The pythontex package

Geoffrey M. Poore
gpoore@gmail.com

Version 0.9beta3 from 2012/07/17

Abstract

PythonTEX allows Python code entered within a LATEX document to be executed, and the output to be included within the original document. This provides access to the full power of Python from within LATEX, simplifying Python-LATEX workflow and making possible a range of document customization and automation. It also allows macro definitions that mix Python and LATEX code. In addition, PythonTEX provides syntax highlighting for many programming languages via the Pygments Python package.

PythonTEX is fast and user-friendly. Python code is only executed when it has been modified. When code is executed, it automatically attempts to run in parallel. If Python code produces errors, the error message line numbers are synchronized with the LATEX document line numbers, so that it is easy to find the misbehaving code.

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 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.

Package status

PythonTEX is currently in "beta." Almost all features intended for version 0.9 are present, and almost all are fully functional (at least, so far as is known!). Testing is the main task that remains. PythonTEX has been primarily developed and tested under Windows with TEX Live and Python 2.7. It has also been tested with Python 3.2 under Windows, and has been used under OS X (10.7) with MacPort's TEX Live and Python 2.7.

[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 **4**

2 Installing and running **7**
- 2.1 Installing PythonTEX . . . 7
- 2.2 Compiling documents using PythonTEX . . . 9

3 Usage **10**
- 3.1 Package options . . . 10
- 3.2 Code commands and environments . . . 13
  - 3.2.1 Inline commands . . . 13
  - 3.2.2 Environments . . . 14
  - 3.2.3 Default families, PythonTEX utilities, and custom code . . . 15
  - 3.2.4 Asynchronous saving of plots . . . 17
  - 3.2.5 Formatting of typeset code . . . 17
  - 3.2.6 Access to printed content (stdout) and error messages (stderr) 18
- 3.3 Pygments commands and environments . . . 20
- 3.4 General code typesetting . . . 20
  - 3.4.1 Listings float . . . 20
  - 3.4.2 Background colors . . . 21
  - 3.4.3 Referencing code by line number . . . 21
  - 3.4.4 Beamer compatibility . . . 22
- 3.5 Advanced PythonTEX usage . . . 22

4 Questions and answers **23**

5 Troubleshooting **24**

6 The future of PythonTEX **24**
- 6.1 To Do . . . 24
  - 6.1.1 Modifications to make . . . 24
  - 6.1.2 Modifications to consider . . . 25

7 Implementation **26**
- 7.1 Package opening . . . 26
- 7.2 Required packages . . . 26
- 7.3 Package options . . . 27
  - 7.3.1 Autoprint . . . 27
  - 7.3.2 stderr . . . 27
  - 7.3.3 stderrfilename . . . 27
  - 7.3.4 Python's `__future__` module . . . 28
  - 7.3.5 Upquote . . . 28
  - 7.3.6 Fix math spacing . . . 28
  - 7.3.7 Keep temporary files . . . 28
  - 7.3.8 Pygments . . . 29
  - 7.3.9 Python console environment . . . 30

7.3.10 De-PythonTeX 31
7.3.11 Process options 31
7.4 Utility macros and input/output setup 31
7.4.1 Automatic counter creation 32
7.4.2 Code context 32
7.4.3 Code groups 32
7.4.4 File input and output 34
7.4.5 Interface to `fancyvrb` 39
7.4.6 Access to printed content (stdout) 41
7.4.7 Access to stderr 43
7.5 Inline commands 44
7.5.1 Inline core macros 44
7.5.2 Inline command constructors 49
7.6 Environments 51
7.6.1 Block and verbatim environment constructors 52
7.6.2 Code environment constructor 57
7.6.3 Console environment constructor 59
7.7 Constructors for macro and environment families 61
7.8 Default commands and environment families 65
7.9 Listings environment 65
7.10 Pygments for general code typesetting 66
7.10.1 Inline Pygments command 67
7.10.2 Pygments environment 68
7.10.3 Special Pygments commands 70
7.10.4 Creating the Pygments commands and environment 72

1 Introduction

LaTeX can do a lot,[2] but the programming required can sometimes be painful.[3] Also, in spite of the many packages available for LaTeX, the libraries and packages of a general-purpose programming language are lacking. For these reasons, there have been multiple attempts to allow other languages to be used within LaTeX.

- 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 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 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.

[2]TeX is a Turing-complete language.
[3]As I learned in creating this package.

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 line 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 likely wish 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 package Pygments to perform highlighting. Pygments can provide superior syntax highlighting, but `minted` can be slow because all code must be highlighted at each compilation. We need syntax highlighting via Pygments that saves all highlighted code, only re-highlighting when there are modifications. Ideally, we would also like a solution that overcomes some of `minted`'s longstanding issues.[4]

Context awareness

It would be nice for the non-LaTeX code to have at least a minimal awareness of its context in the LaTeX document. For example, it would be nice to know whether code is executing within math mode.

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 be able 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.

Printing

It would be nice for the `print` statement/function,[5] 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.

[4]http://code.google.com/p/minted/issues/list

[5]In Python, `print` was a statement until Python 3.0, when it became a function. The function form is available via import from `__future__` in Python 2.6 and later.

Pure code

LaTeX has a number of special characters (`# $ % & ~ _ ^ \ { }`), which complicates the entry of code in a non-LaTeX language 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 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!"[6]

This language hybridization is not terribly difficult to work around in the SageTeX and SympyTeX cases, and might even be considered a feature in LuaTeX in some 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.

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

- It is open-source and has good cross-platform support.[7]

[6] `http://wiki.contextgarden.net/Programming_in_LuaTeX`

[7] Unfortunately, Sage can only run under Windows within a virtual machine at present; otherwise, an extension of SageTeX might have been tempting. Then again, for general computing, an approach that utilizes pure Python is probably superior.

- 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.
- 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! Although PythonTEX is written to support Python within LATEX, the implementation has been specially crafted so that other languages may be supported in the future. See Section 6 for more details.

