontex` is running. The check failed for dependencies that do not exist or were deleted before `pythontex` can read them (#136).

v0.16 (2017/07/20)

- Added preliminary console support for Julia (#98).
- Fixed Python console compatibility with Python 3.6 by setting the `code` module's new `exitmsg` argument to suppress the exit message (#100).
- Improved Rust support, including tracking of created files and dependencies (#91).

v0.15 (2016/07/21)

New features

- The `fvextra` package is now required. This provides line breaking with fine-grained control over break locations, the ability to highlight specific lines or ranges of lines, improved handling of tabs, and several additional features.
- Added `sub` commands and environments (`\pys`, `pysub`, ...). These commands and environments perform string interpolation on text. Fields delimited by `!{...}` are replaced by the result of evaluating and then printing their content. This works for all families of commands and environments, not just Python. See the documentation for details about field delimiters and escaping.
- Added `rust` and `rs` families of commands and environments. These provide essentially complete support for Rust, except that `rstex.formatter()`, `rstex.before()`, and `rstex.after()` will likely need additional refinement (#90).
- Added the `sage` family of commands and environments, which provide support for Sage (#63).
- Added `bash` family of commands and environments. This provides basic support for bash (no utilities class or equivalent). Bash works with Windows if it is installed.
- Improved `console` compatibility under Linux with Python 3 (#70).
- Counters for default sessions are now created automatically. This prevents counter errors under some circumstances when working with `\includeonly`.
- Commands like `\py` can now output verbatim content under LuaTeX.

Bugfixes

- Fixed a bug that could cause an endless loop when a `code` command or environment printed a `code` command or environment of the same family with `autoprint=true`.

v0.14 (2014/07/17)

New features

- All commands for working with code inline are now robust, via `etoolbox`'s `\newrobustcmd`. Among other things, this allows commands like `\py` to work in standard captions that have not been redefined to avoid protection issues.
- Upgraded `syncpdb` to v0.2, which provides better list formatting.

Backward-incompatible changes

- The default working directory is now the main document directory instead of the output directory. Using the output directory was a common source of confusion for new users and was incompatible with plans for future development. Old documents in which the working directory was not specified will continue to use the output directory, but PythonTeX will print an upgrade message; new documents will use the new setting. The output directory may be selected as the working directory manually, or with the shorthand "`\setpythontexworkingdir{<outputdir>}`".
- Standardized version numbering by removing the "v" prefix from the stored version numbers in Python variables and LaTeX macros. Standardized the PythonTeX scripts by renaming `version` to `__version__`.

v0.13 (2014/07/14)

New features

- Added `--interactive` command-line option. This runs a single session in interactive mode, allowing user input. Among other things, this is useful when working with debuggers.
- Added `--debug` command-line option. This runs a single session with the default debugger in interactive mode. Currently, only standard (non-console) Python sessions are supported. The default Python debugger is the new `syncpdb`, which wraps `pdb` and synchronizes code line numbers with document line numbers. All `pdb` commands that take a line number or filename:lineno as an argument will refer to document files and line numbers when the argument has a percent symbol (`%`) as a prefix. For example, `list %50` lists code that came from around line 50 in the document. The `--debug` option will support other languages and provide for customization in the future.
- Added command-line option `--jobs`, which allows the maximum number of concurrent processes to be specified (#35).
- Added support for GNU Octave, via the `octave` family of commands and environments (#36). Parsing of Octave stderr is not ideal, though synchronization works in most cases; this will be addressed by a future rewrite of the stderr parser.
- Installer now automatically works with MiKTeX, not just TeX Live.
- The PythonTeX utilities class has a new `open()` method that opens files and automatically tracks dependencies/created files.
- When `pythontex2.py` and `pythontex3.py` are run directly, the Python interpreter is automatically set to a reasonable default (`py -2` or `py -3` under Windows, using the Python 3.3+ wrapper; `python2` or `python3` under other systems).
- The installer now creates symlinks for the numbered scripts `pythontex*.py` and `depythontex*.py`.
- Added Python version checking to all numbered scripts.

- Under Python, the type of data passed via `\setpythontexcontext` may now be set using YAML-style tags (`!!str`, `!!int`, `!!float`). For example, `\setpythontexcontext{myint=!!int 123}`.
- The `fancyvrb` options `firstline` and `lastline` now work with the `pygments` environment and `\inputpygments` command. This required some additional patching of `fancyvrb`.
- The `pytx@Verbatim` and `pytx@SaveVerbatim` environments are now used for typesetting verbatim code. These are copies of the `fancyvrb` environments. This prevents conflicts when literal `Verbatim` and `SaveVerbatim` environments need to be typeset.
- Improved `latexmk` compatibility (#40). Added discussion of `latexmk` usage to documentation.
- Tildes `~` may now be used in `outputdir` and `workingdir` to refer to the user's home directory, even under Windows.

Bugfixes

- Fixed a bug that prevented created files from being cleaned up when the working directory was not the document root directory and the full path to the files was not provided.
- Fixed a bug that prevented the `fvextfile` option from working when external files were highlighted.

v0.13-beta (2014/02/06)

New features

- Switching to GitHub's Releases for downloads.
- TeX information such as page dimensions may now be easily passed to the programming-language side, using the new `\setpythontexcontext` command. Contextual information is stored in the `context` attribute of the utilities class, which is a dictionary (and also has attributes in Python).
- The utilities class now has `pt_to_in()`, `pt_to_cm()`, and `pt_to_mm()` methods for converting units of TeX points into inches, centimeters, and millimeters. These work with integers and floats, as well as strings that consist of numbers and optionally end in "pt". There is also a `pt_to_bp()` for converting TeX points (1/72.27 inch) into big (DTP or PostScript) points (1/72 inch).
- Expanded Quickstart. Quickstart is now compatible with all LaTeX engines. Quickstart now avoids `microtype` issues on some systems (#32).
- Added information on citing PythonTeX (#28).
- Utilities class has a new attribute `id`, which is a string that joins the command family name, session name, and session restart parameters with underscores. This may be used in creating files that need a name that contains a unique, session-based identifier (for example, names for figures that are saved automatically).

Backward-incompatible changes

- All utilities-class attributes with names of the form `input_*` have been renamed with the "`input_`" removed. Among other things, this makes it easier to access the `context` attribute (`pytex.context` vs. `pytex.input_context`).
- `depythontex` now has `-o` and `--output` command-line options for specifying the name of the output file. If an output file is not specified, then output is written to `stdout`. This allows `depythontex` output to be piped to another program.
- All scripts `*2.py` now have shebangs with `env python2`, and all scripts `*3.py` now have shebangs with `env python3`. This allows the wrapper scripts (`env python` shebang) to be used with the default Python installation, and the numbered scripts to be used with specific versions. Remember that except for console content, the `--interpreter` option is what determines the Python version that actually executes code. The version of Python used to launch `pythontex.py` merely determines the version that manages code execution. (`--interpreter` support for console content is coming.)
- Changed the template style used in the `CodeEngine` class. Replacement fields are now surrounded by single curly braces (as in Python's format string syntax), rather than double curly braces. Literal curly braces are obtained by doubling braces. This allows the use of literal adjacent double braces in templates, which was not possible previously.
- The Julia template now uses the new `in()` function, replacing `contains()`. This requires Julia v0.2.0+.

Bugfixes

- Modified test for LuaTeX, so that `\directlua` is not `\let` to `\relax` if it does not exist. This was causing incompatibility with `babel` under pdfTeX and XeTeX (#33).
- Added missing shebangs to `depythontex*.py`. Handling of `utilspath` is now more forgiving, so that `pythontex_utils.py` can be installed in alternate locations (#23).
- `depythontex` no longer leaves a blank line where `\usepackage{pythontex}` was removed.
- Console environments typeset with `fancyvrb` no longer end with an unnecessary empty line.
- Fixed bug in installer when `kpsewhich` was not found (#21).

v0.12 (2013/08/26)

- Added support for the Julia language, with the `julia` and `jl` families of commands and environments. (Note that Pygments only added Julia support in version 1.6.)

- Warnings and errors are now synchronized with the line numbers of files brought in via `\input`, `\include`, etc. This is accomplished using the `currfile` package.
- Added package option `gobble`. When `gobble=auto`, all code is dedented before being executed and/or typeset. The current implementation is functional but basic; it will be improved and extended in the future.
- The document root directory is now always added to `sys.path` (or its equivalent), even when it is not the working directory. (The working directory has been added to `sys.path` since v0.12beta.) The document directory is added after the working directory, so that the working directory has precedence.
- Fixed a bug in `console` commands and environments; `sys.path` now contains the working and document directories, and the working directory is now the output directory by default. This parallels the behavior of non-`console` commands and environments.
- Added command-line option `--interpreter` that allows an interpreter to be invoked via a specific command. This allows, for example, a specific version of Python to be invoked.
- Improved synchronization of stderr in cases when an error is triggered far after its origin (for example, an error caused by a multiline string that is lacking a closing quote/delimiter, and thus may span several chunks of user code).
- Modified usage of the `shlex` module to work around its lack of Unicode support in Python versions prior to 2.7.3.
- Fixed a bug from v0.12beta that prevented `\inputpygments` from working when `pygments=true`.
- Fixed a bug with counters that caused errors when content spanning multiple columns was created within a `tabular` environment.
- Added checking for compatible Python versions in `pythontex.py`.
- Improved execution of `*.bat` and `*.cmd` files under Windows. The solution from v0.12beta allowed `*.bat` and `*.cmd` to be found and executed when the extension was not given, but did not give correct return codes.

v0.12beta (2013/06/24)

- Merged `pythontex_types*.py` into a single replacement `pythontex_engines.py` compatible with both Python 2 and 3. It is now much simpler to add support for additional languages.
- Added support for the Ruby language as a demonstration of new capabilities. The `ruby` and `rb` families of commands and environments may be enabled via the new `usefamily` package option. Support for additional languages is coming soon. See the new section in the documentation on support for other languages for more information.

- Reimplemented treatment of Pygments content for better efficiency. Now a Pygments process only runs if there is content to highlight. Eliminated redundant highlighting of unmodified code.
- Improved treatment of dependencies. If a dependency is modified (`os.path.getmtime()`) after the current PythonTeX run starts, then code that depends on it will be re-executed the next time PythonTeX runs. A message is also issued to indicate that this is the case.
- The utilities class now has `before()` and `after()` methods that are called immediately before and after user code. These may be redefined to customize output. For example, LaTeX commands could be printed before and after user code; stdout could be redirected to `StringIO` for further processing; or matplotlib figures could be automatically detected, saved, and included in the document.
- Added explanation of how to track dependencies and created files automatically, and how to include matplotlib figures automatically, to the documentation for the PythonTeX utilities class.
- Created a new system for parsing and synchronizing stderr.
  - Exceptions that do not reference a line number in user code (such as those from `warnings.warn()` in a module) are now traced back to a single command or environment. Previously no synchronization was attempted. This is accomplished by writing delimiters to stderr before executing the code from each command/environment.
  - Exceptions that do reference a line in user code are more efficiently synchronized with a document line number. This is accomplished by careful record keeping as each script is assembled. Line number synchronization no longer involves parsing the script that was executed.
  - Improved and generalized parsing of stderr, in preparation for supporting additional languages. Exceptions that cannot be identified as errors or warnings are treated based on `Popen.returncode`.
- Created a new system for `console` content.
  - There are now separate families of `console` commands and environments. No Pygments or `fancyvrb` settings are shared with the non-`console` families, as was previously the case. There is a new family of commands and environments based on `pycon`, including the `\pycon` command (inline reference to console variable), `pyconsole` environment (same as the old one), `\pyconc` and `pyconcode` (execute only), and `\pyconv` and `pyconverbatim` (typeset only). There are equivalent families based on `pylabcon` and `sympycon`.
  - Each console session now runs in its own process and is cached individually. Console output is now cached so that changing Pygments settings no longer requires re-execution.
  - Unicode is now supported under Python 2.
  - The new package option `pyconfuture` allows automatic imports from `__future__` for `console` families under Python 2, paralleling the `pyfuture` option.

  - Any errors or warnings caused by code that is not typeset (`code` command and environment, startup code) are reported in the run summary. This ensures that such code does not create mischief.
  - `customcode` is now supported for `console` content.

- Better support for `latexmk` and similar build tools. PythonTeX creates a file of macros (`*.pytxmcr`) that is always included in a document, and thus can be automatically detected and tracked by `latexmk`. This file now contains the time at which PythonTeX last created files. When new files are created, the macro file will have a new hash, triggering another document compile.
- Improved the way in which the PythonTeX `outputdir` is added to the graphics path. This had been done with `\graphicspath`, but that overwrites any graphics path previously specified by the user. Now the `outputdir` is appended to any pre-existing path.
- Added the `depythontex` option `--graphicspath`. This adds the `outputdir` to the graphics path of the `depythontex` document.
- The installer now provides more options for installation locations. It will now create missing directories if desired.
- The working directory (`workingdir`) is now appended to `sys.path`, so that code there may be imported.
- Under Windows, `subprocess.Popen()` is now invoked with `shell=True` if `shell=False` results in a WindowsError. This allows commands involving `*.bat` and `*.cmd` files to be executed when the extension is not specified; otherwise, only `*.exe` can be found and run.
- The path to utils is now found in `pythontex.py` via `sys.path[0]` rather than `kpsewhich`. This allows the PythonTeX scripts to be executed in an arbitrary location; they no longer must be installed in a texmf tree where `kpsewhich` can find them.
- Added `rerun` value `never`.
- At the end of each run, data and macros are only saved if modified, improving efficiency.
- The number of temporary files required by each process was reduced by one. All macros for commands like `\py` are now returned within stdout, rather than in their own file.
- Fixed a bug with `\stderrpythontex`; it was defaulting to `verb` rather than `verbatim` mode.

v0.11 (2013/04/21)

- As the first non-beta release, this version adds several features and introduces several changes. You should read these release notes carefully, since some changes are not backwards-compatible. Changes are based on a thorough review of all current and planned features. PythonTeX's capabilities

have already grown beyond what was originally intended, and a long list of features still remains to be implemented. As a result, some changes are needed to ensure consistent syntax and naming in the future. Insofar as possible, all command names and syntax will be frozen after this release.

- Added the `pythontex.py` and `depythontex.py` wrapper scripts. When run, these detect the current version of Python and import the correct PythonTeX code. It is still possible to run `pythontex*.py` and `depythontex*.py` directly, but the new wrapper scripts should be used instead for simplicity. There is now only a single `pythontex_utils.py`, which works with both Python 2 and Python 3.

- Added the `beta` package option. This makes the current version behave like v0.11beta, for compatibility. This option is temporary and will probably only be retained for a few releases.

- Backward-incompatible changes (require the `beta` option to restore old behavior)

  - The `pyverb` environment has been renamed `pyverbatim`. The old name was intended to be concise, but promoted confusion with LaTeX's `\verb` macro.

  - For `\printpythontex`, `\stdoutpythontex`, and `\stderrpythontex`, the modes `inlineverb` and `v` have been replaced by `verb`, and the old mode `verb` has been replaced by `verbatim`. This brings naming conventions in line with standard LaTeX `\verb` and `verbatim`, avoiding a source of potential confusion.

  - The `\setpythontexpyglexer`, `\setpythontexpygopt`, and `\setpygmentspygopt` commands now take an optional argument and a mandatory argument, rather than two mandatory arguments. This creates better uniformity among current and planned settings macros.

  - The `\setpythontexformatter` and `\setpygmentsformatter` commands have been replaced by the `\setpythontexprettyprinter` and `\setpygmentsprettyprinter` commands. This anticipates possible upcoming features. It also avoids potential confusion with Pygments's formatters and the utilities class's `formatter()` method.

- Deprecated (still work, but raise warnings; after a few releases, they will raise errors instead, and after that eventually be removed)

  - The `rerun` setting `all` was renamed `always`, in preparation for upcoming features.

  - The `stderr` option is replaced by `makestderr`. The `print`/`stdout` option is replaced by `debug`. These are intended to prevent confusion with future features.

  - The `fixlr` option is deprecated. It was originally introduced to deal with some of SymPy's LaTeX formatting, which has since changed.

  - The utilities class method `init_sympy_latex()` is deprecated. The `sympy_latex()` and `set_sympy_latex()` methods now automatically initialize themselves on first use.

- Added `autostdout` package option and `\setpythontexautostdout`, to complement `autoprint`. Added `prettyprinter` and `prettyprintinline` package options to complement new settings commands.
- Added quickstart guide.
- Installer now installs gallery and quickstart files, if present.

v0.11beta (2013/02/17)

- Commands like `\py` can now bring in any valid LaTeX code, including verbatim content, under the pdfTeX and XeTeX engines. Verbatim content was not allowed previously. LuaTeX cannot bring in verbatim, due to a known bug.
- Added package option `depythontex` and scripts `depythontex*.py`. These allow a PythonTeX document to be converted into a pure LaTeX document, with no Python dependency. The package option creates an auxiliary file with extension `.depytx`. The `depythontex*.py` scripts take this auxiliary file and the original LaTeX document, and combine the two to produce a new document that does not rely on the PythonTeX package. All PythonTeX commands and environments are replaced by their output. All Python-generated content is substituted directly into the document. By default, all typeset code is wrapped in `\verb` and `verbatim`, but `depythontex*.py` has a `--listing` option that allows `fancyvrb`, `listings`, `minted`, or `pythontex` to be used instead.
- The current PythonTeX version is now saved in the `.pytxcode`. If this does not match the version of the PythonTeX scripts, a warning is issued. This makes it easier to determine errors due to version mismatches.
- Fixed an incompatibility with the latest release of `xstring` (version 1.7, 2013/01/13).
- Fixed a bug in the `console` environment that could cause problems when switching from Pygments highlighting to `fancyvrb` when using the `fvextfile` option. Fixed a bug introduced in the v0.10beta series that prevented the `console` environment from working with `fancyvrb`.
- Fixed a bug with PythonTeX verbatim commands and environments that use Pygments. The verbatim commands and environments were incorrectly treated as if they had the attributes of executed code in the v0.10beta series.
- Fixed a bug from the v0.10beta series that sometimes prevented imports from `__future__` from working when there were multiple sessions.
- Fixed a bug related to hashing dependencies' mtime under Python 3.

v0.10beta2 (2013/01/23)

- Improved `pythontex*.py`'s handling of the name of the file being processed. A warning is no longer raised if the name is given with an extension; extensions are now processed (stripped) automatically. The filename may now

contain a path to the file, so you need not run `pythontex*.py` from within the document's directory.

- Added command-line option `--verbose` for more verbose output. Currently, this prints a list of all processes that are launched.
- Fixed a bug that could crash `pythontex*.py` when the package option `pygments=false`.
- Added documentation about `autoprint` behavior in the preamble. Summary: `code` commands and environments are allowed in the preamble as of v0.10beta. `autoprint` only applies to the body of the document, because nothing can be typeset in the preamble. Content printed in the preamble can be brought in by explicitly using `\printpythontex`, but this should be used with great care.
- Revised `\stdoutpythontex` and `\printpythontex` so that they work in the preamble. Again, this should be used with great care if at all.
- Revised treatment of any content that custom code attempts to print. Custom code is not allowed to print to the document (see documentation). If custom code attempts to print, a warning is raised, and the printed content is included in the `pythontex*.py` run summary.
- One-line entries in stderr, such as those produced by Python's `warnings.warn()`, were not previously parsed because they are of the form `:<linenumber>:` rather than `line <linenumber>`. These are now parsed and synchronized with the document. They are also correctly parsed for inclusion in the document via `\stderrpythontex`.
- If the package option `stderrfilename` is changed, all sessions that produced errors or warnings are now re-executed automatically, so that their stderr content is properly updated with the new filename.

v0.10beta (2013/01/09)

- Backward-incompatible: Redid treatment of command-line options for `pythontex*.py`, using Python's `argparse` module. Run `pythontex*.py` with option `-h` to see new command line options.
- Deprecated: `\setpythontexcustomcode` is deprecated in favor of the `\pythontexcustomc` command and `pythontexcustomcode` environment. These allow entry of pure code, unlike `\setpythontexcustomcode`. These also allow custom code to be added to the beginning or end of a session, via an optional argument. Improved treatment of errors and warnings associated with custom code.
- The summary of errors and warnings now correctly differentiates errors and warnings produced by user code, rather than treating all of them as errors. By default, `pythontex*.py` now returns an exit code of 1 if there were errors.

- The PythonTeX utilities class now allows external file dependencies to be specified via `pytex.add_dependencies()`, so that sessions are automatically re-executed when external dependencies are modified (modification is determined via either hash or mtime; this is governed by the new `hashdependencies` option).
- The PythonTeX utilities class now allows created files to be specified via `pytex.add_created()`, so that created files may be automatically cleaned up (deleted) when the code that created them is modified (for example, name change for a saved plot).
- Added the following package options.
  - `stdout` (or `print`): Allows input of stdout to be disabled. Useful for debugging.
  - `runall`: Executes everything. Useful when code depends on external data.
  - `rerun`: Determines when code is re-executed. Code may be set to always run (same as `runall` option), or only run when it is modified or when it produces errors or warnings. By default, code is always re-executed if there are errors or modifications, but not re-executed if there are warnings.
  - `hashdependencies`: Determines whether external dependencies (data, external code files highlighted with Pygments, etc.) are checked for modification via hashing or modification time. Modification time is default for performance reasons.
- Added the following new command line options. The options that are equivalent to package options are overridden by the package options when present.
  - `--error-exit-code`: Determines whether an exit code of 1 is returned if there were errors. On by default, but can be turned off since it is undesirable when working with some editors.
  - `--runall`: Equivalent to new package option.
  - `--rerun`: Equivalent to new package option.
  - `--hashdependencies`: Equivalent to new package option.
- Modified the `fixlr` option, so that it only patches commands if they have not already been patched (avoids package conflicts).
- Added `\setpythontexautoprint` command for toggling autoprint on/off within the body of the document.
- Installer now attempts to create symlinks under OS X and Linux with TeX Live, and under OS X with MacPorts Tex Live.
- Performed compatibility testing under lualatex and xelatex (previously, had only tested with pdflatex). Added documentation for using these TeX engines; at most, slightly different preambles are needed. Modified the PythonTeX gallery to support all three engines.

