The \texttt{fvextra} package

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

v1.7.0 from 2024/05/16

\textbf{Abstract}

\texttt{fvextra} provides several extensions to \texttt{fancyvrb}, including automatic line breaking and improved math mode. \texttt{\Verb} is reimplemented so that it works (with a few limitations) inside other commands, even in movable arguments and PDF bookmarks. The new command \texttt{\EscVerb} is similar to \texttt{\Verb} except that it works everywhere without limitations by allowing the backslash to serve as an escape character. \texttt{fvextra} also patches some \texttt{fancyvrb} internals.
## Contents

1 Introduction 5

2 Usage 5
   2.1 Pandoc compatibility 5

3 General options 6

4 General commands 11
   4.1 Inline-only settings with \fvinlineset 11
   4.2 Custom formatting for inline commands like \Verb with \FancyVerbFormatInline 11
   4.3 Custom formatting for environments like Verbatim with \FancyVerbFormatLine and \FancyVerbFormatText 11

5 Reimplemented commands 12
   5.1 \Verb 13
   5.2 \SaveVerb 14
   5.3 \UseVerb 14

6 New commands and environments 14
   6.1 \EscVerb 14
   6.2 VerbatimEnv 15
   6.3 VerbatimWrite 16
   6.4 VerbatimBuffer 16
   6.5 \VerbatimInsertBuffer 19
   6.6 \VerbatimClearBuffer 20

7 Line breaking 20
   7.1 Line breaking options 20
   7.2 Line breaking and tab expansion 26
   7.3 Advanced line breaking 27
      7.3.1 A few notes on algorithms 27
      7.3.2 Breaks within macro arguments 27
      7.3.3 Customizing break behavior 29

8 Pygments support 29
   8.1 Options for users 29
   8.2 For package authors 30

9 Patches 30
   9.1 Visible spaces 30
   9.2 obeytabs with visible tabs and with tabs inside macro arguments 30
   9.3 Math mode 31
      9.3.1 Spaces 31
      9.3.2 Symbols and fonts 32
   9.4 Orphaned labels 32
   9.5 rulecolor and fillcolor 32
   9.6 Command lookahead tokenization 32
10 Additional modifications to fancyvrb
10.1 Backtick and single quotation mark .......................... 33
10.2 Line numbering .................................................. 33

11 Undocumented features of fancyvrb .......................... 33
11.1 Undocumented options .......................................... 33
11.2 Undocumented macros .......................................... 33

12 Implementation .................................................... 34
12.1 Required packages .............................................. 34
12.2 Utility macros ................................................... 34
   12.2.1 fancyvrb space and tab tokens ......................... 34
   12.2.2 ASCII processing ........................................ 35
   12.2.3 Sentinels ................................................ 35
   12.2.4 Active character definitions ......................... 36
12.3 pdfTeX with inputenc using UTF-8 .......................... 37
12.4 Reading and processing command arguments ................. 39
   12.4.1 Tokenization and lookahead ........................... 39
   12.4.2 Reading arguments ...................................... 40
   12.4.3 Reading and protecting arguments in expansion-only contexts 44
   12.4.4 Converting detokenized tokens into PDF strings ...... 47
   12.4.5 Detokenizing verbatim arguments ..................... 48
   12.4.6 Retokenizing detokenized arguments ................. 65
12.5 Hooks .......................................................... 66
12.6 Escaped characters ............................................ 67
12.7 Inline-only options ........................................... 67
12.8 Reimplementations ............................................. 68
   12.8.1 extra option ............................................ 68
   12.8.2 \FancyVerbFormatInline ................................ 68
   12.8.3 \Verb .................................................. 68
   12.8.4 \SaveVerb ............................................. 70
   12.8.5 \UseVerb ............................................. 70
12.9 New commands and environments ............................. 72
   12.9.1 \EscVerb .............................................. 72
   12.9.2 \VerbEnv .............................................. 73
   12.9.3 VerbatimWrite ......................................... 75
   12.9.4 VerbatimBuffer ........................................ 76
   12.9.5 \VerbatimInsertBuffer ................................. 78
   12.9.6 \VerbatimClearBuffer ................................. 79
12.10 Patches ........................................................ 80
   12.10.1 Delimiting characters for verbatim commands ........ 80
   12.10.2 \CustomVerbatimCommand compatibility with \FVExtraRobustCommand ................................... 80
   12.10.3 Visible spaces .......................................... 81
   12.10.4 \obeytabs with visible tabs and with tabs inside macro arguments .................................. 81
   12.10.5 Spacing in math mode .................................. 85
   12.10.6 Fonts and symbols in math mode .................... 85
   12.10.7 Orphaned label ......................................... 86
   12.10.8 \rulecolor and \fillcolor ............................ 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.11</td>
<td>Extensions</td>
<td>87</td>
</tr>
<tr>
<td>12.11.1</td>
<td>New options requiring minimal implementation</td>
<td>87</td>
</tr>
<tr>
<td>12.11.2</td>
<td>Formatting with \FancyVerbFormatLine, \FancyVerbFormatText, and \FancyVerbHighlightLine</td>
<td>90</td>
</tr>
<tr>
<td>12.11.3</td>
<td>Line numbering</td>
<td>91</td>
</tr>
<tr>
<td>12.11.4</td>
<td>Line highlighting or emphasis</td>
<td>95</td>
</tr>
<tr>
<td>12.12</td>
<td>Line breaking</td>
<td>97</td>
</tr>
<tr>
<td>12.12.1</td>
<td>Options and associated macros</td>
<td>97</td>
</tr>
<tr>
<td>12.12.2</td>
<td>Line breaking implementation</td>
<td>105</td>
</tr>
<tr>
<td>12.13</td>
<td>Pygments compatibility</td>
<td>123</td>
</tr>
</tbody>
</table>
1 Introduction

The fancyvrb package had its first public release in January 1998. In July of the same year, a few additional features were added. Since then, the package has remained almost unchanged except for a few bug fixes. fancyvrb has become one of the primary \LaTeX{} packages for working with verbatim text.

Additional verbatim features would be nice, but since fancyvrb has remained almost unchanged for so long, a major upgrade could be problematic. There are likely many existing documents that tweak or patch fancyvrb internals in a way that relies on the existing implementation. At the same time, creating a completely new verbatim package would require a major time investment and duplicate much of fancyvrb that remains perfectly functional. Perhaps someday there will be an amazing new verbatim package. Until then, we have fvextra.

fvextra is an add-on package that gives fancyvrb several additional features, including automatic line breaking. Because fvextra patches and overwrites some of the fancyvrb internals, it may not be suitable for documents that rely on the details of the original fancyvrb implementation. fvextra tries to maintain the default fancyvrb behavior in most cases. All reimplementations (section 5), patches (section 9), and modifications to fancyvrb defaults (section 10) are documented. In most cases, there are options to switch back to original implementations or original default behavior.

Some features of fvextra were originally created as part of the pythontex and minted packages. fancyvrb-related patches and extensions that currently exist in those packages will gradually be migrated into fvextra.

2 Usage

fvextra may be used as a drop-in replacement for fancyvrb. It will load fancyvrb if it has not yet been loaded, and then proceeds to patch fancyvrb and define additional features.

The upquote package is loaded to give correct backticks (`) and typewriter single quotation marks ('). When this is not desirable within a given environment, use the option curlyquotes. fvextra modifies the behavior of these and other symbols in typeset math within verbatim, so that they will behave as expected (section 9.3). fvextra uses the lineno package for working with automatic line breaks. lineno gives a warning when the csquotes package is loaded before it, so fvextra should be loaded before csquotes. The etoolbox package is required. color or xcolor should be loaded manually to use color-dependent features.

While fvextra attempts to minimize changes to the fancyvrb internals, in some cases it completely overwrites fancyvrb macros with new definitions. New definitions typically follow the original definitions as much as possible, but code that depends on the details of the original fancyvrb implementation may be incompatible with fvextra.

2.1 Pandoc compatibility

fvextra supports line breaking in Pandoc \LaTeX{} output that includes highlighted source code. Enabling basic line breaking at spaces is as simple as adding
\usepackage{fvextra} and \fvset{breaklines} to the Pandoc Markdown header-includes.

By default, more advanced line breaking features such as breakanywhere, breakbefore, and breakafter will not work with Pandoc highlighted output, due to the presence of the syntax highlighting macros. This can be fixed by using breaknonspaceingroup, which enables all line breaking features within macros. For example, the following YAML metadata in a Markdown document would redefine the Pandoc Highlighting environment to enable line breaking anywhere.

```yaml
---
header-includes:
- \`
- `{=latex}
\usepackage{fvextra}
\DefineVerbatimEnvironment{Highlighting}{Verbatim}{
  commandchars=\\{\},
  breaklines, breaknonspaceingroup, breakanywhere}
...  
---
```

### 3 General options

fvextra adds several general options to fancyvrb. All options related to automatic line breaking are described separately in section 7. All options related to syntax highlighting using Pygments are described in section 8.

**beameroverlays** *(boolean)* *(default: false)*

Give the < and > characters their normal text meanings, so that beamer overlays of the form \only<1>{...} will work. Note that something like commandchars=\\{\} is required separately to enable macros. This is not incorporated in the beameroverlays option because essentially arbitrary command characters could be used; only the < and > characters are hard-coded for overlays.

With some font encodings and language settings, beameroverlays prevents literal (non-overlay) < and > characters from appearing correctly, so they must be inserted using commands.

**curlyquotes** *(boolean)* *(default: false)*

Unlike fancyvrb, fvextra requires the upquote package, so the backtick ('') and typewriter single quotation mark (’) always appear literally by default, instead of becoming the left and right curly single quotation marks (‘’). This option allows these characters to be replaced by the curly quotation marks when that is desirable.

```
\begin{Verbatim}
`quoted text`
\end{Verbatim}
`quoted text`
```
\begin{Verbatim}[curlyquotes] `quoted text' \end{Verbatim}

\texttt{extra} \hspace{1cm} (boolean) \hspace{1cm} (default: \texttt{true})

Use \texttt{fvextra} reimplementations of \texttt{fancyvrb} commands and environments when available. For example, use \texttt{fvextra}'s reimplemented \texttt{Verb} that works (with a few limitations) inside other commands, rather than the original \texttt{fancyvrb} implementation that essentially functions as \texttt{verbatim} inside other commands.

\texttt{fontencoding} \hspace{1cm} (string) \hspace{1cm} (default: \texttt{(document font encoding)})

Set the font encoding inside \texttt{fancyvrb} commands and environments. Setting \texttt{fontencoding=none} resets to the default document font encoding.

\texttt{highlightcolor} \hspace{1cm} (string) \hspace{1cm} (default: \texttt{LightCyan})

Set the color used for \texttt{highlightlines}, using a predefined color name from \texttt{color} or \texttt{xcolor}, or a color defined via \texttt{definecolor}.

\texttt{highlightlines} \hspace{1cm} (string) \hspace{1cm} (default: \texttt{none})

This highlights a single line or a range of lines based on line numbers. The line numbers refer to the line numbers that \texttt{fancyvrb} would show if \texttt{numbers=left}, etc. They do not refer to original or actual line numbers before adjustment by \texttt{firstnumber}.

The highlighting color can be customized with \texttt{highlightcolor}.

\begin{Verbatim}[numbers=left, highlightlines={1, 3-4}] First line Second line Third line Fourth line Fifth line \end{Verbatim}

\begin{verbatim}
  1  First line
  2  Second line
  3  Third line
  4  Fourth line
  5  Fifth line
\end{verbatim}

The actual highlighting is performed by a set of commands. These may be customized for additional fine-tuning of highlighting. See the default definition of \texttt{FancyVerbHighlightLineFirst} as a starting point.

- \texttt{FancyVerbHighlightLineFirst}: First line in a range.
- \texttt{FancyVerbHighlightLineMiddle}: Inner lines in a range.
- \texttt{FancyVerbHighlightLineLast}: Last line in a range.
- \texttt{FancyVerbHighlightLineSingle}: Single highlighted lines.
- \texttt{FancyVerbHighlightLineNormal}: Normal lines without highlighting.
If these are customized in such a way that indentation or inter-line spacing is changed, then \FancyVerbHighlightLineNormal may be modified as well to make all lines uniform. When working with the First, Last, and Single commands, keep in mind that \f{extra} merges all numbers ranges, so that \{1, 2-3, 3-5\} is treated the same as \{1-5\}.

Highlighting is applied after \FancyVerbFormatText, so any text formatting defined via that command will work with highlighting. Highlighting is applied before \FancyVerbFormatLine, so if \FancyVerbFormatLine puts a line in a box, the box will be behind whatever is created by highlighting. This prevents highlighting from vanishing due to user-defined customization.

\begin{Verbatim}[numbers=left, stepnumber=2, numberfirstline]
First line
Second line
Third line
Fourth line
\end{Verbatim}

1 First line
2 Second line
3 Third line
4 Fourth line

(numbers)\textbf{ \textit{linenos} (boolean) (default: false)}
\textbf{\textit{fancyvrb}} allows line numbers via the options \texttt{numbers=(position)}. This is essentially an alias for \texttt{numbers=left}. It primarily exists for better compatibility with the minted package.

\textbf{\textit{mathescape} (boolean) (default: false)}
This causes everything between dollar signs $...$ to be typeset as math. The ampersand &, caret ^, and underscore _ have their normal math meanings.

This is equivalent to
\begin{verbatim}
codes={\catcode`\$=3\catcode`\&=4\catcode`\^=7\catcode`\_=8}
\end{verbatim}
\texttt{mathescape} is always applied \textbf{before codes}, so that \texttt{codes} can be used to override some of these definitions.

Note that \texttt{f{extra}} provides several patches that make math mode within verbatim as close to normal math mode as possible (section 9.3).

\begin{Verbatim}
\begin{Verbatim}[numbers=left, stepnumber=2, numberfirstline]
First line
Second line
Third line
Fourth line
\end{Verbatim}
\end{Verbatim}

(numbers)\textbf{ \textit{numbers} (none | left | right | both) (default: none)}
\texttt{f{extra}} adds the \texttt{both} option for line numbering.
<table>
<thead>
<tr>
<th>\begin{Verbatim}[numbers=both]</th>
<th>\end{Verbatim}</th>
</tr>
</thead>
<tbody>
<tr>
<td>First line</td>
<td>1</td>
</tr>
<tr>
<td>Second line</td>
<td>2</td>
</tr>
<tr>
<td>Third line</td>
<td>3</td>
</tr>
<tr>
<td>Fourth line</td>
<td>4</td>
</tr>
</tbody>
</table>

**retokenize** (boolean)  (default: false)
By default, \UseVerb inserts saved verbatim material with the catcodes (commandchars, codes, etc.) under which it was originally saved with \SaveVerb. When retokenize is used, the saved verbatim material is retokenized under the settings in place at \UseVerb.

This only applies to the reimplemented \UseVerb, when paired with the reimplemented \SaveVerb. It may be extended to environments (\UseVerbatim, etc.) in the future, if the relevant commands and environments are reimplemented.