2 Installing and running

2.1 Installing PythonTEX

PythonTEX requires a TEX installation. TEX Live or MiKTEX are preferred. PythonTEX requires the `Kpathsea` library, which is available in both of these distributions. The following LATEX packages, with their dependencies, are also required: `fancyvrb`, `etex`, `etoolbox`, `xstring`, `pgfopts`, `newfloat`, and `color` or `xcolor`. If you are creating and importing graphics using Python, you will also need `graphicx`. The `mdframed` package is recommended for enclosing typeset code in boxes with fancy borders and/or background colors.

PythonTEX also requires a Python installation. Python 2.7 is recommended for the greatest compatibility with scientific tools. Python 3.1 and later will work as well. Earlier versions of Python 2 and 3 are not compatible, at least not without some modifications to the PythonTEX scripts. 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 or experiment with the PythonTEX gallery file, the following are also recommended: NumPy, SciPy, matplotlib, and SymPy.

PythonTEX consists of the following files:

- Installer file `pythontex.ins`

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

- Documented LaTeX source file `pythontex.dtx`, from which `pythontex.pdf` and `pythontex.sty` are generated
- Main Python scripts `pythontex2.py` and `pythontex3.py`
- Helper scripts `pythontex_utils2.py`, `pythontex_types2.py`, `pythontex_utils3.py`, `pythontex_types3.py` and `async_pylab_save.py`
- Installation script `pythontex_install_texlive` (for TeX Live)
- README
- Optional batch file `pythontex.bat` for use in launching `pythontex*.py` under Windows

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`. Two versions of all of the Python scipts are supplied, one for Python 2 and one for Python 3.[9]

Until PythonTeX is submitted to CTAN, it must be installed manually. The PythonTeX files should be installed within the TeX directory structure as follows.

- ⟨*TeX tree root*⟩`/doc/latex/pythontex/`
  - `pythontex.pdf`
  - `README`
- ⟨*TeX tree root*⟩`/scripts/pythontex/`
  - `pythontex2.py` and `pythontex3.py`
  - `pythontex_types2.py` and `pythontex_types3.py`
  - `pythontex_utils2.py` and `pythontex_utils3.py`
  - `async_pylab_save.py`
- ⟨*TeX tree root*⟩`/source/latex/pythontex/`
  - `pythontex.dtx`
- ⟨*TeX tree root*⟩`/tex/latex/pythontex/`
  - `pythontex.sty`

[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.

After the files are installed, the system must be made aware of their existence. Run `mktexlsr` or `texhash` to do this. In order for `pythontex*.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/` directory. For TeX Live under Windows, simply copy `bin/win32/runscript.exe` to `bin/win32/pythontex*.exe` to create the wrapper (replace the `*` with the appropriate version).[10]

A Python installation script is provided for use with TeX Live. It may need to be slightly modified based on your system. It performs all steps described above, except for creating a symlink under Linux.

2.2 Compiling documents using PythonTeX

To compile a document that uses PythonTeX, you should run LaTeX, then run `pythontex*.py` (preferably via a symlink, wrapper, or batch file, as described above), and finally run LaTeX again. `pythontex*.py` requires a single command-line argument, which must be passed to it directly or via symlink/wrapper/batch file: the name of the .tex file. The filename can be passed with or without the .tex extension, but no extension is preferred.[11] The file name should be wrapped in double quotes `"` to allow for space characters.[12] For example, under Windows with TeX Live and Python 2.7 we would create the wrapper `pythontex2.exe`. Then we could run PythonTeX on a file ⟨*file name*⟩.tex using the command `pythontex2.exe "`⟨*file name*⟩`"`. In practice, you will probably want to configure your TeX editor with a shortcut key for running PythonTeX.

A second argument specifying the file encoding may also be passed to PythonTeX: `pythontex*.py` ⟨*file*⟩ `--encoding` ⟨*encoding*⟩. Any encoding supported by Python's codecs module may be used. If an encoding is not specified, PythonTeX uses UTF-8. Note that the encoding **must** be used consistently; the .tex source, the PythonTeX output, and any external code files that PythonTeX highlights should all use the same encoding. If support for characters beyond ASCII is required, then the LaTeX packages `fontenc` and `inputenc` should be used.

PythonTeX currently does not provide means to choose between multiple Python installations; it will use the default Python installation. Support for multiple installations is unlikely to be added, since a cross-platform solution would be required. If you need to work with multiple installations, you may wish to modify `pythontex_types*.py` to create additional command and environment families that invoke different versions of Python, based on your system.

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

[11] `pythontex*.py` will be happy to work with a file that does not have the .tex extension, so long as the file cooperates with `pythontex.sty`. In this case, the file extension should **not** be passed to `pythontex*.py`, because it won't be expecting it and won't be able to determine that it is indeed an extension. `pythontex*.py` just needs to know `\jobname`.

[12] 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 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 5 for more details regarding Troubleshooting.

3 Usage

3.1 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, what ⟨*none*⟩ does is described. Each option lists its default setting, if the option is not invoked when the package is loaded.

`autoprint=`⟨*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. In that case, the printed content must be included using the `\printpythontex` or `\stdoutpythontex` commands, or one of their variants.

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 `inlineverb` (`v`) options.

`stderr=`⟨*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*⟩`_`⟨*group*⟩`.`⟨*extension*⟩. For example, an error produced by a `py` command or environment, in the session `mysession`, using the default group (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 imported from `__future__` for all code. `none` imports nothing from `__future__`; `all` imports everything (`absolute_import`, `division`, `print_function`, and `unicode_literals`); and `default` imports everything except `unicode_literals`, since `unicode_literals` can conflict with some packages.

This option has no effect under Python 3.

`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:true` ⟨*none*⟩`=true`

This option fixes extra spacing around `\left` and `\right` in math mode. See the implementation for details.

`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.

`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`.

Note that the package-level Pygments option can be overridded for individual command and environment families, using the `\setpythontexformatter` macro; the `\setpygmentsformatter` provides equivalent functionality for the Pygments commands and environments. 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.

`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.

`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.

`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.

3.2 Code 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 a `console` environment. 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, we describe the different commands and environments, using the `py` base name (the `py` family) as an example.

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*⟩`]`

3.2.1 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 only works 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 try to expand the code). The inline commands can work properly within other macros, but only when all that they contain is also valid LATEX code (and you should stick with curly braces for delimiters in this case).