- Code commands and environments may now be used in the preamble. This, combined with the new treatment of custom code, allows PythonTeX to be used in creating LaTeX packages.
- Added documentation for using PythonTeX in LaTeX programming.
- Fixed a bug that sometimes caused incorrect line numbers with `stderr` content. Improved processing of stderr.
- Fixed a bug in automatic detection of pre-existing listings environment.
- Improved the detection of imports from `__future__`. Detection should now be stricter, faster, and more accurate.

v0.9beta3 (2012/07/17)

- Added Unicode support, which required the Python code to be split into one set for Python 2 and another set for Python 3. This will require any old installation to be completely removed, and a new installation created from scratch.
- Refactoring of Python code. Documents should automatically re-execute all code after updating to the new version. Otherwise, you should delete the PythonTeX directory and run PythonTeX.
- Improved installation script.
- Added package options: pyfuture, stderr, upquote, pyglexer, pyginline. Renamed the pygextfile option to fvextfile.
- Added custom code and workingdir commands.
- Added the console environment and associated options.
- Rewrote pythontex_utils*.py, creating a new, context-aware interface to SymPy's LatexPrinter class.
- Content brought in via macros no longer uses labels. Rather, long defs are used, which allows line breaks.
- Pygments highlighting is now default for PythonTeX commands and environments

v0.9beta2 (2012/05/09)

- Changed Python output extension to .stdout.

v0.9beta (2012/04/27)

- Initial public beta release.

10 Implementation

This section describes the technical implementation of the package. Unless you wish to understand all the fine details or need to use the package in extremely sophisticated ways, you should not need to read it.

The prefix `pytx@` is used for all PythonTeX macros, to prevent conflict with other packages. Macros that simply store text or a value for later retrieval are given names completely in lower case. For example, `\pytx@packagename` stores the name of the package, `PythonTeX`. Macros that actually perform some operation in contrast to simple storage are named using CamelCase, with the first letter after the prefix being capitalized. For example, `\pytx@CheckCounter` checks to see if a counter exists, and if not, creates it. Thus, macros are divided into two categories based on their function, and named accordingly.

10.1 Package opening

We store the name of the package in a macro for later use in warnings and error messages.

```
\newcommand{\pytx@packagename}{PythonTeX}
\newcommand{\pytx@packageversion}{0.18}
```

10.2 Required packages

A number of packages are required. `fvextra`, which loads and extends `fancyvrb`, is used to typeset all code that is not inline. `fancyvrb` internals are used to format inline code. `etoolbox` is used for string comparison and boolean flags. `xstring` provides string manipulation. `pgfopts` is used to process package options, via the `pgfkeys` package. `newfloat` allows the creation of a floating environment for code listings. `currfile` is needed to allow errors and warnings to be synchronized with content brought in via `\input`, `\include`, etc. `xcolor` or `color` is needed for syntax highlighting with Pygments.

```
\RequirePackage{fvextra}
\RequirePackage{etoolbox}
\RequirePackage{xstring}
\RequirePackage{pgfopts}
\RequirePackage{newfloat}
\@ifpackageloaded{currfile}{}{\RequirePackage{currfile}}
\AtEndPreamble{\@ifpackageloaded{color}{}{\RequirePackage{xcolor}}}
```

10.3 Package options

We now proceed to define package options, using the `pgfopts` package that provides a package-level interface to `pgfkeys`. All keys for package-level options are placed in the key tree under the path `/PYTX/pkgopt/`, to prevent conflicts with any other packages that may be using `pgfkeys`.

10.3.1 Enabling command and environment families

`\pytx@families` This option determines which command and environment families are defined beyond `py`, `pylab`, and `sympy`. Additional families are not automatically defined

since some of them create commands or environment that may conflict with other packages.[36]

```
\def\pytx@families{}
\pgfkeys{/PYTX/pkgopt/usefamily/.estore in=\pytx@families}
```

10.3.2 Gobble

`\pytx@opt@gobble` This option determines how leading whitespace in user code is treated.

```
\def\pytx@opt@gobble{none}
\pgfkeys{/PYTX/pkgopt/gobble/.is choice}
\pgfkeys{/PYTX/pkgopt/gobble/none/.code=\def\pytx@opt@gobble{none}}
\pgfkeys{/PYTX/pkgopt/gobble/auto/.code=\def\pytx@opt@gobble{auto}}
```

10.3.3 Beta

`pytx@opt@beta` This option provides compatibility with the beta releases from before the full v0.11 release. It should be removed after a few major releases.

```
\newbool{pytx@opt@beta}
\pgfkeys{/PYTX/pkgopt/beta/.default=true}
\pgfkeys{/PYTX/pkgopt/beta/.is choice}
\pgfkeys{/PYTX/pkgopt/beta/true/.code=\booltrue{pytx@opt@beta}}
\pgfkeys{/PYTX/pkgopt/beta/false/.code=\boolfalse{pytx@opt@beta}}
```

10.3.4 Runall

`pytx@opt@rerun` This option causes all code to be executed, regardless of whether it has been modified. It is primarily useful for re-executing code that has not changed, when the code depends on external files that **have** changed. Since it shares functionality with the `rerun` option, both options share a single macro. Note that the macro is initially set to `default`, rather than the default value of `errors`, so that the Python side can distinguish whether a value was actually set by the user on the TEX side, and thus any potential conflicts between command-line options and package options can be resolved in favor of package options.

```
\def\pytx@opt@rerun{default}
\pgfkeys{/PYTX/pkgopt/runall/.default=true}
\pgfkeys{/PYTX/pkgopt/runall/.is choice}
\pgfkeys{/PYTX/pkgopt/runall/true/.code=\def\pytx@opt@rerun{always}}
\pgfkeys{/PYTX/pkgopt/runall/false/.code=\relax}
```

10.3.5 Rerun

This option determines the conditions under which code is rerun. It stores its state in a macro shared with `runall`.

```
\pgfkeys{/PYTX/pkgopt/rerun/.is choice}
\pgfkeys{/PYTX/pkgopt/rerun/never/.code=\def\pytx@opt@rerun{never}}
\pgfkeys{/PYTX/pkgopt/rerun/modified/.code=\def\pytx@opt@rerun{modified}}
\pgfkeys{/PYTX/pkgopt/rerun/errors/.code=\def\pytx@opt@rerun{errors}}
\pgfkeys{/PYTX/pkgopt/rerun/warnings/.code=\def\pytx@opt@rerun{warnings}}
\pgfkeys{/PYTX/pkgopt/rerun/always/.code=\def\pytx@opt@rerun{always}}
```

[36] For example, a `\ruby` command for Ruby code, and the `\ruby` command defined by the Ruby package in the CJK package.

```
\pgfkeys{/PYTX/pkgopt/rerun/all/.code=\def\pytx@opt@rerun{always}%
    \PackageWarning{\pytx@packagename}{rerun=all is deprecated; use rerun=always}}
```

10.3.6 Hashdependencies

pytx@opt@hashdependencies This option determines whether dependencies (either code to be highlighted, or dependencies such as data that have been specified within a session) are checked for modification via modification time or via hashing.

```
\def\pytx@opt@hashdependencies{default}
\pgfkeys{/PYTX/pkgopt/hashdependencies/.is choice}
\pgfkeys{/PYTX/pkgopt/hashdependencies/.default=true}
\pgfkeys{/PYTX/pkgopt/hashdependencies/true/.code=\def\pytx@opt@hashdependencies{true}}
\pgfkeys{/PYTX/pkgopt/hashdependencies/false/.code=\def\pytx@opt@hashdependencies{false}}
```

10.3.7 Autoprint

pytx@opt@autoprint The `autoprint` option determines whether content printed within a code command or environment is automatically included at the location of the command or environment. If the option is not used, `autoprint` is turned on by default. If the option is used, but without a setting (`\usepackage[autoprint]{pythontex}`), it is true by default. We use the key handler ⟨*key*⟩`/.is choice` to ensure that only true/false values are allowed. The code for the true branch is redundant, but is included for symmetry.

```
\newbool{pytx@opt@autoprint}
\booltrue{pytx@opt@autoprint}
\pgfkeys{/PYTX/pkgopt/autoprint/.default=true}
\pgfkeys{/PYTX/pkgopt/autoprint/.is choice}
\pgfkeys{/PYTX/pkgopt/autoprint/true/.code=\booltrue{pytx@opt@autoprint}}
\pgfkeys{/PYTX/pkgopt/autoprint/false/.code=\boolfalse{pytx@opt@autoprint}}
\pgfkeys{/PYTX/pkgopt/autostdout/.default=true}
\pgfkeys{/PYTX/pkgopt/autostdout/.is choice}
\pgfkeys{/PYTX/pkgopt/autostdout/true/.code=\booltrue{pytx@opt@autoprint}}
\pgfkeys{/PYTX/pkgopt/autostdout/false/.code=\boolfalse{pytx@opt@autoprint}}
```

\setpythontexautoprint
\setpythontexautostdout
Sometimes it may be useful to switch `autoprint` on and off within different parts of a document, rather than setting it to a single setting for the entire document. So we provide a command for that purpose. Note that the command overrides the package-level option.

```
\newcommand{\setpythontexautoprint}[1]{%
    \Depythontex{cmd:setpythontexautoprint:m:n}%
    \ifstrequal{#1}{true}{\booltrue{pytx@opt@autoprint}}{}%
    \ifstrequal{#1}{false}{\boolfalse{pytx@opt@autoprint}}{}%
}
\newcommand{\setpythontexautostdout}[1]{%
    \Depythontex{cmd:setpythontexautostdout:m:n}%
    \ifstrequal{#1}{true}{\booltrue{pytx@opt@autoprint}}{}%
    \ifstrequal{#1}{false}{\boolfalse{pytx@opt@autoprint}}{}%
}
```

10.3.8 Debug

pytx@opt@stdout This option determines whether printed content/content written to stdout is included in the document. Disabling the inclusion of printed content is useful when

the printed content contains LaTeX errors that would prevent successful compilation.

```
\newbool{pytx@opt@stdout}
\booltrue{pytx@opt@stdout}
\pgfkeys{/PYTX/pkgopt/debug/.code=\boolfalse{pytx@opt@stdout}}
\pgfkeys{/PYTX/pkgopt/stdout/.default=true}
\pgfkeys{/PYTX/pkgopt/stdout/.is choice}
\pgfkeys{/PYTX/pkgopt/stdout/true/.code=\booltrue{pytx@opt@stdout}%
    \PackageWarning{\pytx@packagename}{Option stdout is deprecated; use option debug}}
\pgfkeys{/PYTX/pkgopt/stdout/false/.code=\boolfalse{pytx@opt@stdout}%
    \PackageWarning{\pytx@packagename}{Option stdout is deprecated; use option debug}}
\pgfkeys{/PYTX/pkgopt/print/.default=true}
\pgfkeys{/PYTX/pkgopt/print/.is choice}
\pgfkeys{/PYTX/pkgopt/print/true/.code=\booltrue{pytx@opt@stdout}%
    \PackageWarning{\pytx@packagename}{Option print is deprecated; use option debug}}
\pgfkeys{/PYTX/pkgopt/print/false/.code=\boolfalse{pytx@opt@stdout}%
    \PackageWarning{\pytx@packagename}{Option print is deprecated; use option debug}}
\AtBeginDocument{%
    \ifbool{pytx@opt@stdout}{}{%
        \PackageWarning{\pytx@packagename}{Using package option debug}%
    }%
}
```

10.3.9 makestderr

`pytx@opt@stderr` The `makestderr` option determines whether stderr is saved and may be included in the document via `\stderrpythontex`.

```
\newbool{pytx@opt@stderr}
\pgfkeys{/PYTX/pkgopt/makestderr/.default=true}
\pgfkeys{/PYTX/pkgopt/makestderr/.is choice}
\pgfkeys{/PYTX/pkgopt/makestderr/true/.code=\booltrue{pytx@opt@stderr}}
\pgfkeys{/PYTX/pkgopt/makestderr/false/.code=\boolfalse{pytx@opt@stderr}}
\pgfkeys{/PYTX/pkgopt/stderr/.default=true}
\pgfkeys{/PYTX/pkgopt/stderr/.is choice}
\pgfkeys{/PYTX/pkgopt/stderr/true/.code=\booltrue{pytx@opt@stderr}%
    \PackageWarning{\pytx@packagename}{Option stderr is deprecated; use option makestderr}}
\pgfkeys{/PYTX/pkgopt/stderr/false/.code=\boolfalse{pytx@opt@stderr}%
    \PackageWarning{\pytx@packagename}{Option stderr is deprecated; use option makestderr}}
```

10.3.10 stderrfilename

`\pytx@opt@stderrfilename` This option determines how the file name appears in `stderr`.

```
\def\pytx@opt@stderrfilename{full}
\pgfkeys{/PYTX/pkgopt/stderrfilename/.default=full}
\pgfkeys{/PYTX/pkgopt/stderrfilename/.is choice}
\pgfkeys{/PYTX/pkgopt/stderrfilename/full/.code=\def\pytx@opt@stderrfilename{full}}
\pgfkeys{/PYTX/pkgopt/stderrfilename/session/.code=\def\pytx@opt@stderrfilename{session}}
\pgfkeys{/PYTX/pkgopt/stderrfilename/genericfile/.code=%
    \def\pytx@opt@stderrfilename{genericfile}}
\pgfkeys{/PYTX/pkgopt/stderrfilename/genericscript/.code=%
    \def\pytx@opt@stderrfilename{genericscript}}
```

10.3.11 Python's `__future__` module

`\pytx@opt@pyfuture` The `pyfuture` option determines what is imported from the `__future__` module under Python 2. It has no effect under Python 3.

```
\def\pytx@opt@pyfuture{default}
\pgfkeys{/PYTX/pkgopt/pyfuture/.is choice}
\pgfkeys{/PYTX/pkgopt/pyfuture/default/.code=\def\pytx@opt@pyfuture{default}}
\pgfkeys{/PYTX/pkgopt/pyfuture/all/.code=\def\pytx@opt@pyfuture{all}}
\pgfkeys{/PYTX/pkgopt/pyfuture/none/.code=\def\pytx@opt@pyfuture{none}}
```

`\pytx@opt@pyconfuture` The `pyconfuture` option determines what is automatically imported from the `__future__` module under Python 2, for `console` content. It has no effect under Python 3.

```
\def\pytx@opt@pyconfuture{none}
\pgfkeys{/PYTX/pkgopt/pyconfuture/.is choice}
\pgfkeys{/PYTX/pkgopt/pyconfuture/default/.code=\def\pytx@opt@pyconfuture{default}}
\pgfkeys{/PYTX/pkgopt/pyconfuture/all/.code=\def\pytx@opt@pyconfuture{all}}
\pgfkeys{/PYTX/pkgopt/pyconfuture/none/.code=\def\pytx@opt@pyconfuture{none}}
```

10.3.12 Upquote

`pytx@opt@upquote` The `upquote` option determines whether the `upquote` package is loaded. It makes quotes within verbatim contexts `'` rather than `'`. This is important, because it means that code may be copied directly from the compiled PDF and executed without any errors due to quotes `'` being copied as acute accents ´.

```
\newbool{pytx@opt@upquote}
\booltrue{pytx@opt@upquote}
\pgfkeys{/PYTX/pkgopt/upquote/.default=true}
\pgfkeys{/PYTX/pkgopt/upquote/.is choice}
\pgfkeys{/PYTX/pkgopt/upquote/true/.code=\booltrue{pytx@opt@upquote}}
\pgfkeys{/PYTX/pkgopt/upquote/false/.code=\boolfalse{pytx@opt@upquote}}
```

10.3.13 Fix math spacing

`pytx@opt@fixlr` The `fixlr` option fixes extra, undesirable spacing in mathematical formulae introduced by the commands `\left` and `\right`. For example, compare the results of `$\sin(x)$` and `$\sin\left(x\right)$`: $\sin(x)$ and $\sin\left(x\right)$. The `fixlr` option fixes this, using a solution proposed by Mateus Araújo, Philipp Stephani, and Heiko Oberdiek.[37]

```
\newbool{pytx@opt@fixlr}
\pgfkeys{/PYTX/pkgopt/fixlr/.default=true}
\pgfkeys{/PYTX/pkgopt/fixlr/.is choice}
\pgfkeys{/PYTX/pkgopt/fixlr/true/.code=\booltrue{pytx@opt@fixlr}}
\pgfkeys{/PYTX/pkgopt/fixlr/false/.code=\boolfalse{pytx@opt@fixlr}}
```

10.3.14 Keep temporary files

`\pytx@opt@keeptemps` By default, PythonTEX tries to be very tidy. It creates many temporary files, but deletes all that are not required to compile the document, keeping the overall file count very low. At times, particularly during debugging, it may be useful to keep

[37] http://tex.stackexchange.com/questions/2607/spacing-around-left-and-right

these temporary files, so that code, errors, and output may be examined more directly. The `keeptemps` option makes this possible.

```
\def\pytx@opt@keeptemps{none}
\pgfkeys{/PYTX/pkgopt/keeptemps/.default=all}
\pgfkeys{/PYTX/pkgopt/keeptemps/.is choice}
\pgfkeys{/PYTX/pkgopt/keeptemps/all/.code=\def\pytx@opt@keeptemps{all}}
\pgfkeys{/PYTX/pkgopt/keeptemps/code/.code=\def\pytx@opt@keeptemps{code}}
\pgfkeys{/PYTX/pkgopt/keeptemps/none/.code=\def\pytx@opt@keeptemps{none}}
```

10.3.15 Pygments

`pytx@opt@pygments` By default, PythonTEX uses `fancyvrb` to typeset code. This provides nice formatting and font options, but no syntax highlighting. The `prettyprinter` options, and `pygments` alias, determine whether Pygments or `fancyvrb` is used to typeset code. Pygments is a generic syntax highlighter written in Python. Since PythonTEX sends code to Python anyway, having Pygments process the code is only a small additional step and in many cases takes little if any extra time to execute.[38]

Command and environment families obey the `prettyprinter` option by default, but they may be set to override it and always use Pygments or always use `fancyvrb`, via `\setpythontexprettyprinter` and `\setpygmentsprettyprinter`.

```
\newbool{pytx@opt@pygments}
\booltrue{pytx@opt@pygments}
\pgfkeys{/PYTX/pkgopt/prettyprinter/.is choice}
\pgfkeys{/PYTX/pkgopt/prettyprinter/pygments/.code=\booltrue{pytx@opt@pygments}}
\pgfkeys{/PYTX/pkgopt/prettyprinter/fancyvrb/.code=\boolfalse{pytx@opt@pygments}}
\pgfkeys{/PYTX/pkgopt/pygments/.default=true}
\pgfkeys{/PYTX/pkgopt/pygments/.is choice}
\pgfkeys{/PYTX/pkgopt/pygments/true/.code=\booltrue{pytx@opt@pygments}}
\pgfkeys{/PYTX/pkgopt/pygments/false/.code=\boolfalse{pytx@opt@pygments}}
```

`pytx@opt@pyginline` This option governs whether, when Pygments is in use, it highlights inline content.

```
\newbool{pytx@opt@pyginline}
\booltrue{pytx@opt@pyginline}
\pgfkeys{/PYTX/pkgopt/prettyprintinline/.default=true}
\pgfkeys{/PYTX/pkgopt/prettyprintinline/.is choice}
\pgfkeys{/PYTX/pkgopt/prettyprintinline/true/.code=\booltrue{pytx@opt@pyginline}}
\pgfkeys{/PYTX/pkgopt/prettyprintinline/false/.code=\boolfalse{pytx@opt@pyginline}}
\pgfkeys{/PYTX/pkgopt/pyginline/.default=true}
\pgfkeys{/PYTX/pkgopt/pyginline/.is choice}
\pgfkeys{/PYTX/pkgopt/pyginline/true/.code=\booltrue{pytx@opt@pyginline}}
\pgfkeys{/PYTX/pkgopt/pyginline/false/.code=\boolfalse{pytx@opt@pyginline}}
```

`pytx@pyglexer` For completeness, we need a way to set the Pygments lexer for all content. Note that in general, resetting the lexers for all content is not desirable.

```
\def\pytx@pyglexer{}
\pgfkeys{/PYTX/pkgopt/pyglexer/.code=\def\pytx@pyglexer{#1}}
```

[38] Pygments code highlighting is executed as a separate process by `pythontex.py`, so it runs in parallel on a multicore system. Pygments usage is optimized by saving highlighted code and only reprocessing it when changed.

\pytx@pygopt We also need a way to specify Pygments options at the package level. This is accomplished via the `pygopt` option: `pygopt`=$\{\langle options \rangle\}$. Note that the options must be enclosed in curly braces since they contain equals signs and thus must be distinguishable from package options.