**space** (macro)  (default: _)
Redefine the visible space character. Note that this is only used if showspaces=true.

The color of the character may be set with spacecolor.

**spacebreak** (macro)  (default: \discretionary{}{}{})
This determines the break that is inserted around spaces when breaklines=true and one or more of the following conditions applies: breakcollapsespaces=false, showspaces=true, or the space is affected by breakbefore or breakafter. If it is redefined, it should typically be similar to \FancyVerbBreakAnywhereBreak, \FancyVerbBreakBeforeBreak, and \FancyVerbBreakAfterBreak to obtain consistent breaks.

**spacecolor** (string)  (default: none)
Set the color of visible spaces. By default (none), they take the color of their surroundings.

```latex
\color{gray}
\begin{Verbatim}[showspaces, spacecolor=red]
One two three
\end{Verbatim}
```

**stepnumberfromfirst** (boolean)  (default: false)
By default, when line numbering is used with stepnumber \neq 1, only line numbers that are a multiple of stepnumber are included. This offsets the line numbering from the first line, so that the first line, and all lines separated from it by a multiple of stepnumber, are numbered.
First line
Second line
Third line
Fourth line

\begin{Verbatim}[numbers=left, stepnumber=2, stepnumberfromfirst]
First line
Second line
Third line
Fourth line
\end{Verbatim}

\begin{Verbatim}[numbers=left, stepnumber=2, firstnumber=4, stepnumberoffsetvalues]
First line
Second line
Third line
Fourth line
\end{Verbatim}

\begin{Verbatim}[numbers=left, stepnumber=2, stepnumberfromfirst, firstnumber=4, stepnumberoffsetvalues]
First line
Second line
Third line
Fourth line
\end{Verbatim}

stepnumberoffsetvalues \hspace{1cm} (boolean) \hspace{1cm} (default: false)

By default, when line numbering is used with \texttt{stepnumber} \neq 1, only line numbers that are a multiple of \texttt{stepnumber} are included. Using \texttt{firstnumber} to offset the numbering will change which lines are numbered and which line gets which number, but will not change which \textit{numbers} appear. This option causes \texttt{firstnumber} to be ignored in determining which line numbers are a multiple of \texttt{stepnumber}. \texttt{firstnumber} is still used in calculating the actual numbers that appear. As a result, the line numbers that appear will be a multiple of \texttt{stepnumber}, plus \texttt{firstnumber} minus 1.

This option gives the original behavior of \texttt{fancyverb} when \texttt{firstnumber} is used with \texttt{stepnumber} \neq 1 (section 10.2).

tab \hspace{1cm} (macro) \hspace{1cm} (default: \texttt{fancyverb}'s \texttt{\FancyVerbTab}, \texttt{\textendash})

Redefine the visible tab character. Note that this is only used if \texttt{showtabs=true}. The color of the character may be set with \texttt{tabcolor}.

When redefining the tab, you should include the font family, font shape, and text color in the definition. Otherwise these may be inherited from the surrounding text. This is particularly important when using the tab with syntax highlighting, such as with the \texttt{minted} or \texttt{pythontex} packages.
fvextra patches fancyverb tab expansion so that variable-width symbols such as \rightarrowfill may be used as tabs. For example,

\begin{Verbatim}[obeytabs, showtabs, breaklines, tab=\rightarrowfill, tabcolor=orange] −⟩|First −⟩|Second −⟩|Third −⟩|And more text that goes on for a while until wrapping is needed −⟩|First −⟩|Second −⟩|Third −⟩|Forth \end{Verbatim}

−−−−−−→First − − →Second − →Third − − →And more text that goes on for a while until wrapping is needed −−−−−−→First − − →Second − →Third − − →Forth

tabcolor (string) (default: none)
Set the color of visible tabs. By default (none), they take the color of their surroundings.

4 General commands

4.1 Inline-only settings with \fvinlineset

\fvinlineset{⟨options⟩}
This is like \fvset, except that options only apply to commands that typeset inline verbatim, like \Verb and \EscVerb. Settings from \fvinlineset override those from \fvset.

Note that \fvinlineset only works with commands that are reimplemented, patched, or defined by fvextra; it is not compatible with the original fancyverb definitions.

4.2 Custom formatting for inline commands like \Verb with \FancyVerbFormatInline

\FancyVerbFormatInline
This can be used to apply custom formatting to inline verbatim text created with commands like \Verb. It only works with commands that are reimplemented, patched, or defined by fvextra; it is not compatible with the original fancyverb definitions. The default definition does nothing; it is equivalent to \newcommand{\FancyVerbFormatInline}[1]{#1}.

This is the inline equivalent of \FancyVerbFormatLine and \FancyVerbFormatText. In the inline context, there is no need to distinguish between entire line formatting and only text formatting, so only \FancyVerbFormatInline exists.

4.3 Custom formatting for environments like Verbatim with \FancyVerbFormatLine and \FancyVerbFormatText
fancyvrb defines \texttt{\textbackslash FancyVerbFormatLine}, which can be used to apply custom formatting to each individual line of text in environments like \texttt{Verbatim}. By default, it takes a line as an argument and inserts it with no modification. This is equivalent to \texttt{\newcommand\{\texttt{\textbackslash FancyVerbFormatLine}\}[1]\{#1\}.\footnote{The actual definition in fancyvrb is \texttt{\def\texttt{\textbackslash FancyVerbFormatLine}\{}\texttt{\textbackslash FV@ObeyTabs\{}\texttt{\textbackslash #1\}}. This is problematic because redefining the macro could easily eliminate \texttt{\textbackslash FV@ObeyTabs}, which governs tab expansion. \texttt{fvextra} redefines the macro to \texttt{\def\texttt{\textbackslash FancyVerbFormatLine}\{}\texttt{\textbackslash #1\}} and patches all parts of fancyvrb that use \texttt{\textbackslash FancyVerbFormatLine} so that \texttt{\textbackslash FV@ObeyTabs} is explicitly inserted at the appropriate points.}

\texttt{fvextra} introduces line breaking, which complicates line formatting. We might want to apply formatting to the entire line, including line breaks, line continuation symbols, and all indentation, including any extra indentation provided by line breaking. Or we might want to apply formatting only to the actual text of the line. \texttt{fvextra} leaves \texttt{\textbackslash FancyVerbFormatLine} as applying to the entire line, and introduces a new command \texttt{\textbackslash FancyVerbFormatText} that only applies to the text part of the line.\footnote{When \texttt{\textbackslash breaklines=true}, each line is wrapped in a \texttt{\textbackslash parbox}. \texttt{\textbackslash FancyVerbFormatLine} is outside the \texttt{\textbackslash parbox}, and \texttt{\textbackslash FancyVerbFormatText} is inside.} By default, \texttt{\textbackslash FancyVerbFormatText} inserts the text unmodified. When it is customized, it should not use boxes that do not allow line breaks to avoid conflicts with line breaking code.

\begin{verbatim}
\renewcommand{\texttt{\textbackslash FancyVerbFormatLine}}[1]{\fcolorbox{DarkBlue}{LightGray}{#1}}
\renewcommand{\texttt{\textbackslash FancyVerbFormatText}}[1]{\textcolor{Green}{#1}}
\begin{Verbatim}[\texttt{\textbackslash breaklines}]
Some text that proceeds for a while and finally wraps onto another line
Some more text
\end{Verbatim}
\end{verbatim}

5 Reimplemented commands

\texttt{fvextra} reimplements parts of fancyvrb. These new implementations stay close to the original definitions while allowing for new features that otherwise would not be possible. Reimplemented versions are used by default. The original implementations may be used via \texttt{\textbackslash fvset\{\textbackslash extra=\textbackslash false\}} or by using \texttt{\textbackslash extra=\textbackslash false} in the optional arguments to a command or environment.

Reimplemented commands restrict the scope of catcode-related options compared to the original fancyvrb versions. This prevents catcode-related options from interfering with new features such as \texttt{\textbackslash FancyVerbFormatInline}. With \texttt{fvextra}, the \texttt{\textbackslash codes} option should only be used for catcode modifications. Including non-catcode commands in \texttt{\textbackslash codes} will typically have no effect, unlike with fancyvrb. If you want
to customize verbatim content using general commands, consider \texttt{formatcom}.

\section{\texttt{\Verb}}
\texttt{\Verb}[(\verb\options)](\verb\text\verb\text\{\verb\delimiter\text\verb\text\})

The new \texttt{\Verb} works as expected (with a few limitations) inside other
commands. It even works in movable arguments (for example, in \texttt{\section}), and is
compatible with \texttt{hyperref} for generating PDF strings (for example, PDF bookmarks).
The \texttt{fancyverb} definition did work inside some other commands, but essentially func-
tioned as \texttt{\texttt{\texttt{\verb}}} in that context.

\texttt{\Verb} is compatible with \texttt{breaklines} and the relevant line-breaking options.
Like the original \texttt{fancyverb} implementation, the new \texttt{\Verb} can be starred
(\texttt{\Verb*}) and accepts optional arguments. While \texttt{fancyverb}'s starred command
\texttt{\Verb*} is a shortcut for \texttt{showspaces}, \texttt{fvextra}'s \texttt{\Verb*} is a shortcut for both
\texttt{showspaces} and \texttt{showtabs}. This is more similar to the current behavior of \LaTeX's
\texttt{\verb*}, except that \texttt{\verb*} converts tabs into visible spaces instead of displaying
them as visible tabs.

\textbf{Delimiters} A repeated character like normal \texttt{\verb}, or a pair of curly braces
\{\ldots\}. If curly braces are used, then \{\texttt{text}\} cannot contain unpaired curly
braces. Note that curly braces should be preferred when using \texttt{\Verb} inside
other commands, and curly braces are \textit{required} when \texttt{\Verb} is in a movable
argument, such as in a \texttt{\section}. Non-ASCII characters now work as delimiters
under pdfTeX with \texttt{inputenc} using UTF-8. For example, \texttt{\verb§verb§} now works as expected.

\textbf{Limitations inside other commands} While the new \texttt{\Verb} does work inside
arbitrary other commands, there are a few limitations.

\begin{itemize}
\item # and \% cannot be used. If you need them, consider \texttt{\EscVerb} or perhaps
\texttt{\SaveVerb\Verb}.
\item Curly braces are only allowed in pairs.
\item Multiple adjacent spaces will be collapsed into a single space.
\item Be careful with backslashes. A backslash that is followed by one or more
ASCII letters will cause a following space to be lost, if the space is not
immediately followed by an ASCII letter. For example, \texttt{\Verb\r\n} becomes
\r\n, but \texttt{\Verb\r\n} becomes \r\n. Basically, anything
that looks like a \LaTeX command (control word) will gobble following
spaces, unless the next character after the spaces is an ASCII letter.
\item A single ~ is fine, but avoid "" because it will serve as an escape sequence
for an ASCII command character.
\end{itemize}

\textbf{Using in movable arguments} \texttt{\Verb} works automatically in movable argu-
ments, such as in a \texttt{\section}. \texttt{\protect} or similar measures are not needed
for \texttt{\Verb} itself, or for any of its arguments, and should not be used. \texttt{\Verb}
performs operations that amount to applying \texttt{\protect} to all of these automatic-
ally.

\footnote{Under pdfTeX, non-ASCII code points are processed at the byte rather than code point level,
so \Verb must treat a sequence of multiple bytes as the delimiter.}

13
hyperref PDF strings  \Verb is compatible with hyperref for generating PDF strings such as PDF bookmarks. Note that the PDF strings are always a literal rendering of the verbatim text, with all fancyvrb options ignored. For example, things like showspaces and commandchars have no effect. If you need options to be applied to obtain desired PDF strings, consider a custom approach, perhaps using \texorpdfstring.

Line breaking breaklines allows breaks at spaces. breakbefore, breakafter, and breakanywhere function as expected, as do things like breakaftersymbolpre and breakaftersymbolpost. Break options that are only applicable to block text like a Verbatim environment do not have any effect. For example, breakindent and breaksymbol do nothing.

5.2 \SaveVerb
\SaveVerb*[\langle options\rangle]{\langle name\rangle}{\langle delim char or \{} ⟨text⟩ ⟨delim char or \}⟩}
\SaveVerb is reimplemented so that it is equivalent to the reimplemented \Verb. Like the new \Verb, it accepts ⟨text⟩ delimited by a pair of curly braces {...}. It supports \fvinlineset. It also adds support for the new retokenize option for \UseVerb.

5.3 \UseVerb
\UseVerb*[\langle options\rangle]{\langle name\rangle}
\UseVerb is reimplemented so that it is equivalent to the reimplemented \Verb. It supports \fvinlineset and breaklines.
Like \Verb, \UseVerb is compatible with hyperref for generating PDF strings such as PDF bookmarks. Note that the PDF strings are always a literal rendering of the verbatim text, with all fancyvrb options ignored. For example, things like showspaces and commandchars have no effect. The new option retokenize also has no effect. If you need options to be applied to obtain desired PDF strings, consider a custom approach, perhaps using \texorpdfstring.

There is a new option retokenize for \UseVerb. By default, \UseVerb inserts saved verbatim material with the catcodes (commandchars, codes, etc.) under which it was originally saved with \SaveVerb. When retokenize is used, the saved verbatim material is retokenized under the settings in place at \UseVerb.

For example, consider \SaveVerb{save}{\textcolor{red}{#\%}}:
- \UseVerb{save} ⇒ \textcolor{red}{#\%}
- \UseVerb[commandchars=\\\{}]{save} ⇒ \textcolor{red}{#\%}
- \UseVerb[retokenize, commandchars=\\\{}]{save} ⇒ #\%

6 New commands and environments

6.1 \EscVerb
\EscVerb*[\langle options\rangle]{\langle backslash-escaped text\rangle}
This is like \Verb but with backslash escapes to allow for characters such as # and %. For example, \EscVerb{\\Verb{\#\%}} gives \Verb{#\%}. It behaves
exactly the same regardless of whether it is used inside another command. Like the reimplemented \Verb, it works in movable arguments (for example, in \section), and is compatible with hyperref for generating PDF strings (for example, PDF bookmarks).

**Delimiter** Text must *always* be delimited with a pair of curly braces {...}. This ensures that \EscVerb is always used in the same manner regardless of whether it is inside another command.

**Escaping rules**

- Only printable, non-alphanumeric ASCII characters (symbols, punctuation) can be escaped with backslashes.\(^4\)
- Always escape these characters: \, %, #.
- Escape spaces when there are more than one in a row.
- Escape ~ if there are more than one in a row.
- Escape unpaired curly braces.
- Additional symbols or punctuation characters may require escaping if they are made \active, depending on their definitions.

**Using in moveable arguments** \EscVerb works automatically in moveable arguments, such as in a \section. \protect or similar measures are not needed for \EscVerb itself, or for any of its arguments, and should not be used. \EscVerb performs operations that amount to applying \protect to all of these automatically.