`\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. 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 value 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.[13]

The text returned by Python must be valid LaTeX code. If you need to include complex text within your document, or if you need to include verbatim text, you should use the `print` statement/function within one of the other commands or environments. The primary reason to use `\py` rather than `print` is that `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.[14]

`\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 (which is the default setting).

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

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

`\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` and `\stdoutpythontex` macros.

3.2.2 Environments

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

This environment encloses code that is executed but not typeset. The second optional argument ⟨*fancyvrb settings*⟩ is irrelevant since nothing is typeset, but

[13]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.

[14]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.

it is accepted to maintain parallelism with the `verb` 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 (which is the default setting).

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

This environment encloses code that is typeset but not executed. The suffix `verb` is an abbreviation for `verbatim`.

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

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.

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

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. Unlike the other environments, `pyconsole` has no inline equivalent. Currently, non-ASCII characters are not supported in `console` environments under Python 2.

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.

3.2.3 Default families, PythonTEX utilities, and custom code

By default, three command and environment families are defined.

- Python
  - Base name `py`: `\py`, `\pyc`, `\pyv`, `\pyb`, `pycode`, `pyverb`, `pyblock`, `pyconsole`
  - Imports: None.
- Python + pylab (matplotlib module)
  - Base name `pylab`: `\pylab`, `\pylabc`, `\pylabv`, `\pylabb`, `pylabcode`, `pylabverb`, `pylabblock`, `pylabconsole`
  - 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 *`.
- Python + SymPy
  - Base name `sympy`: `\sympy`, `\sympyc`, `\sympyv`, `\sympyb`, `sympycode`, `sympyverb`, `sympyblock`, `sympyconsole`
  - 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 families import `absolute_import`, `division`, and `print_function` from `__future__` by default. This may be changed using the package option `pyfuture`. 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 `\setpythontexcustomcode` command (described below), or simply enter the import code immediately at the beginning of a session.

All families import `pythontex_utils*.py`, and create an instance of the PythonTeX utilities class called `pytex`. This provides various utilities for interfacing with LaTeX. In particular, it provides an interface for determining how Python objects are converted into strings in commands such as `\py`. The `pytex.set_formatter(`⟨*formatter*⟩`)` method determines 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 $\times$, but only when math is in displaystyle.[15] 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 PythonTeX utilities formatter may also 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`.

The context-sensitive interface to SymPy's `LatexPrinter` is always available via `pytex.sympy_latex()`. If you wish to use it outside the `sympy` com-

[15]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.

mand and environment family, you must initialize it before use via the command `pytex.init_sympy_latex()`.

`\setpythontexcustomcode{⟨family⟩}{⟨quoted list⟩}`

This macro allows custom code to be added to all sessions within a command and environment family. ⟨*quoted list*⟩ should be a comma-separated list of lines of code, each line enclosed in quotes (single or double). For example, `\setpythontexcustomcode{py}{'a=1', 'b=2'}` would create the variables `a` and `b` within all sessions of the `py` family, by invisibly adding the following lines at the beginning of each session:

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

Note that custom code is executed, but never typeset. Only code that is actually entered within a `block` (or `verb`) command or environment is every 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.

⟨*quoted list*⟩ may contain imports from `__future__`. It is best if these are the first elements in the list, since future imports are only possible at the very beginning of a Python script. This is not strictly required, however. When PythonTEX writes the individual scripts that are executed, it checks ⟨*quoted list*⟩ for future imports, and automatically moves them to the appropriate location.

⟨*quoted list*⟩ may **not** contain LATEX macros. ⟨*quoted list*⟩ is interpreted as verbatim content, since in general the custom code will not be valid LATEX.

3.2.4 Asynchronous saving of plots

To avoid locking up the main program flow when saving figures the utility `async_pylab_save.py` is included.

This works by making `asp = AsyncPylabSave()` available in every family and then calling `asp.join()` at the end. Calling `asp.savefig(...)` will call `plt.savefig(...)` without blocking the main program (using multiprocessing). Or eqvivalently `asp.savefig(filename, fig=fig, **kwargs)`, where fig is a figure handle, will work as `fig.savefig(filename, **kwargs)`.

Please note that the figure will be closed after saving, so manipulation of the figure after calling save will not work. Saving multiple figures to the same filename will of course have unexpected results!

TODO: I guess this feature only makes sense in the pylab family...

3.2.5 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.

`\setpythontexformatter{⟨family⟩}{⟨formatter⟩}`

This should generally not be needed. It allows the formatter used by ⟨*family*⟩ to be set. Valid options for ⟨*formatter*⟩ are `auto`, `fancyvrb`, and `pygments`. Using `auto` means that the formatter will be determined based on the package `pygments` option. Using either of the other two options will force ⟨*family*⟩ to use that formatter, regardless of the package-level options. By default, families use the `auto` formatter.

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

This allows the Pygments lexer to be set for ⟨*family*⟩. ⟨*pygments lexer*⟩ should 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.

`\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.

3.2.6 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[⟨verbatim options⟩][⟨fancyvrb options⟩]`

`\stdoutpythontex[⟨verbatim options⟩][⟨fancyvrb 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.

By default, printed content is brought in raw—it is pulled in directly from the external file in which it is saved and interpreted 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 consider the `verb` and `inlineverb` (also accesible as `v`) options, which bring in code verbatim. If code is brought in verbatim, then ⟨*fancyvrb options*⟩ are applied to it.

`\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*⟩`}`. ⟨*verbatim options*⟩ must be either `verb` or `inlineverb` (also accessible as `v`), specifying how content is brought in verbatim. If content is brought in verbatim, then ⟨*fancyvrb options*⟩ are applied.

`\stderrpythontex[⟨verbatim options⟩][⟨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. ⟨*verbatim options*⟩ may be set to `verb` (default), `inlineverb` (or `v`), and `raw`. 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 `\py` command or 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[⟨verbatim options⟩][⟨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.

3.3 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⟩}`

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.