Currently, three options may be passed in this manner: `style`=$\langle style \rangle$, which sets the formatting style; `texcomments`, which allows LaTeX in code comments to be rendered; and `mathescape`, which allows LaTeX math mode (`$...$`) in comments. The `texcomments` and `mathescape` options may be used with a boolean argument; if an argument is not supplied, true is assumed. As an example of `pygopt` usage, consider the following:

```
pygopt={style=colorful, texcomments=true, mathescape=false}
```

While the package-level `pygments` option may be overridden by individual commands and environments (though it is not by default), the package-level Pygments options cannot be overridden by individual commands and environments. While we're defining storage for `pygopt`, go ahead and define parsing to extract `style` for later use under all cicumstances. This should be reorganized during the next refactoring.

```
\def\pytx@pygopt{}
\pgfkeys{/PYTX/pkgopt/pygopt/.code=\def\pytx@pygopt{#1}\pgfkeys{/PYTX/gopt/pygopt/.cd, #1}}
\pgfkeys{/PYTX/gopt/pygopt/.is choice}
\pgfkeys{/PYTX/gopt/pygopt/texcomments/.code=\relax}
\pgfkeys{/PYTX/gopt/pygopt/mathescape/.code=\relax}
\pgfkeys{/PYTX/gopt/pygopt/style/.code=\ifstrempty{#1}{}{\def\pytx@style{#1}}}
\pgfkeys{/PYTX/lopt/pygopt/.is choice}
\pgfkeys{/PYTX/lopt/pygopt/name/.code=\def\pytx@tmp@name{#1}}
\pgfkeys{/PYTX/lopt/pygopt/texcomments/.code=\relax}
\pgfkeys{/PYTX/lopt/pygopt/mathescape/.code=\relax}
\pgfkeys{/PYTX/lopt/pygopt/style/.code=\ifstrempty{#1}{}{%
  \expandafter\def\csname pytx@style@\pytx@tmp@name\endcsname{#1}}}
\pgfkeys{/PYTX/popt/pygopt/.is choice}
\pgfkeys{/PYTX/popt/pygopt/name/.code=\def\pytx@tmp@name{#1}}
\pgfkeys{/PYTX/popt/pygopt/texcomments/.code=\relax}
\pgfkeys{/PYTX/popt/pygopt/mathescape/.code=\relax}
\pgfkeys{/PYTX/popt/pygopt/style/.code=\ifstrempty{#1}{}{%
  \expandafter\def\csname pytx@style@PYG\pytx@tmp@name\endcsname{#1}}}
```

\pytx@fvextfile By default, code highlighted by Pygments, the `console` environment, and some other content is brought back via `fancyvrb`'s `SaveVerbatim` macro, which saves verbatim content into a macro and then allows it to be restored. This makes it possible for all Pygments content to be brought back in a single file, keeping the total file count low (which is a major priority for PythonTeX!). This approach does have a disadvantage, though, because `SaveVerbatim` slows down as the length of saved code increases.[39] To deal with this issue, we create the `fvextfile` option. This option takes an integer, `fvextfile`=$\langle integer \rangle$. All content that is more than $\langle integer \rangle$ lines long will be saved to its own external file and inputted from there, rather than saved and restored via `SaveVerbatim` and `UseVerbatim`. This provides a workaround should speed ever become a hindrance for large blocks of code.

[39] The macro in which code is saved is created by grabbing the code one line at a time, and for each line redefining the macro to be its old value with the additional line tacked on. This is rather inefficient, but apparently there isn't a good alternative.

A default value of 25 is set. There is nothing special about 25; it is just a relatively reasonably cutoff. If the option is unused, it has a value of -1, which is converted to the maximum integer on the Python side.

```
\def\pytx@fvextfile{-1}
\pgfkeys{/PYTX/pkgopt/fvextfile/.default=25}
\pgfkeys{/PYTX/pkgopt/fvextfile/.code=\IfInteger{#1}{%
    \ifnum#1>0\relax
        \def\pytx@fvextfile{#1}%
    \else
        \PackageError{\pytx@packagename}{option fvextfile must be an integer > 0}{}%
    \fi}%
    {\PackageError{\pytx@packagename}{option fvextfile must be an integer > 0}{}}%
}
```

10.3.16 Python console environment

`\pytx@opt@pyconbanner` This option governs the appearance (or disappearance) of a banner at the beginning of Python console environments. The options `none` (no banner), `standard` (standard Python banner), `default` (default banner for Python's `code` module, standard banner plus interactive console class name), and `pyversion` (banner in the form `Python x.y.z`) are accepted.

```
\def\pytx@opt@pyconbanner{none}
\pgfkeys{/PYTX/pkgopt/pyconbanner/.is choice}
\pgfkeys{/PYTX/pkgopt/pyconbanner/none/.code=\def\pytx@opt@pyconbanner{none}}
\pgfkeys{/PYTX/pkgopt/pyconbanner/standard/.code=\def\pytx@opt@pyconbanner{standard}}
\pgfkeys{/PYTX/pkgopt/pyconbanner/default/.code=\def\pytx@opt@pyconbanner{default}}
\pgfkeys{/PYTX/pkgopt/pyconbanner/pyversion/.code=\def\pytx@opt@pyconbanner{pyversion}}
```

`\pytx@opt@pyconfilename` This option governs the file name that appears in error messages in the console. The file name may be either `stdin`, as it is in a standard interactive interpreter, or `console`, as it would typically be for the Python `code` module.

```
Traceback (most recent call last):
  File "<file name>", line <line no>, in <module>
```

```
\def\pytx@opt@pyconfilename{stdin}
\pgfkeys{/PYTX/pkgopt/pyconfilename/.is choice}
\pgfkeys{/PYTX/pkgopt/pyconfilename/stdin/.code=\def\pytx@opt@pyconfilename{stdin}}
\pgfkeys{/PYTX/pkgopt/pyconfilename/console/.code=\def\pytx@opt@pyconfilename{console}}
```

10.3.17 depythontex

`pytx@opt@depythontex` This option governs whether PythonTeX saved data that can be used to create a version of the .tex file that does not require PythonTeX to be compiled. This option should be useful for converting a PythonTeX document into a more standard TeX document when sharing or publishing documents.

While we're at it, we go ahead and define dummy versions of the `depythontex` macros, so that they can be used in defining commands that are used within the package, not just outside of it.

```
\newbool{pytx@opt@depythontex}
\pgfkeys{/PYTX/pkgopt/depythontex/.default=true}
\pgfkeys{/PYTX/pkgopt/depythontex/.is choice}
```

```
\pgfkeys{/PYTX/pkgopt/depythontex/true/.code=\booltrue{pytx@opt@depythontex}}
\pgfkeys{/PYTX/pkgopt/depythontex/false/.code=\boolfalse{pytx@opt@depythontex}}
\let\Depythontex\@gobble
\let\DepyFile\@gobble
\let\DepyMacro\@gobble
\let\DepyListing\@empty
```

10.3.18 Process options

Now we process the package options.

```
\ProcessPgfPackageOptions{/PYTX/pkgopt}
```

The `fixlr` option only affects one thing, so we go ahead and take care of that. Notice that before we patch `\left` and `\right`, we make sure that they have not already been patched by checking how `\left` is expanded. This is important if the user has manually patched these commands, is using the `mleftright` package, or accidentally loads PythonTeX twice.

```
\ifbool{pytx@opt@fixlr}{
    \IfStrEq{\detokenize\expandafter{\left}}{\detokenize{\left}}{
        \let\originalleft\left
        \let\originalright\right
        \renewcommand{\left}{\mathopen{}\mathclose\bgroup\originalleft}
        \renewcommand{\right}{\aftergroup\egroup\originalright}
    }{}
}{}
```

Likewise, the `upquote` option.

```
\ifbool{pytx@opt@upquote}{\RequirePackage{upquote}}{}
```

If the `depythontex` option is used, we also need to disable Pygments highlighting. This is necessary because some content, such as `console` environments, is needed in a non-highlighted form, so that it will not contain any special macros.

```
\ifbool{pytx@opt@depythontex}{\boolfalse{pytx@opt@pygments}}{}
```

10.4 Utility macros and input/output setup

Once options are processed, we proceed to define a number of utility macros and setup the file input/output that is required by PythonTeX. We also create macros and perform setup needed by depythontex, since these are closely related to input/output.

10.4.1 Automatic counter creation

`\pytx@CheckCounter` We will be using counters to give each command/environment a unique identifier, as well as to manage line numbering of code when desired. We don't know the names of the counters ahead of time (this is actually determined by the user's naming of code sessions), so we need a macro that checks whether a counter exists, and if not, creates it.

```
\def\pytx@CheckCounter#1{%
    \ifcsname c@#1\endcsname\else\newcounter{#1}\fi
}
```

10.4.2 Saving verbatim content in macros

\pytx@SVMCR Commands like \py bring in string representations of objects. Printed content is saved to external files, but commands like \py bring in content by saving it in macros. A single large file of macro definitions is brought in, rather than many external files.

This prevents the creation of unnecessary files, but it also has a significant drawback: only some content can be saved in a standard macro. In particular, verbatim content using \verb and verbatim will not work. So we need a way to save anything in a macro. The solution is to create a special macro that captures its argument verbatim. The argument is then tokenized when it is used via \scantokens. All of this requires a certain amount of catcode trickery.

```
\def\pytx@SVMCR#1{%
    \edef\pytx@tmp{\csname #1\endcsname}%
    \begingroup
    \endlinechar`\^^J
    \let\do\@makeother\dospecials
    \pytx@SVMCR@i}
\begingroup
\catcode`!=0
!catcode`!\=12
!long!gdef!pytx@SVMCR@i#1\endpytx@SVMCR^^J{%
    !endgroup
    !expandafter!gdef!pytx@tmp{%
        !expandafter!scantokens!expandafter{#1!empty}}%
}%
!endgroup
```

pytx@Verbatim pytx@SaveVerbatim We need custom versions of fancyvrb's Verbatim and SaveVerbatim environments, because we don't want to have to worry about the possibility of these environments containing literal Verbatim and SaveVerbatim environments.

```
\DefineVerbatimEnvironment{pytx@Verbatim}{Verbatim}{}
\DefineVerbatimEnvironment{pytx@SaveVerbatim}{SaveVerbatim}{}
```

10.4.3 Code context

\pytx@context \pytx@SetContext \setpythontexcontext It would be nice if when our code is executed, we could know something about its context, such as the style of its surroundings or information about page size.

By default, no contextual information is passed to LaTeX. There is a wide variety of information that could be passed, but most use cases would only need a very specific subset. Instead, the user can customize what information is passed to LaTeX. The \setpythontexcontext macro defines what is passed. It creates the \pytx@SetContext macro, which creates \pytx@context, in which the expanded context information is stored. The context should only be defined in the preamble, so that it is consistent throughout the document.

If you are interested in typesetting mathematics based on math styles, you should use the \mathchoice macro rather than attempting to pass contextual information.

```
\newcommand{\setpythontexcontext}[1]{%
    \Depythontex{cmd:setpythontexcontext:m:n}%
    \def\pytx@SetContext{%
        \edef\pytx@context{#1}%
```

```
    }%
}
\setpythontexcontext{}
\@onlypreamble\setpythontexcontext
```

10.4.4 Code groups

By default, PythonTEX executes code based on sessions. All of the code entered within a command and environment family is divided based on sessions, and each session is saved to a single external file and executed. If you have a calculation that will take a while, you can simply give it its own named session, and then the code will only be executed when there is a change within that session.

While this approach is appropriate for many scenarios, it is sometimes inefficient. For example, suppose you are writing a document with multiple chapters, and each chapter needs its own session. You could manually do this, but that would involve a lot of commands like `\py[chapter x]{`⟨*some code*⟩`}`, which means lots of extra typing and extra session names. So we need a way to subdivide or restart sessions, based on context such as chapter, section, or subsection.

"Groups" provide a solution to this problem. Each session is subdivided based on groups behind the scenes. By default, this changes nothing, because each session is put into a single default group. But the user can redefine groups based on chapter, section, and other counters, so that sessions are automatically subdivided accordingly. Note that there is no continuity between sessions thus subdivided. For example, if you set groups to change between chapters, there will be no continuity between the code of those chapters, even if all the code is within the same named session. If you require continuity, the groups approach is probably not appropriate. You could consider saving results at the end of one chapter and loading them at the beginning of the next, but that introduces additional issues in keeping all code properly synchronized, since code is executed only when it changes, not when any data it loads may have changed.

`\restartpythontexsession` `\pytx@group` `\pytx@SetGroup` `\pytx@SetGroupVerb` `\pytx@SetGroupCons`

We begin by creating the `\restartpythontexsession` macro. It creates the `\pytx@SetGroup*` macros, which create `\pytx@group`, in which the expanded context information is stored. The context should only be defined in the preamble, so that it is consistent throughout the document. Note that groups should be defined so that they will only contain characters that are valid in file names, because groups are used in naming temporary files. It is also a good idea to avoid using periods, since LaTeX input of file names containing multiple periods can sometimes be tricky. For best results, use A-Z, a-z, 0-9, and the hyphen and underscore characters to define groups. If groups contain numbers from multiple sources (for example, chapter and section), the numbers should be separated by a non-numeric character to prevent unexpected collisions (for example, distinguishing chapter 1-11 from 11-1). For example, `\restartpythontexsession{\arabic{chapter}-\arabic{section}}` could be a good approach.

Three forms of `\pytx@SetGroup*` are provided. `\pytx@SetGroup` is for general code use. `\pytx@SetGroupVerb` is for use in verbatim contexts. It splits verbatim content off into its own group. That way, verbatim content does not affect the instance numbers of code that is actually executed. This prevents code from needing to be run every time verbatim content is added or removed; code is only executed when it is actually changed. `pytx@SetGroupCons` is for `console` environments. It

separate console content from executed code and from verbatim content, again for efficiency reasons.

```
\newcommand{\restartpythontexsession}[1]{%
    \Depythontex{cmd:restartpythontexsession:m:n}%
    \def\pytx@SetGroup{%
        \edef\pytx@group{#1}%
    }%
    \def\pytx@SetGroupVerb{%
        \edef\pytx@group{#1verb}%
    }%
    \def\pytx@SetGroupCons{%
        \edef\pytx@group{#1cons}%
    }%
    \AtBeginDocument{%
        \pytx@SetGroup
        \IfSubStr{\pytx@group}{verb}{%
            \PackageError{\pytx@packagename}%
                {String "verb" is not allowed in \string\restartpythontexsession}%
                {Use \string\restartpythontexsession with a valid argument}}{}%
        \IfSubStr{\pytx@group}{cons}{%
            \PackageError{\pytx@packagename}%
                {String "cons" is not allowed in \string\restartpythontexsession}%
                {Use \string\restartpythontexsession with a valid argument}}{}%
    }%
}
```

For the sake of consistency, we only allow group behaviour to be set in the preamble. And if the group is not set by the user, then we use a single default group for each session.

```
\@onlypreamble\restartpythontexsession
\restartpythontexsession{default}
```

10.4.5 File input and output

`\pytx@jobname` We will need to create directories and files for PythonTeX output, and some of these will need to be named using `\jobname`. This presents a problem. Ideally, the user will choose a job name that does not contain spaces. But if the job name does contain spaces, then we may have problems bringing in content from a directory or file that is named based on the job, due to the space characters. So we need a "sanitized" version of `\jobname`. We replace spaces with hyphens. We replace double quotes `"` with nothing. Double quotes are placed around job names containing spaces by TeX Live, and thus may be the first and last characters of `\jobname`. Since we are replacing any spaces with hyphens, quote delimiting is no longer needed, and in any case, some operating systems (Windows) balk at creating directories or files with names containing double quotes. We also replace asterisks with hyphens, since MiKTeX (at least v. 2.9) apparently replaces spaces with asterisks in `\jobname`,[40] and some operating systems may not be happy with names containing asterisks.

This approach to "sanitizing" `\jobname` is not foolproof. If there are ever two files in a directory that both use PythonTeX, and if their names only differ by these

[40] http://tex.stackexchange.com/questions/14949/why-does-jobname-give-s-instead-of-spaces-and-how-do-i-fix-this

substitutions for spaces, quotes, and asterisks, then the output of the two files will collide. We believe that it is better to graciously handle the possibility of space characters at the expense of nearly identical file names, since nearly identical file names are arguably a much worse practice than file names containing spaces, and since such nearly identical file names should be much rarer. At the same time, in rare cases a collision might occur, and in even rarer cases it might go unnoticed.[41] To prevent such issues, `pythontex.py` checks for collisions and issues a warning if a potential collision is detected.

```
\StrSubstitute{\jobname}{ }{-}[\pytx@jobname]
\StrSubstitute{\pytx@jobname}{"}{}[\pytx@jobname]
\StrSubstitute{\pytx@jobname}{*}{-}[\pytx@jobname]
```

`\pytx@outputdir`
`\setpythontexoutputdir`

To keep things tidy, all PythonTEX files are stored in a directory that is created in the document root directory. By default, this directory is called `pythontex-files-`⟨*sanitized jobname*⟩, but we want to provide the user with the option to customize this. For example, when ⟨*sanitized jobname*⟩ is very long, it might be convenient to use `pythontex-`⟨*abbreviated name*⟩.

The command `\setpythontexoutputdir` stores the name of PythonTEX's output directory in `\pytx@outputdir`. The command `\setpythontexoutputdir` is only allowed in the preamble, because the location of PythonTEX content should be specified before the body of the document is typeset.

```
\def\pytx@outputdir{pythontex-files-\pytx@jobname}
\newcommand{\setpythontexoutputdir}[1]{%
    \Depythontex{cmd:setpythontexoutputdir:m:n}%
    \def\pytx@outputdir{#1}}
\@onlypreamble\setpythontexoutputdir
```

`pytx@workingdir`
`\setpythontexworkingdir`

We need to be able to set the current working directory for the scripts executed by PythonTEX. By default, the working directory should be the same as the document root directory. But in some cases the user may wish to specify a different working directory. We want to be able to use "`<outputdir>`" as a shortcut for setting the working directory to the output directory.

If the `graphicx` package is loaded, and the output directory is being used as the working directory, then the output directory is added to the graphics path at the beginning of the document, so that files in the output directory may be included within the main document without the necessity of specifying path information.

```
\def\pytx@workingdir{.}
\def\pytx@workingdirset{false}
\newcommand{\setpythontexworkingdir}[1]{%
    \Depythontex{cmd:setpythontexworkingdir:m:n}%
    \def\pytx@workingdir{#1}%
    \def\pytx@workingdirset{true}%
}
\@onlypreamble\setpythontexworkingdir
\AtBeginDocument{%
    \ifdefstring{\pytx@workingdir}{<outputdir>}%
        {\let\pytx@workingdir\pytx@outputdir}{}%
```

[41] In general, a collision would produce errors, and the user would thereby become aware of the collision. The dangerous case is when the two files with similar names use exactly the same PythonTEX commands, the same number of times, so that the naming of the output is identical. In that case, no errors would be issued.

```
    \ifdefstrequal{\pytx@workingdir}{\pytx@outputdir}{%
        \@ifpackageloaded{graphicx}{%
            \ifx\Ginput@path\@undefined
                \graphicspath{{\pytx@outputdir/}}%
            \else
                \g@addto@macro\Ginput@path{{\pytx@outputdir/}}%
            \fi
        }{}%
    }{}%
}
```

`pytx@usedpygments` Once we have specified the output directory, we are free to pull in content from it. Most content from the output directory will be pulled in manually by the user (for example, via `\includegraphics`) or automatically by PythonTEX as it goes along. But content "printed" by code commands and environments (via macros) as well as code typeset by Pygments needs to be included conditionally, based on whether it exists and on user preferences.