**hyperref PDF strings** \EscVerb is compatible with hyperref for generating PDF strings such as PDF bookmarks. Note that the PDF strings are *always* a literal rendering of the verbatim text after backslash escapes have been applied, with all fancyvrb options ignored. For example, things like showspaces and commandchars have no effect. If you need options to be applied to obtain desired PDF strings, consider a custom approach, perhaps using texorpdfstring.

### 6.2 VerbEnv

\begin{VerbEnv}[⟨options⟩]
⟨single line⟩
\end{VerbEnv}

This is an environment variant of \Verb. The environment must contain only a single line of text, and the closing \end{VerbEnv} must be on a line by itself. The ⟨options⟩ and ⟨single line⟩ are read and then passed on to \Verb internally for actual typesetting.

While VerbEnv can be used by document authors, it is primarily intended for package creators. For example, it is used in minted to implement \mintinline. In that case, highlighted code is always generated within a Verbatim environment. It is possible to process this as inline rather than block verbatim by \letting \Verbatim to \VerbEnv.

\(^4\)Allowing backslash escapes of letters would lead to ambiguity regarding spaces; see \Verb.
VerbEnv is not implemented using the typical fancyvrb environment implementation style, so it is not compatible with \RecustomVerbatimEnvironment.

6.3 VerbatimWrite

\begin{VerbatimWrite}[[opt]]
\langle\lines\rangle
\end{VerbatimWrite}

This writes environment contents verbatim to an external file. It is similar to fancyvrb’s VerbatimOut, except that (1) it allows writing to a file multiple times (multiple environments can write to the same file) and (2) by default it uses \detokenize to guarantee truly verbatim output.

By default, all fancyvrb options except for VerbatimWrite-specific options are ignored. This can be customized on a per-environment basis via environment optional arguments.

Options defined specifically for VerbatimWrite:

\begin{tabular}{ll}
\textbf{writefilehandle} & (file handle) \hspace{1cm} (default: \texttt{(none)}) \\
& File handle for writing. For example,
\end{tabular}

\begin{verbatim}
\newwrite\myfile
\immediate\openout\myfile=myfile.txt\relax

\begin{VerbatimWrite}[writefilehandle=\myfile]
...\end{VerbatimWrite}

\immediate\closeout\myfile
\end{verbatim}

\begin{tabular}{ll}
\textbf{writer} & (macro) \hspace{1cm} (default: \texttt{\FancyVerbDefaultWriter}) \\
& This is the macro that processes each line of text in the environment and then writes it to file. This is the default implementation:
\end{tabular}

\begin{verbatim}
\def\FancyVerbDefaultWriter#1{%
  \immediate\write\FancyVerbWriteFileHandle{\detokenize{#1}}%
}
\end{verbatim}

6.4 VerbatimBuffer

\begin{VerbatimBuffer}[[opt]]
\langle\lines\rangle
\end{VerbatimBuffer}

This environment stores its contents verbatim in a “buffer,” a sequence of numbered macros each of which contains one line of the environment. The “buffered” lines can then be looped over for further processing or later use. This is similar to fancyvrb’s SaveVerbatim, which saves an environment for later use. VerbatimBuffer offers additional flexibility by capturing truly verbatim environment contents using \detokenize and saving environment contents in a format designed for further processing.
By default, all fancyverb options except for VerbatimBuffer-specific options are ignored. This can be customized on a per-environment basis via environment optional arguments.

Below is an extended example that demonstrates what is possible with VerbatimBuffer combined with \VerbatimInsertBuffer. This uses \ifdefstring from the etoolbox package.

- \setformatter defines an empty \formatter macro. Then it loops over the lines in a buffer looking for a line containing only the text “red”. If this is found, it redefines \formatter to \color{red}. FancyVerbBufferIndex is a counter that is always available for buffer looping. FancyVerbBufferLength is the default counter containing the buffer length (number of lines). \FancyVerbBufferLineName contains the base name for buffer line macros (default FancyVerbBufferLine).

- afterbuffer involves two steps: (1) \setformatter loops through the buffer and defines \formatter based on the buffer contents, and (2) \VerbatimInsertBuffer typesets the buffer, using formatcom=\formatter to format the text based on whether any line contains only the text “red”.

```latex
\def\setformatter{
  \def\formatter{}
  \setcounter{FancyVerbBufferIndex}{1}
  \loop\unless\ifnum\value{FancyVerbBufferIndex}>\value{FancyVerbBufferLength}\relax
    \expandafter\let\expandafter\bufferline\csname FancyVerbBufferLineName\arabic{FancyVerbBufferIndex}\endcsname
    \ifdefstring{\bufferline}{red}{\def\formatter{\color{red}}}{}
  \stepcounter{FancyVerbBufferIndex}
  \repeat

  \begin{VerbatimBuffer}[
    afterbuffer={\setformatter\VerbatimInsertBuffer[formatcom=\formatter]}
  ]
    first
    second
    red
  \end{VerbatimBuffer}
```

Here is the same example, but rewritten to use a global buffer with custom buffer names instead.

```
first
second
red
```
\begin{VerbatimBuffer}[globalbuffer, bufferlinename=exbuff, bufferlengthname=exbuff]
first
second
red
\end{VerbatimBuffer}

\def\formatter{}
\setcounter{FancyVerbBufferIndex}{1}
\loop\unless\ifnum\value{FancyVerbBufferIndex}>\value{exbuff}\relax
  \expandafter\let\expandafter\bufferline\csname exbuff\arabic{FancyVerbBufferIndex}\endcsname
  \ifdefstring{\bufferline}{red}{\def\formatter{\color{red}}}{}
  \stepcounter{FancyVerbBufferIndex}
\repeat
\VerbatimInsertBuffer[
  formatcom=\formatter,
  bufferlinename=exbuff,
  bufferlengthname=exbuff
]

Options defined specifically for \texttt{VerbatimBuffer}:

\texttt{afterbuffer} \hspace{1em} (macro) \hspace{1em} \texttt{(default: (none))}
Macro or macros invoked at the end of the environment, after all lines of the environment have been buffered. This is outside the \texttt{\begin{group}\ldots\end{group}} that wraps verbatim processing, so \texttt{fancyvrb} settings are no longer active. However, the buffer line macros and the buffer length counter are still accessible.

\texttt{bufferer} \hspace{1em} (macro) \hspace{1em} \texttt{(default: \texttt{\FancyVerbDefaultBufferer})}
This is the macro that adds lines to the buffer. The default is designed to create a truly verbatim buffer via \texttt{\detokenize}. This can be customized if you wish to use \texttt{fancyvrb} options related to catcodes to create a buffer that is only partially verbatim (that contains macros).

\texttt{\def\FancyVerbDefaultBufferer#1{\%}
  \expandafter\xdef\csname\texttt{FancyVerbBufferLineName}\arabic{FancyVerbBufferIndex}\endcsname{\%}
  \texttt{\detokenize[#1]}}\%}

A custom \texttt{bufferer} must take a single argument \#1 (a line of the environment text) and ultimately store the processed line in a macro called

\texttt{\csname\texttt{FancyVerbBufferLineName}\arabic{FancyVerbBufferIndex}\endcsname}

This macro must be defined globally, so \texttt{\xdef} or \texttt{\gdef} is necessary (this does not interfere with scoping from \texttt{globalbuffer}). Otherwise, there are no restrictions. The \texttt{\xdef} and \texttt{\detokenize} in the default definition guarantee that the buffer

18
consists only of the literal text from the environment, but this is not required for a custom bufferer.

`bufferlengthname` (string) (default: `FancyVerbBufferLength`) Name of the counter (`\newcounter`) storing the length of the buffer. This is the number of lines stored.

`bufferlinename` (string) (default: `FancyVerbBufferLine`) The base name of the buffer line macros. The default is `FancyVerbBufferLine`, which will result in buffer macros `\FancyVerbBufferLine<n>` with integer `n` greater than or equal to one and less than or equal to the number of lines (one-based indexing). Since buffer macro names contain a number, they must be accessed differently than typical macros:

```
\csname FancyVerbBufferLine<n>\endcsname
\@nameuse{FancyVerbBufferLine<n>}
```

Typically the buffer macros will be looped over with a counter that is incremented, in which case `<n>` should be the counter value `\arabic{<counter>}`.

`buffername` (string) (default: `⟨none⟩`) Shortcut for setting `bufferlengthname` and `bufferlinename` simultaneously, using the same root name. This sets `bufferlengthname` to `<buffername>length` and `bufferlinename` to `<buffername>line`.

`globalbuffer` (bool) (default: `false`) This determines whether buffer line macros are defined globally, that is, whether they are accessible after the end of the `VerbatimBuffer` environment. If the line macros are defined globally, then the buffer length counter is also increased appropriately outside the environment. `globalbuffer` does not affect any afterbuffer macro, since that is invoked inside the environment.

When buffered lines are used immediately, consider using `afterbuffer` instead of `globalbuffer`. When buffered lines must be used later in a document, consider using `globalbuffer` with custom (and perhaps unique) `bufferlinename` and `bufferlengthname`.

When `globalbuffer=true`, `VerbatimBuffer` environments with the same buffer name will append to a single buffer, so that it ultimately contains the concatenated contents of all environments. A `VerbatimBuffer` environment with `globalbuffer=false` will append to the buffer created by any previous `VerbatimBuffer` that had `globalbuffer=true` and shared the same buffer name. Any `afterbuffer` macro will have access to a buffer containing the concatenated data. At the very end of the environment with `globalbuffer=false`, after any `afterbuffer`, this appended content will be removed. All buffer line macros (from `bufferlinename`) that were created by that environment are “deleted” (`\let` to an undefined macro), and the buffer length counter (from `bufferlengthname`) is reduced proportionally.

### \VerbatimInsertBuffer

\VerbatimInsertBuffer[(options)]

This inserts an existing buffer created by `VerbatimBuffer` as a `Verbatim` environment. It customizes `Verbatim` internals to function with a buffer in a
command context. See the VerbatimBuffer documentation for an example of usage.

Options related to catcodes cause the buffer to be retokenized during typesetting. That is, the fancyvrb options used for \VerbatimInsertBuffer are not restricted by those that were in effect when VerbatimBuffer originally created the buffer, so long as the buffer contains a complete representation of the original VerbatimBuffer environment contents.

\VerbatimInsertBuffer is not implemented using the typical fancyvrb command and environment implementation styles, so it is not compatible with \RecustomVerbatimCommand or \RecustomVerbatimEnvironment.

6.6 \VerbatimClearBuffer
\VerbatimClearBuffer\{\options\}

Clear an existing buffer created with VerbatimBuffer. \global\let all buffer line macros to an undefined macro and set the buffer length counter to zero.

7 Line breaking

Automatic line breaking may be turned on with breaklines=true. By default, breaks only occur at spaces. Breaks may be allowed anywhere with breakanywhere, or only before or after specified characters with breakbefore and breakafter. Many options are provided for customizing breaks. A good place to start is the description of breaklines.

When a line is broken, the result must fit on a single page. There is no support for breaking a line across multiple pages.

7.1 Line breaking options

Options are provided for customizing typical line breaking features. See section 7.3 for details about low-level customization of break behavior.

breakafter (string) (default: \{none\})

Break lines after specified characters, not just at spaces, when breaklines=true. For example, breakafter=-/ would allow breaks after any hyphens or slashes. Special characters given to breakafter should be backslash-escaped (usually #, {, }, %, [, ]), and the comma ,; the backslash \ may be obtained via \space).\footnote{breakafter expands each token it is given once, so when it is given a macro like \%, the macro should expand to a literal character that will appear in the text to be typeset. fwextra defines special character escapes that are activated for breakafter so that this will work with common escapes. The only exception to token expansion is non-ASCII characters under pdfTeX; these should appear literally. breakafter is not catcode-sensitive.}

For an alternative, see breakbefore. When breakbefore and breakafter are used for the same character, breakbeforeinrun and breakafterinrun must both have the same setting.

Note that when commandchars or codes are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work unless the entire macro including arguments is wrapped in a group (curly
braces { }, or other characters specified with `commandchars`. See section 7.3 for details, and consider `breaknonspaceingroup` as a solution in simple cases.

```verbatim
\begin{Verbatim}[breaklines, breakafter=d]
 some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}

some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCould\_NeverFitOnOneLine'
```

`breakafterrun` (boolean) (default: `false`)
When `breakafter` is used, insert breaks within runs of identical characters. If `false`, treat sequences of identical characters as a unit that cannot contain breaks. When `breakbefore` and `breakafter` are used for the same character, `breakbeforeinrun` and `breakafterinrun` must both have the same setting.

`breakersymbolpre` (string) (default: \textbackslash \footnotesize \textbackslash ensuremath{\rfloor}, \)}
The symbol inserted pre-break for breaks inserted by `breakafter`. This does not apply to breaks inserted next to spaces; see `spacebreak`.

`breakersymbolpost` (string) (default: `{none}`)
The symbol inserted post-break for breaks inserted by `breakafter`. This does not apply to breaks inserted next to spaces; see `spacebreak`.

`breakanywhere` (boolean) (default: `false`)
Break lines anywhere, not just at spaces, when `breaklines=true`.

Note that when `commandchars` or `codes` are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work unless the entire macro including arguments is wrapped in a group (curly braces { }, or other characters specified with `commandchars`). See section 7.3 for details, and consider `breaknonspaceingroup` as a solution in simple cases.

```verbatim
\begin{Verbatim}[breaklines, breakanywhere]
 some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}

some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
```

`breakanywheresymbolpre` (string) (default: \textbackslash \footnotesize \textbackslash ensuremath{\rfloor}, \)}
The symbol inserted pre-break for breaks inserted by `breakanywhere`. This does not apply to breaks inserted next to spaces; see `spacebreak`.

`breakanywheresymbolpost` (string) (default: `{none}`)
The symbol inserted post-break for breaks inserted by `breakanywhere`. This does
not apply to breaks inserted next to spaces; see \texttt{spacebreak}.

\texttt{breakautoindent} \hspace{1em} (boolean) \hspace{1em} (default: \texttt{true})

When a line is broken, automatically indent the continuation lines to the indentation level of the first line. When \texttt{breakautoindent} and \texttt{breakindent} are used together, the indentations add. This indentation is combined with \texttt{breaksymbolindentleft} to give the total actual left indentation.

\texttt{breakbefore} \hspace{1em} (string) \hspace{1em} (default: \texttt{(none)})

Break lines before specified characters, not just at spaces, when \texttt{breaklines=true}. For example, \texttt{breakbefore=A} would allow breaks before capital A’s. Special characters given to \texttt{breakbefore} should be backslash-escaped (usually \#, \{, \}, \%, [], and the comma, ; the backslash \ may be obtained via \ and the space via \space).\footnote{\texttt{breakbefore} expands each token it is given once, so when it is given a macro like \%, the macro should expand to a literal character that will appear in the text to be typeset. \texttt{fvextra} defines special character escapes that are activated for \texttt{breakbefore} so that this will work with common escapes. The only exception to token expansion is non-ASCII characters under pdfTeX; these should appear literally. \texttt{breakbefore} is not catcode-sensitive.}

For an alternative, see \texttt{breakafter}. When \texttt{breakbefore} and \texttt{breakafter} are used for the same character, \texttt{breakbeforeinrun} and \texttt{breakafterinrun} must both have the same setting.