`\setpygmentsformatter{⟨formatter⟩}`

This usually should not be needed. It allows the formatter for Pygments content to be set. Valid options for ⟨*formatter*⟩ are `auto`, `fancyvrb`, and `pygments`. Using `auto` means that the formatter will be determined based on the package `pygments` option. Using either of the other two options will force Pygments content to use that formatter, regardless of the package-level options. The `auto` formatter is used by default.

3.4 General code typesetting

3.4.1 Listings float

`listing`

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`.

3.4.2 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.

3.4.3 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 references are created within comments, the references 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!\ref{lst:important}
```

would appear as

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

If a comment only contains the `\ref` command, then only the comment character `#` would actually be visible in the typeset code.

3.4.4 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 them back in. This use of an external file breaks PythonTEX's error line number synchronization. A fix is expected in a future version of PythonTEX.

3.5 Advanced PythonTEX usage

`\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 conventient 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). ⟨*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.[16]

This command can only be used in the preamble.

`\setpythontexoutputdir{⟨output directory⟩}`

By default, PythonTEX saves all 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.

`\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 output 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 output directory `pythontex-files-my_file`. 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`. Or you could set the working directory to `plots` using `\setpythontexworkingdir{plots}`, so that all content would automatically be saved there.

If you want your working directory to be the document root directory, you should use a period (`.`) for ⟨*working directory*⟩: `\setpythontexworkingdir{.}`.

4 Questions and answers

Will you add a plot command that automates the saving and inclusion of plots or other graphics created by matplotlib or similar packages? There are no plans to add a plot command like `\pyplot`. A plot command would add a little convenience, but at the expense of power. Automated saving would give the plot an automatically generated name, making the file harder to find. Automated inclusion would involve collecting a lot of settings and then passing them on to `\includegraphics`, perhaps within `figure` and `center` environments. It is much simpler for the user to choose a meaningful name and then include the file in the desired manner.

[16]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 and should be avoided.

5 Troubleshooting

A more extensive troubleshooting section will be added in the future.

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. The simplest 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.

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}`.

6 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 are in progress or under consideration.

6.1 To Do

6.1.1 Modifications to make

- Fix error line number synchronization with Beamer. 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`.
- User-defined custom commands and environments for general Pygments typesetting.
- Testing under Linux.
- Additional documentation for the Python code (Sphinx?).
- Establish a testing framework.
- Keep track of any Pygments errors for future runs, so we know what to run again? How easy is it to get Pygments errors? There don't seem to have been any in any of the testing so far.
- It might nice to include some methods in the PythonTEX utilities for formatting numbers (especially with SymPy and Pylab). Also, it would be nice to have shortcuts for Matplotlib2tikz integration.

6.1.2 Modifications to consider

- 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`?

- Consider support for executing other languages. It might be nice to support a few additional languages at a basic level by version 1.0. Languages currently under consideration: Perl, MATLAB, Mathematica, Lua, Sage, R. But note that there are ways to interface with many or perhaps all of these from within Python. Also, consider general command line-access, similar to `\write18`. The `bashful` package can do some nice command-line things. But it would probably require some real finesse to get that kind of `bash` access cross-platform. Probably could figure out a way to access Cygwin's bash or GnuWin32 or MSYS.

- 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. Maybe there should be a `\typesetpythontex` command that parallels `\printpythontex`?

- 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.

- Consider methods of taking PythonTeX documents and removing their dependence on `pythontex.sty`. Something that could convert a PythonTeX document into a document that would be more readily acceptable by a publisher. SageTeX has something like this.

Acknowledgements

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

7 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.

7.1 Package opening

We begin according to custom by specifying the version of LaTeX that we require and stating the package that we are providing. We also store the name of the package in a macro for later use in warnings and error messages.

```
\NeedsTeXFormat{LaTeX2e}[1999/12/01]
\ProvidesPackage{pythontex}[2012/07/17 v0.9beta3]
\newcommand{\pytx@packagename}{PythonTeX}
```

7.2 Required packages

A number of packages are required. `fancyvrb` is used to typeset all code that is not inline, and its internals are used to format inline code as well. `etex` provides extra registers, to avoid the (probably unlikely) possibility that the many counters required by PythonTeX will exhaust the supply. `etoolbox` is used for string comparison and boolean flags. `xstring` provides the `\tokenize` macro. `pgfopts` is used to process package options, via the `pgfkeys` package. `newfloat` allows the creation of a floating environment for code listings. `xcolor` or `color` is needed for syntax highlighting with Pygments.

```
\RequirePackage{fancyvrb}
\RequirePackage{etex}
\RequirePackage{etoolbox}
\RequirePackage{xstring}
\RequirePackage{pgfopts}
\RequirePackage{newfloat}
\AtBeginDocument{\@ifpackageloaded{color}{}{\RequirePackage{xcolor}}}
```

7.3 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`.

7.3.1 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}}
```

7.3.2 stderr

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

```
\newbool{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}}
\pgfkeys{/PYTX/pkgopt/stderr/false/.code=\boolfalse{pytx@opt@stderr}}
```

7.3.3 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}}
```

7.3.4 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}}
```

7.3.5 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}}
```

7.3.6 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.[17]

```
\newbool{pytx@opt@fixlr}
\booltrue{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}}
```

7.3.7 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 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}
```

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

```
\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}}
```

7.3.8 Pygments

`pytx@opt@pygments`

By default, PythonTEX uses `fancyvrb` to typeset code. This provides nice formatting and font options, but no syntax highlighting. The `pygments` option determines 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.[18]

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

Pygments has been used previously to highlight code for LATEX, most notably in the `minted` package.

```
\newbool{pytx@opt@pygments}
\booltrue{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@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}}

```

`\pytx@pygopt`

We also need a way to specify Pygments options at the package level. This is accomplished via the `pygopt` option: `pygopt`={⟨*options*⟩}. 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=`⟨*style*⟩, 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}
```

[18] 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.

The usage of capitalized `True` and `False` is more pythonic, but is not strictly require.

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.

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

`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/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@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.[19] To deal with this issue, we create the `fvextfile` option. This option takes an integer, `fvextfile`=⟨*integer*⟩. All content that is more than ⟨*integer*⟩ 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.

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=\def\pytx@fvextfile{#1}}
```

7.3.9 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}}
```

[19]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.