This gets a little tricky. We only want to pull in the Pygments content if it is actually used, since Pygments content will typically use `fancyvrb`'s `SaveVerb` environment, and this can slow down compilation when very large chunks of code are saved. It doesn't matter if the code is actually used; saving it in a macro is what potentially slows things down. So we create a bool to keep track of whether Pygments is ever actually used, and only bring in Pygments content if it is.[42] This bool must be set to `true` whenever a command or environment is created that makes use of Pygments (in practice, we will simply set it to true when a family is created). Note that we cannot use the `pytx@opt@pygments` bool for this purpose, because it only tells us if the package option for Pygments usage is `true` or `false`. Typically, this will determine if any Pygments content is used. But it is possible for the user to create a command and environment family that overrides the package option (indeed, this may sometimes be desirable, for example, if the user wishes code in a particular language never to be highlighted). Thus, a new bool is needed to allow detection in such nonstandard cases.

```
\newbool{pytx@usedpygments}
```

Now we can conditionally bring in the Pygments content. Note that we must use the `etoolbox` macro `\AfterEndPreamble`. This is because commands and environments are created using `\AtBeginDocument`, so that the user can change their properties in the preamble before they are created. And since the commands and environments must be created before we know the final state of `pytx@usedpygments`, we must bring in Pygments content after that. We typically need to patch the Pygments single quote macro so that it cooperates with `upquote`.

```
\AfterEndPreamble{%
  \ifbool{pytx@usedpygments}%
```

[42] The same effect could be achieved by having `pythontex.py` delete the Pygments content whenever it is run and Pygments is not used. But that approach is faulty in two regards. First, it requires that `pythontex.py` be run, which is not necessarily the case if the user simply sets the package option `pygments` to `false` and the recompiles. Second, even if it could be guaranteed that the content would be deleted, such an approach would not be optimal. It is quite possible that the user wishes to temporarily turn off Pygments usage to speed compilation while working on other parts of the document. In this case, deleting the Pygments content is simply deleting data that must be recreated when Pygments is turned back on.

```
    {\InputIfFileExists{\pytx@outputdir/\pytx@jobname.pytxpyg}{}{}%
     \ifcsname PYGZsq\endcsname
       \ifdefstring{\PYGZsq}{\char`\'}{\pytx@patch@PYGZsq}{}%
     \fi}%
    {}%
}
\begingroup
\catcode`\'=\active
\gdef\pytx@patch@PYGZsq{\gdef\PYGZsq{'}}
\endgroup
```

While we are pulling in content, we also pull in the file of macros that stores some inline "printed" content, if the file exists. Since we need this file in general, and since it will not typically invole a noticeable speed penalty, we bring it in at the beginning of the document without any special conditions.

```
\AtBeginDocument{%
    \makeatletter
    \InputIfFileExists{\pytx@outputdir/\pytx@jobname.pytxmcr}{}%
        {\ifstrempty{\pytx@outputdir}%
            {\typeout{No file \pytx@jobname.pytxmcr.}}%
            {}%
         \IfStrEq{\pytx@outputdir}{.}%
            {\typeout{No file \pytx@jobname.pytxmcr.}}%
            {\typeout{No file \pytx@outputdir/\pytx@jobname.pytxmcr.}}%
         \typeout{Run \pytx@packagename\space to create it.}}%
    \makeatother
}
```

`\pytx@codefile` We create a new write, named `\pytx@codefile`, to which we will save code. All the code from the document will be written to this single file, interspersed with information specifying where in the document it came from. PythonTeX parses this file to separate the code into individual sessions and groups. These are then executed, and the identifying information is used to tie code output back to the original code in the document.[43]

```
\newwrite\pytx@codefile
\immediate\openout\pytx@codefile=\jobname.pytxcode
```

In the code file, information from PythonTeX must be interspersed with the code. Some type of delimiting is needed for PythonTeX information. All PythonTeX content is written to the file in the form `=>PYTHONTEX#`⟨*content*⟩`#`. When this content involves package options, the delimiter is modified to the form `=>PYTHONTEX:SETTINGS#`⟨*content*⟩`#`. The `#` symbol is also used as a subdelimiter within ⟨*content*⟩. The `#` symbol is convenient as a delimiter since it has a

[43] The choice to write all code to a single file is the result of two factors. First, TeX has a limited number of output registers available (16), so having a separate output stream for each group or session is not possible. The `morewrites` package from Bruno Le Floch potentially removes this obstacle, but since this package is very recent (README from 2011/7/10), we will not consider using additional writes in the immediate future. Second, one of the design goals of PythonTeX is to minimize the number of persistent files created by a run. This keeps directories cleaner and makes file synchronization/transfer somewhat simpler. Using one write per session or group could result in numerous code files, and these could only be cleaned up by `pythontex.py` since LaTeX cannot delete files itself (well, without unrestricted `write18`). Using a single output file for code does introduce a speed penalty since the code does not come pre-sorted by session or group, but in typical usage this should be minimal. Adding an option for single or multiple code files may be something to reconsider at a later date.

special meaning in TeX and is very unlikely to be accidentally entered by the user in unexpected locations without producing errors. Note that the usage of "`=>PYTHONTEX#`" as a beginning delimiter for PythonTeX data means that this string should **never** be written by the user at the beginning of a line, because `pythontex.py` will try to intepret it as data and will fail.

`\pytx@delimchar` We create a macro to store the delimiting character.

```
\edef\pytx@delimchar{\string#}
```

`\pytx@delim` We create a macro to store the starting delimiter.

```
\edef\pytx@delim{=\string>PYTHONTEX\string#}
```

`\pytx@delimsettings` And we create a second macro to store the starting delimiter for settings that are passed to Python.

```
\edef\pytx@delimsettings{=\string>PYTHONTEX:SETTINGS\string#}
```

Settings need to be written to the code file. Some of these settings are not final until the beginning of the document, since they may be modified by the user within the preamble. Thus, all settings should be written at the end of the document, so that they will all be together and will not be interspersed with any code that was entered in the preamble. The order in which the settings are written is not significant, but for symmetry it should mirror the order in which they were defined.

```
\AtEndDocument{%
    \immediate\write\pytx@codefile{\pytx@delimsettings}%
    \immediate\write\pytx@codefile{version=\pytx@packageversion}%
    \immediate\write\pytx@codefile{outputdir=\pytx@outputdir}%
    \immediate\write\pytx@codefile{workingdir=\pytx@workingdir}%
    \immediate\write\pytx@codefile{workingdirset=\pytx@workingdirset}%
    \immediate\write\pytx@codefile{gobble=\pytx@opt@gobble}%
    \immediate\write\pytx@codefile{rerun=\pytx@opt@rerun}%
    \immediate\write\pytx@codefile{hashdependencies=\pytx@opt@hashdependencies}%
    \immediate\write\pytx@codefile{makestderr=\ifbool{pytx@opt@stderr}{true}{false}}%
    \immediate\write\pytx@codefile{stderrfilename=\pytx@opt@stderrfilename}%
    \immediate\write\pytx@codefile{keeptemps=\pytx@opt@keeptemps}%
    \immediate\write\pytx@codefile{pyfuture=\pytx@opt@pyfuture}%
    \immediate\write\pytx@codefile{pyconfuture=\pytx@opt@pyconfuture}%
    \immediate\write\pytx@codefile{pygments=\ifbool{pytx@opt@pygments}{true}{false}}%
    \immediate\write\pytx@codefile{pygglobal=:GLOBAL|\pytx@pyglexer|\pytx@pygopt}%
    \immediate\write\pytx@codefile{fvextfile=\pytx@fvextfile}%
    \immediate\write\pytx@codefile{pyconbanner=\pytx@opt@pyconbanner}%
    \immediate\write\pytx@codefile{pyconfilename=\pytx@opt@pyconfilename}%
    \immediate\write\pytx@codefile{depythontex=\ifbool{pytx@opt@depythontex}{true}{false}}%
}
```

`\pytx@WriteCodefileInfo` `\pytx@WriteCodefileInfoExt` Later, we will frequently need to write PythonTeX information to the code file in standardized form. We create a macro to simplify that process. We also create an alternate form, for use with external files that must be inputted or read in by PythonTeX and processed.[44]

[44] The external-file form also takes an optional argument. This corresponds to a command-line argument that is passed to an external file during the file's execution. Currently, executing external files, with or without arguments, is not implemented. But this feature is under consideration, and the macro retains the optional argument for the potential future compatibility. Originally, the external version used a fixed instance, but that conflicted with the `fancyvrb` options `firstline` and `lastline`, so instances had to be added.

```
\def\pytx@argsrun{}
\def\pytx@argspprint{}
\def\pytx@WriteCodefileInfo{%
    \ifcurrfile{\currfilebase}{\jobname}%
        {\let\pytx@currfile\@empty}{\let\pytx@currfile\currfilename}%
    \immediate\write\pytx@codefile{\pytx@delim\pytx@type\pytx@delimchar%
        \pytx@session\pytx@delimchar\pytx@group\pytx@delimchar%
        \arabic{\pytx@counter}\pytx@delimchar\pytx@cmd\pytx@delimchar%
        \pytx@context\pytx@delimchar\pytx@argsrun\pytx@delimchar%
        \pytx@argspprint\pytx@delimchar%
        \pytx@currfile\pytx@delimchar%
        \the\inputlineno\pytx@delimchar}%
}
\newcommand{\pytx@WriteCodefileInfoExt}[1][]{%
    \ifcurrfile{\currfilebase}{\jobname}%
            {\let\pytx@currfile\@empty}{\let\pytx@currfile\currfilename}%
    \immediate\write\pytx@codefile{\pytx@delim\pytx@type\pytx@delimchar%
        \pytx@session\pytx@delimchar\pytx@group\pytx@delimchar%
        \arabic{\pytx@counter}\pytx@delimchar\pytx@cmd\pytx@delimchar%
        \pytx@context\pytx@delimchar\pytx@argsrun\pytx@delimchar%
        \pytx@argspprint\pytx@delimchar%
        \pytx@currfile\pytx@delimchar%
        \the\inputlineno\pytx@delimchar#1}%
}
```

10.4.6 Interface to `fancyvrb`

The `fancyvrb` package is used to typeset lines of code, and its internals are also used to format inline code snippets. We need a way for each family of PythonTeX commands and environments to have its own independent `fancyvrb` settings.

`\pytx@fvsettings` `\setpythontexfv` The macro `\setpythontexfv[`⟨*family*⟩`]{`⟨*settings*⟩`}` takes ⟨*settings*⟩ and stores them in a macro that is run through `fancyvrb`'s `\fvset` at the beginning of PythonTeX code. If a ⟨*family*⟩ is specified, the settings are stored in `\pytx@fvsettings@`⟨*family*⟩, and the settings only apply to typeset code belonging to that family. If no optional argument is given, then the settings are stored in `\pytx@fvsettings`, and the settings apply to all typeset code.

In the current implementation, `\setpythontexfv` and `\fvset` differ because the former is not persistent in the same sense as the latter. If we use `\fvset` to set one property, and then use it later to set another property, the setting for the original property is persistent. It remains until another `\fvset` command is issued to change it. In contrast, every time `\setpythontexfv` is used, it clears all prior settings and only the current settings actually apply. This is because `\fvset` stores the state of each setting in its own macro, while `\setpythontexfv` simply stores a string of settings that is passed to `\fvset` at the appropriate times. For typical use scenarios, this distinction shouldn't be important—usually, we will want to set the behavior of `fancyvrb` for all PythonTeX content, or for a family of PythonTeX content, and leave those settings constant throughout the document. Furthermore, environments that typeset code take `fancyvrb` commands as their second optional argument, so there is already a mechanism in place for changing the settings for a single environment. However, if we ever want to change the typesetting of code for only a small portion of a document (larger than a single

environment), this persistence distinction does become important.[45]

```
\newcommand{\setpythontexfv}[2][]{%
    \Depythontex{cmd:setpythontexfv:om:n}%
    \ifstrempty{#1}%
        {\gdef\pytx@fvsettings{#2}}%
        {\expandafter\gdef\csname pytx@fvsettings@#1\endcsname{#2}}%
}%
```

Now that we have a mechanism for applying global settings to typeset PythonTeX code, we go ahead and set a default tab size for all environments. If `\setpythontexfv` is ever invoked, this setting will be overwritten, so that must be kept in mind.

```
\setpythontexfv{tabsize=4}
```

`\pytx@FVSet` Once the `fancyvrb` settings for PythonTeX are stored in macros, we need a way to actually invoke them. `\pytx@FVSet` applies family-specific settings first, then PythonTeX-wide settings second, so that PythonTeX-wide settings have precedence and will override family-specific settings. Note that by using `\fvset`, we are overwriting `fancyvrb`'s settings. Thus, to keep the settings local to the PythonTeX code, `\pytx@FVSet` must always be used within a `\begingroup ... \endgroup` block.

```
\def\pytx@FVSet{%
    \expandafter\let\expandafter\pytx@fvsettings@@%
        \csname pytx@fvsettings@\pytx@type\endcsname
    \ifdefstring{\pytx@fvsettings@@}{}%
        {}%
        {\expandafter\fvset\expandafter{\pytx@fvsettings@@}}%
    \ifdefstring{\pytx@fvsettings}{}%
        {}%
        {\expandafter\fvset\expandafter{\pytx@fvsettings}}%
}
```

`\pytx@FVB@SaveVerbatim` `pytx@FancyVerbLineTemp` `fancyvrb`'s `SaveVerbatim` environment will be used extensively to include code highlighted by Pygments and other processed content. Unfortunately, when the saved content is included in a document with the corresponding `UseVerbatim`, line numbering does not work correctly. Based on a web search, this appears to be a known bug in `fancyvrb`. We begin by fixing this, which requires patching `fancyvrb`'s `\FVB@SaveVerbatim` and `\FVE@SaveVerbatim`. We create a patched `\pytx@FVB@SaveVerbatim` by inserting `\FV@StepLineNo` and `\FV@CodeLineNo=1` at appropriate locations. We also delete an unnecessary `\gdef\SaveVerbatim@Name{#1}`. Then we create a `\pytx@FVE@SaveVerbatim`, and add code so that the two macros work together to prevent `FancyVerbLine` from incorrectly being incremented within the `SaveVerbatim` environment. This involves using the counter `pytx@FancyVerbLineTemp` to temporarily store the value of `FancyVerbLine`, so that it may be restored to its original value after verbatim content has been saved.

[45] An argument could be made for having `\setpythontexfv` behave exactly like `\fvset`. Properly implementing this behavior would be tricky, because of inheritance issues between PythonTeX-wide and family-specific settings (this is probably a job for `pgfkeys`). Full persistence would likely require a large number of macros and conditionals. At least from the perspective of keeping the code clean and concise, the current approach is superior, and probably introduces minor annoyances at worst.

There is an additional line-numbering issue when the `firstline` option is used with `SaveVerbatim`. This is fixed by globally reseting `\FV@CodeLineNo` to zero. That was originally done in `fancyvrb`, via `\FV@FormattingPrep`, but this macro is commented out in the current version of `fancyvrb`, which throws off line numbering.

Typically, we `\let` our own custom macros to the corresponding macros within `fancyvrb`, but only within a command or environment. In this case, however, we are fixing behavior that should be considered a bug even for normal `fancyvrb` usage. So we let the buggy macros to the patched macros immediately after defining the patched versions.

```
\newcounter{pytx@FancyVerbLineTemp}
\def\pytx@FVB@SaveVerbatim#1{%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \global\FV@CodeLineNo\z@
    \@bsphack
    \begingroup
    \FV@UseKeyValues
    \def\SaveVerbatim@Name{#1}%
    \def\FV@ProcessLine##1{%
        \expandafter\gdef\expandafter\FV@TheVerbatim\expandafter{%
        \FV@TheVerbatim\FV@StepLineNo\FV@ProcessLine{##1}}}%
    \gdef\FV@TheVerbatim{\FV@CodeLineNo=1}%
    \FV@Scan}
\def\pytx@FVE@SaveVerbatim{%
    \expandafter\global\expandafter\let
    \csname FV@SV@\SaveVerbatim@Name\endcsname\FV@TheVerbatim
    \endgroup\@esphack
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}}
\let\FVB@SaveVerbatim\pytx@FVB@SaveVerbatim
\let\FVE@SaveVerbatim\pytx@FVE@SaveVerbatim
```

10.4.7 Enabling `fvextra` support for Pygments macros

`\pytx@ConfigPygments`

The `fvextra` package provides Pygments support. We need a macro that can be used to turn this on at the appropriate points.

```
\def\pytx@ConfigPygments{%
  \def\pytx@currentstyle{default}%
  \ifcsname pytx@style\endcsname
    \let\pytx@currentstyle\pytx@style
  \else
    \ifcsname pytx@style@\pytx@type\endcsname
      \expandafter\let\expandafter\pytx@currentstyle\csname pytx@style@\pytx@type\endcsname
    \fi
  \fi
  \expandafter\let\expandafter\PYG@style\csname PYG\pytx@currentstyle\endcsname
  \VerbatimPygments{\PYG}{\PYG@style}}
```

10.4.8 Access to printed content (stdout)

The `autoprint` package option automatically pulls in printed content from `code` commands and environments. But this does not cover all possible use cases, because we could have print statements/functions in `block` commands and environ-

ments as well. Furthermore, sometimes we may print content, but then desire to bring it back into the document multiple times, without duplicating the code that creates the content. Here, we create a number of macros that allow access to printed content. All macros are created in two identical forms, one based on the name `print` and one based on the name `stdout`. Which macros are used depends on user preference. The macros based on `stdout` provide symmetry with `stderr` access.

`\pytx@stdfile` We begin by defining a macro to hold the base name for stdout and stderr content. The name of this file is updated by most commands and environments so that it stays current.[46] It is important, however, to initially set the name empty for error-checking purposes.

```
\def\pytx@stdfile{}
```

`\pytx@FetchStdoutfile` Now we create a generic macro for bringing in the stdout file. This macro can input the content in verbatim form, applying `fancyvrb` options if present. Usage: `\pytx@FetchStdoutfile[`⟨*mode*⟩`][`⟨*options*⟩`]{`⟨*file path*⟩`}`. We must disable the macro in the event that the `debug` option is false. Also, the warning text should not be included if we are in the preamble.

```
\def\pytx@stdout@warntext{}
\def\pytx@FetchStdoutfile[#1][#2]#3{%
    \IfFileExists{\pytx@outputdir/#3.stdout}{%
        \ifstrempty{#1}{\input{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{raw}{\input{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{verb}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{verbatim}{\VerbatimInput[#2]{\pytx@outputdir/#3.stdout}}{}%
        \DepyFile{p:\pytx@outputdir/#3.stdout:mode=#1}%
    }%
    {\pytx@stdout@warntext
        \PackageWarning{\pytx@packagename}{Non-existent printed content}}%
}
\ifbool{pytx@opt@stdout}{}{\def\pytx@FetchStdoutfile[#1][#2]#3{}}
\AtBeginDocument{\def\pytx@stdout@warntext{\textbf{??~\pytx@packagename~??}}}
```

`\printpythontex` `\stdoutpythontex` We define a macro that pulls in the content of the most recent stdout file, accepting verbatim settings and also `fancyvrb` settings if they are given.

```
\def\stdoutpythontex{%
    \Depythontex{cmd:stdoutpythontex:oo:p}%
    \@ifnextchar[{\pytx@Stdout}{\pytx@Stdout[raw]}%
}
\def\pytx@Stdout[#1]{%
    \@ifnextchar[{\pytx@Stdout@i[#1]}{\pytx@Stdout@i[#1][]}%
}
\def\pytx@Stdout@i[#1][#2]{%
    \pytx@FetchStdoutfile[#1][#2]{\pytx@stdfile}%
}
\def\printpythontex{%
    \Depythontex{cmd:printpythontex:oo:p}%
```

[46] It is only updated by those commands and environments that interact with `pythontex.py` and thus increment a type-session-group counter so that they can be distinguished. `verb` commands and environments that use `fancyvrb` for typesetting do not interact with `pythontex.py`, do not increment a counter, and thus do not update the stdout file.

```
    \@ifnextchar[{\pytx@Stdout}{\pytx@Stdout[raw]}%
}
```

\saveprintpythontex
\savestdoutpythontex

Sometimes, we may wish to use printed content at multiple locations in a document. Because \pytx@stdfile is changed by every command and environment that could print, the printed content that \printpythontex tries to access is constantly changing. Thus, \printpythontex is of use only immediately after content has actually been printed, before any additional PythonTeX commands or environments change the definition of \pytx@stdfile. To get around this, we create \saveprintpythontex{⟨*name*⟩}. This macro saves the current name of \pytx@stdfile so that it is associated with ⟨*name*⟩ and thus can be retrieved later, after \pytx@stdfile has been redefined.

```
\def\savestdoutpythontex{%
    \Depythontex{cmd:savestdoutpythontex:m:n}%
    \savestdoutpythontex@i
}
\def\savestdoutpythontex@i#1{%
    \ifcsname pytx@SVout@#1\endcsname
        \PackageError{\pytx@packagename}%
            {Attempt to save content using an already-defined name}%
            {Use a name that is not already defined}%
    \else
        \expandafter\edef\csname pytx@SVout@#1\endcsname{\pytx@stdfile}%
    \fi
}
\def\saveprintpythontex{%
    \Depythontex{cmd:saveprintpythontex:m:n}%
    \savestdoutpythontex@i
}
```

\useprintpythontex
\usestdoutpythontex

Now that we have saved the current \pytx@stdoutfile under a new, user-chosen name, we need a way to retrieve the content of that file later, using the name.

```
\def\usestdoutpythontex{%
    \Depythontex{cmd:usestdoutpythontex:oom:p}%
    \@ifnextchar[{\pytx@UseStdout}{\pytx@UseStdout[]}%
}
\def\pytx@UseStdout[#1]{%
    \@ifnextchar[{\pytx@UseStdout@i[#1]}{\pytx@UseStdout@i[#1][]}%
}
\def\pytx@UseStdout@i[#1][#2]#3{%
    \ifcsname pytx@SVout@#3\endcsname
        \pytx@FetchStdoutfile[#1][#2]{\csname pytx@SVout@#3\endcsname}%
    \else
        \textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent saved printed content}%
    \fi
}
\def\useprintpythontex{%
    \Depythontex{cmd:useprintpythontex:oom:p}%
    \@ifnextchar[{\pytx@UseStdout}{\pytx@UseStdout[]}%
}
```

10.4.9 Access to stderr

We need access to stderr, if it is enabled via the package `makestderr` option.

Both stdout and stderr share the same base file name, stored in `\pytx@stdfile`. Only the file extensions, .stdout and .stderr, differ.

stderr and stdout are treated identically, except that stderr is brought in verbatim by default, while stdout is brought in raw by default.

`\pytx@FetchStderrfile` Create a generic macro for bringing in the stderr file.

```
\def\pytx@FetchStderrfile[#1][#2]#3{%
    \IfFileExists{\pytx@outputdir/#3.stderr}{%
        \ifstrequal{#1}{raw}{\input{\pytx@outputdir/#3.stderr}}{}%
        \ifstrempty{#1}{\VerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{verb}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{verbatim}{\VerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \DepyFile{p:\pytx@outputdir/#3.stderr:mode=#1}%
    }%
    {\textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent stderr content}}%
}
```

`\stderrpythontex` We define a macro that pulls in the content of the most recent error file, accepting verbatim settings and also `fancyvrb` settings if they are given.

```
\def\stderrpythontex{%
    \Depythontex{cmd:stderrpythontex:oo:p}%
    \@ifnextchar[{\pytx@Stderr}{\pytx@Stderr[verbatim]}%
}
\def\pytx@Stderr[#1]{%
    \@ifnextchar[{\pytx@Stderr@i[#1]}{\pytx@Stderr@i[#1][]}%
}
\def\pytx@Stderr@i[#1][#2]{%
    \pytx@FetchStderrfile[#1][#2]{\pytx@stdfile}%
}
```

A mechanism is provided for saving and later using stderr. This should be used with care, since stderr content may lose some of its meaning if isolated from the larger code context that produced it.