Note that when \texttt{commandchars} or \texttt{codes} are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work unless the entire macro including arguments is wrapped in a group (curly braces \{\}, or other characters specified with \texttt{commandchars}). See section 7.3 for details, and consider \texttt{breaknonspaceingroup} as a solution in simple cases.

\begin{verbatim}
\begin{Verbatim}[breaklines, breakbefore=A]
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}
\end{verbatim}

\begin{verbatim}
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{verbatim}

\texttt{breakbeforeinrun} \hspace{1em} (boolean) \hspace{1em} (default: \texttt{false})

When \texttt{breakbefore} is used, insert breaks within runs of identical characters. If \texttt{false}, treat sequences of identical characters as a unit that cannot contain breaks. When \texttt{breakbefore} and \texttt{breakafter} are used for the same character, \texttt{breakbeforeinrun} and \texttt{breakafterinrun} must both have the same setting.

\texttt{breakbeforesymbolpre} \hspace{1em} (string) \hspace{1em} (default: \texttt{\footnotesize\textbackslash\textbackslash\textbackslash\texttt{footnotesize\ensuremath{\lfloor}}, \})

The symbol inserted pre-break for breaks inserted by \texttt{breakbefore}. This does not apply to breaks inserted next to spaces; see \texttt{spacebreak}.

\texttt{breakbeforesymbolpost} \hspace{1em} (string) \hspace{1em} (default: \texttt{(none)})

The symbol inserted post-break for breaks inserted by \texttt{breakbefore}. This does not apply to breaks inserted next to spaces; see \texttt{spacebreak}.
breakcollapsespaces (bool) (default: true)
When true (default), a line break within a run of regular spaces (showspaces=false) replaces all spaces with a single break, and the wrapped line after the break starts with a non-space character. When false, a line break within a run of regular spaces preserves all spaces, and the wrapped line after the break may start with one or more spaces. This causes regular spaces to behave exactly like the visible spaces produced with showspaces; both give identical line breaks, with the only difference being the appearance of spaces.

breakindent (dimension) (default: \langle breakindentnchars \rangle)
When a line is broken, indent the continuation lines by this amount. When breakautoindent and breakindent are used together, the indentations add. This indentation is combined with breaksymbolindentleft to give the total actual left indentation.

breakindentnchars (integer) (default: 0)
This allows breakindent to be specified as an integer number of characters rather than as a dimension (assumes a fixed-width font).

breaklines (boolean) (default: false)
Automatically break long lines. When a line is broken, the result must fit on a single page. There is no support for breaking a line across multiple pages.\footnote{Following the implementation in fancyvrb, each line is typeset within an \hbox, so page breaks are not possible.}

By default, automatic breaks occur at spaces (even when showspaces=true). Use breakanywhere to enable breaking anywhere; use breakbefore and breakafter for more fine-tuned breaking.

To customize the indentation of broken lines, see breakindent and breakautoindent.
To customize the line continuation symbols, use breaksymbolleft and breaksymbolright.
To customize the separation between the continuation symbols and the text, use breaksymbolsepleft and breaksymbolsepright. To customize the extra indentation that is supplied to make room for the break symbols, use breaksymbolindentleft and breaksymbolindentright. Since only the left-hand symbol is used by default, it may also be modified using the alias options breaksymbol, breaksymbolsep, and breaksymbolindent.

An example using these options to customize the Verbatim environment is shown below. This uses the \carriagereturn symbol from the dingbat package.

\begin{Verbatim}[breaklines]
def f(x):
    return 'Some text ' + str(x)
\end{Verbatim}

\begin{Verbatim}[breaklines]
def f(x):
    return 'Some text ' +
        str(x)
\end{Verbatim}

\begin{Verbatim}[breaklines]
def f(x):
    return 'Some text ' +
        str(x)
\end{Verbatim}
```python
def f(x):
    return 'Some text ' + str(x) + ' some more text ' +
           ' even more text that goes on for a while'
```

Beginning in version 1.6, automatic line breaks work with `showspaces=true` by default. Defining `breakbefore` or `breakafter` for `\space` is no longer necessary. For example,

```latex
some_string = 'Some Text That Goes On And On For So Long That It Could Never Fit'
```

`breaknonspacegroup` *(boolean)* *(default: false)*

By using `commandchars`, it is possible to include \LaTeX{} commands within otherwise verbatim text. In these cases, there can be groups (typically \{...\}) but depends on `commandchars` within verbatim. Spaces within groups are treated as potential line break locations when `breaklines=true`, but by default no other break locations are inserted (`breakbefore`, `breakafter`, `breakanywhere`). This is because inserting non-space break locations can interfere with command functionality. For example, in `\textcolor{red}{text}`, breaks shouldn’t be inserted within `red`.

`breaknonspaceingrooup` allows non-space breaks to be inserted within groups. This option should only be used when `commandchars` is including \LaTeX{} commands that do not take optional arguments and only take mandatory arguments that are typeset. Something like `\textit{text}` is fine, but `\textcolor{red}{text}` is not because one of the mandatory arguments is not typeset but rather provides a setting. For more complex commands, it is typically better to redefine them to insert breaks in appropriate locations using `\FancyVerbBreakStart\ldots\FancyVerbBreakStop`. 

24
breaksymbol (string) (default: \tiny\ensuremath{\hookrightarrow}, ↪ →)
Alias for breaksymbolleft.

breaksymbolleft (string) (default: \tiny\ensuremath{\hookrightarrow}, ¬)
The symbol used at the beginning (left) of continuation lines when breaklines=true.
To have no symbol, simply set breaksymbolleft to an empty string (“=,” or “=\{”).
The symbol is wrapped within curly braces {} when used, so there is no danger of formatting commands such as \tiny “escaping.”

The \hookrightarrow and \hookleftarrow may be further customized by the use of the \rotatebox command provided by graphicx. Additional arrow-type symbols that may be useful are available in the dingbat (\carriagereturn) and mnsymbol (hook and curve arrows) packages, among others.

breaksymbolright (string) (default: (none))
The symbol used at breaks (right) when breaklines=true. Does not appear at the end of the very last segment of a broken line.

breaksymbolindent (dimension) (default: \langle breaksymbolindentleftnchars \rangle)
Alias for breaksymbolindentleft.

breaksymbolindentleftnchars (integer) (default: \langle breaksymbolindentleftnchars \rangle)
Alias for breaksymbolindentleftnchars.

breaksymbolindentleft (dimension) (default: \langle breaksymbolindentleftnchars \rangle)
The extra left indentation that is provided to make room for breaksymbolleft.
This indentation is only applied when there is a breaksymbolleft.

breaksymbolindentright (dimension) (default: \langle breaksymbolindentrightnchars \rangle)
The extra right indentation that is provided to make room for breaksymbolright.
This indentation is only applied when there is a breaksymbolright.

breaksymbolindentrightnchars (integer) (default: 4)
This allows breaksymbolindentright to be specified as an integer number of characters rather than as a dimension (assumes a fixed-width font).

breaksymbolsep (dimension) (default: \langle breaksymbolsepleftnchars \rangle)
Alias for breaksymbolsepleft.

breaksymbolsepnchars (integer) (default: \langle breaksymbolsepleftnchars \rangle)
Alias for breaksymbolsepleftnchars.

breaksymbolsepleft (dimension) (default: \langle breaksymbolsepleftnchars \rangle)
The separation between the breaksymbolleft and the adjacent text.

breaksymbolsepleftnchars (integer) (default: 2)
Allows breaksymbolsepleft to be specified as an integer number of characters rather than as a dimension (assumes a fixed-width font).

breaksymbolsepright (dimension) (default: \langle breaksymbolseprightnchars \rangle)
The minimum separation between the breaksymbolright and the adjacent text.
This is the separation between breaksymbolright and the furthest extent to which
adjacent text could reach. In practice, \texttt{\linewidth} will typically not be an exact integer multiple of the character width (assuming a fixed-width font), so the actual separation between the \texttt{breaksymbolright} and adjacent text will generally be larger than \texttt{breaksymbolsepright}. This ensures that break symbols have the same spacing from the margins on both left and right. If the same spacing from text is desired instead, \texttt{breaksymbolsepright} may be adjusted. (See the definition of \texttt{\FV@makeLineNumber} for implementation details.)

\begin{itemize}
  \item \texttt{breaksymbolseprightnchars} (integer) (default: 2)
    \begin{itemize}
      \item Allows \texttt{breaksymbolsepright} to be specified as an integer number of characters rather than as a dimension (assumes a fixed-width font).
    \end{itemize}
  \item \texttt{spacebreak} (macro) (default: \texttt{\discretionary{}{}{}})
    \begin{itemize}
      \item This determines the break that is inserted around spaces when \texttt{breaklines=true} and one or more of the following conditions applies: \texttt{breakcollapsespaces=false}, \texttt{showspaces=true}, or the space is affected by \texttt{breakbefore} or \texttt{breakafter}. If it is redefined, it should typically be similar to \texttt{\FancyVerbBreakAnywhereBreak}, \texttt{\FancyVerbBreakBeforeBreak}, and \texttt{\FancyVerbBreakAfterBreak} to obtain consistent breaks.
    \end{itemize}
\end{itemize}

### 7.2 Line breaking and tab expansion

\texttt{fancyvrb} provides an \texttt{obeytabs} option that expands tabs based on tab stops rather than replacing them with a fixed number of spaces (see \texttt{fancyvrb}'s \texttt{tabsize}). The \texttt{fancyvrb} implementation of tab expansion is not directly compatible with \texttt{fvextra}'s line-breaking algorithm, but \texttt{fvextra} builds on the \texttt{fancyvrb} approach to obtain identical results.

Tab expansion in the context of line breaking does bring some additional considerations that should be kept in mind. In each line, all tabs are expanded exactly as they would have been had the line not been broken. This means that after a line break, any tabs will not align with tab stops unless the total left indentation of continuation lines is a multiple of the tab stop width. The total indentation of continuation lines is the sum of \texttt{breakindent}, \texttt{breakautoindent}, and \texttt{breaksymbolindentleft} (alias \texttt{breaksymbolindent}).

A sample \texttt{Verbatim} environment that uses \texttt{obeytabs} with \texttt{breaklines} is shown below, with numbers beneath the environment indicating tab stops (\texttt{tabsize=8} by default). The tab stops in the wrapped and unwrapped lines are identical. However, the continuation line does not match up with the tab stops because by default the width of \texttt{breaksymbolindentleft} is equal to four monospace characters. (By default, \texttt{breakautoindent=true}, so the continuation line gets a tab plus \texttt{breaksymbolindentleft}.)

\begin{verbatim}
\begin{Verbatim}[obeytabs, showtabs, breaklines]
  ➕First ➕Second ➕Third ➕And more text that goes on for a
  ➕ while until wrapping is needed
  ➕First ➕Second ➕Third ➕Forth
\end{Verbatim}
\end{verbatim}

123456781234567812345678123456781234567812345678123456781234567812345678
We can set the symbol indentation to eight characters by creating a dimen,

\newdimen\temporarydimen
%

setting its width to eight characters,

\settowidth{\temporarydimen}{\ttfamily AaAaAaAa}
%

and finally adding the option \texttt{breaksymbolindentleft=\temporarydimen} to the \texttt{Verbatim} environment to obtain the following:

\begin{verbatim}
  First \& Second \& Third \& And more text that goes on for a 
  \hline
  First \& Second \& Third \& Forth
\end{verbatim}

\section{Advanced line breaking}

\subsection{A few notes on algorithms}

\texttt{breakanywhere}, \texttt{breakbefore}, and \texttt{breakafter} work by scanning through the tokens in each line and inserting line breaking commands wherever a break should be allowed. By default, they skip over all groups (\{\ldots\}) and all math (\$\ldots\$). Note that this refers to curly braces and dollar signs with their normal \LaTeX{} meaning (catcodes), not verbatim curly braces and dollar signs; such non-verbatim content may be enabled with \texttt{commandchars} or \texttt{codes}. This means that math and macros that only take mandatory arguments (\{\ldots\}) will function normally within otherwise verbatim text. However, macros that take optional arguments may not work because \{\ldots\} is not treated specially, and thus break commands may be inserted within [\ldots] depending on settings. Wrapping an entire macro, including its arguments, in a group will protect the optional argument: {\texttt{\{\{macro\}}[\{\{oarg\}]}\{\{marg\}}].

\texttt{breakbefore} and \texttt{breakafter} insert line breaking commands around specified characters. This process is catcode-independent; tokens are \texttt{\detokenized} before they are checked against characters specified via \texttt{breakbefore} and \texttt{breakafter}.

\subsection{Breaks within macro arguments}

When \texttt{commandchars} or \texttt{codes} are used to include macros within verbatim content, the options \texttt{breakanywhere}, \texttt{breakbefore}, and \texttt{breakafter} will not generate breaks within mandatory macro arguments. Macros with optional arguments may not work, depending on settings, unless they are wrapped in a group (curly braces \{}), or other characters specified via \texttt{commandchars}.

If you want to allow breaks within macro arguments (optional or mandatory), then you should (re)define your macros so that the relevant arguments are wrapped in the commands
For example, suppose you have the macro

\begin{verbatim}
\newcommand{\mycmd}[1]{\_before:#1:after\_}
\end{verbatim}

Then you would discover that line breaking does not occur:

\begin{verbatim}
\mycmd{1}\mycmd{2}\mycmd{3}\mycmd{4}\mycmd{5}
\end{verbatim}

Instead of completely redefining macros, it may be more convenient to use `\let`. For example,

\begin{verbatim}
\let\originalmycmd\mycmd
\renewcommand{\mycmd}[1]{\expandafter\originalmycmd{#1}\FancyVerbBreakStop}
\end{verbatim}

Notice that in this case `\expandafter` is required, because `\FancyVerbBreakStart` does not permit any expansion and thus will skip over `\originalmycmd{#1}` unless it is already expanded. The `etoolbox` package provides commands that may be useful for patching macros to insert line breaks.

When working with `\FancyVerbBreakStart` and `\FancyVerbBreakStop`, keep in mind that any groups `{...}` or math `$...$` between the two commands will be skipped as far as line breaks are concerned, and breaks may be inserted within any optional arguments `[...]` depending on settings. Inserting breaks within groups requires another level of `\FancyVerbBreakStart` and `\FancyVerbBreakStop`, and protecting optional arguments requires wrapping the entire macro in a group `{...}`. Also, keep in mind that `\FancyVerbBreakStart` cannot introduce line breaks in a context in which they are never allowed, such as in an `\hbox`. 28
7.3.3 Customizing break behavior

These macros govern the behavior of breaks introduced by breakanywhere, breakbefore, and breakafter. These do not apply to breaks inserted next to spaces; see spacebreak.