```
\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}}
```

7.3.10 De-PythonTeX

`pytx@opt@depythontex` This option governs whether PythonTEX creates 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.

```
\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}}
```

7.3.11 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.

```
\ifbool{pytx@opt@fixlr}{
    \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}}{}
```

7.4 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.

7.4.1 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
}
```

7.4.2 Code context

`\pytx@context` `\pytx@SetContext` `\definepythontexcontext` 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 `\definepythontexcontext` 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{\definepythontexcontext}[1]{%
    \def\pytx@SetContext{%
        \edef\pytx@context{#1}%
    }%
}
\definepythontexcontext{}
\@onlypreamble\definepythontexcontext
```

7.4.3 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]{%
    \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}
```

7.4.4 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`,[20] 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 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.[21]

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

[21] 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.

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. If the graphicx package is loaded, the output directory is also added to the graphics path, so that files in the output directory may be included within the main document without the necessity of specifying path information. The command \setpythontexoutputdir is only allowed in the preamble, because the location of PythonTEX content must be specified before the body of the document is typeset. If \setpythontexoutputdir is not invoked by the user, then we automatically invoke it at the beginning of the document to set the default directory name.

```
\newcommand{\setpythontexoutputdir}[1]{%
    \def\pytx@outputdir{#1}%
    \AtBeginDocument{\@ifpackageloaded{graphicx}{\graphicspath{{#1/}}}{}}%
}
\@onlypreamble\setpythontexoutputdir
\AtBeginDocument{%
    \ifcsname pytx@outputdir\endcsname\else
        \setpythontexoutputdir{pythontex-files-\pytx@jobname}\fi
}
```

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 output directory. That way, any files saved in the current working directory will be in the PythonTEX output directory, and will thus be kept separate. But in some cases the user may wish to specify a different working directory, such as the document root.

```
\newcommand{\setpythontexworkingdir}[1]{%
    \def\pytx@workingdir{#1}%
}
\@onlypreamble\setpythontexworkingdir
\AtBeginDocument{%
    \ifcsname pytx@workingdir\endcsname\else
        \setpythontexworkingdir{\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.[22] 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.

```
\AfterEndPreamble{%
    \ifbool{pytx@usedpygments}%
        {\InputIfFileExists{\pytx@outputdir/\pytx@jobname.pytxpyg}{}{}}{}%
}
```

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{%
    \InputIfFileExists{\pytx@outputdir/\pytx@jobname.pytxmcr}{}{}%
}
```

[22]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.

`\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.[23]

```
\newwrite\pytx@codefile
```

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 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 beginning of the document. The order in which the settings are written is not significant, but for symmetry it should mirror the order in which they were defined.

```
\AtBeginDocument{
```

[23]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.

```
    \immediate\openout\pytx@codefile=\jobname.pytxcode
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings outputdir=\pytx@outputdir\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings workingdir=\pytx@workingdir\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings stderr=%
        \ifbool{pytx@opt@stderr}{true}{false}\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings stderrfilename=\pytx@opt@stderrfilename\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings keeptemps=\pytx@opt@keeptemps\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pyfuture=\pytx@opt@pyfuture\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pygments=%
        \ifbool{pytx@opt@pygments}{true}{false}\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pyglexer=\pytx@pyglexer\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pygmentsglobal:\string{\pytx@pygopt\string}\pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings fvextfile=\pytx@fvextfile \pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pyconbanner=\pytx@opt@pyconbanner \pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings pyconfilename=\pytx@opt@pyconfilename \pytx@delimchar}%
    \immediate\write\pytx@codefile{%
        \pytx@delimsettings depythontex=%
        \ifbool{pytx@opt@depythontex}{true}{false}\pytx@delimchar}%
}
```

`\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. While the standard form employs a counter that is incremented elsewhere, the version for external files substitutes a zero (0) for the counter, because each external file must be unique in name and thus numbering via a counter is redundant.[24]

```
\def\pytx@WriteCodefileInfo{%
    \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\the\inputlineno\pytx@delimchar}%
}
```

[24]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.

```
\newcommand{\pytx@WriteCodefileInfoExt}[1][]{%
    \immediate\write\pytx@codefile{\pytx@delim\pytx@type\pytx@delimchar%
        \pytx@session\pytx@delimchar\pytx@group\pytx@delimchar%
        0\pytx@delimchar\pytx@cmd\pytx@delimchar%
        \pytx@context\pytx@delimchar\the\inputlineno\pytx@delimchar#1}%
}
```

At the end of the document, we need to close the code file, so we go ahead and issue the commands for that. From now on, we may simply write to the code file when necessary, and need not otherwise concern ourselves with the file.

```
\AtEndDocument{%
    \immediate\closeout\pytx@codefile
}
```

7.4.5 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.[25]

[25] 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.

```
\newcommand{\setpythontexfv}[2][]{%
    \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.

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}}%
    \@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
```

7.4.6 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 environments 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.[26] 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[`⟨*verbatim options*⟩`][`⟨*fancyvrb options*⟩`]{`⟨*file path*⟩`}`.

[26]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.

```
\def\pytx@FetchStdoutfile[#1][#2]#3{%
    \IfFileExists{\pytx@outputdir/#3.stdout}{%
        \ifstrequal{#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}}{}%
    }%
    {\textbf{??~\pytx@packagename~??}%
        \PackageWarning{\pytx@packagename}{Non-existent printed content}}%
}
```

`\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{%
    \@ifnextchar[{\pytx@Stdout}{\pytx@Stdout[]}%
}
\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}%
}
\let\printpythontex\stdoutpythontex
```

`\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#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
}
\let\saveprintpythontex\savestdoutpythontex
```

`\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{%
    \@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
}
\let\useprintpythontex\usestdoutpythontex
```

7.4.7 Access to stderr

We need access to stderr, if it is enabled via the package `stderr` 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}{}{\VerbatimInput[#2]{\pytx@outputdir/#3.stderr}}{}%
        \ifstrequal{#1}{raw}{\input{\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}}{}%
    }%
    {\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{%
    \@ifnextchar[{\pytx@Stderr}{\pytx@Stderr[]}%
}
\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{%
    \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{%
    \@ifnextchar[{\pytx@UseStderr}{\pytx@UseStderr[]}%
}
\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
}
```

7.5 Inline commands

7.5.1 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.[27] 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]{%
```

[27]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.

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

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.

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}}%
    \edef\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}%
}%
```

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`. In this case, 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. `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:[28]

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

```
\def\pytx@InlineShowFV{%
    \begingroup
```

[28]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.

```
    \let\do\@makeother\dospecials
    \catcode`\ =\active
    \catcode`\^^I=\active
    \tokenize{\pytx@argretok}{\pytx@argdetok}%
    \endgroup
    \begingroup
    \pytx@FVSet
    \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
    \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.