`\savestderrpythontex`

```
\def\savestderrpythontex#1{%
    \Depythontex{cmd:savestderrpythontex:m:n}%
    \ifcsname pytx@SVerr@#1\endcsname
        \PackageError{\pytx@packagename}%
            {Attempt to save content using an already-defined name}%
            {Use a name that is not already defined}%
    \else
        \expandafter\edef\csname pytx@SVerr@#1\endcsname{\pytx@stdfile}%
    \fi
}
```

`\usestderrpythontex`

```
\def\usestderrpythontex{%
    \Depythontex{cmd:usestderrpythontex:oom:p}%
```

```
    \@ifnextchar[{\pytx@UseStderr}{\pytx@UseStderr[verb]}%
}
\def\pytx@UseStderr[#1]{%
    \@ifnextchar[{\pytx@UseStderr@i[#1]}{\pytx@UseStderr@i[#1][]}%
}
\def\pytx@UseStderr@i[#1][#2]#3{%
    \ifcsname pytx@SVerr@#3\endcsname
        \pytx@FetchStderrfile[#1][#2]{\csname pytx@SVerr@#3\endcsname}%
    \else
        \textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent saved stderr content}%
    \fi
}
```

10.4.10 `depythontex`

The purpose of `depythontex` is to create a version of the original LaTeX document that does not rely on the PythonTeX package. All uses of PythonTeX are replaced by their output. This is particularly useful when submitting a paper to a journal, because PythonTeX can simplify the writing process, but many journals frown upon "special" packages or custom macros. Note that if you just need to share a PythonTeX document with someone, you can always include a copy of `pythontex.sty` and the PythonTeX output directory with the document, and then non-Python parts of the document can be edited just like a normal LaTeX document, without running any Python code.

The general strategy for `depythontex` is to write an auxiliary file that contains information about all environments and macros that need to be replaced, including location, format, and the content with which they are to be replaced. This auxiliary file is then used to performed substitutions on a copy of the original document. It would be possible to simply create a list of all PythonTeX macros and environments, and use that to perform substitutions. But that approach would have to track the state of PythonTeX more carefully than the auxiliary file approach. For example, in the auxiliary file approach, it is easy to track whether `autoprint` is on or off, because commands and environments will write to the auxiliary file if they do indeed use `autoprint`. But without an auxiliary file, you would have to search for `\setpythontexautoprint` and devise an algorithm for determining where it is on or off. Furthermore, once there is a large set of macros, a general search-and-replace could be quite expensive computationally.

These commands need to be defined after all the other settings commands, because some of the other settings commands are used within this package after being defined, and thus don't need replacement because they're in the package. At the same time, the `depythontex` commands have to exist so that other commands can be defined with them. So dummy versions are created earlier. During the next refactoring, the order will be cleaned up and clarified.

`\pytx@depyfile` If the `depythontex` package option is on, we need to open an auxiliary file for writing depythontex information.

```
\ifbool{pytx@opt@depythontex}{%
    \newwrite\pytx@depyfile
    \immediate\openout\pytx@depyfile=\jobname.depytx
}{}
```

\Depythontex Each command or environment that is to work with depythontex will write the following information to the auxiliary file:

```
=>DEPYTHONTEX#<type>:<name>:<args>:<typeset>:<line>:[<Pygments lexer>]#
```

where `<type>` is `cmd` or `env`; `<name>` is the complete name of the command or environment; `<args>` is a string representing the arguments taken (`o`=optional, `m`=mandatory, `v`=mandatory verbatim, `n`=none); `<typeset>` is a string representing what is typeset (`c`=code, `p`=printed, `n`=null), and `<line>` is `\the\inputlineno`. The last one can be determined automatically without user input, but the first four must be entered when a macro is created. Optionally, the Pygments lexer is written to file if it is available (if `\pytx@lexer` is not `\relax`). These pieces of information are needed for the following reasons.

- `<type>` We need to know whether we are dealing with a command or environment, so we know how to deal with it. There is no way to detect this automatically, since a command could always be inside some environment.
- `<name>` We need to know the name of what is to be replaced. There's no way to automatically get this.
- `<args>` We need to know the form of the arguments, so we can assemble an appropriate regular expression. In some cases, a command might be created in such a way that this could be detected or easily passed on to PythonTeX (for example, if the command was defined using the `xparse` package), but in general there isn't a simple way to detect it.
- `<typeset>` Technically, this could be determined from `\pytx@cmd` in many instances. But it couldn't be determined for cases like `\printpythontex` and `\stderrpythontex`. Furthermore, we want a very general interface suitable for users writing custom commands and environments.
- `<line>` This can be determined automatically.
- `<Pygments lexer>` This is needed if so that the language can be specified in the output. In general, `\pytx@lexer` can be defined automatically by a command and environment generator.

We need a command that writes this information to the auxiliary file. Since this command may be employed by users writing custom macros, we choose a capitalized name not containing any ampersands `@`. Since we need to be able to easily disable the macro, we create the real macro with name ending in `@orig`, and then `\let` the intended name to it.

```
\let\pytx@lexer\relax
\def\Depythontex@orig#1{%
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX\pytx@delimchar#1:%
        \the\inputlineno:\ifx\pytx@lexer\relax\else\pytx@lexer\fi\pytx@delimchar}%
    \let\pytx@lexer\relax}
\ifbool{pytx@opt@depythontex}%
    {\let\Depythontex\Depythontex@orig}%
    {\let\Depythontex\@gobble}
\ifbool{pytx@opt@depythontex}{%
\AtEndDocument{%
```

```
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX:SETTINGS\pytx@delimchar version=%
        \pytx@packageversion\pytx@delimchar}%
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX:SETTINGS\pytx@delimchar macrofile=%
        \pytx@outputdir/\pytx@jobname.pytxmcr\pytx@delimchar}%
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX:SETTINGS\pytx@delimchar outputdir=%
        \pytx@outputdir\pytx@delimchar}%
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX:SETTINGS\pytx@delimchar graphicx=%
            \ifcsname graphicspath\endcsname true\else false\fi\pytx@delimchar}%
    \immediate\write\pytx@depyfile{=>DEPYTHONTEX:SETTINGS\pytx@delimchar gobble=%
            \pytx@opt@gobble\pytx@delimchar}%
}%
}{}
```

`\DepyMacro` We need a macro that will write the appropriate information to the auxiliary file if substitution with the contents of a macro is needed. The argument is of the form `<typeset>:<macro name>`, where `<typeset>` is the type of content (`p`=print, `c`=code).

```
\def\DepyMacro@orig#1{%
    \immediate\write\pytx@depyfile{MACRO:#1}%
}
\ifbool{pytx@opt@depythontex}%
    {\let\DepyMacro\DepyMacro@orig}%
    {\let\DepyMacro\@gobble}
```

`\DepyFile` We also need a macro that will write the appropriate information to the auxiliary file if substitution with the contents of a file is needed. As an argument, this command takes `<typeset>:<filename>[:mode=<format>]`, where `<typeset>` is the type of content (`p`=print, `c`=code), `<filename>` is the full filename, and the optional `mode` is the format in which the file is brought in (`raw`, `verb`, `verbatim`). If `mode` is not specified, it defaults to reasonable defaults. In general, `mode` is only needed for `p` content; `c` content is verbatim of one form or another.

```
\def\DepyFile@orig#1{%
    \immediate\write\pytx@depyfile{FILE:#1}%
}%
\ifbool{pytx@opt@depythontex}%
    {\let\DepyFile\DepyFile@orig}%
    {\let\DepyFile\@gobble}
```

`\DepyListing` We need a macro that will write information to the auxiliary file for code listings, specifically whether line numbers were used, and if so, what the starting number was. This is non-trivial, because it is possible to change both of these via an environment's second optional argument. One approach would be to capture all optional arguments, pass the second to `fancyvrb`, and then attempt to evaluate the status of line numbers via an examination of `fancyvrb`'s internals. That approach would require a good deal of work and would likely involve a patch for `fancyvrb`. Instead, we redefine `\theFancyVerbLine`, so that it saves the line number to file the first time it is used, and then redefines itself to its original form.

Since we are redefining `\theFancyVerbLine`, `\DepyListing` can only be used with commands (such as `\inputpygments`) if it is in a group (`\begingroup` ... `\endgroup`). This prevents the redefinition from "escaping," if line numbering is not used. (Environments are wrapped in groups automatically, so this doesn't apply to them.)

```
\newcommand{\pytx@DepyListing@write}{%
    \immediate\write\pytx@depyfile{LISTING:firstnumber=\arabic{FancyVerbLine}}%
}
\def\DepyListing@orig{%
    \let\oldFancyVerbLine\theFancyVerbLine
    \def\theFancyVerbLine{%
        \pytx@DepyListing@write
        \expandafter\gdef\expandafter\theFancyVerbLine\expandafter{\oldFancyVerbLine}%
        \theFancyVerbLine
    }%
}
\ifbool{pytx@opt@depythontex}%
    {\let\DepyListing\DepyListing@orig}%
    {\let\DepyListing\@empty}
```

`\DepythontexOn` `\DepythontexOff` We need a way to switch `depythontex` on and off. When `depythontex` is being used, it needs to be on throughout the entire main document. But it must be switched off for any commands or environments that are brought in via external files (for example, in a package). Since anything that is brought in isn't actually in the text of the main document, substitution is both impossible and unnecessary.

```
\newcommand{\DepythontexOn}{%
    \let\Depythontex\Depythontex@orig
    \let\DepyMacro\DepyMacro@orig
    \let\DepyFile\DepyFile@orig
    \let\DepyListing\DepyListing@orig
}
\newcommand{\DepythontexOff}{%
    \let\Depythontex\@gobble
    \let\DepyMacro\@gobble
    \let\DepyFile\@gobble
    \let\DepyListing\@empty
}
```

10.5 Inline commands

10.5.1 Inline core macros

All inline commands use the same core of inline macros. Inline commands invoke the `\pytx@Inline` macro, and this then branches through a number of additional macros depending on the details of the command and the usage context. `\@ifnextchar` and `\let` are used extensively to control branching.

`\pytx@Inline`, and the macros it calls, perform the following series of operations.

- If there is an optional argument, capture it. The optional argument is the session name of the command. If there is no session name, use the "`default`" session.
- Determine the delimiting character(s) used for the code encompassed by the command. Any character except for the space character and the opening curly brace `{` may be used as a delimiting character, just as for `\verb`. The opening curly brace `{` may be used, but in this case the closing delimiting character is the closing curly brace `}`. If paired curly braces are used as delimiters, then the code enclosed may only contain paired curly braces.

- Using the delimiting character(s), capture the code. Perform some combination of the following tasks: typeset the code, save it to the code file, and bring in content created by the code.

`\pytx@Inline` This is the gateway to all inline core macros. It is called by all inline commands. Because the delimiting characters could be almost anything, we need to turn off all special category codes before we peek ahead with `\@ifnextchar` to see if an optional argument is present, since `\@ifnextchar` sets the category code of the character it examines. But we set the opening curly brace `{` back to its standard catcode, so that matched braces can be used to capture an argument as usual. The catcode changes are enclosed withing `\begingroup ... \endgroup` so that they may be contained.

The macro `\pytx@InlineOarg` which is called at the end of `\pytx@Inline` takes an argument enclosed by square brackets. If an optional argument is not present, then we supply an empty one, which invokes default treatment in `\pytx@InlineOarg`.

```
\def\pytx@Inline{%
    \begingroup
    \let\do\@makeother\dospecials
    \catcode`\{=1
    \@ifnextchar[{\endgroup\pytx@InlineOarg}{\endgroup\pytx@InlineOarg[]}%
}%
```

`\pytx@InlineOarg` This macro captures the optional argument (or the empty default substitute), which corresponds to the code session. Then it determines whether the delimiters of the actual code are a matched pair of curly braces or a pair of other, identical characters, and calls the next macro accordingly.

We begin by testing for an empty argument (either from the user or from the default empty substitute), and setting the default value if this is indeed the case. It is also possible that the user chose a session name containing a colon. If so, we substitute a hyphen for the colon. This is because temporary files are named based on session, and file names often cannot contain colons.

Then we turn off all special catcodes and set the catcodes of the curly braces back to their default values. This is necessary because we are about to capture the actual code, and we need all special catcodes turned off so that the code can contain any characters. But curly braces still need to be active just in case they are being used as delimiters. We also make the space and tab characters active, since `fancyvrb` needs them that way.[47] Using `\@ifnextchar` we determine whether the delimiters are curly braces. If so, we proceed to `\pytx@InlineMargBgroup` to capture the code using curly braces as delimiters. If not, we reset the catcodes of the braces and proceed to `\pytx@InlineMargOther`, which uses characters other than the opening curly brace as delimiters.

```
\def\pytx@InlineOarg[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \begingroup
    \let\do\@makeother\dospecials
    \catcode`\{=1
    \catcode`\}=2
```

[47] Part of this may be redundant, since we detokenize later and then retokenize during typesetting if Pygments isn't used. But the detokenizing and saving eliminates tab characters if they aren't active here.

```
    \catcode`\ =\active
    \catcode`\^^I=\active
    \@ifnextchar\bgroup
        {\pytx@InlineMargBgroup}%
        {\catcode`\{=12
            \catcode`\}=12
            \pytx@InlineMargOther}%
}
```

`\pytx@InlineMargOther` `\pytx@InlineMargOtherGet` This macro captures code delimited by a pair of identical non-brace characters. Then it passes the code on to `\pytx@InlineMargBgroup` for processing. This approach means that the macro definition may be kept concise, and that the processing code must only be defined once.

The macro captures only the next character. This will be the delimiting character. We must begin by ending the group that was left open by `\pytx@InlineOarg`, so that catcodes return to normal. Next we check to see if the delimiting character is a space character. If so, we issue an error, because that is not allowed. If the delimiter is valid, we define a macro `\pytx@InlineMargOtherGet` that will capture all content up to the next delimiting character and pass it to the `\pytx@InlineMargBgroup` macro for processing. That macro does exactly what is needed, except that part of the retokenization is redundant since curly braces were not active when the code was captured.

Once the custom capturing macro has been created, we turn off special catcodes and call the capturing macro.

```
\def\pytx@InlineMargOther#1{%
    \endgroup
    \ifstrequal{#1}{ }{%
        \PackageError{\pytx@packagename}%
            {The space character cannot be used as a delimiting character}%
            {Choose another delimiting character}}{}%
    \def\pytx@InlineMargOtherGet##1#1{\pytx@InlineMargBgroup{##1}}%
    \begingroup
    \let\do\@makeother\dospecials
    \pytx@InlineMargOtherGet
}
```

`\pytx@InlineMargBgroup` We are now ready to capture code using matched curly braces as delimiters, or to process previously captured code that used another delimiting character.

At the very beginning, we must end the group that was left open from `\pytx@InlineOarg` (or by `\pytx@InlineMargOther`), so that catcodes return to normal.

We save a detokenized version of the argument in `\pytx@argdetok`. Even though the argument was captured under special catcode conditions, this is still necessary. If the argument was delimited by curly braces, then any internal curly braces were active when the argument was captured, and these need their catcodes corrected. If the code contains any unicode characters, detokenization is needed so that they may be correctly saved to file.

We save a retokenized version of the argument in `\pytx@argretok`. This is needed for typesetting with `fancyvrb`. The code must be retokenized so that space characters are active, since `fancyvrb` allows space characters to be visible or invisible by making them active.

The **name** of the counter corresponding to this code is assembled. It is needed for keeping track of the instance, and is used for bringing in content created by the code and for bringing in highlighting created by Pygments.

Next we call a series of macros that determine whether the code is shown (typeset), whether it is saved to the code file, and whether content created by the code ("printed") should be brought in. These macros are `\let` to appropriate values when an inline command is called; they are not defined independently.

Finally, the counter for the code is incremented.

```
\def\pytx@InlineMargBgroup#1{%
    \endgroup
    \def\pytx@argdetok{\detokenize{#1}}%
    \def\pytx@arg{#1}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \pytx@InlineShow
    \pytx@InlineSave
    \pytx@InlinePrint
    \stepcounter{\pytx@counter}%
}%
```

`\pytx@InlineShow` `\pytx@InlineSave` `\pytx@InlinePrint`

The three macros `\pytx@InlineShow`, `\pytx@InlineSave`, and `\pytx@InlinePrint` will be `\let` to appropriate values when commands are called. We must now define the macros to which they may be `\let`.

`\pytx@InlineShowFV`

Code may be typeset with `fancyvrb`. `fancyvrb` settings are invoked via `pytx@FVSet`, but this must be done within a group so that the settings remain local. Most of the remainder of the commands are from `fancyvrb`'s `\FV@FormattingPrep`, and take care of various formatting matters, including spacing, font, whether space characters are shown, and any user-defined formatting. Finally, we create an `\hbox` and invoke `\FancyVerbFormatLine` to maintain parallelism with `BVerbatim`, which is used for inline content highlighted with Pygments. `\FancyVerbFormatLine` may be redefined to alter the typeset code, for example, by putting it in a colorbox via the following command:[48]

```
\renewcommand{\FancyVerbFormatLine}[1]{\colorbox{green}{#1}}
```

```
\def\pytx@InlineShowFV{%
    \def\pytx@argretok{%
        \begingroup
        \everyeof{\noexpand}%
        \endlinechar-1\relax
        \let\do\@makeother\dospecials
        \catcode`\ =\active
        \catcode`\^^I=\active
        \expandafter\scantokens\expandafter{\pytx@arg}%
        \endgroup}%
    \begingroup
    \pytx@FVSet
```

[48] Currently, `\FancyVerbFormatLine` is global, as in `fancyvrb`. Allowing a family-specific variant may be considered in the future. In most cases, the `fancyvrb` option `formatcom`, combined with external formatting from packages like `mdframed`, should provide all formatting desired. But something family-specific might occasionally prove useful.

```
    \FV@BeginVBox
    \frenchspacing
    \FV@SetupFont
    \FV@DefineWhiteSpace
    \FancyVerbDefineActive
    \FancyVerbFormatCom
    \FV@ObeyTabsInit
    \hbox{\FancyVerbFormatLine{\pytx@argretok}}%
    \FV@EndVBox
    \endgroup
}
```

`\pytx@InlineShowPyg` Code may be typeset with Pygments. Processed Pygments content is saved in the .pytxmcr file, wrapped in `fancyvrb`'s `SaveVerbatim` environment. The content is then restored, in a form suitable for inline use, via `BUseVerbatim`. Unlike non-inline content, which may be brought in either via macro or via separate external file, inline content is always brought in via macro. The counter `pytx@FancyVerbLineTemp` is used to prevent `fancyvrb`'s line count from being affected by PythonTeX content. A group is necessary to confine the `fancyvrb` settings created by `\pytx@FVSet`.

```
\def\pytx@InlineShowPyg{%
    \begingroup
    \pytx@FVSet
    \pytx@ConfigPygments
    \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
        \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
        \BUseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \else
        \textbf{??}%
        \PackageWarning{\pytx@packagename}{Non-existent Pygments content}%
    \fi
    \endgroup
}
```

`\pytx@InlineSaveCode` This macro writes PythonTeX information to the code file and then writes the actual code.

```
\def\pytx@InlineSaveCode{%
    \pytx@WriteCodefileInfo
    \immediate\write\pytx@codefile{\pytx@argdetok}%
}
```

`\pytx@InlineAutoprint` This macro brings in printed content automatically, if the package `autoprint` option is true. Otherwise, it does nothing. We must disable the macro in the event that the `debug` option is false. We wait until the beginning of the document to create the real macro, since any code commands and environments in the preamble shouldn't be printing and in any case we can't know what the outputdir is until the beginning of the document.

```
\let\pytx@InlineAutoprint\@empty
\AtBeginDocument{
    \def\pytx@InlineAutoprint{%
        \ifbool{pytx@opt@autoprint}{%
```

```
            \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}%
                {\DepyFile{p:\pytx@outputdir/\pytx@stdfile.stdout}}{}{}%
    }
    \ifbool{pytx@opt@stdout}{}{\let\pytx@InlineAutoprint\@empty}
}
```

`\pytx@InlineAlwaysprint` This is like `\pytx@InlineAutoprint`, except that it always prints rather than depending on `autoprint`. It is used for the `s` commands, which are always expected to have output.

```
\def\pytx@InlineAlwaysprint{%
    \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}%
       {\DepyFile{p:\pytx@outputdir/\pytx@stdfile.stdout}}%
       {\textbf{??}%
        \PackageWarning{\pytx@packagename}{Missing sub content}}}
```

`\pytx@InlineMacroprint` This macro brings in "printed" content that is brought in via macros in the .pytxmcr file. We must disable the macro in the event that the `debug` option is false.

```
\def\pytx@InlineMacroprint{%
    \edef\pytx@mcr{pytx@MCR@\pytx@type @\pytx@session @\pytx@group @\arabic{\pytx@counter}}%
    \ifcsname\pytx@mcr\endcsname
        \csname\pytx@mcr\endcsname
        \DepyMacro{p:\pytx@mcr}%
    \else
        \textbf{??}%
        \PackageWarning{\pytx@packagename}{Missing autoprint content}%
    \fi
}
\ifbool{pytx@opt@stdout}{}{\let\pytx@InlineMacroprint\@empty}
```

`\pytx@InlineMacroprintFV` This macro brings in "printed" content that is brought in via `SaveVerbatim` (only inline console references at the moment). We must disable the macro in the event that the `debug` option is false.

```
\def\pytx@InlineMacroprintFV{%
    \edef\pytx@mcr{pytx@\pytx@type @\pytx@session @\pytx@group @\arabic{\pytx@counter}}%
    \ifcsname FV@SV@\pytx@mcr\endcsname
        \BUseVerbatim{\pytx@mcr}%
        \DepyMacro{c:\pytx@mcr}%
    \else
        \textbf{??}%
        \PackageWarning{\pytx@packagename}{Missing autoprint content}%
    \fi
}
\ifbool{pytx@opt@stdout}{}{\let\pytx@InlineMacroprint\@empty}
```

10.5.2 Inline command constructors

With the core inline macros complete, we are ready to create constructors for different kinds of inline commands. All of these consctructors take a string and define an inline command named using that string as a base name. Two forms of each constructor are created, one that uses Pygments and one that does not. The Pygments variants have names ending in "Pyg". All commands are created

using `etoolbox`'s `\newrobustcmd`. Among other things, this is needed so that commands will work within the default caption command.