By default, these macros use \discretionary. \discretionary takes three arguments: commands to insert before the break, commands to insert after the break, and commands to insert if there is no break. For example, the default definition of \FancyVerbBreakAnywhereBreak:

```latex
\newcommand{\FancyVerbBreakAnywhereBreak}{\% \discretionary{\FancyVerbBreakAnywhereSymbolPre}{\FancyVerbBreakAnywhereSymbolPost}{}\%}
```

The other macros are equivalent, except that “Anywhere” is swapped for “Before” or “After”.

\discretionary will generally only insert breaks when breaking at spaces simply cannot make lines short enough (this may be tweaked to some extent with hyphenation settings). This can produce a somewhat ragged appearance in some cases. If you want breaks exactly at the margin (or as close as possible) regardless of whether a break at a space is an option, you may want to use \allowbreak instead. Another option is \linebreak[⟨n⟩], where ⟨n⟩ is between 0 to 4, with 0 allowing a break and 4 forcing a break.

8 Pygments support

8.1 Options for users

fvextra defines additional options for working code that has been highlighted with Pygments. These options work with the minted and pythontex packages, and may be enabled for other packages that work with Pygments output (section 8.2).

**breakbytoken** (boolean) (default: false)
When breaklines=true, do not allow breaks within Pygments tokens. This would prevent, for example, line breaking within strings.

**breakbytokenanywhere** (boolean) (default: false)
When breaklines=true, do not allow breaks within Pygments tokens, but always allow breaks between tokens even when they are immediately adjacent (not separated by spaces). This option should be used with care. Due to the details of how each Pygments lexer works, and due to the tokens defined in each lexer, this may result in breaks in locations that might not be anticipated. Also keep in mind that this will not allow breaks between tokens if those tokens are actually “subtokens” within another token.

\FancyVerbBreakByTokenAnywhereBreak

This defines the break inserted when breakbytokenanywhere=true. By default, it is \allowbreak.
8.2 For package authors

By default, line breaking will only partially work with Pygments output; `breakbefore` and `breakafter` will not work with any characters that do not appear literally in Pygments output but rather are replaced with a character macro. Also, `breakbytoken` and `breakbytokenanywhere` will not function at all.

To enable full Pygments support, use this macro before \begin{Verbatim}, etc. This macro must be used within \begingroup...\endgroup to prevent settings from escaping into the rest of the document. It may be used safely at the beginning of a \newenvironment definition. When used with \newcommand, though, the \begingroup...\endgroup will need to be inserted explicitly.

\begin{VerbatimPygments}{⟨literal_macro⟩}{⟨actual_macro⟩}

\end{VerbatimPygments}

\VerbatimPygments takes the two Pygments macros and redefines ⟨literal_macro⟩ so that it will invoke ⟨actual_macro⟩ while fully supporting line breaks, \breakbytoken, and \breakbytokenanywhere. No further modification of either ⟨literal_macro⟩ or ⟨actual_macro⟩ is possible after \VerbatimPygments is used.

In packages that do not make a distinction between ⟨literal_macro⟩ and ⟨actual_macro⟩, simply use \VerbatimPygments with two identical arguments; \VerbatimPygments is defined to handle this case.

9 Patches

\texttt{fvextra} modifies some \texttt{fancyvrb} behavior that is the result of bugs or omissions.

9.1 Visible spaces

The command \texttt{\FancyVerbSpace} defines the visible space when \texttt{showspaces=true}. The default \texttt{fancyvrb} definition allows a font command to escape under some circumstances, so that all following text is forced to be teletype font. The command is redefined following \url{https://tex.stackexchange.com/a/120231/10742}.

9.2 obeytabs with visible tabs and with tabs inside macro arguments

The original \texttt{fancyvrb} treatment of visible tabs when \texttt{showtabs=true} and \texttt{obeytabs=true} did not allow variable-width tab symbols such as \texttt{\rightarrowfill} to function correctly. This is fixed through a redefinition of \texttt{\FV@TrueTab}.

Various macros associated with \texttt{obeytabs=true} are also redefined so that tabs may be expanded regardless of whether they are within a group (within \{...\} with the normal \LaTeX{} meaning due to \texttt{commandchars}, etc.). In the \texttt{fancyvrb} implementation, using \texttt{obeytabs=true} when a tab is inside a group typically causes the entire line to vanish. \texttt{fvextra} patches this so that the tab is expanded and will be visible if \texttt{showtabs=true}. Note, though, that the tab expansion in
these cases is only guaranteed to be correct for leading whitespace that is inside a group. The start of each run of whitespace that is inside a group is treated as a tab stop, whether or not it actually is, due to limitations of the tab expansion algorithm. A more detailed discussion is provided in the implementation.

The example below shows correct tab expansion of leading whitespace within a macro argument. With fancyvrb, the line of text would simply vanish in this case.

\begin{Verbatim}[obeytabs, showtabs, showspaces, tabsize=4, commandchars=\\\{\}, tab=\textcolor{orange}{\rightarrowfill}]
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
Text after 1 space + 2 tabs
\end{Verbatim}

The next example shows that tab expansion inside macros in the midst of text typically does not match up with the correct tab stops, since in such circumstances the beginning of the run of whitespace must be treated as a tab stop.

\begin{Verbatim}[obeytabs, showtabs, commandchars=\\\{\}, tab=\textcolor{orange}{\rightarrowfill}]
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
\textcolor{blue}{−⟩}
2 leading tabs
\textcolor{blue}{Text −⟩ −⟩ −⟩ then 2 tabs}
\end{Verbatim}

9.3 Math mode

9.3.1 Spaces

When typeset math is included within verbatim material, fancyvrb makes spaces within the math appear literally.

\begin{Verbatim}[commandchars=\\\{\}, mathescape]
Verbatim $\displaystyle\frac{1}{ x^2 + y^2 }$ verbatim
\end{Verbatim}

fvextra patches this by redefining fancyvrb’s space character within math mode so that it behaves as expected:

Verbatim $\frac{1}{ x^2 + y^2 }$ verbatim
9.3.2 Symbols and fonts

With \texttt{fancyvrb}, using a single quotation mark (’) in typeset math within verbatim material results in an error rather than a prime symbol (’).\footnote{The single quotation mark is made active within verbatim material to prevent ligatures, via \texttt{@noligs}. The default definition is incompatible with math mode.} \texttt{fvextra} redefines the behavior of the single quotation mark within math mode to fix this, so that it will become a proper prime.

The \texttt{amsmath} package provides a \texttt{text} command for including normal text within math. With \texttt{fancyvrb}, \texttt{text} does not behave normally when used in typeset math within verbatim material. \texttt{fvextra} redefines the backtick (‘) and the single quotation mark so that they function normally within \texttt{text}, becoming left and right quotation marks. It redefines the greater-than sign, less-than sign, comma, and hyphen so that they function normally as well. \texttt{fvextra} also switches back to the default document font within \texttt{text}, rather than using the verbatim font, which is typically a monospace or typewriter font.

The result of these modifications is a math mode that very closely mimics the behavior of normal math mode outside of verbatim material.

\begin{Verbatim}
\verb|Verbatim $\displaystyle f'''(x) = \text{``Some quoted text---''}$|
\end{Verbatim}

Verbatim $f'''(x) = \text{"Some quoted text—"}$

9.4 Orphaned labels

When \texttt{frame=lines} is used with a \texttt{label}, \texttt{fancyvrb} does not prevent the label from being orphaned under some circumstances. \texttt{\FV@BeginListFrame@Lines} is patched to prevent this.

9.5 \texttt{rulecolor} and \texttt{fillcolor}

The \texttt{rulecolor} and \texttt{fillcolor} options are redefined so that they accept color names directly, rather than requiring \texttt{\color{⟨color\_name⟩}}. The definitions still allow the old usage.

9.6 Command lookahead tokenization

\texttt{\FV@Command} is used internally by commands like \texttt{\Verb} to read stars (\texttt{*}) and optional arguments (\texttt{[...]}). This is redefined so that lookahead tokenizes under a verbatim catcode regime. The original definition could prevent commands like \texttt{\Verb} from using characters like \texttt{\%} as delimiters, because the lookahead for a star and optional argument could read the \texttt{\%} and give it its normal meaning of comment character. The new definition fixes this, so that commands like \texttt{\Verb} behave as closely to \texttt{\verb} as possible.
10 Additional modifications to fancyvrb

fvextra modifies some fancyvrb behavior with the intention of improving logical consistency or providing better defaults.

10.1 Backtick and single quotation mark

With fancyvrb, the backtick ` and typewriter single quotation mark ’ are typeset as the left and right curly single quotation marks ‘ ‘. fvextra loads the upquote package so that these characters will appear literally by default. The original fancyvrb behavior can be restored with the fvextra option curlyquotes (section 3).

10.2 Line numbering

With fancyvrb, using firstnumber to offset line numbering in conjunction with stepnumber changes which line numbers appear. Lines are numbered if their original line numbers, without the firstnumber offset, are a multiple of stepnumber. But the actual numbers that appear are the offset values that include firstnumber. Thus, using firstnumber=2 with stepnumber=5 would cause the original lines 5, 10, 15, ... to be numbered, but with the values 6, 11, 16, ....

fvextra changes line numbering so that when stepnumber is used, the actual line numbers that appear are always multiples of stepnumber by default, regardless of any firstnumber offset. The original fancyvrb behavior may be turned on by setting stepnumberoffsetvalues=true (section 3).

11 Undocumented features of fancyvrb

fancyvrb defines some potentially useful but undocumented features.

11.1 Undocumented options

codes* (macro) (default: \{empty\})

fancyvrb’s codes is used to specify catcode changes. It overwrites any existing codes. codes* appends changes to existing settings.

defineactive* (macro) (default: \{empty\})

fancyvrb’s defineactive is used to define the effect of active characters. It overwrites any existing defineactive. defineactive* appends changes to existing settings.

formatcom* (macro) (default: \{empty\})

fancyvrb’s formatcom is used to execute commands before verbatim text. It overwrites any existing formatcom. formatcom* appends changes to existing settings.

11.2 Undocumented macros

\FancyVerbTab

This defines the visible tab character (−⟩|) that is used when showtabs=true. The default definition is
While this may be redefined directly, fvextra also defines a new option `tab`.

This defines the visible space character (\textordf淥) that is used when `showspaces=true`. The default definition (as patched by fvextra, section 9.1) follows https://tex.stackexchange.com/a/120231/10742. While this may be redefined directly, fvextra also defines a new option `space`.

### 12 Implementation

#### 12.1 Required packages

The `upquote` package performs some font checks when it is loaded to determine whether `textcomp` is needed, but errors can result if the font is changed later in the preamble, so duplicate the package’s font check at the end of the preamble. Also check for a package order issue with `lineno` and `csquotes`.

```latex
\AtEndPreamble{%
\ifx\encodingdefault\upquote@OTone
\ifx\ttdefault\upquote@cmtt\else\RequirePackage{textcomp}\fi
\else
\RequirePackage{textcomp}
\fi
\if\pdftex
\fi}
\RequirePackage{lineno}
\@ifpackageloaded{csquotes}{}
\PackageWarning{fvextra}{csquotes should be loaded after fvextra, %
to avoid a warning from the lineno package}{}
```

#### 12.2 Utility macros

##### 12.2.1 `fancyvrb` space and tab tokens

`\FV@ActiveSpaceToken` Active space for `\ifx` token comparisons.

```latex
\begingroup
\catcode`\ =\active%
\edef\FV@ActiveSpaceToken{ }%
\endgroup%
```

`\FV@SpaceCatTen` Space with catcode 10. Used instead of `\ ` and `\space` in some contexts to avoid issues in the event that these are redefined.

```latex
\edef\FV@SpaceCatTen{\detokenize{ }}
```
\FV@FVSpaceToken \ Macro with the same definition as fancyvrb’s active space. Useful for \ifx comparisons, such as \@ifnextchar lookaheads.
\def\FV@FVSpaceToken{\FV@Space}

\FV@FVTabToken \ Macro with the same definition as fancyvrb’s active tab. Useful for \ifx comparisons, such as \@ifnextchar lookaheads.
\def\FV@FVTabToken{\FV@Tab}

12.2.2 ASCII processing
\FVExtraDoSpecials \ Apply \do to all printable, non-alphanumeric ASCII characters (codepoints 0x20 through 0x7E except for alphanumeric characters).
These punctuation marks and symbols are the most likely characters to be made \active, so it is convenient to be able to change the catcodes for all of them, not just for those in the \dospecials defined in latex.ltx:
\%
some \texttt{fvextra} applications, the tokens will have been incorrectly tokenized under
a normal catcode regime, and need to be retokenized as verbatim, in which case
undefined macros must be expected. Thus, a sentinel macro whose expansion is
resistant to collisions is needed.

\texttt{\FV@<Sentinel> } This is the standard default \texttt{fvextra} delimited-macro sentinel. It is used with
\texttt{\makeatletter} by changing \texttt{<} and \texttt{>} to catcode 11. The \texttt{<} and \texttt{>} add an extra
level of collision resistance. Because it is undefined, it is \textit{only} appropriate for use
in delimited macro arguments.

\texttt{\FV@Sentinel} This is the standard \texttt{fvextra} \texttt{\ifx} comparison sentinel. It expands to the control
word \texttt{\FV@<Sentinel>}, which is very unlikely to be in any other macro since it
requires that \texttt{\@}, \texttt{<}, and \texttt{>} all have catcode 11 and appear in the correct sequence.
Because its definition is itself undefined, this sentinel will result in an error if it
escapes.

\begin{verbatim}
\begingroup
\catcode`\<=11
\catcode`\>=11
\gdef\FV@Sentinel{\FV@<Sentinel>}
\endgroup
\end{verbatim}

12.2.4 Active character definitions

\texttt{\FV@OuterDefEOLEmpty} Macro for defining the active end-of-line character \texttt{^^M} (\texttt{\r}), which \texttt{fancyverb}
uses to prevent runaway command arguments. \texttt{fancyverb} uses macro definitions of the form

\begin{verbatim}
\begingroup
%\catcode`\^^M=\active%
%\gdef\macro{%
% \outer\def\^^M{}
% ...
%}%
%\endgroup
%
\end{verbatim}

While this works, it is nice to avoid the \texttt{\begingroup\ldots\endgroup} and especially
the requirement that all lines now end with \% to discard the \texttt{^^M} that would
otherwise be inserted.

\begin{verbatim}
\begingroup
\catcode`\^^M=\active%
\gdef\FV@OuterDefEOLEmpty{\outer\def\^^M{}}
\endgroup
\end{verbatim}

\texttt{\FV@DefEOLEmpty} The same thing, without the \texttt{\outer}. This is used to ensure that \texttt{\^^M} is not \texttt{\outer}
when it should be read.

\begin{verbatim}
\begingroup
\catcode`\^^M=\active%
\gdef\FV@DefEOLEmpty{\def\^^M{}}
\endgroup
\end{verbatim}
Define start-of-text (STX) `^B so that it cannot be used inside other macros. This makes it possible to guarantee that `^B is not part of a verbatim argument, so that it can be used later as a sentinel in retokenizing the argument.

```latex
\begingroup
\catcode`\^B=\active
\gdef\FV@OuterDefSTXEmpty{\outer\def\^B{}}
\endgroup
```

Define end-of-text (ETX) `^C so that it cannot be used inside other macros. This makes it possible to guarantee that `^C is not part of a verbatim argument, so that it can be used later as a sentinel in retokenizing the argument.

```latex
\begingroup
\catcode`\^C=\active
\gdef\FV@OuterDefETXEmpty{\outer\def\^C{}}
\endgroup
```

12.3 pdfTeX with inputenc using UTF-8

Working with verbatim text often involves handling individual code points. While these are treated as single entities under LuaTeX and XeTeX, under pdfTeX code points must be handled at the byte level instead. This means that reading a single code point encoded in UTF-8 may involve a macro that reads up to four arguments.