```
\ifbool{pytx@opt@autoprint}%
    {\def\pytx@InlineAutoprint{%
        \InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}{}{}}}%
    {\let\pytx@InlineAutoprint\@empty}
```

`\pytx@InlineMacroprint` This macro brings in "printed" content that is brought in via macros in the .pytxmcr file.

```
\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
    \else
        \textbf{??}%
        \PackageWarning{\pytx@packagename}{Missing autoprint content}%
    \fi
}
```

7.5.2 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".

`\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\newcommand\expandafter{\csname #1b\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inlineb}%
        \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\newcommand\expandafter{\csname #1b\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inlineb}%
        \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\newcommand\expandafter{\csname #1v\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inlinev}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowFV
        \let\pytx@InlineSave\@empty
        \let\pytx@InlinePrint\@empty
        \pytx@Inline
    }%
}%
\newcommand{\pytx@MakeInlinevPyg}[1]{%
    \expandafter\newcommand\expandafter{\csname #1v\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inlinev}%
        \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\newcommand\expandafter{\csname #1c\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inlinec}%
        \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@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\newcommand\expandafter{\csname #1\endcsname}{%
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{inline}%
        \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
```

7.6 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.

7.6.1 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
        \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]}%
    \edef\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 `verb` 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{#1verb}{%
        \VerbatimEnvironment
        \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}%
    }%
}
```

`\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]}%
    \edef\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}{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}%
    }%
}%
```

\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]}%
    \edef\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 pyverb between two Verbatim's).

```
\def\pytx@BeginEnvPyg@i[#1]{%
    \def\pytx@fvopttmp{#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}{verb}}
```

7.6.2 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@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
        \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}}%
    \ifbool{pytx@opt@autoprint}%
        {\InputIfFileExists{\pytx@outputdir/\pytx@stdfile.stdout}{}{}}{}%
    \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    \stepcounter{\pytx@counter}%
    }%
}%
```

`\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]}%
    \edef\pytx@counter{pytx@\pytx@type @\pytx@session @\pytx@group}%
    \pytx@CheckCounter{\pytx@counter}%
    \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]{%
    \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
```

7.6.3 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.

`\pytx@MakeConsoleFV`

```
\newcommand{\pytx@MakeConsFV}[1]{%
    \expandafter\newenvironment{#1console}{%
        \VerbatimEnvironment
        \xdef\pytx@type{#1}%
        \edef\pytx@cmd{console}%
        \pytx@SetContext
        \pytx@SetGroupCons
        \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}}%
    \else
        \InputIfFileExists{\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}%
    }%
}
```

`\pytx@BeginConsEnvFV`

```
\def\pytx@BeginConsEnvFV[#1]{%
    \ifstrempty{#1}{\edef\pytx@session{default}}{\StrSubstitute{#1}{:}{-}[\pytx@session]}%
    \edef\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}%
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
    \begin{VerbatimOut}%
}%
```

`\pytx@MakeConsPyg` The console environment saves code and then brings back the result of console-style evaluation. Whether Pygments is used to highlight the code depends on the family settings, so the Pygments and non-Pygments forms of the environment are identical.

```
\let\pytx@MakeConsPyg\pytx@MakeConsFV
```

7.7 Constructors for macro and environment families

Everything is now in place to create inline 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, based on a base name. For example, we need a way to easily create all commands and environments based off of the py base name.

\makepythontexfamilyfv This is a mass constructor for all commands and environments. It takes a single mandatory argument: a base name. It creates almost all commands and environments using the base name; the console environment is created conditionally, based on an optional argument. The console environment is only created conditionally because support for it will probably be very limited if languages other than Python are added in the future. The macro also creates fancyvrb settings corresponding to the family, and sets them to a null default.

The macro checks for the base name PYG, which is not allowed. This is for two reasons. First, given that the family py is already defined by default, another family with such a similar name would not be a good idea. Second, and more importantly, the prefix PYG is used for other purposes. Although PythonTeX is primarily intended for executing and typesetting Python code, provision has also been made for typesetting code in any language supported by Pygments. The PYG prefix is used by the macros that perfom that function.

The constructor macro should only be allowed in the preamble, since commands and environments must be defined before the document begins.

```
\newcommand{\makepythontexfamilyfv}[2][]{%
    \IfBeginWith{#2}{PYG}%
        {\PackageError{\pytx@packagename}%
            {Attempt to create macros with reserved prefix PYG}{}}{}%
    \pytx@MakeInlinebFV{#2}%
    \pytx@MakeInlinevFV{#2}%
    \pytx@MakeInlinecFV{#2}%
    \pytx@MakeInlineFV{#2}%
    \pytx@MakeBlockFV{#2}%
    \pytx@MakeVerbFV{#2}%
    \pytx@MakeCodeFV{#2}%
    \ifstrequal{#1}{console}{\pytx@MakeConsFV{#2}}{}%
    \ifstrequal{#1}{all}{\pytx@MakeConsFV{#2}}{}%
    \setpythontexfv[#2]{}%
}
\@onlypreamble\makepythontexfamilyfv
```

\makepythontexfamilypyg Creating a family of Pygments commands and environments is a little more involved. This macro takes three mandatory arguments: the base name, the Pygments lexer to be used, and Pygments options for typesetting. Currently, three options may be passed to Pygments 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. Note that these settings may be overridden by the package option `pygments`. Again, the `console` environment is created conditionally, based on an optional argument.