`\pytx@MakeInlinebFV` `\pytx@MakeInlinebPyg` These macros creates inline block commands, which both typeset code and save it so that it may be executed. The base name of the command is stored in `\pytx@type`. A string representing the kind of command is stored in `\pytx@cmd`. Then `\pytx@SetContext` is used to set `\pytx@context` and `\pytx@SetGroup` is used to set `\pytx@group`. Macros for showing, saving, and printing are set to appropriate values. Then the core inline macros are invoked through `\pytx@Inline`.

```
\newcommand{\pytx@MakeInlinebFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1b\endcsname}{%
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{cmd:#1b:ov:c}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{b}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\pytx@InlineShowFV
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\@empty
        \pytx@Inline
    }%
}%
\newcommand{\pytx@MakeInlinebPyg}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1b\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{b}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\pytx@InlineShowPyg
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\@empty
        \pytx@Inline
    }%
}%
```

`\pytx@MakeInlinevFV` `\pytx@MakeInlinevPyg` This macro creates inline verbatim commands, which only typeset code. `\pytx@type`, `\pytx@cmd`, `\pytx@context`, and `\pytx@group` are still set, for symmetry with other commands. They are not needed for `fancyvrb` typesetting, though. We use `\pytx@SetGroupVerb` to split verbatim content (`v` and `verb`) off into its own group. That way, verbatim content doesn't affect the instance numbers of executed code, and thus executed code is not affected by the addition or removal of verbatim content.

```
\newcommand{\pytx@MakeInlinevFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1v\endcsname}{%
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{cmd:#1v:ov:c}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{v}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowFV
        \let\pytx@InlineSave\@empty
```

```
        \let\pytx@InlinePrint\@empty
        \pytx@Inline
    }%
}%
\newcommand{\pytx@MakeInlinevPyg}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1v\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{v}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowPyg
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\@empty
        \pytx@Inline
    }%
}%
```

`\pytx@MakeInlinecFV` `\pytx@MakeInlinecPyg`

This macro creates inline code commands, which save code for execution but do not typeset it. If the code prints content, this content is inputted automatically if the package option `autoprint` is on. Since no code is typeset, there is no difference between the `fancyvrb` and Pygments forms.

```
\newcommand{\pytx@MakeInlinecFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1c\endcsname}{%
        \Depythontex{cmd:#1c:ov:p}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{c}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\@empty
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\pytx@InlineAutoprint
        \pytx@Inline
    }%
}%
\let\pytx@MakeInlinecPyg\pytx@MakeInlinecFV
```

`\pytx@MakeInlinesFV` `\pytx@MakeInlinesPyg`

This macro behaves almost exactly like code commands on the LaTeX side, but on the Python side, the argument is treated as a template in which fields are evaluated and replaced with the result. Since no code is typeset, there is no difference between the `fancyvrb` and Pygments forms.

```
\newcommand{\pytx@MakeInlinesFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1s\endcsname}{%
        \Depythontex{cmd:#1s:ov:p}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{s}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\@empty
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\pytx@InlineAlwaysprint
        \pytx@Inline
    }%
}%
\let\pytx@MakeInlinesPyg\pytx@MakeInlinesFV
```

\pytx@MakeInlineFV
\pytx@MakeInlinePyg

This macro creates plain inline commands, which save code and then bring in the output of `pytex.formatter(`⟨*code*⟩`)` (`pytex.formatter()` is the formatter function in Python sessions that is provided by `pythontex_utils*.py`). The Python output is saved in a TEX macro, and the macro is written to a file shared by all PythonTEX sessions. This greatly reduces the number of external files needed. Since no code is typeset, there is no difference between the `fancyvrb` and Pygments forms.

```
\newcommand{\pytx@MakeInlineFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1\endcsname}{%
        \Depythontex{cmd:#1:ov:p}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{i}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\@empty
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\pytx@InlineMacroprint
        \pytx@Inline
    }%
}%
\let\pytx@MakeInlinePyg\pytx@MakeInlineFV
```

\pytx@MakeInlineConsFV
\pytx@MakeInlineConsPyg

This is the inline form for `console` types. It brings in `SaveVerbatim`.

```
\newcommand{\pytx@MakeInlineConsFV}[1]{%
    \expandafter\newrobustcmd\expandafter{\csname #1\endcsname}{%
        \Depythontex{cmd:#1:ov:c}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{i}%
        \pytx@SetContext
        \pytx@SetGroup
        \let\pytx@InlineShow\@empty
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\pytx@InlineMacroprintFV
        \pytx@Inline
    }%
}%
\let\pytx@MakeInlineConsPyg\pytx@MakeInlineConsFV
```

\pythontexcustomc

This macro takes a single line of code and adds it to all sessions within a family. It is the inline version of the `pythontexcustomcode` environment.

```
\newrobustcmd{\pythontexcustomc}[2][begin]{%
    \Depythontex{cmd:pythontexcustomc:omv:p}%
    \ifstrequal{#1}{begin}{}{%
        \ifstrequal{#1}{end}{}{\PackageError{\pytx@packagename}%
            {Invalid optional argument for \string\pythontexcustomc}{}
        }%
    }%
    \xdef\pytx@type{CC:#2:#1}%
    \edef\pytx@cmd{c}%
    \pytx@SetContext
    \def\pytx@group{none}%
    \let\pytx@InlineShow\@empty
    \let\pytx@InlineSave\pytx@InlineSaveCode
```

```
    \let\pytx@InlinePrint\@empty
    \pytx@Inline[none]%
}%
```

\setpythontexcustomcode This macro is a holdover from 0.9beta3. It has been deprecated in favor of \pythontexcustomc and pythontexcustomcode.

```
\def\setpythontexcustomcode#1{%
    \Depythontex{cmd:setpythontexcustomcode:mv:p}%
    \PackageWarning{\pytx@packagename}{The command
        \string\setpythontexcustomcode\space has been deprecated;
        use \string\pythontexcustomc\space or pythontexcustomcode instead}%
    \begingroup
    \let\do\@makeother\dospecials
    \catcode`\{=1
    \catcode`\}=2
    \catcode`\^^M=10\relax
    \pytx@SetCustomCode{#1}%
}
\long\def\pytx@SetCustomCode#1#2{%
    \endgroup
    \pythontexcustomc{#1}{pythontexcustomcode=[#2];
        exec('for expr in pythontexcustomcode: exec(expr)');
        del(pythontexcustomcode)}
}
\@onlypreamble\setpythontexcustomcode
```

10.6 Environments

The inline commands were all created using a common core set of macros, combined with short, command-specific constructors. In the case of environments, we do not have a common core set of macros. Each environment is coded separately, though there are similarities among environments. In the future, it may be worthwhile to attempt to consolidate the environment code base.

One of the differences between inline commands and environments is that environments may need to typeset code with line numbers. Each family of code needs to have its own line numbering (actually, its own numbering for code, verbatim, and console groups), and this line numbering should not overwrite any line numbering that may separately be in use by fancyvrb. To make this possible, we use a temporary counter extensively. When line numbers are used, fancyvrb's line counter is copied into pytx@FancyVerbLineTemp, lines are numbered, and then fancyvrb's line counter is restored from pytx@FancyVerbLineTemp. This keeps fancyvrb and PythonTeX's line numbering separate, even though PythonTeX is using fancyvrb and its macros internally.

10.6.1 Block and verbatim environment constructors

We begin by creating block and verb environment constuctors that use fancyvrb. Then we create Pygments versions.

\pytx@FancyVerbGetLine The block environment needs to both typeset code and save it so it can be executed. fancyvrb supports typesetting, but doesn't support saving at the same time. So we create a modified version of fancyvrb's \FancyVerbGetLine macro

which does. This is identical to the `fancyvrb` version, except that we add a line that writes to the code file. The material that is written is detokenized to avoid catcode issues and make unicode work correctly.

```
\begingroup
\catcode`\^^M=\active
\gdef\pytx@FancyVerbGetLine#1^^M{%
    \@nil%
    \FV@CheckEnd{#1}%
    \ifx\@tempa\FV@EnvironName%
        \ifx\@tempb\FV@@@CheckEnd\else\FV@BadEndError\fi%
        \let\next\FV@EndScanning%
    \else%
        \def\FV@Line{#1}%
        \def\next{\FV@PreProcessLine\FV@GetLine}%
        \immediate\write\pytx@codefile{\detokenize{#1}}%
    \fi%
    \next}%
\endgroup
```

`\pytx@MakeBlockFV` Now we are ready to actually create block environments. This macro takes an environment base name ⟨*name*⟩ and creates a block environment ⟨*name*⟩`block`, using `fancyvrb`.

The block environment is a `Verbatim` environment, so we declare that with the `\VerbatimEnvironment` macro, which lets `fancyvrb` find the end of the environment correctly. We define the type, define the command, and set the context and group.

We need to check for optional arguments, so we begin a group and use `\obeylines` to make line breaks active. Then we check to see if the next char is an opening square bracket. If so, there is an optional argument, so we end our group and call the `\pytx@BeginBlockEnvFV` macro, which will capture the argument and finish preparing for the block content. If not, we end the group and call the same `\pytx@BeginBlockEnvFV` macro with an empty argument. The line breaks need to be active during this process because we don't care about content on the next line, including opening square brackets on the next line; we only care about content in the line on which the environment is declared, because only on that line should there be an optional argument. The problem is that since we are dealing with code, it is quite possible for there to be an opening square bracket at the beginning of the next line, so we must prevent that from being misinterpreted as an optional argument.

After the environment, we need to clean up several things. Much of this relates to what is done in the `\pytx@BeginBlockEnvFV` macro. The body of the environment is wrapped in a `Verbatim` environment, so we must end that. It is also wrapped in a group, so that `fancyvrb` settings remain local; we end the group. Then we define the name of the outfile for any printed content, so that it may be accessed by `\printpythontex` and company. Finally, we rearrange counters. The current code line number needs to be stored in `\pytx@linecount`, which was defined to be specific to the current type-session-group set. The `fancyvrb` line number needs to be set back to its original value from before the environment began, so that PythonTeX content does not affect the line numbering of `fancyvrb` content. Finally, the `\pytx@counter`, which keeps track of commands and environments within the current type-session-group set, needs to be incremented.

```
\newcommand{\pytx@MakeBlockFV}[1]{%
    \expandafter\newenvironment{#1block}{%
        \VerbatimEnvironment
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{env:#1block:oo|:c}%
        \DepyListing
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{block}%
        \pytx@SetContext
        \pytx@SetGroup
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginBlockEnvFV}{\endgroup\pytx@BeginBlockEnvFV[]}%
    }%
    {\end{Verbatim}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}
```

`\pytx@BeginBlockEnvFV` This macro finishes preparations to actually begin the block environment. It captures the optional argument (or the empty argument supplied by default). If this argument is empty, then it sets the value of the argument to the default value. If not, then colons in the optional argument are replaced with underscores, and the modified argument is stored in `\pytx@session`. Colons are replaced with underscores because session names must be suitable for file names, and colons are generally not allowed in file names. However, we want to be able to *enter* session names containing colons, since colons provide a conventient method of indicating relationships, and are commonly used in LaTeX labels. For example, we could have a session named `plots:specialplot`.

Once the session is established, we are free to define the counter for the current type-session-group, and make sure it exists. We also define the counter that will keep track of line numbers for the current type-session-group, and make sure it exists. Then we do some counter trickery. We don't want `fancyvrb` line counting to be affected by PythonTeX content, so we store the current line number held by `FancyVerbLine` in `pytx@FancyVerbLineTemp`; we will restore `FancyVerbLine` to this original value at the end of the environment. Then we set `FancyVerbLine` to the appropriate line number for the current type-session-group. This provides proper numbering continuity between different environments within the same type-session-group.

Next, we write environment information to the code file, now that all the necessary information is assembled. We begin a group, to keep some things local. We `\let` a `fancyvrb` macro to our custom macro. We set `fancyvrb` settings to those of the current type using `\pytx@FVSet`. Once this is done, we are finally ready to start the `Verbatim` environment. Note that the `Verbatim` environment will capture a second optional argument delimited by square brackets, if present, and apply this argument as `fancyvrb` formatting. Thus, the environment actually takes up to two optional arguments, but if you want to use `fancyvrb` formatting, you must supply an empty (default session) or named (custom session) optional

argument for the PythonTeX code.

```
\def\pytx@BeginBlockEnvFV[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@WriteCodefileInfo
    \let\FancyVerbGetLine\pytx@FancyVerbGetLine
    \pytx@FVSet
    \begin{Verbatim}%
}
```

`\pytx@MakeVerbFV` The `verbatim` environments only typeset code; they do not save it for execution. Thus, we just use a standard `fancyvrb` environment with a few enhancements.

As in the `block` environment, we declare that we are using a `Verbatim` environment, define type and command, set context and group (note the use of the `Verb` group), and take care of optional arguments before calling a macro to wrap things up (in this case, `\pytx@BeginVerbEnvFV`). Currently, much of the saved information is unused, but it is provided to maintain parallelism with the `block` environment.

Ending the environment involves ending the `Verbatim` environment begun by `\pytx@BeginVerbEnvFV`, ending the group that kept `fancyvrb` settings local, and resetting counters. We define a `stdfile` and step the counter, even though there will never actually be any output to pull in, to force `\printpythontex` and company to be used immediately after the code they refer to and to maintain parallelism.

```
\newcommand{\pytx@MakeVerbFV}[1]{%
    \expandafter\newenvironment{#1verbatim}{%
        \VerbatimEnvironment
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{env:#1verbatim:oo|:c}%
        \DepyListing
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{verbatim}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginVerbEnvFV}{\endgroup\pytx@BeginVerbEnvFV[]}%
    }%
    {\end{Verbatim}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}
```

`\pytx@BeginVerbEnvFV` This macro captures the optional argument of the environment (or the default empty argument that is otherwise supplied). If the argument is empty, it assignes

a default value; otherwise, it substitutes underscores for colons in the argument. The argument is assigned to `\pytx@session`. A line counter is created, and its existence is checked. We do the standard line counter trickery. Then we begin a group to keep `fancyvrb` settings local, invoke the settings via `\pytx@FVSet`, and begin the `Verbatim` environment.

```
\def\pytx@BeginVerbEnvFV[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \begin{Verbatim}%
}
```

Now for the Pygments forms of `block` and `verb`. Since all code must be saved now (either to be executed or processed by Pygments, or both), the environment code may be simplified compared to the non-Pygments case.

`\pytx@MakePygEnv` The `block` and `verb` environments are created via the same macro. The `\pytx@MakePygEnv` macro takes two arguments: first, the code type, and second, the environment (`block` or `verb`). The reason for using the same macro is that both must now save their code externally, and bring back the result typeset by Pygments. Thus, on the LaTeX side, their behavior is identical. The only difference is on the Python side, where the block code is executed and thus there may be output available via `\printpythontex` and company.

The actual workings of the macro are a combination of those of the non-Pygments macros, so please refer to those for details. The only exception is the code for bringing in Pygments output, but this is done using almost the same approach as that used for the inline Pygments commands. There are two differences: first, the `block` and `verb` environments use `\UseVerbatim` rather than `\BUseVerbatim`, since they are not typesetting code inline; and second, they accept a second, optional argument containing `fancyvrb` commands and this is used in typesetting the saved content. Any `fancyvrb` commands are saved in `\pytx@fvopttmp` by `\pytx@BeginEnvPyg@i`, and then used when the code is typeset.

Note that the positioning of all the `FancyVerbLine` trickery in what follows is significant. Saving the `FancyVerbLine` counter to a temporary counter before the beginning of `VerbatimOut` is important, because otherwise the `fancyvrb` numbering can be affected.

```
\newcommand{\pytx@MakePygEnv}[2]{%
    \expandafter\newenvironment{#1#2}{%
        \VerbatimEnvironment
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{#2}%
        \pytx@SetContext
        \ifstrequal{#2}{block}{\pytx@SetGroup}{}
        \ifstrequal{#2}{verbatim}{\pytx@SetGroupVerb}{}
        \begingroup
        \obeylines
```

```
        \@ifnextchar[{\endgroup\pytx@BeginEnvPyg}{\endgroup\pytx@BeginEnvPyg[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
    \pytx@ConfigPygments
    \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
        \UseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
    \else
        \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.pygtex}{}%
            {\textbf{??~\pytx@packagename~??}%
                \PackageWarning{\pytx@packagename}{Non-existent Pygments content}}%
    \fi
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}%
```

`\pytx@BeginEnvPyg` This macro finishes preparing for the content of a `verb` or `block` environment with Pygments content. It captures an optional argument corresponding to the session name and sets up instance and line counters. Finally, it calls an additional macro that handles the possibility of a second optional argument.

```
\def\pytx@BeginEnvPyg[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \pytx@WriteCodefileInfo
    \begingroup
    \obeylines
    \@ifnextchar[{\endgroup\pytx@BeginEnvPyg@i}{\endgroup\pytx@BeginEnvPyg@i[]}%
}%
```

`\pytx@BeginEnvPyg@i` This macro captures a second optional argument, corresponding to `fancyvrb` options. Note that not all `fancyvrb` options may be passed to saved content when it is actually used, particularly those corresponding to how the content was read in the first place (for example, command characters). But at least most formatting options such as line numbering work fine. As with the non-Pygments environments, `\begin{VerbatimOut}` doesn't take a second mandatory argument, since we are using a custom version and don't need to specify the file in which Verbatim content is saved. It is important that the `FancyVerbLine` saving be done here; if it is done later, after the end of `VerbatimOut`, then numbering can be off in some circumstances (for example, a single `pyverbatim` between two `Verbatim`'s).

```
\def\pytx@BeginEnvPyg@i[#1]{%
    \def\pytx@fvopttmp{#1}%
    \def\pytx@argspprint{#1}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
```

```
    \begin{VerbatimOut}%
}%
```

Since we are using the same code to create both `block` and `verb` environments, we now create a specific macro for creating each case, to make usage equivalent to that for the non-Pygments case.

`\pytx@MakeBlockPyg` The block environment is constructed via the `\pytx@MakePygEnv` macro.

```
\newcommand{\pytx@MakeBlockPyg}[1]{\pytx@MakePygEnv{#1}{block}}
```

`\pytx@MakeVerbPyg` The verb environment is constructed likewise.

```
\newcommand{\pytx@MakeVerbPyg}[1]{\pytx@MakePygEnv{#1}{verbatim}}
```

10.6.2 Code environment constructor

The `code` environment merely saves code to the code file; nothing is typeset. To accomplish this, we use a slightly modified version of `fancyvrb`'s `VerbatimOut`.

`\pytx@WriteDetok` We can use `fancyvrb` to capture the code, but we will need a way to write the code in detokenized form. This is necessary so that TEX doesn't try to process the code as it is written, which would generally be disastrous.

```
\def\pytx@WriteDetok#1{%
    \immediate\write\pytx@codefile{\detokenize{#1}}}%
```

`\pytx@FVB@VerbatimOut` We need a custom version of the macro that begins `VerbatimOut`. We don't need `fancyvrb`'s key values, and due to our use of `\detokenize` to write content, we don't need its space and tab treatment either. We do need `fancyvrb` to write to our code file, not the file to which it would write by default. And we don't need to open any files, because the code file is already open. These last two are the only important differences between our version and the original `fancyvrb` version. Since we don't need to write to a user-specified file, we don't require the mandatory argument of the original macro.

```
\def\pytx@FVB@VerbatimOut{%
    \@bsphack
    \begingroup
    \let\FV@ProcessLine\pytx@WriteDetok
    \let\FV@FontScanPrep\relax
    \let\@noligs\relax
    \FV@Scan}%
```

`\pytx@FVE@VerbatimOut` Similarly, we need a custom version of the macro that ends `VerbatimOut`. We don't want to close the file to which we are saving content.

```
\def\pytx@FVE@VerbatimOut{\endgroup\@esphack}%
```

`\pytx@EnvAutoprint` We also need a macro for bringing in autoprint content. This is a separate macro so that it can be easily disabled by the package option `debug`. We wait until the beginning of the document to create the real macro, since any code commands and environments in the preamble shouldn't be printing and in any case we can't know what the outputdir is until the beginning of the document.

```
\let\pytx@EnvAutoprint\@empty
\AtBeginDocument{
    \def\pytx@EnvAutoprint{%
```

```
        \ifbool{pytx@opt@autoprint}{%
            \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}%
                {\DepyFile{p:\pytx@outputdir/\pytx@stdfile.stdout}}{}{}%
    }
    \ifbool{pytx@opt@stdout}{}{\let\pytx@EnvAutoprint\@empty}
}
```

`\pytx@MakeCodeFV` Now that the helper macros for the `code` environment have been defined, we are ready to create the macro that makes `code` environments. Everything at the beginning of the environment is similar to the `block` and `verb` environments.