Macros are defined for working with non-ASCII code points under pdfTeX. These are only for use with the inputenc package set to utf8 encoding.

```latex
\ifFV@pdfTeXinputenc
\else
\fi
```

All of the UTF macros are only needed with pdfTeX when inputenc is loaded, so they are created conditionally, inspired by the approach of the iftex package. The tests deal with the possibility that a previous test using `\iffx` rather than the cleaner `\ifcsname` has already been performed. These assume that inputenc will be loaded before \fvextra. The `\inputencodingname` tests should be redundant after the `\@ifpackageloaded` test, but do provide some additional safety if another package is faking inputenc being loaded but not providing an equivalent encoding interface.

Note that an encoding test of the form

```latex
/%\ifdefstring{\inputencodingname}{utf8}{<true>}{<false>}
```

is still required before switching to the UTF variants in any given situation. A document using inputenc can switch encodings (for example, around an `\input`), so simply checking encoding when \fvextra is loaded is not sufficient.
Define UTF macros conditionally:

\ifFV@pdfTeXinputenc

Define macros of the form \FV@U8:<byte> for each active byte. These are used for determining whether a token is the first byte in a multi-byte sequence, and if so, invoking the necessary macro to capture the remaining bytes. The code is adapted from the beginning of utf8.def. Completely capitalized macro names are used to avoid having to worry about \uppercase.

\begingroup
\catcode`\~=13
\catcode`\"=12
\def\FV@UTFviii@loop{\uccode`\~\count@\uppercase\expandafter{\FV@UTFviii@Tmp}\advance\count@\@ne\ifnum\count@<\@tempcnta\expandafter\FV@UTFviii@loop\fi}

Setting up 2-byte UTF-8:
\count@`C2\@tempcnta`E0\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{\FV@UTF@two@octets}}\FV@UTFviii@loop

Setting up 3-byte UTF-8:
\count@`E0\@tempcnta`F0\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{\FV@UTF@three@octets}}\FV@UTFviii@loop

Setting up 4-byte UTF-8:
\count@`F0\@tempcnta`F4\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{\FV@UTF@four@octets}}\FV@UTFviii@loop
\endgroup

These are variants of the utf8.def macros that capture all bytes of a multi-byte code point and then pass them on to \FV@UTF@octets@after as a single argument for further processing. The invoking macro should \let or \def'ed \FV@UTF@octets@after to an appropriate macro that performs further processing.

Typical use will involve the following steps:

1. Read a token, say #1.
2. Use \ifcsname FV@U8:\detokenize{#1}\endcsname to determine that the token is the first byte of a multi-byte code point.
3. Ensure that \FV@UTF@octets@after has an appropriate value, if this has not already been done.

4. Use \csname FV@U8:\detokenize{#1}\endcsname#1 at the end of the original reading macro to read the full multi-byte code point and then pass it on as a single argument to \FV@UTF@octets@after.

All code points are checked for validity here so as to raise errors as early as possible. Otherwise an invalid terminal byte sequence might gobble a sentinel macro in a scanning context, potentially making debugging much more difficult. It would be possible to use \UTFviii@defined{(bytes)} to trigger an error directly, but the current approach is to attempt to typeset invalid code points, which should trigger errors without relying on the details of the utf8.def implementation.

\begin{verbatim}
\def\FV@UTF@two@octets#1#2{% 
  \ifcsname u8:\detokenize{#1#2}\endcsname 
  \else 
    \if#1\% 
      \else 
    \fi 
    \FV@UTF@octets@after{#1#2}}
\def\FV@UTF@three@octets#1#2#3{% 
  \ifcsname u8:\detokenize{#1#2#3}\endcsname 
  \else 
    \if#1\% 
      \else 
    \fi 
    \FV@UTF@octets@after{#1#2#3}}
\def\FV@UTF@four@octets#1#2#3#4{% 
  \ifcsname u8:\detokenize{#1#2#3#4}\endcsname 
  \else 
    \if#1\% 
      \else 
    \fi 
    \FV@UTF@octets@after{#1#2#3#4}}
\end{verbatim}

End conditional creation of UTF macros:

110 \fi

12.4 Reading and processing command arguments

\texttt{fvextra} provides macros for reading and processing verbatim arguments. These are primarily intended for creating commands that take verbatim arguments but can still be used within other commands (with some limitations). These macros are used in reimplementing \texttt{fancyverb} commands like \texttt{\Verb}. They may also be used in other packages; \texttt{minted} and \texttt{pythontex} use them for handling inline code.

All macros meant for internal use have names of the form \texttt{\FV@<Name>}, while all macros meant for use in other packages have names of the form \texttt{\FVExtra<Name>}. Only the latter are intended to have a stable interface.

12.4.1 Tokenization and lookahead

\texttt{\FVExtra@ifnextcharAny} A version of \texttt{\@ifnextchar} that can detect any character, including catcode 10 spaces. This is an exact copy of the definition from \texttt{latex.ltx}, modified with the “\let\reserved@d= #1\%” (note space!) trick from amsgen.

\begin{verbatim}
\long\def\FVExtra@ifnextcharAny#1#2#3{% 
\end{verbatim}

39
This is a wrapper for `@ifnextchar` from `latex.ltx` (ltdefns.dtx) that tokenizes lookaheads under a mostly verbatim catcode regime rather than the current catcode regime. This is important when looking ahead for stars * and optional argument delimiters [], because if these are not present when looking ahead for a verbatim argument, then the first thing tokenized will be the verbatim argument’s delimiting character. Ideally, the delimiter should be tokenized under a verbatim catcode regime. This is necessary for instance if the delimiter is `\active` and `\outer`.

The catcode of the space is preserved (in the unlikely event it is `\active`) and curly braces are given their normal catcodes for the lookahead. This simplifies space handling in an untokenized context, and allows paired curly braces to be used as verbatim delimiters.

```
\long\def\FVExtra@ifnextcharVArg#1#2#3{%
  \begingroup
    \edef\FV@TmpSpaceCat{\the\catcode`}\relax
    \catcode`\{=1
    \catcode`\}=2
    \@ifnextchar#1{\endgroup#2}{\endgroup#3}}
```

A starred command behaves differently depending on whether it is followed by an optional star or asterisk *. `@ifstar` from `latex.ltx` is typically used to check for the *. In the process, it discards following spaces (catcode 10) and tokenizes the next non-space character under the current catcode regime. While this is fine for normal commands, it is undesirable if the next character turns out to be not a * but rather a verbatim argument’s delimiter. This reimplementation prevents such issues for all printable ASCII symbols via `\FVExtra@ifnextcharVArg`.

```
\begingroup
\catcode`*=12
\gdef\FVExtra@ifstarVArg#1{\FVExtra@ifnextcharVArg*{\@firstoftwo{#1}}}
\endgroup
```

### 12.4.2 Reading arguments

`\FV@ReadOArgContinue` Read a macro followed by an optional argument, then pass the optional argument to the macro for processing and to continue.

```
\def\FV@ReadOArgContinue#1[#2]{#1{#2}}
```

`\FVExtraReadOArgBeforeVArg` Read an optional argument that comes before a verbatim argument. The lookahead for the optional argument tokenizes with a verbatim catcode regime in case it encounters the delimiter for the verbatim argument rather than [. If the lookahead
doesn’t find [], the optional argument for \FVExtraReadOArgBeforeVArg can be used to supply a default optional argument other than (empty).

\FVExtraReadOArgBeforeVEnv Read an optional argument at the start of a verbatim environment, after the \begin{⟨environment⟩} but before the start of the next line where the verbatim content begins. Check for extraneous content after the optional argument and discard the following newline. Note that this is not needed when an environment takes a mandatory argument that follows the optional argument.

The case with only an optional argument is tricky because the default behavior of \@ifnextchar is to read into the next line looking for the optional argument. Setting \^^M as \active prevents this. That does mean, though, that the end-of-line token will have to be read and removed later as an \active \^^M.

\@ifnextchar is used instead of \FVExtra@ifnextcharVArg because the latter is not needed since there is an explicit, required delimiter (\^^M) before the actual start of verbatim content. Lookahead can never tokenize verbatim content under an incorrect catcode regime.

\FVExtraReadVArg Read a verbatim argument that is bounded by two identical characters or by paired curly braces. This uses the \outer \^^M with \FV@EOL trick from fancyvrb to prevent runaway arguments. An \outer \^^C is used to prevent \^^C from being part of arguments, so that it can be used later as a sentinel if retokenization is needed. \^^B is handled in the same manner for symmetry with later usage, though technically it is not used as a sentinel so this is not strictly necessary. Alternate UTF macros, defined later, are invoked when under pdfTeX with inputenc using UTF-8.
The lookahead for the type of delimiting character is done under a verbatim catcode regime, except that the space catcode is preserved and curly braces are given their normal catcodes. This provides consistency with any \FVExtra@ifnextcharVArg or \FVExtra@ifstarVArg that may have been used previously, allows characters like # and % to be used as delimiters when the verbatim argument is read outside any other commands (untokenized), and allows paired curly braces to serve as delimiters. Any additional command-specific catcode modifications should only be applied to the argument after it has been read, since they do not apply to the delimiters.

Once the delimiter lookahead is complete, catcodes revert to full verbatim, and are then modified appropriately given the type of delimiter. The space and tab must be \active to be preserved correctly when the verbatim argument is not inside any other commands (otherwise, they collapse into single spaces).

\FVExtraReadVArg The argument is read under the verbatim catcode regime already in place from \FVExtraReadVArg. The \endgroup returns to prior catcodes. Any command-specific catcodes can be applied later via \scantokens. Using them here in reading the argument would have no effect as far as later processing with \scantokens is concerned, unless the argument were read outside any other commands and additional characters were given catcodes 1 or 2 (like the curly braces). That scenario is not allowed because it makes reading the argument overly dependent on the argument content. (Technically, reading the argument is already dependent
on the argument content in the sense that the argument cannot contain unescaped unpaired curly braces, given that it is delimited by curly braces.)

193 \def\FV@ReadVArg@Group#1#2#3{%
194 \endgroup
195 #1(#3)}

\FV@ReadVArg@Char The delimiting character is read under the verbatim catcode regime in place from \FVExtraReadVArg. If the command is not inside a normal command, then this means the delimiting character will typically have catcode 12 and that characters like # and \% can be used as delimiters; otherwise, the delimiter may have any catcode that is possible for a single character captured by a macro. If the argument is read inside another command (already tokenized), then it is possible for the delimiter to be a control sequence rather than a single character. An error is raised in this case. The \endgroup in \FV@ReadVArg@Char@i returns to prior catcodes after the argument is captured.

It would be possible to read the argument using any command-specific catcode settings, but that would result in different behavior depending on whether the argument is already tokenized, and would make reading the argument overly dependent on the argument content.

196 \def\FV@ReadVArg@Char#1#2#3{%
197 \expandafter\expandafter\expandafter
198 \if\expandafter\expandafter\expandafter\relax\expandafter\@gobble\detokenize{#3}\relax
199 \\expandafter\@gobble
200 \else
201 \\expandafter\@firstofone
202 \fi
203 {\PackageError{fvextra}{Verbatim delimiters must be single characters, not commands}{Try a different delimiter}}%
204 {Try a different delimiter}}%
205 \def\FV@ReadVArg@Char@i##1##2##3#3{%
206 \endgroup
207 \#1(#3)}%
208 \FV@ReadVArg@Char@i{#1}\FV@EOL%

Alternate implementation for pdfTeX with inputenc using UTF-8

Start conditional creation of macros:

210 \ifFV@pdfTeXinputenc

\FV@ReadVArg@Char@UTF This is a variant of \FV@ReadVArg@Char that allows non-ASCII codepoints as delimiters under the pdfTeX engine with inputenc using UTF-8. Under pdfTeX, non-ASCII codepoints must be handled as a sequence of bytes rather than as a single entity. \FV@ReadVArg@Char is automatically \let to this version when appropriate. This uses the \FV@U8:<byte> macros for working with inputenc's UTF-8.

211 \def\FV@ReadVArg@Char@UTF#1#2#3{%
212 \expandafter\expandafter\expandafter
213 \if\expandafter\expandafter\expandafter\relax\expandafter\@gobble\detokenize{#3}\relax
214 \\expandafter\@gobble
215 \else
216 \\expandafter\@firstofone

43
\PackageError{fvextra}{}
{Verbatim delimiters must be single characters, not commands}{}
{Try a different delimiter}{}
\ifcsname FV@U8:\detokenize{#3}\endcsname
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
\fi
\if\csname FV@U8:\detokenize{#3}\endcsname#3
\FV@ReadVArg@Char@UTF@i{#1}{#3}
\FV@ReadVArg@Char@UTF@i
\def\FV@ReadVArg@Char@UTF@i#1#2{\def\FV@ReadVArg@Char@i##1##2##3#2{##1{##3}
\FV@ReadVArg@Char@i{#1}\FV@EOL}
\def\FV@ReadVArg@Char@UTF@i{#1}{\FV@EOL}
End conditional creation of UTF macros:
\fi

12.4.3 Reading and protecting arguments in expansion-only contexts

The objective here is to make possible commands that can function correctly after
being in expansion-only contexts like \edef. The general strategy is to allow
commands to be defined like this:

%\def\cmd{\FVExtraRobustCommand\robustcmd\reader}
%
\robustcmd is the actual command, including argument reading and process-
ing, and is \texttt{protected}. \texttt{reader} is an expandable macro that reads all
of \texttt{robustcmd}’s arguments, then wraps them in \texttt{\FVExtraAlwaysUnexpanded}. When \texttt{\FVExtraAlwaysUnexpanded{⟨args⟩}} is expanded, the result is always
\texttt{\FVExtraAlwaysUnexpanded{⟨args⟩}}. \texttt{\FVExtraRobustCommand} is \texttt{protected}
and manages everything in a context-sensitive manner.