After checking for the disallowed prefix `PYG`, we begin by creating all commands and environments, and creating a macro in which to store default `fancyvrb` setting. We save the Pygments settings in a macro of the form `\pytx@pygopt@`⟨*base name*⟩. We also set the bool `pytx@usedpygments` to true, so that Pygments content will be inputted at the beginning of the document. Then we request that the base name, lexer, and any Pygments settings be written to the code file at the beginning of the document, so that Pygments can access them. The options are saved in a macro, and then the macro is saved to file only at the beginning of the document, so that the user can modify default options for default code and environment families.

This macro should only be allowed in the preamble.

```
\newcommand{\makepythontexfamilypyg}[4][]{%
    \IfBeginWith{#2}{PYG}%
        {\PackageError{\pytx@packagename}%
            {Attempt to create macros with reserved prefix PYG}{}}{}%
    \ifbool{pytx@opt@pyginline}%
        {\pytx@MakeInlinebPyg{#2}%
            \pytx@MakeInlinevPyg{#2}}%
        {\pytx@MakeInlinebFV{#2}%
            \pytx@MakeInlinevFV{#2}}%
    \pytx@MakeInlinecPyg{#2}%
    \pytx@MakeInlinePyg{#2}%
    \pytx@MakeBlockPyg{#2}%
    \pytx@MakeVerbPyg{#2}%
    \pytx@MakeCodePyg{#2}%
    \ifstrequal{#1}{console}{\pytx@MakeConsPyg{#2}}{}%
    \ifstrequal{#1}{all}{\pytx@MakeConsPyg{#2}}{}%
    \setpythontexfv[#2]{}%
    \booltrue{pytx@usedpygments}%
    \expandafter\xdef\csname pytx@pygopt@#2\endcsname{#4}%
    \AtBeginDocument{\immediate\write\pytx@codefile{%
        \pytx@delimsettings pygmentsfamily:#2,#3,%
        \string{\csname pytx@pygopt@#2\endcsname\string}\pytx@delimchar}%
    }%
}
\@onlypreamble\makepythontexfamilypyg
```

`\setpythontexpyglexer` We need to be able to reset the lexer associated with a family after the family has already been created.

```
\def\setpythontexpyglexer#1#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]{%
    \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
}
\@onlypreamble\setpythontexpygopt
```

`\makepythontexfamily` While the `\makepythontexfamilyfv` and `\makepythontexfamilypyg` macros allow the creation of families that use `fancyvrb` and Pygments, respectively, we want to be able to create families that can switch between the two possibilities, based on the package option `pygments`. In some cases, we may want to force a family to use either `fancyvrb` or Pygments, but generally we will want to be able to control the method of typesetting of all families at the package level. We create a new macro for this purpose. This macro takes the same arguments that `\makepythontexfamilypyg` does: the family base name, the lexer to be used by Pygments, and Pygments options for typesetting, plus an optional argument governing the `console` environment. The actual creation of macros is delayed using `\AtBeginDocument`, so that the user has the option to choose whether `fancyvrb` or Pygments usage should be forced for the family.

This macro should always be used for defining new families, unless there is a particular reason to always force `fancyvrb` or Pygments usage.

```
\newcommand{\makepythontexfamily}[4][]{%
    \expandafter\xdef\csname pytx@macroformatter@#2\endcsname{auto}
    \expandafter\xdef\csname pytx@pyglexer@#2\endcsname{#3}
    \expandafter\xdef\csname pytx@pygopt@#2\endcsname{#4}
    \AtBeginDocument{%
        \ifcsstring{pytx@macroformatter@#2}{auto}{%
            \ifbool{pytx@opt@pygments}%
                {\makepythontexfamilypyg[#1]{#2}{\csname pytx@pyglexer@#2\endcsname}%
                    {\csname pytx@pygopt@#2\endcsname}}%
                {\makepythontexfamilyfv[#1]{#2}}}{}%
        \ifcsstring{pytx@macroformatter@#2}{fancyvrb}%
            {\makepythontexfamilyfv[#1]{#2}}{}%
        \ifcsstring{pytx@macroformatter@#2}{pygments}%
            {\makepythontexfamilypyg[#1]{#2}{\csname pytx@pyglexer@#2\endcsname}%
                {\csname pytx@pygopt@#2\endcsname}}{}%
    }%
}
\@onlypreamble\makepythontexfamily
```

`\setpythontexformatter` We need to be able to reset the formatter used by a family among the options `auto`, `fancyvrb`, and `pygments`.

```
\def\setpythontexformatter#1#2{%
    \ifcsname pytx@macroformatter@#1\endcsname
        \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
```

`\setpythontexcustomcode` One additional customization macro is needed. This macro allows custom code to be added to the start of all code for a specified family. It applies to **all** commands and environments within a family, including the `console` environment. Custom code is included as a comma-delimited list of quoted code strings. Some catcode trickery is required to allow the code to contain arbitrary characters. Currently, the code may contain anything except unmatched curly braces.

There are multiple ways that a custom code macro could be implemented. The current approach is based on several factors. Usually, only a few lines of custom code should need to be specified (if you have a lot of code, you should create a separate file and import it). For this use case, an inline macro taking quoted strings should be adequate. Also, such a macro is typical of the type of macros that appear in the preamble. An environment in which verbatim code is written (say, `pythontexcustomcode`) could be created. This would require that an environment be used in the preamble, which is possible but uncommon.

```
\def\setpythontexcustomcode#1{%
    \begingroup
    \let\do\@makeother\dospecials
    \catcode`\{=1
    \catcode`\}=2
    \catcode`\^^M=10\relax
    \pytx@SetCustomCode{#1}%
}
\long\def\pytx@SetCustomCode#1#2{%
    \endgroup
    \AtBeginDocument{%
```

```
        \immediate\write\pytx@codefile{%
            \pytx@delimsettings customcode:#1,%
            [\detokenize{#2}]\pytx@delimchar}%
    }%
}
\@onlypreamble\setpythontexcustomcode
```

7.8 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[all]{py}{python}{}
\makepythontexfamily[all]{sympy}{python}{}
\makepythontexfamily[all]{pylab}{python}{}
```

7.9 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{
    \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 conflicting "listing" environment already exists}%
                {Use \string\setpythontexlistingenv to create a custom environment}}%
    \else
        \ifbool{pytx@listingenv}{}{\DeclareFloatingEnvironment[fileext=lopytx]{listing}}
    \fi
}
```