After the environment, we need to close the `VerbatimOut` environment begun by `\pytx@BeginCodeEnv@i` and end the group it began. We define the outfile, and bring in any printed content if the `autoprint` setting is on. We must still perform some `FancyVerbLine` trickery to prevent the `fancyvrb` line counter from being affected by **writing** content! Finally, we step the counter.

```
\newcommand{\pytx@MakeCodeFV}[1]{%
    \expandafter\newenvironment{#1code}{%
        \VerbatimEnvironment
        \Depythontex{env:#1code:oo|:p}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{code}%
        \pytx@SetContext
        \pytx@SetGroup
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginCodeEnv}{\endgroup\pytx@BeginCodeEnv[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \ifcsname pytx@nonpyconsole@\pytx@type\endcsname
        \ifcsname pytx@code@as@console\endcsname
            \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
            \pytx@FVSet
            \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}
            \pytx@ConfigPygments
            \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}%
               {\DepyFile{p:\pytx@outputdir/\pytx@stdfile.stdout}}%
               {\par\textbf{??~\pytx@packagename~??}\par
                \PackageWarning{\pytx@packagename}{Non-existent console content}}%
            \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
        \else
        \fi
        \let\pytx@EnvAutoprint\relax
    \else
    \fi
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    \pytx@EnvAutoprint
    }%
}%
```

`\pytx@BeginCodeEnv` This macro finishes setting things up before the `code` environment contents. It processes the optional argument, defines a counter and checks its existence, writes

info to the code file, and then calls the `\pytx@BeginCodeEnv@i` macro. This macro is necessary so that the environment can accept two optional arguments. Since the `block` and `verb` environments can accept two optional arguments (the first is the name of the session, the second is `fancyvrb` options), the code environment also should be able to, to maintain parallelism (for example, `pyblock` should be able to be swapped with `pycode` without changing environment arguments—it should just work). However, `VerbatimOut` doesn't take an optional argument. So we need to capture and discard any optional argument, before starting `VerbatimOut`.

```
\def\pytx@BeginCodeEnv[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \pytx@WriteCodefileInfo
    \begingroup
    \obeylines
    \@ifnextchar[{\endgroup\pytx@BeginCodeEnv@i}{\endgroup\pytx@BeginCodeEnv@i[]}%
}%
```

`\pytx@BeginCodeEnv@i` As described above, this macro captures a second optional argument, if present, and then starts the `VerbatimOut` environment. Note that `VerbatimOut` does not have a mandatory argument, because we are invoking our custom `\pytx@FVB@VerbatimOut` macro. The default `fancyvrb` macro needs an argument to tell it the name of the file to which to save the verbatim content. But in our case, we are always writing to the same file, and the custom macro accounts for this by not having a mandatory file name argument. We must perform the typical `FancyVerbLine` trickery, to prevent the `fancyvrb` line counter from being affected by **writing** content!

```
\def\pytx@BeginCodeEnv@i[#1]{%
    \def\pytx@fvopttmp{#1}%
    \def\pytx@argspprint{#1}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
    \begin{VerbatimOut}%
}%
```

`\pytx@MakeCodePyg` Since the code environment simply saves code for execution and typesets nothing, the Pygments version is identical to the non-Pygments version, so we simply let the former to the latter.

```
\let\pytx@MakeCodePyg\pytx@MakeCodeFV
```

`pythontexcustomcode` This environment is used for adding custom code to all sessions within a command and environment family. It is the environment equivalent of the inline command `\pythontexcustomc`.

```
\newenvironment{pythontexcustomcode}[2][begin]{%
    \VerbatimEnvironment
    \Depythontex{env:pythontexcustomcode:om:n}%
    \ifstrequal{#1}{begin}{}{%
        \ifstrequal{#1}{end}{}{\PackageError{\pytx@packagename}%
            {Invalid optional argument for pythontexcustomcode}{}
        }%
```

```
    }%
    \xdef\pytx@type{CC:#2:#1}%
    \edef\pytx@cmd{code}%
    \pytx@SetContext
    \def\pytx@group{none}%
    \pytx@BeginCodeEnv[none]}%
{\end{VerbatimOut}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
}%
```

10.6.3 Sub environment constructor

The `sub` environment behaves exactly like a `code` environment on the LaTeX side: environment content is saved to the code file, and then the output is brought back in. The difference is on the Python side, where the environment content is treated as a template in which fields are evaluated and replaced with the result.

`\pytx@MakeSubFV` Create a `sub` environment compatible with `fancyvrb`, reusing the `code` approach almost entirely.

```
\newcommand{\pytx@MakeSubFV}[1]{%
    \expandafter\newenvironment{#1sub}{%
        \VerbatimEnvironment
        \Depythontex{env:#1sub:oo|:p}%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{sub}%
        \pytx@SetContext
        \pytx@SetGroup
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginCodeEnv}{\endgroup\pytx@BeginCodeEnv[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}%
       {\DepyFile{p:\pytx@outputdir/\pytx@stdfile.stdout}}%
       {\textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent substituted content}}%
    }%
}%
```

`\pytx@MakeSubPyg` The Pygments-compatible version is the same.

```
\let\pytx@MakeSubPyg\pytx@MakeSubFV
```

10.6.4 Console environment constructor

The `console` environment needs to write all code contained in the environment to the code file, and then bring in the console output.

An environment suffix is not enforced for flexibility. For Python, the convention is that `console` type names will end with `con`, and then the environment will use the suffix `sole`. For example, the `pycon` type has the `pyconsole` environment.

\pytx@MakeConsoleFV

```
\newcommand{\pytx@MakeConsFV}[2]{%
    \expandafter\newenvironment{#1#2}{%
        \VerbatimEnvironment
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{env:#1#2:oo|:c}%
        \DepyListing
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{console}%
        \pytx@SetContext
        \pytx@SetGroup
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginConsEnvFV}{\endgroup\pytx@BeginConsEnvFV[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
    \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
        \UseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
        \DepyMacro{c:\pytx@counter @\arabic{\pytx@counter}}%
    \else
        \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.tex}%
            {\DepyFile{c:\pytx@outputdir/\pytx@stdfile.tex}}%
            {\textbf{??~\pytx@packagename~??}%
                \PackageWarning{\pytx@packagename}{Non-existent console content}}%
    \fi
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}
```

\pytx@BeginConsEnvFV

```
\def\pytx@BeginConsEnvFV[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \pytx@WriteCodefileInfo
    \begingroup
    \obeylines
    \@ifnextchar[{\endgroup\pytx@BeginConsEnvFV@i}{\endgroup\pytx@BeginConsEnvFV@i[]}%
}%
```

\pytx@BeginConsEnvFV@i

```
\def\pytx@BeginConsEnvFV@i[#1]{%
    \def\pytx@fvopttmp{#1}%
    \def\pytx@argspprint{#1}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
```

```
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
    \begin{VerbatimOut}%
}%
```

\pytx@MakeConsPyg The Pygments version of the `console` environment is identical to the `fancyvrb` version, except that `.pygtex` rather than `.tex` files are brought in.

```
\newcommand{\pytx@MakeConsPyg}[2]{%
    \expandafter\newenvironment{#1#2}{%
        \VerbatimEnvironment
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{env:#1#2:oo|:c}%
        \DepyListing
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{console}%
        \pytx@SetContext
        \pytx@SetGroup
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginConsEnvFV}{\endgroup\pytx@BeginConsEnvFV[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
    \pytx@ConfigPygments
    \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
        \UseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
        \DepyMacro{c:\pytx@counter @\arabic{\pytx@counter}}%
    \else
        \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.pygtex}%
            {\DepyFile{c:\pytx@outputdir/\pytx@stdfile.pygtex}}%
            {\textbf{??~\pytx@packagename~??}%
                \PackageWarning{\pytx@packagename}{Non-existent console content}}%
    \fi
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}
```

10.7 Constructors for command and environment families

Everything is now in place to create commands and environments, with and without Pygments usage. To make all of this more readily usable, we need macros that will create a whole family of commands and environments at once, using a base name. For example, we need a way to create all commands and environments based off of the `py` base name.

\makepythontexfamily This macro creates a family of commands. It needs a some `pgfkeys` to handle the optional arguments. The actual creation of all non-code commands and environments is delayed using `\AtBeginDocument`, so that the user has the option to choose whether `fancyvrb` or Pygments is used for the family.

We need to create a counter for the default session for each family to avoid (some of the) issues with `\includeonly` and counters. See http://tug.org/pipermail/macostex-archives/2010-December/046007.html for more on the problematic counter behavior with `\includeonly`.

```
\pgfkeys{
    /PYTX/family/.is family,
    /PYTX/family,
    name/.estore in = \pytx@tmp@name,
    prettyprinter/.estore in = \pytx@tmp@pprinter,
    pyglexer/.estore in = \pytx@tmp@pyglexer,
    pygopt/.code = \def\pytx@tmp@pygopt{#1}\pgfkeys{/PYTX/lopt/pygopt/.cd, #1},
    console/.estore in = \pytx@tmp@console,
    default/.style = {prettyprinter=auto, pyglexer=text, pygopt={}, console=false}
}
\def\pytx@MakeFamilyFV#1{%
    \pytx@MakeInlinebFV{#1}%
    \pytx@MakeInlinevFV{#1}%
    \pytx@MakeInlineFV{#1}%
    \pytx@MakeBlockFV{#1}%
    \pytx@MakeVerbFV{#1}%
}
\def\pytx@MakeFamilyPyg#1{%
    \ifbool{pytx@opt@pyginline}%
        {\pytx@MakeInlinebPyg{#1}\pytx@MakeInlinevPyg{#1}}%
        {\pytx@MakeInlinebFV{#1}\pytx@MakeInlinevFV{#1}}%
    \pytx@MakeInlinePyg{#1}%
    \pytx@MakeBlockPyg{#1}%
    \pytx@MakeVerbPyg{#1}%
    \booltrue{pytx@usedpygments}%
    \AtEndDocument{\immediate\write\pytx@codefile{pygfamily=#1|%
        \csname pytx@pyglexer@#1\endcsname|%
        \csname pytx@pygopt@#1\endcsname}%
    }%
}
\def\pytx@MakeFamilyFVCons#1{%
    \pytx@MakeInlinevFV{#1}%
    \pytx@MakeInlineConsFV{#1}%
    \pytx@MakeConsFV{#1}{sole}%
    \pytx@MakeVerbFV{#1}%
}
\def\pytx@MakeFamilyPygCons#1{%
    \ifbool{pytx@opt@pyginline}%
        {\pytx@MakeInlinevPyg{#1}}%
        {\pytx@MakeInlinevFV{#1}}%
    \pytx@MakeInlineConsPyg{#1}%
    \pytx@MakeConsPyg{#1}{sole}%
    \pytx@MakeVerbPyg{#1}%
    \booltrue{pytx@usedpygments}%
    \AtEndDocument{\immediate\write\pytx@codefile{pygfamily=#1|%
        \csname pytx@pyglexer@#1\endcsname|%
        \csname pytx@pygopt@#1\endcsname}%
    }%
}
\newcommand{\makepythontexfamily}[2][]{%
```

```
    \IfBeginWith{#2}{PYG}%
        {\PackageError{\pytx@packagename}%
            {Attempt to create macros with reserved prefix PYG}{}}{}%
    \pgfkeys{/PYTX/family, name=#2, default, #1}
    \expandafter\xdef\csname pytx@macroformatter@#2\endcsname{\pytx@tmp@pprinter}
    \expandafter\gdef\csname pytx@fvsettings@#2\endcsname{}
    \expandafter\xdef\csname pytx@pyglexer@#2\endcsname{\pytx@tmp@pyglexer}
    \expandafter\xdef\csname pytx@pygopt@#2\endcsname{\pytx@tmp@pygopt}
    \expandafter\xdef\csname pytx@console@#2\endcsname{\pytx@tmp@console}
    \pytx@MakeInlinecFV{#2}
    \pytx@MakeInlinesFV{#2}
    \pytx@MakeCodeFV{#2}
    \pytx@MakeSubFV{#2}
    \AtBeginDocument{%
        \ifcsstring{pytx@macroformatter@#2}{auto}{%
            \ifbool{pytx@opt@pygments}%
                {\ifcsstring{pytx@console@#2}{true}%
                    {\pytx@MakeFamilyPygCons{#2}}{\pytx@MakeFamilyPyg{#2}}}%
                {\ifcsstring{pytx@console@#2}{true}%
                    {\pytx@MakeFamilyFVCons{#2}}{\pytx@MakeFamilyFV{#2}}}%
        }{}%
        \ifcsstring{pytx@macroformatter@#2}{fancyvrb}%
            {\ifcsstring{pytx@console@#2}{true}%
               {\pytx@MakeFamilyFVCons{#2}}{\pytx@MakeFamilyFV{#2}}}{}%
        \ifcsstring{pytx@macroformatter@#2}{pygments}%
            {\ifcsstring{pytx@console@#2}{true}%
                {\pytx@MakeFamilyPygCons{#2}}{\pytx@MakeFamilyPyg{#2}}}{}%
    }%
    \newcounter{pytx@#2@default@default}%
}
\@onlypreamble\makepythontexfamily
```

`\makepythontexfamily@con` This macro creates `console` and `code` environments for non-Python consoles. PythonTeX was not designed with commands and environments for non-Python consoles. Non-Python consoles are currently created via specially customized `code` environments. Note that simply creating these `console` and `code` environments is typically not enough to create non-Python consoles; `pythontex2.py` and `pythontex3.py` usually also require customization. This macro's definition should not be treated as stable; it will change in the future. The ultimate long-term goal is to eliminate it entirely, by redesigning the code execution core of PythonTeX to accomodate non-Python consoles more easily.

```
\newcommand{\makepythontexfamily@con}[2][text]{%
    \pgfkeys{/PYTX/family, name=#2con, default, pyglexer=#1, console=true}%
    \expandafter\xdef\csname pytx@macroformatter@#2con\endcsname{\pytx@tmp@pprinter}%
    \expandafter\gdef\csname pytx@fvsettings@#2con\endcsname{}%
    \expandafter\xdef\csname pytx@pyglexer@#2con\endcsname{\pytx@tmp@pyglexer}%
    \expandafter\xdef\csname pytx@pygopt@#2con\endcsname{\pytx@tmp@pygopt}%
    \expandafter\xdef\csname pytx@console@#2con\endcsname{\pytx@tmp@console}%
    \AtEndDocument{\immediate\write\pytx@codefile{pygfamily=#2con|%
        \csname pytx@pyglexer@#2con\endcsname|%
        \csname pytx@pygopt@#2con\endcsname}%
    }%
    \pytx@MakeCodeFV{#2con}%
```

```
    \expandafter\global\expandafter\let\csname pytx@nonpyconsole@#2con\endcsname\relax
    \newenvironment{#2console}%
       {\VerbatimEnvironment
        \def\pytx@type{#2con}%
        \let\pytx@code@as@console\relax
        \begin{#2concode}}%
       {\end{#2concode}}%
}
```

`\setpythontexpyglexer` We need to be able to reset the lexer associated with a family after the family has already been created.

```
\newcommand{\setpythontexpyglexer}[2][]{%
    \Depythontex{cmd:setpythontexpyglexer:om:n}%
    \ifstrempty{#1}{\def\pytx@pyglexer{#2}}{%
        \ifcsname pytx@pyglexer@#1\endcsname
            \expandafter\xdef\csname pytx@pyglexer@#1\endcsname{#2}%
        \else
            \PackageError{\pytx@packagename}%
                {Cannot modify a non-existent family}{}%
        \fi
    }%
}%
\@onlypreamble\setpythontexpyglexer
```

`\setpythontexpygopt` The user may wish to modify the Pygments options associated with a family. This macro takes two arguments: first, the family base name; and second, the Pygments options to associate with the family. This macro is particularly useful in changing the Pygments style of default command and environment families.

Due to the implementation (and also in the interest of keeping typesetting consistent), the Pygments style for a family must remain constant throughout the document. Thus, we only allow changes to the style in the preamble.

```
\newcommand{\setpythontexpygopt}[2][]{%
    \Depythontex{cmd:setpythontexpygopt:om:n}%
    \ifstrempty{#1}%
     {\def\pytx@pygopt{#2}\pgfkeys{/PYTX/gopt/pygopt/.cd, #2}}%
     {\ifcsname pytx@pygopt@#1\endcsname
        \expandafter\xdef\csname pytx@pygopt@#1\endcsname{#2}%
        \pgfkeys{/PYTX/lopt/pygopt/.cd, name=#1, #2}
      \else
        \PackageError{\pytx@packagename}%
         {Cannot modify Pygments options for a non-existent family}{}%
      \fi}%
}
\@onlypreamble\setpythontexpygopt
```

`\setpythontexprettyprinter` We need to be able to reset the pretty printer used by a family among the options `auto`, `fancyvrb`, and `pygments`.

```
\newcommand{\setpythontexprettyprinter}[2][]{%
    \Depythontex{cmd:setpythontexprettyprinter:om:n}%
    \ifstrempty{#1}{%
        \ifstrequal{#2}{fancyvrb}{\boolfalse{pytx@opt@pygments}}%
        \ifstrequal{#2}{pygments}{\booltrue{pytx@opt@pygments}}%
    }{%
```

```
        \ifcsname pytx@macroformatter@#1\endcsname
            \ifbool{pytx@opt@depythontex}{}{%
                \expandafter\xdef\csname pytx@macroformatter@#1\endcsname{#2}}
        \else
            \PackageError{\pytx@packagename}%
                {Cannot modify a family that does not exist or does not allow formatter choi
                {Create the family with \string\makepythontexfamily}%
        \fi
    }%
}
\@onlypreamble\setpythontexprettyprinter
```

10.8 Default commands and environment families

We are finally prepared to create the default command and environment families. We create a basic Python family with the base name `py`. We also create customized Python families for the SymPy package, using the base name `sympy`, and for the pylab module, using the base name `pylab`. All of these are created with a `console` environment.

All of these command and environment families are created conditionally, depending on whether the package option `pygments` is used, via `\makepythontexfamily`. We recommend that any custom families created by the user be constructed in the same manner.

```
\makepythontexfamily[pyglexer=python3]{py}
\makepythontexfamily[pyglexer=pycon, console=true]{pycon}
\makepythontexfamily[pyglexer=python3]{sympy}
\makepythontexfamily[pyglexer=pycon, console=true]{sympycon}
\makepythontexfamily[pyglexer=python3]{pylab}
\makepythontexfamily[pyglexer=pycon, console=true]{pylabcon}
```

We also need to create any additional families specified via the `usefamily` package option.[49]

```
\renewcommand{\do}[1]{%
    \ifstrequal{#1}{ruby}{\makepythontexfamily[pyglexer=ruby]{ruby}}{}%
    \ifstrequal{#1}{rb}{\makepythontexfamily[pyglexer=ruby]{rb}}{}%
    \ifstrequal{#1}{julia}{\makepythontexfamily[pyglexer=julia]{julia}}{}%
    \ifstrequal{#1}{juliacon}{\makepythontexfamily@con[jlcon]{julia}}{}%
    \ifstrequal{#1}{jl}{\makepythontexfamily[pyglexer=julia]{jl}}{}%
    \ifstrequal{#1}{matlab}{\makepythontexfamily[pyglexer=matlab]{matlab}}{}%
    \ifstrequal{#1}{octave}{\makepythontexfamily[pyglexer=octave]{octave}}{}%
    \ifstrequal{#1}{bash}{\makepythontexfamily[pyglexer=bash]{bash}}{}%
    \ifstrequal{#1}{sage}{\makepythontexfamily[pyglexer=sage]{sage}}{}%
    \ifstrequal{#1}{rust}{\makepythontexfamily[pyglexer=rust]{rust}}{}%
    \ifstrequal{#1}{rs}{\makepythontexfamily[pyglexer=rust]{rs}}{}%
    \ifstrequal{#1}{R}{\makepythontexfamily[pyglexer=r]{R}}{}%
    \ifstrequal{#1}{Rcon}{\makepythontexfamily@con[rconsole]{R}}{}%
    \ifstrequal{#1}{perl}{\makepythontexfamily[pyglexer=perl]{perl}}{}%
    \ifstrequal{#1}{pl}{\makepythontexfamily[pyglexer=perl]{pl}}{}%
    \ifstrequal{#1}{perlsix}{\makepythontexfamily[pyglexer=perl6]{perlsix}}{}%
    \ifstrequal{#1}{psix}{\makepythontexfamily[pyglexer=perl6]{psix}}{}%
    \ifstrequal{#1}{javascript}{\makepythontexfamily[pyglexer=js]{javascript}}{}%
```

[49] The loop here is accomplished via `etoolbox`. `pgffor` might be an alternative, but making definitions global requires trickery.

```
    \ifstrequal{#1}{js}{\makepythontexfamily[pyglexer=js]{js}}{}%
}
\expandafter\docsvlist\expandafter{\pytx@families}
```

10.9 Listings environment

`fancyvrb`, especially when combined with Pygments, provides most of the formatting options we could want. However, it simply typesets code within the flow of the document and does not provide a floating environment. So we create a floating environment for code listings via the `newfloat` package.

It is most logical to name this environment `listing`, but that is already defined by the `minted` package (although PythonTeX and `minted` are probably not likely to be used together, due to overlapping features). Furthermore, the `listings` package specifically avoided using the name `listing` for an environment due to the use of this name by other packages.

We have chosen to make a compromise. We create a macro that creates a float environment with a custom name for listings. If this macro is invoked, then a float environment for listings is created and nothing else is done. If it is not invoked, the package attempts to create an environment called `listing` at the beginning of the document, and issues a warning if another macro with that name already exists. This approach makes the logical `listing` name available in most cases, and provides the user with a simple fallback in the event that another package defining `listing` must be used alongside PythonTeX.