- In a normal context, \texttt{\FVExtraRobustCommand} reads two arguments, which
will be \texttt{\robustcmd} and \texttt{\reader}. It detects that \texttt{\reader} has not expanded
to \texttt{\FVExtraAlwaysUnexpanded{⟨args⟩}}, so it discards \texttt{\reader} and reinserts
\texttt{\robustcmd} so that it can operate normally.

- In an expansion-only context, neither \texttt{\FVExtraRobustCommand} nor \texttt{\robustcmd}
will expand, because both are \texttt{protected}. \texttt{\reader} will read \texttt{\robustcmd}’s
arguments and protect them with \texttt{\FVExtraAlwaysUnexpanded}. When this
is used later in a normal context, \texttt{\FVExtraRobustCommand} reads two argu-
ments, which will be \texttt{\robustcmd} and \texttt{\FVExtraAlwaysUnexpanded}. It
detects that \texttt{\reader} did expand, so it discards \texttt{\FVExtraAlwaysUnexpanded}
and reads its argument to discard the wrapping braces. Then it reinserts
\texttt{\robustcmd{⟨args⟩}} so that everything can proceed as if expansion had not
occurred.
\FVExtrapdfstringdef
Conditionally allow alternate definitions for PDF bookmarks when hyperref is in use. This is helpful for working with \protected or otherwise unexpandable commands.

\def\FVExtrapdfstringdef#1#2{\%\AfterPreamble\%\ifcsname pdfstringdef\endcsname\iffalse\pdfstringdef\relax\else\pdfstringdef#1{#2}\fi\fi\}%\def\FVExtrapdfstringdefDisableCommands#1{\%\AfterPreamble\%\ifcsname pdfstringdefDisableCommands\endcsname\iffalse\pdfstringdefDisableCommands{#1}\else\pdfstringdefDisableCommands{#1}\fi\fi\}%

\FVExtraAlwaysUnexpanded
Always expands to itself, thanks to \unexpanded.\long\def\FVExtraAlwaysUnexpanded#1{\unexpanded{\FVExtraAlwaysUnexpanded{#1}}}\def\FVExtraRobustCommand{\protect\def\FVExtraRobustCommand#1#2{\expandafter#1\@gobble#2}}\FVExtrapdfstringdefDisableCommands{\def\FVExtraRobustCommand{}}

\FVExtraRobustCommandExpanded
Boolean to track whether expansion occurred. Set in \FVExtraRobustCommand. Useful in creating commands that behave differently depending on whether expansion occurred.
\newbool{FVExtraRobustCommandExpanded}

\FVExtraRobustCommand
\protected\def\FVExtraRobustCommand#1#2{\%\ifeq#2\FVExtraAlwaysUnexpanded\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{\booltrue{FVExtraRobustCommandExpanded}\FV@RobustCommand@i{#1}}{\boolfalse{FVExtraRobustCommandExpanded}#1}}\def\FVExtraRobustCommand{\protect\def\FVExtraRobustCommand{}\long\def\FVExtraRobustCommand@i#1{}}

\FVExtraRobustCommand@i #2 will be the argument of \FVExtraAlwaysUnexpanded. Reading this strips the braces. At the beginning of #2 will be the reader macro, which must be \@@gobble'd.
\def\FVExtraRobustCommand@i#1#2{\expandafter\@firstoftwo\expandafter\@@gobble\expandafter\expandafter{#1\expandafter\expandafter{#2}}

\FVExtraUnexpandedReadStarOArgMArg
Read the arguments for a command that may be starred, may have an optional argument, and has a single brace-delimited mandatory argument. Then protect them with \FVExtraAlwaysUnexpanded. The reader macro is itself maintained in the protected result, so that it can be redefined to provide a simple default value for hyperref.

Note the argument signature #1#2. This reads everything up to, but not including, the next brace group.

45
This is a variant of \FVExtraUnexpandedReadStarOArgMArg customized for \UseVerb. It would be tempting to use \pdfstringdef to define a PDF string based on the final tokenization in \UseVerb, rather than applying \FVExtraPDFStringVerbatimDetokenize to the original raw (read) tokenization. Unfortunately, \pdfstringdef apparently can’t handle catcode 12 and %. Since the final tokenization could contain arbitrary catcodes, that approach might fail even if the \ and % issue were resolved. It may be worth considering more sophisticated approaches in the future.

Same as \FVExtraUnexpandedReadStarOArgMArg, except \BVArg, brace-delimited verbatim argument.

Same as \FVExtraUnexpandedReadStarOArgMArg, except \BEscVArg, brace-delimited escaped verbatim argument.


12.4.4 Converting detokenized tokens into PDF strings

At times it will be convenient to convert detokenized tokens into PDF strings, such as bookmarks. Define macros to escape such detokenized content so that it is in a suitable form.

Note that this does not apply any special treatment to spaces. If there are multiple adjacent spaces, then the octal escape \040 is needed to prevent them from being merged. In the detokenization macros where \FVExtraPDFStringEscapeChar is currently used, spaces are processed separately without \FVExtraPDFStringEscapeChar, and literal spaces or \040 are inserted in a context-dependent manner.

\FVExtraPDFStringEscapeChar

\begingroup
\catcode`\&=14
\catcode`\%=12&
\catcode`\(=12&
\catcode`\)=12&
\catcode`\^^J=12&
\catcode`\^^M=12&
\catcode`\^^I=12&
\catcode`\^^H=12&
\catcode`\^^L=12&
\catcode`\!=0\relax
\endgroup

Detokenizing verbatim arguments

Ensure correct catcodes for this subsection (note `<` and `>` for `\FV@<Sentinel>`):

```
\begingroup
\catcode `\=10
\catcode `\a=11
\catcode `\<=11
\catcode `\>=11
\catcode `\^^C=\active
```

Detokenize as if the original source were tokenized verbatim

```
\detokenize
```

This yields spaces with catcode 12, not spaces with catcode 10 like `\detokenize`. Spaces with catcode 10 require special handling when being read by macros, so detokenizing them to catcode 10 makes further processing difficult. Spaces with catcode 12 may be used just like any other catcode 12 token.

This requires that the `\active` end-of-text (ETX) `^^C` (U+0003) not be defined as `\outer`, since `^^C` is used as a sentinel. Usually, it should not be defined at
all, or defined to an error sequence. When in doubt, it may be worth explicitly defining \textbackslash{}C before using \FVExtraVerbatimDetokenize:

\begingroup
\catcode\textbackslash{}\textbackslash{}C=\active
\def\textbackslash{}\textbackslash{}C{}
\...
\FVExtraVerbatimDetokenize{...}
\...
\endgroup

\detokenize inserts a space after each control word (control sequence with a name composed of catcode 11 tokens, ASCII letters [a-zA-Z]). For example,

\detokenize{\acroA\acroB{}\csname name\endcsname123}

yields

\acroA \acroB {}\csname name\endcsname 123

That is the correct behavior when detokenizing text that will later be retokenized for normal use. The space prevents the control word from accidentally merging with any letters that follow it immediately, and will be gobbled by the macro when retokenized. However, the inserted spaces are unwanted in the current context, because

\FVExtraVerbatimDetokenize{\acroA\acroB{}\csname name\endcsname123}

should yield

\acroA\acroB{}\csname name\endcsname123

Note that the space is visible since it is catcode 12.

Thus, \FVExtraVerbatimDetokenize is essentially a context-sensitive wrapper around \detokenize that removes extraneous space introduced by \detokenize. It iterates through the tokens, detokenizing them individually and then removing any trailing space inserted by \detokenize.

\FV@VDetok@Scan This scans through a token sequence while performing two tasks:

1. Replace all catcode 10 spaces with catcode 12 spaces.
2. Insert macros that will process groups, after which they will insert yet other macros to process individual tokens.

Usage must always have the form

\FV@VDetok@Scan\{\tokens\}^^C \FV@<Sentinel>

where ^^C is \active, the catcode 10 space after ^^C is mandatory, and \FV@<Sentinel> is a single, undefined control word (this is accomplished via catcodes).
\texttt{\textbackslash FV@VDetok@Scan} searches for spaces to replace. After any spaces in \texttt{\langle tokens\rangle} have been handled, the space in \texttt{\textasciitilde\textbackslash C\textbackslash FV@<Sentinel>} triggers space processing. When \texttt{\textbackslash FV@VDetok@Scan} detects the sentinel macro \texttt{\textbackslash FV@<Sentinel>}, scanning stops.

The \texttt{\{\}} protects the beginning of \texttt{\langle tokens\rangle}, so that if \texttt{\langle tokens\rangle} is a group, its braces won’t be gobbled. Later, the inserted \texttt{\{\}} must be stripped so that it does not become part the processed \texttt{\langle tokens\rangle}.

\texttt{\textasciitilde\textbackslash C} is a convenient separator between \texttt{\langle tokens\rangle} and the rest of the sentinel sequence.

- Since \texttt{\textbackslash FV@VDetok@Scan} has delimited arguments, a leading catcode 10 space in \texttt{(tokens)} will be preserved automatically. Preserving a trailing catcode 10 space is much easier if it is immediately adjacent to a non-space character in the sentinel sequence; two adjacent catcode 10 spaces would be difficult to handle with macro pattern matching. However, the sentinel sequence must contain a catcode 10 space, so the sentinel sequence must contain at least 3 tokens.

- Since \texttt{\textasciitilde\textbackslash C} is not a control word, it does not gobble following spaces. That makes it much easier to assemble macro arguments that contain a catcode 10 space. This is useful because the sentinel sequence \texttt{\textasciitilde\textbackslash C\textbackslash FV@<Sentinel>} may have to be inserted into processing multiple times (for example, in recursive handling of groups).

- \texttt{\textbackslash FVExtraReadVArg} defines \texttt{\textasciitilde\textbackslash C} as \texttt{\textbackslash outer}, so any verbatim argument read by it is guaranteed not to contain \texttt{\textasciitilde\textbackslash C}. This is in contrast to \texttt{\textbackslash active} ASCII symbols and to two-character sequences \texttt{<backslash><symbol>} that should be expected in arbitrary verbatim content. It is a safe sentinel from that perspective.

- A search of a complete TeX Live 2018 installation revealed no other uses of \texttt{\textasciitilde\textbackslash C} that would clash (thanks, \texttt{ripgrep}!). As a control character, it should not be in common use except as a sentinel or for similar special purposes.

If \texttt{(tokens)} is empty or contains no spaces, then \texttt{\#1} will contain \texttt{\{\langle tokens\rangle\textasciitilde\textbackslash C}} and \texttt{\#2} will be empty. Otherwise, \texttt{\#1} will contain \texttt{\{\langle tokens\rangle_to_space\}} and \texttt{\#2} will contain \texttt{\langle tokens\rangle_after_space\textasciitilde\textbackslash C\textbackslash C\textbackslash C}}.

This uses the \texttt{\textbackslash if\textbackslash relax\textbackslash detokenize{\langle argument\rangle}\textbackslash relax} approach to check for an empty argument. If \texttt{\#2} is empty, then the space that was just removed by \texttt{\textbackslash FV@VDetok@Scan} reading its arguments was the space in the sentinel sequence, in which case scanning should end. \texttt{\#1} is passed on raw so that \texttt{\textbackslash FV@VDetok@ScanEnd} can strip the \texttt{\textasciitilde\textbackslash C} from the end, which is the only remaining token from the sentinel sequence \texttt{\textasciitilde\textbackslash C\textbackslash FV@<Sentinel>}. Otherwise, if \texttt{\#2} is not empty, continue. In that case, the braces in \texttt{\{\#1\}\{\#2\}} ensure arguments remain intact.

Note that \texttt{\textbackslash FV@<Sentinel>} is removed during each space search, and thus must be reinserted in \texttt{\textbackslash FV@VDetok@ScanCont}. It would be possible to use the macro signature \texttt{\#1 \#2} instead of \texttt{\#1 \textbackslash FV@<Sentinel> \#2}, and then do an \texttt{\textbackslash ifx} test on \texttt{\#2} for \texttt{\textbackslash FV@<Sentinel>}. However, that is problematic, because \texttt{\#2} may contain an arbitrary sequence of arbitrary tokens, so it cannot be used safely without \texttt{\textbackslash detokenize}.
This removes the `^^C` from the sentinel sequence `^^C\FV@<Sentinel>`, so the sentinel sequence is now completely gone. If `#1` is empty, there is nothing to do (``#1`` being empty means that `#1` consumed the `{}` that was inserted to protect anything following, because there was nothing after it). Otherwise, \texttt{\@gobble} the inserted `{}` before starting a different scan to deal with groups. The group scanner `\FV@VDetok@ScanGroup` has its own sentinel sequence `\{\FV@<Sentinel>\}`.

#1 is everything before the space. If `#1` is empty, there is nothing to do related to it; `#1` simply consumed an inserted `{}` that preceded nothing (that would be a leading space). Otherwise, start a different scan on `#1` to deal with groups. A non-empty `#1` will start with the `{}` that was inserted to protect groups, hence the \texttt{\@gobble} before group scanning.

Then insert a literal catcode 12 space to account for the space removed in `\FV@VDetok@Scan`. Note the catcode, and thus the lack of indentation and the `\%' to avoid unwanted catcode 12 spaces.

#2 is everything after the space, ending with `^^C` from the sentinel sequence `^^C\FV@<Sentinel>`. This needs continued scanning to deal with spaces, with `{}` inserted in front to protect a leading group and `\FV@<Sentinel>` after to complete the sentinel sequence.

The macro argument `#1` reads up to the next group. When this macro is invoked, the sentinel sequence `\{\FV@<Sentinel>\}` is inserted, so there is guaranteed to be at least one group.
Everything in #1 contains no spaces and no groups, and thus is ready for token scanning, with the sentinel `\FV@Sentinel`. Note that `\FV@Sentinel`, which is defined as `\def\FV@Sentinel{\FV@<Sentinel>}`, is used here, not `\FV@<Sentinel>`. `\FV@<Sentinel>` is not defined and is thus unsuitable for `\ifx` comparisons with tokens that may have been tokenized under an incorrect catcode regime and thus are undefined. `\FV@Sentinel` is defined, and its definition is resistant against accidental collisions.

\gdef\FV@VDetok@ScanGroup@i#1{% 
\FV@VDetok@ScanToken@i\FV@Sentinel 
\FV@VDetok@ScanGroup@ii}

\FV@VDetok@ScanGroup@ii The braces from the group are stripped during reading #1. Proceed based on whether the group is empty. If the group is not empty, {} must be inserted to protect #1 in case it is a group, and the new sentinel sequence `\FV@<Sentinel>^^C` is added for the group contents. `\FV@<Sentinel>` cannot be used as a sentinel for the group contents, because if this is the sentinel group `{\FV@<Sentinel>}`, then #1 is `{\FV@<Sentinel>}`.