7.10 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

7.10.1 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@type{PYG##1}%
        \edef\pytx@cmd{inlinev}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowFV
        \let\pytx@InlineSave\@empty
        \let\pytx@InlinePrint\@empty
        \ifcsname pytx@fvsettings@\pytx@type\endcsname
        \else
            \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
        \fi
        \pytx@Inline[]%
    }%
}
\def\pytx@MakePygmentsInlinePyg{%
    \newcommand{\pygment}[1]{%
        \edef\pytx@type{PYG##1}%
        \edef\pytx@cmd{inlinev}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \let\pytx@InlineShow\pytx@InlineShowPyg
        \let\pytx@InlineSave\pytx@InlineSaveCode
        \let\pytx@InlinePrint\@empty
        \ifcsname pytx@fvsettings@\pytx@type\endcsname
        \else
            \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
        \fi
        \pytx@Inline[]
    }%
}
```

7.10.2 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}%
    \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@cmd{verb}%
    \edef\pytx@session{default}%
    \edef\pytx@linecount{pytx@\pytx@type @\pytx@session @\pytx@group @line}%
    \pytx@CheckCounter{\pytx@linecount}%
    \ifcsname pytx@fvsettings@\pytx@type\endcsname
    \else
        \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
    \fi
    \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}}%
        \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}%
    \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{verb}%
    \edef\pytx@session{default}%
    \edef\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
    \ifcsname pytx@fvsettings@\pytx@type\endcsname
    \else
        \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
    \fi
    \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
    \let\FVB@VerbatimOut\pytx@FVB@VerbatimOut
    \let\FVE@VerbatimOut\pytx@FVE@VerbatimOut
    \begin{VerbatimOut}%
}
```

7.10.3 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*⟩.[29] Next we define a `fancyvrb` settings macro for the type if it does not already exist. We write info to the code file using `\pytx@WriteCodefileInfoExt`, which writes the standard info to the code file but uses zero for the instance, since external files that are not executed can only have one instance.

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.

[29]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.

```
\def\pytx@MakePygmentsInputFV{
    \newcommand{\inputpygments}[3][]{%
        \edef\pytx@type{PYG##2}%
        \edef\pytx@cmd{verb}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \edef\pytx@session{EXT:##3}%
        \ifcsname pytx@fvsettings@\pytx@type\endcsname
        \else
            \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
        \fi
        \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
        \begingroup
        \pytx@FVSet
        \fvset{firstnumber=auto}%
        \IfFileExists{##3}%
            {\VerbatimInput[##1]{##3}}%
            {\PackageWarning{\pytx@packagename}{Input file <##3> doesn't exist}}%
        \endgroup
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    }%
}
\def\pytx@MakePygmentsInputPyg{
    \newcommand{\inputpygments}[3][]{%
        \edef\pytx@type{PYG##2}%
        \edef\pytx@cmd{verb}%
        \pytx@SetContext
        \pytx@SetGroupVerb
        \edef\pytx@session{EXT:##3}%
        \ifcsname pytx@fvsettings@\pytx@type\endcsname
        \else
            \expandafter\gdef\csname pytx@fvsettings@\pytx@type\endcsname{}%
        \fi
        \pytx@WriteCodefileInfoExt
        \IfFileExists{##3}{}{\PackageWarning{\pytx@packagename}%
            {Input file <##3> does not exist}{}}
        \setcounter{pytx@FancyVerbLineTemp}{\value{FancyVerbLine}}%
        \begingroup
        \pytx@FVSet
        \fvset{firstnumber=auto}%
        \ifcsname FV@SV@pytx@\pytx@type @\pytx@session @\pytx@group @0\endcsname
            \UseVerbatim[##1]{pytx@\pytx@type @\pytx@session @\pytx@group @0}%
        \else
            \InputIfFileExists{\pytx@outputdir/##3_##2.pygtex}{}%
                {\textbf{??~\pytx@packagename~??}%
                    \PackageWarning{\pytx@packagename}{Non-existent Pygments content}}%
        \fi
        \endgroup
        \setcounter{FancyVerbLine}{\value{pytx@FancyVerbLineTemp}}%
    }%
}
```

7.10.4 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][]{%
    \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.

```
\def\setpygmentspygopt#1#2{%
    \AtBeginDocument{\immediate\write\pytx@codefile{%
        \pytx@delimsettings pygmentsfamily:PYG#1,#1,%
        \string{#2\string}\pytx@delimchar}%
    }%
}
\@onlypreamble\setpygmentspygopt
```

`\setpygmentsformatter` This macro sets the formatter (Pygments or `fancyvrb`) that is used by the Pygments commands and environment. There are three options: `auto`, which depends on the package `pygments` option; and `pygments` and `fancyvrb`, which override the package option. By default, `auto` is used. Since the package Pygments option is true by default, this means that Pygments content will automatically be highlighted by Pygments, and that the behavior of Pygments content will follow the package option.

The parallel PythonTeX command allows for setting the formatting for individual families. The rationale is that the user might use a PythonTeX family for executing and typesetting code, but not wish to use Pygments to highlight the code. The Pygments command does not allow for setting the formatter for individual lexers, which would be the closest parallel to that behavior. The primary reason that the user might use the Pygments commands and environments is for highlighting purposes. Otherwise, there is little reason not to use `fancyvrb` or an equivalent directly.[30]

[30] The user might want to use Pygments commands for the `fancyvrb` style and line num-

```
\def\setpygmentsformatter#1{\xdef\pytx@macroformatter@PYG{#1}}
\@onlypreamble\setpygmentsformatter
\setpygmentsformatter{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 as well as the `\setpygmentsformatter` command that may be used to override the package 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
```

bering continuity they provide. In that case, a custom Pygments lexer, with formatter set to `fancyvrb` should be considered. The verbatim part of a PythonTeX family could also be used. Alternatively, the Pygments `TextLexer` (aka `text`) may be used; it is a null lexer, so nothing is highlighted.