`\setpythontexlistingenv` We define a bool `pytx@listingenv` that keeps track of whether a listings environment has been created. Then we define a macro that creates a floating environment with a custom name, with appropriate settings for a listing environment. We only allow this macro to be used in the preamble, since later use would wreak havok.

```
\newbool{pytx@listingenv}
\def\setpythontexlistingenv#1{%
    \Depythontex{cmd:setpythontexlistingenv:m:n}%
    \DeclareFloatingEnvironment[fileext=lopytx,listname={List of Listings},name=Listing]{#1}
    \booltrue{pytx@listingenv}
}
\@onlypreamble\setpythontexlistingenv
```

At the beginning of the document, we issue a warning if the `listing` environment needs to be created but cannot be due to a pre-existing macro (and no version with a custom name has been created). Otherwise, we create the `listing` environment.

```
\AtBeginDocument{
    \ifcsname listing\endcsname
        \ifbool{pytx@listingenv}{}%
            {\PackageWarning{\pytx@packagename}%
                {A "listing" environment already exists \MessageBreak
                 \pytx@packagename\space will not create one \MessageBreak
                 Use \string\setpythontexlistingenv\space to create a custom listing environ
    \else
        \ifbool{pytx@listingenv}{}{\DeclareFloatingEnvironment[fileext=lopytx]{listing}}
    \fi
}
```

10.10 Pygments for general code typesetting

After all the work that has gone into PythonTeX thus far, it would be a pity not to slightly expand the system to allow Pygments typesetting of any language Pygments supports. While PythonTeX currently can only *execute* Python code, it is relatively easy to add support for *highlighting* any language supported by Pygments. We proceed to create a `\pygment` command, a `pygments` environment, and an `\inputpygments` command that do just this. The functionality of these is very similar to that provided by the `minted` package.

Both the commands and the environment are created in two forms: one that actually uses Pygments, which is the whole point in the first place; and one that uses `fancyvrb`, which may speed compilation or make editing faster since `pythontex.py` need not be invoked. By default, the two forms are switched between based on the package `pygments` option, but this may be easily modified as described below.

The Pygments commands and environment operate under the code type `PYG`⟨*lexer name*⟩. This allows Pygments typesetting of general code to proceed with very few additions to `pythontex.py`; in most situations, the Pygments code types behave just like standard PythonTeX types that don't execute any code. Due to the use of the `PYG` prefix for all Pygments content, the use of this prefix is not allowed at the beginning of a base name for standard PythonTeX command and environment families.

We have previously used the suffix `Pyg` to denote macro variants that use Pygments rather than `fancyvrb`. We continue that practice here. To distinguish the special Pygments typesetting macros from the regular PythonTeX macros, we use `Pygments` in the macro names, in addition to any `Pyg` suffix

10.11 Pygments utilities macros

`\pytx@CheckPygmentsInit` We need to see if macros exist for storing Pygments `fv` settings and `pygopt`. If not, create them, and make sure they will be written to file.

```
\def\pytx@CheckPygmentsInit#1{%
    \ifcsname pytx@fvsettings@PYG#1\endcsname\else
        \expandafter\gdef\csname pytx@fvsettings@PYG#1\endcsname{}%
        \expandafter\gdef\csname pytx@pygopt@PYG#1\endcsname{}%
        \AtEndDocument{\immediate\write\pytx@codefile{pygfamily=PYG#1|#1|%
            \csname pytx@pygopt@PYG#1\endcsname}}%
    \fi
}
```

10.11.1 Inline Pygments command

`\pytx@MakePygmentsInlineFV` `\pytx@MakePygmentsInlinePyg` `\pygment` These macros create an inline command. They reuse the `\pytx@Inline` macro sequence. The approach is very similar to the constructors for inline commands, except for the way in which the type is defined and for the fact that we have to check to see if a macro for `fancyvrb` settings exists. Just as for the PythonTeX inline commands, we do not currently support `fancyvrb` options in Pygments inline commands, since almost all options are impractical for inline usage, and the few that might conceivably be practical, such as showing spaces, should probably be used throughout an entire document rather than just for a tiny code snippet within a paragraph.

We supply an empty optional argument to `\pytx@Inline`, so that the `\pygment` command can only take two mandatory arguments, and no optional argument (since sessions don't make sense for code that is merely typeset):

`\pygment{`⟨*lexer*⟩`}{`⟨*code*⟩`}`

```
\def\pytx@MakePygmentsInlineFV{%
    \newcommand{\pygment}[1]{%
        \edef\pytx@lexer{##1}%
        \Depythontex{cmd:pygment:mv:c}%
        \edef\pytx@type{PYG##1}%
        \edef\pytx@cmd{v}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowFV
        \let\pytx@InlineSave\@empty
        \let\pytx@InlinePrint\@empty
        \pytx@CheckPygmentsInit{##1}%
        \pytx@Inline[]%
    }%
}
\def\pytx@MakePygmentsInlinePyg{%
    \newcommand{\pygment}[1]{%
        \edef\pytx@type{PYG##1}%
        \edef\pytx@cmd{v}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowPyg
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\@empty
        \pytx@CheckPygmentsInit{##1}%
        \pytx@Inline[]
    }%
}
```

10.11.2 Pygments environment

`\pytx@MakePygmentsEnvFV`
`pygments`

The `pygments` environment is created to take an optional argument, which corresponds to `fancyvrb` settings, and one mandatory argument, which corresponds to the Pygments lexer to be used in highlighting the code.

The `pygments` environment begins by declaring that it is a `Verbatim` environment and setting variables. Again, some variables are unnecessary, but they are created to maintain uniformity with other PythonTeX environments. The environment code is very similar to that of PythonTeX `verb` environments.

```
\def\pytx@MakePygmentsEnvFV{%
    \newenvironment{pygments}{%
        \VerbatimEnvironment
        \pytx@SetContext
        \pytx@SetGroupVerb
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BEPygmentsFV}{\endgroup\pytx@BEPygmentsFV[]}%
    }%
    {\end{Verbatim}%
```

```
        \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    }%
}
```

`\pytx@BEPygmentsFV` This macro captures the optional argument containing `fancyvrb` commands.

```
\def\pytx@BEPygmentsFV[#1]{%
    \def\pytx@fvopttmp{#1}%
    \def\pytx@argspprint{#1}%
    \begingroup
    \obeylines
    \pytx@BEPygmentsFV@i
}
```

`\pytx@BEPygmentsFV@i` This macro captures the mandatory argument, containing the lexer name, and proceeds.

```
\def\pytx@BEPygmentsFV@i#1{%
    \endgroup
    \edef\pytx@type{PYG#1}%
    \edef\pytx@lexer{#1}%
    \Depythontex{env:pygments:om:c}%
    \DepyListing
    \edef\pytx@cmd{verbatim}%
    \edef\pytx@session{default}%
    \edef\pytx@linecount{pytx@\pytx@type @\pytx@session @\pytx@group @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \pytx@CheckPygmentsInit{#1}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
    \begin{Verbatim}%
}
```

`\pytx@MakePygmentsEnvPyg` `pygments` The Pygments version is very similar, except that it must bring in external Pygments content.

```
\def\pytx@MakePygmentsEnvPyg{%
    \newenvironment{pygments}{%
        \VerbatimEnvironment
        \pytx@SetContext
        \pytx@SetGroupVerb
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BEPygmentsPyg}{\endgroup\pytx@BEPygmentsPyg[]}%
    }%
    {\end{VerbatimOut}%
        \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
        \pytx@FVSet
        \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
        \pytx@ConfigPygments
        \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
            \UseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
        \else
            \InputIfFileExists{\pytx@outputdir/%
```

```
                \pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}.pygtex}{}%
                {\textbf{??~\pytx@packagename~??}%
                    \PackageWarning{\pytx@packagename}{Non-existent Pygments content}}%
        \fi
        \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
        \stepcounter{\pytx@counter}%
    }%
}
```

`\pytx@BEPygmentsPyg` This macro captures the optional argument, which corresponds to `fancyvrb` settings.

```
\def\pytx@BEPygmentsPyg[#1]{%
    \def\pytx@fvopttmp{#1}%
    \def\pytx@argspprint{#1}%
    \begingroup
    \obeylines
    \pytx@BEPygmentsPyg@i
}
```

`\pytx@BEPygmentsPyg@i` This macro captures the mandatory argument, containing the lexer name, and proceeds.

```
\def\pytx@BEPygmentsPyg@i#1{%
    \endgroup
    \edef\pytx@type{PYG#1}%
    \edef\pytx@cmd{verbatim}%
    \edef\pytx@session{default}%
    \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \edef\pytx@linecount{\pytx@counter @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \pytx@WriteCodefileInfo
    \pytx@CheckPygmentsInit{#1}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
    \begin{VerbatimOut}%
}
```

10.11.3 Special Pygments commands

Code highlighting may be used for some tasks that would never appear in a code execution context, which is what the PythonTEX part of this package focuses on. We create some special Pygments macros to handle these highlighting cases.

`\pytx@MakePygmentsInputFV` `\pytx@MakePygmentsInputPyg` For completeness, we need to be able to read in a file and highlight it. This is done through some trickery with the current system. We define the type as `PYG`⟨*lexer*⟩, and the command as `verb`. We set the context for consistency. We set the session as `EXT:`⟨*file name*⟩.[50] Next we define a `fancyvrb` settings macro

[50] There is no possibility of this session being confused with a user-defined session, because colons are substituted for hyphens in all user-defined sessions, before they are written to the code file.

for the type if it does not already exist. We write info to the code file using `\pytx@WriteCodefileInfoExt`.

Then we check to see if the file actually exists, and issue a warning if not. This saves the user from running `pythontex.py` to get the same error. We perform our typical `FancyVerbLine` trickery. Next we make use of the saved content in the same way as the `pygments` environment. Note that we do not create a counter for the line numbers. This is because under typical usage an external file should have its lines numbered beginning with 1. We also encourage this by setting `firstnumber=auto` before bringing in the content.

The current naming of the macro in which the Pygments content is saved is probably excessive. In almost every situation, a unique name could be formed with less information. The current approach has been taken to maintain parallelism, thus simplifying `pythontex.py`, and to avoid any rare potential conflicts.

```
\def\pytx@MakePygmentsInputFV{
    \newcommand{\inputpygments}[3][]{%
        \edef\pytx@lexer{##2}%
        \Depythontex{cmd:inputpygments:omm:c}%
        \edef\pytx@type{PYG##2}%
        \edef\pytx@cmd{verbatim}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \edef\pytx@session{EXT:##3}%
        \pytx@CheckPygmentsInit{##2}%
        \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
        \pytx@CheckCounter{\pytx@counter}%
        \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
        \begingroup
        \DepyListing %Always must be in a group
        \pytx@FVSet
        \fvset{firstnumber=auto}%
        \IfFileExists{##3}%
            {\DepyFile{c:##3:mode=verbatim}\VerbatimInput[##1]{##3}}%
            {\PackageWarning{\pytx@packagename}{Input file <##3> doesn't exist}}%
        \endgroup
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
        \stepcounter{\pytx@counter}%
    }%
}
\def\pytx@MakePygmentsInputPyg{
    \newcommand{\inputpygments}[3][]{%
        \begingroup
        \edef\pytx@type{PYG##2}%
        \edef\pytx@cmd{verbatim}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \def\pytx@argspprint{##1}%
        \edef\pytx@session{EXT:##3}%
        \pytx@CheckPygmentsInit{##2}%
        \xdef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
        \pytx@CheckCounter{\pytx@counter}%
        \pytx@WriteCodefileInfoExt
        \IfFileExists{##3}{}{\PackageWarning{\pytx@packagename}%
            {Input file <##3> does not exist}}%
```

```
        \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
        \begingroup
        \pytx@FVSet
        \fvset{firstnumber=auto}%
        \pytx@ConfigPygments
        \ifcsname FV@SV@pytx@\pytx@type @\pytx@session @\pytx@group
                    @\arabic{\pytx@counter}\endcsname
            \UseVerbatim[##1]{pytx@\pytx@type @\pytx@session @\pytx@group
                    @\arabic{\pytx@counter}}%
        \else
            \InputIfFileExists{\pytx@outputdir/\pytx@type_##3_\pytx@group
                    _\arabic{\pytx@counter}.pygtex}{}%
                {\textbf{??~\pytx@packagename~??}%
                    \PackageWarning{\pytx@packagename}{Non-existent Pygments content}}%
        \fi
        \endgroup
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
        \stepcounter{\pytx@counter}%
        \endgroup
    }%
}
```

10.11.4 Creating the Pygments commands and environment

We are almost ready to actually create the Pygments commands and environments. First, though, we create some macros that allow the user to set `fancyvrb` settings, Pygments options, and formatting of Pygments content.

`\setpygmentsfv` This macro allows `fancyvrb` settings to be specified for a Pygments lexer. It takes the lexer name as the optional argument and the settings as the mandatory argument. If no optional argument (lexer) is supplied, then it sets the document-wide `fancyvrb` settings, and is in that case equivalent to `\setpythontexfv`.

```
\newcommand{\setpygmentsfv}[2][]{%
    \Depythontex{cmd:setpygmentsfv:om:n}%
    \ifstrempty{#1}%
        {\gdef\pytx@fvsettings{#2}}%
        {\expandafter\gdef\csname pytx@fvsettings@PYG#1\endcsname{#2}}%
}%
```

`\setpygmentspygopt` This macro allows the Pygments option to be set for a lexer. It takes the lexer name as the first argument and the options as the second argument. If this macro is used multiple times for a lexer, it will write the settings to the code file multiple times. But `pythontex.py` will simply process all settings, and each subsequent set of settings will overwrite any prior settings, so this is not a problem.

```
\newcommand{\setpygmentspygopt}[2][]{%
    \Depythontex{cmd:setpygmentspygopt:om:n}%
    \ifstrempty{#1}%
     {\def\pytx@pygopt{#2}\pgfkeys{/PYTX/gopt/pygopt/.cd, #2}}%
     {\expandafter\gdef\csname pytx@pygopt@PYG#1\endcsname{#2}%
      \pgfkeys{/PYTX/popt/pygopt/.cd, name=#1, #2}}%
}
\@onlypreamble\setpygmentspygopt
```

\setpygmentsprettyprinter This macro parallels \setpythontexprettyprinter. Currently, it takes no optional argument. Eventually, it may be desirable to allow an optional argument that sets the pretty printer on a per-lexer basis.

```
\newcommand{\setpygmentsprettyprinter}[1]{%
    \Depythontex{cmd:setpygmentsprettyprinter:m:n}%
    \ifstrequal{#1}{fancyvrb}{\boolfalse{pytx@opt@pygments}}%
    \ifstrequal{#1}{pygments}{\booltrue{pytx@opt@pygments}}%
}
\@onlypreamble\setpygmentsprettyprinter
\xdef\pytx@macroformatter@PYG{auto}
```

\makepygmentsfv This macro creates the Pygments commands and environment using `fancyvrb`, as a fallback when Pygments is unavailable or when the user desires maximum speed.

```
\def\makepygmentsfv{%
    \pytx@MakePygmentsInlineFV
    \pytx@MakePygmentsEnvFV
    \pytx@MakePygmentsInputFV
}%
\@onlypreamble\makepygmentsfv
```

\makepygmentspyg This macro creates the Pygments commands and environment using Pygments. We must set the bool `pytx@usedpygments` true so that `pythontex.py` knows that Pygments content is present and must be highlighted.

```
\def\makepygmentspyg{%
    \ifbool{pytx@opt@pyginline}%
        {\pytx@MakePygmentsInlinePyg}%
        {\pytx@MakePygmentsInlineFV}%
    \pytx@MakePygmentsEnvPyg
    \pytx@MakePygmentsInputPyg
    \booltrue{pytx@usedpygments}
}%
\@onlypreamble\makepygmentspyg
```

\makepygments This macro uses the two preceding macros to conditionally define the Pygments commands and environments, based on the package Pygments settings.

```
\def\makepygments{%
    \AtBeginDocument{%
        \ifdefstring{\pytx@macroformatter@PYG}{auto}%
            {\ifbool{pytx@opt@pygments}%
                {\makepygmentspyg}{\makepygmentsfv}}{}
        \ifdefstring{\pytx@macroformatter@PYG}{pygments}%
            {\makepygmentspyg}{}
        \ifdefstring{\pytx@macroformatter@PYG}{fancyvrb}%
            {\makepygmentsfv}{}
    }%
}%
\@onlypreamble\makepygments
```

We conclude by actually creating the Pygments commands and environments.

```
\makepygments
```

10.12 Final cleanup

At the end of the document, we need to close files.

```
\AfterEndDocument{%
    \immediate\closeout\pytx@codefile
    \ifbool{pytx@opt@depythontex}{\immediate\closeout\pytx@depyfile}{}%
}
```

10.13 Compatibility with beta releases

The following code maintains compatibility with the beta releases when the package option `beta` is used. It will be retained for several releases before being removed.

```
\ifbool{pytx@opt@beta}{

% Revert changes in stdout and stderr modes
\def\pytx@FetchStdoutfile[#1][#2]#3{%
    \IfFileExists{\pytx@outputdir/#3.stdout}{%
        \ifstrempty{#1}{\input{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{raw}{\input{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{verb}{\VerbatimInput[#2]{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{inlineverb}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stdout}}{}%
        \ifstrequal{#1}{v}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stdout}}{}%
        \DepyFile{p:\pytx@outputdir/#3.stdout:mode=#1}%
    }%
    {\pytx@stdout@warntext
        \PackageWarning{\pytx@packagename}{Non-existent printed content}}%
}
\def\pytx@FetchStderrfile[#1][#2]#3{%
    \IfFileExists{\pytx@outputdir/#3.stderr}{%
        \ifstrequal{#1}{raw}{\input{\pytx@outputdir/#3.stderr}}{}%
        \ifstrempty{#1}{\VerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{verb}{\VerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{inlineverb}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{v}{\BVerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \DepyFile{p:\pytx@outputdir/#3.stderr:mode=#1}%
    }%
    {\textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent stderr content}}%
}


% Old verb environment
\renewcommand{\pytx@MakeVerbFV}[1]{%
    \expandafter\newenvironment{#1verb}{%
        \VerbatimEnvironment
        \expandafter\let\expandafter\pytx@lexer\csname pytx@pyglexer@#1\endcsname
        \Depythontex{env:#1verb:oo|:c}%
        \DepyListing
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{verb}%
        \pytx@SetContext
        \pytx@SetGroupVerb
```

```
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginVerbEnvFV}{\endgroup\pytx@BeginVerbEnvFV[]}%
    }%
    {\end{Verbatim}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}
\renewcommand{\pytx@MakePygEnv}[2]{%
    \expandafter\newenvironment{#1#2}{%
        \VerbatimEnvironment
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{#2}%
        \pytx@SetContext
        \ifstrequal{#2}{block}{\pytx@SetGroup}{}
        \ifstrequal{#2}{verb}{\pytx@SetGroupVerb}{}
        \begingroup
        \obeylines
        \@ifnextchar[{\endgroup\pytx@BeginEnvPyg}{\endgroup\pytx@BeginEnvPyg[]}%
    }%
    {\end{VerbatimOut}%
    \xdef\pytx@stdfile{\pytx@type_\pytx@session_\pytx@group_\arabic{\pytx@counter}}%
    \setcounter{FancyVerbLine}{\value{\pytx@linecount}}%
    \pytx@FVSet
    \ifdefstring{\pytx@fvopttmp}{}{}{\expandafter\fvset\expandafter{\pytx@fvopttmp}}%
    \ifcsname FV@SV@\pytx@counter @\arabic{\pytx@counter}\endcsname
        \UseVerbatim{\pytx@counter @\arabic{\pytx@counter}}%
    \else
        \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.pygtex}{}%
            {\textbf{??~\pytx@packagename~??}%
                \PackageWarning{\pytx@packagename}{Non-existent Pygments content}}%
    \fi
    \setcounter{\pytx@linecount}{\value{FancyVerbLine}}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}%
\renewcommand{\pytx@MakeVerbPyg}[1]{\pytx@MakePygEnv{#1}{verb}}


% Settings macros
\def\setpythontexpyglexer#1#2{%
    \Depythontex{cmd:setpythontexpyglexer:mm:n}%
    \ifcsname pytx@pyglexer@#1\endcsname
        \expandafter\xdef\csname pytx@pyglexer@#1\endcsname{#2}%
    \else
        \PackageError{\pytx@packagename}%
            {Cannot modify a non-existent family}{}%
    \fi
}%
\renewcommand{\setpythontexpygopt}[2]{%
```

```
    \Depythontex{cmd:setpythontexpygopt:mm:n}%
    \ifcsname pytx@pygopt@#1\endcsname
        \expandafter\xdef\csname pytx@pygopt@#1\endcsname{#2}%
    \else
        \PackageError{\pytx@packagename}%
            {Cannot modify Pygments options for a non-existent family}{}%
    \fi
}
\def\setpygmentspygopt#1#2{%
    \Depythontex{cmd:setpygmentspygopt:mm:n}%
    \AtEndDocument{\immediate\write\pytx@codefile{%
        \pytx@delimsettings pygfamily=PYG#1,#1,%
        \string{#2\string}\pytx@delimchar}%
    }%
}


% Old formatters
\def\setpythontexformatter#1#2{%
    \Depythontex{cmd:setpythontexformatter:mm:n}%
    \ifcsname pytx@macroformatter@#1\endcsname
        \ifbool{pytx@opt@depythontex}{}{%
            \expandafter\xdef\csname pytx@macroformatter@#1\endcsname{#2}}
    \else
        \PackageError{\pytx@packagename}%
            {Cannot modify a family that does not exist or does not allow formatter choices}
            {Create the family with \string\makepythontexfamily}%
    \fi
}
\@onlypreamble\setpythontexformatter
\def\setpygmentsformatter#1{%
    \Depythontex{cmd:setpygmentsformatter:m:n}%
    \ifbool{pytx@opt@depythontex}{}{\xdef\pytx@macroformatter@PYG{#1}}}
\@onlypreamble\setpygmentsformatter



}{} %End beta
```