\gdef\FV@VDetok@ScanEmptyGroup({}\FV@VDetok@ScanGroup)

\FV@VDetok@ScanEmptyGroup Insert {} to handle the empty group, then continue group scanning.

\begingroup \catcode`\(=1 \catcode`\)=2 \catcode`\{=12 \catcode`\}=12 \gdef\FV@VDetok@ScanEmptyGroup({}\FV@VDetok@ScanGroup@i\FV@<Sentinel>^^C)}\endgroup

\FV@VDetok@ScanGroup@ii The group is not empty, so determine whether it contains `\FV@<Sentinel>` and thus is the sentinel group. The group contents are followed by the sentinel sequence `\FV@<Sentinel>^^C` inserted in `\FV@VDetok@ScanGroup@i`. This means that if #2 is empty, the group did not contain `\FV@<Sentinel>` and thus is not the sentinel group. Otherwise, #2 will be `{\FV@<Sentinel>}`.

If this is not the sentinel group, then the group contents must be scanned, with surrounding literal braces inserted. #1 already contains an inserted leading {} to protect groups; see `\FV@VDetok@ScanGroup@i`. A sentinel sequence `^^C` is needed, though. Then group scanning must continue.

\begingroup \catcode`\(=1 \catcode`\)=2 \catcode`\{=12 \catcode`\}=12 \gdef\FV@VDetok@ScanGroup@ii({}\FV@VDetok@ScanGroup@i\FV@<Sentinel>^^C)}\endgroup
\FV@VDetok@ScanToken
Scan individual tokens. At this point, all spaces and groups have been handled,
so this will only ever encounter individual tokens that can be iterated with a
#1 argument. The sentinel for token scanning is \FV@Sentinel. This is the
appropriate sentinel because ifx comparisons are now safe (individual tokens)
and \FV@Sentinel is defined. Processing individual detokenized tokens requires
the same sentinel sequence as handling spaces, since it can produce them.

\gdef\FV@VDetok@ScanToken#1{%
  \ifx\FV@Sentinel#1%
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  \detokenize{#1}\relax\FV@<Sentinel>}}

\FV@VDetok@ScanToken@i
If #2 is empty, then there are no spaces in the detokenized token, so it is either an
\active character other than the space, or a two-character sequence of the form
<backslash><symbol> where the second character is not a space. Thus, #1 contains
\langle detokenized\rangle\`C. Otherwise, #1 contains (detokenized without space), and #2
may be discarded since it contains \`C, \FV@Sentinel. (If the detokenized token
contains a space, it is always at the end.)

\gdef\FV@VDetok@ScanToken@i#1 #2\FV@<Sentinel>{%
  \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  \detokenize{#1}\relax\FV@<Sentinel>}}

\FV@VDetok@ScanTokenNoSpace
Strip \`C sentinel in reading, then insert character(s) and continue scanning.

\gdef\FV@VDetok@ScanTokenNoSpace#1\relax\FV@<Sentinel>{#1\FV@VDetok@ScanToken}

\FV@VDetok@ScanTokenWithSpace
Handle a token that when detokenized produces a space. If there is nothing left
once the space is removed, this is the \active space. Otherwise, process further.

\gdef\FV@VDetok@ScanTokenWithSpace#1{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  \detokenize{#1}\relax\FV@<Sentinel>}}

\FV@VDetok@ScanTokenActiveSpace
\begin{group}
\catcode`\ =12%
\gdef\FV@VDetok@ScanTokenActiveSpace{ \FV@VDetok@ScanToken}\endgroup

53
If there is only one character left once the space is removed, this is the escaped space \. Otherwise, this is a command word that needs further processing.

\begingroup
\catcode`\ =12
\gdef\ScanTokenEscSpace#1{#1 \ScanToken}
\endgroup

Process control words in a context-sensitive manner by looking ahead to the next token (#2). The lookahead must be reinserted into processing, hence the \ScanTokenCW{#2}.

A control word will detokenize to a sequence of characters followed by a space. If the following token has catcode 11, then this space represents one or more space characters that must have been present in the original source, because otherwise the catcode 11 token would have become part of the control word’s name. If the following token has another catcode, then it is impossible to determine whether a space was present, so assume that one was not.

\begingroup
\catcode`\ =12
\gdef\ScanTokenCW#1#2{\ifcat\noexpand#2a\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{#1 \ScanToken#2}{#1\ScanToken#2}}
\endgroup

Detokenize as if the original source were tokenized verbatim, then convert to PDF string

\begingroup
\catcode`\ =12
\gdef\VerbatimDetokenize#1{\if\relax\detokenize{#1}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi\ScanTokenCW{#1}^{\@<Sentinel>}}
\endgroup

This is identical to \VerbatimDetokenize, except that the output is converted to a valid PDF string. Some spaces are represented with the octal escape \040 to prevent adjacent spaces from being merged.

\gdef\PDFStringVerbatimDetokenize#1{\PDFStrVDetok@Scan{}#1^{\@<Sentinel>}}
If there is only one character left once the space is removed, this is the escaped space \. Otherwise, this is a command word that needs further processing.

\begin{verbatim}
\gdef\FV@PDFStrVDetok@ScanTokenEscSpace\{#1\}{% 
\if\relax\detokenize{#2}\relax 
\expandafter\@firstoftwo 
\else 
\expandafter\@secondoftwo 
\fi
\FV@PDFStrVDetok@ScanToken{#1}{% 
\FVExtraPDFStringEscapeChar{#1}{040} \FV@PDFStrVDetok@ScanToken}
\end{verbatim}

This is modified to add \FVExtraPDFStringEscapeChar and use \040 for the space, since a space could follow.

\begin{verbatim}
\begingroup 
\catcode`\!=0\relax 
\catcode`\=12\relax 
\gdef\FV@PDFStrVDetok@ScanTokenCW\{#1#2\}{% 
\ifcat\noexpand#2a% 
\expandafter\@firstoftwo% 
\else% 
\expandafter\@secondoftwo% 
\fi% 
\FVExtraPDFStringEscapeChars{#1} \FV@PDFStrVDetok@ScanToken#2}
\end{verbatim}

This is modified to add \FVExtraPDFStringEscapeChars.

\begin{verbatim}
\begingroup 
\catcode`\!=12% 
\gdef\FV@PDFStrVDetok@ScanTokenCW\{#1\}{% 
\ifcat\noexpand#2a%
\expandafter\@firstoftwo%
\else%
\expandafter\@secondoftwo%
\fi%
\FVExtraPDFStringEscapeChars\{#1\} \FV@PDFStrVDetok@ScanToken#2%
\FVExtraPDFStringEscapeChars\{#1\} \FV@PDFStrVDetok@ScanToken#2%
\end{verbatim}

\FV@PDFStrVDetok@ScanTokenCW

Detokenize as if the original source were tokenized verbatim, except for backlash escapes of non-catcode 11 characters

\begin{verbatim}
\begingroup 
\catcode`\!=12% 
\gdef\FV@PDFStrVDetok@ScanTokenCW#1\{#2\}{% 
\ifcat\noexpand#2a% 
\expandafter\@firstoftwo% 
\else% 
\expandafter\@secondoftwo% 
\fi% 
\FVExtraPDFStringEscapeChars\{#1\} \FV@PDFStrVDetok@ScanToken#2%
\FVExtraPDFStringEscapeChars\{#1\} \FV@PDFStrVDetok@ScanToken#2%
\end{verbatim}

ExtraEscapedVerbatimDetokenize

This is a variant of \FVExtraVerbatimDetokenize that treats character sequences of the form \<char> as escapes for <char>. It is primarily intended for making \<symbol> escapes for <symbol>, but allowing arbitrary escapes simplifies the default behavior and implementation. This is useful in constructing nearly verbatim commands that can be used inside other commands, because the backlash escapes allow for characters like # and %, as well as making possible multiple adjacent spaces via \. It should be applied to arguments that are read verbatim insofar as is possible, except that the backlash \ should have its normal meaning (catcode 0). Most of the implementation is identical to that for \FVExtraVerbatimDetokenize. Only the token processing requires modification to handle backlash escapes.

It is possible to restrict escapes to ASCII symbols and punctuation. See \FVExtraDetokenizeREscVArg. The disadvantage of restricting escapes is that it
prevents functioning in an expansion-only context (unless you want to use undefined macros as a means of raising errors). The advantage is that it eliminates ambiguity introduced by allowing arbitrary escapes. Backslash escapes of characters with catcode 11 (ASCII letters, [A-Za-z]) are typically not necessary, and introduce ambiguity because something like \x will gobble following spaces since it will be tokenized originally as a control word.

\gdef\FVExtraEscapedVerbatimDetokenize#1{%
  \FV@EscVDetok@Scan{}#1^^C \FV@<Sentinel>}

\FV@EscVDetok@Scan

\gdef\FV@EscVDetok@Scan#1 #2\FV@<Sentinel>{%
  \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FV@EscVDetok@ScanEnd#1}%
  {\FV@EscVDetok@ScanCont{#1}{#2}}}

\FV@EscVDetok@ScanEnd

\gdef\FV@EscVDetok@ScanEnd#1^^C{%
  \if\relax\detokenize{#1}\relax%
    \expandafter\@gobble%
  \else%
    \expandafter\@firstofone%
  \fi%
  {\expandafter\FV@EscVDetok@ScanGroup\@gobble#1{\FV@<Sentinel>}}%
  %<-catcode 12 space
  \FV@EscVDetok@Scan{}#2\FV@<Sentinel>}}

\FV@EscVDetok@ScanCont

\begingroup
  \catcode`\ =12%
\gdef\FV@EscVDetok@ScanCont#1#2{%
  \if\relax\detokenize{#1}\relax%
    \expandafter\@firstoftwo%
  \else%
    \expandafter\@secondoftwo%
  \fi%
  {\expandafter\FV@EscVDetok@ScanGroup\@gobble#1{\FV@<Sentinel>}}%
  %<-catcode 12 space
  \FV@EscVDetok@Scan{}#2\FV@<Sentinel>}}%
\endgroup

\FV@EscVDetok@ScanGroup

\gdef\FV@EscVDetok@ScanGroup#1#2{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\expandafter\FV@EscVDetok@ScanGroup#1}{#2}}

\FV@EscVDetok@ScanGroup@i
Parallel implementations, with a restricted option  Starting here, there are alternate macros for restricting escapes to ASCII punctuation and symbols. These alternates have names of the form \texttt{\textbackslash FV\textbackslash REscVDetok\textbackslash <name>}. They are used in \texttt{\textbackslash FVExtraDetokenizeREscVArg}. The alternate \texttt{\textbackslash FV\textbackslash REscVDetok\textbackslash <name>} macros replace invalid escape sequences with the undefined \texttt{\textbackslash FV\textbackslash <InvalidEscape>}, which is later scanned for with a delimited macro.

59
This was modified from \FV@EscVDetok@ScanTokenNoSpace to discard the first character of multi-character sequences (that would be the backslash \).

\gdef\FV@EscVDetok@ScanTokenNoSpace#1#2^^C{%\if\relax\detokenize{#2}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{#1\FV@EscVDetok@ScanToken}{#2\FV@EscVDetok@ScanToken}}

If there is only one character left once the space is removed, this is the escaped space \. Otherwise, this is a command word. A command word is passed on so as to keep the backslash and letters separate.

\gdef\FV@EscVDetok@ScanTokenWithSpace#1{\if\relax\detokenize{#1}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{\FV@EscVDetok@ScanTokenActiveSpace}{\FV@EscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}

\gdef\FV@EscVDetok@ScanTokenWithSpace@i#1#2\FV@<Sentinel>{%\if\relax\detokenize{#2}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{\FV@EscVDetok@ScanTokenEscSpace{#1}}{\FV@EscVDetok@ScanTokenCW{#1}{#2}}}

\gdef\FV@EscVDetok@ScanTokenWithSpace0i{\begin{group}
\catcode`\ =12\gdef\FV@EscVDetok@ScanTokenActiveSpace{ \FV@EscVDetok@ScanToken}%
\endgroup}

\gdef\FV@EscVDetok@ScanTokenWithSpace0i01#1#2\FV@<Sentinel>{%\if\relax\detokenize{#2}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{\FV@EscVDetok@ScanTokenEscSpace{#1}}{\FV@EscVDetok@ScanTokenCW{#1}{#2}}}

\gdef\FV@EscVDetok@ScanTokenWithSpace0i02#1#2\FV@<Sentinel>{%\if\relax\detokenize{#2}\relax
This is modified to drop \#1, which will be the backslash.

\begingroup
\catcode`\ =12\%
\gdef\FV@EscVDetok@ScanTokenEscSpace#1{ \FV@EscVDetok@ScanToken\space }
\endgroup
\FV@EscVDetok@ScanTokenCW

This is modified to accept an additional argument, since the control word is now split into backslash plus letters.

\begingroup
\catcode`\ =12\%
\gdef\FV@EscVDetok@ScanTokenCW#1#2#3{\ifcat\noexpand#2a\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{#2 \FV@EscVDetok@ScanToken#3}{}{#2\FV@EscVDetok@ScanToken#3}}
\endgroup

Detokenize as if the original source were tokenized verbatim, except for backslash escapes of non-catcode 11 characters, then convert to PDF string

\FVExtraPDFStringEscapedVerbatimDetokenize This is identical to \FVExtraEscapedVerbatimDetokenize, except that the output is converted to a valid PDF string. All spaces are represented with the octal escape \040 to prevent adjacent spaces from being merged. There is no alternate implementation for restricting escapes to ASCII symbols and punctuation. Typically, this would be used in an expansion-only context to create something like bookmarks, while \FVExtraEscapedVerbatimDetokenize (potentially with escape restrictions) would be used in parallel to generate whatever is actually typeset. Escape errors can be handled in generating what is typeset.

\begingroup
\catcode`\ =12\%
\gdef\FVExtraPDFStringEscapedVerbatimDetokenize#1{%
\FV@PDFStrEscVDetok@Scan{}#1^^C \FV@<Sentinel>}%
\endgroup

\FV@PDFStrEscVDetok@Scan

\gdef\FV@PDFStrEscVDetok@Scan#1 #2\FV@<Sentinel>{\if\relax\detokenize{#2}\relax\expandafter\@firstoftwo\else\expandafter\@secondoftwo\fi{\FV@PDFStrEscVDetok@ScanEnd#1}{\FV@PDFStrEscVDetok@ScanCont{#1}{#2}}}

\FV@PDFStrEscVDetok@ScanEnd

\FV@PDFStrEscVDetok@ScanCont
This is modified to use \040 for the space. In the unescaped case, using a normal space here is fine, but in the escaped case, the preceding or following token could be an escaped space.
\catcode`\(=1
\catcode`\)=2
\catcode`\{=12
\catcode`\}=12
\gdef\FV@PDFStrEscVDetok@ScanGroup@ii#1\FV@<Sentinel>\#2``C (%
    \if\relax\detokenize(#2)\relax
    \expandafter\@firstofone
    \else
    \expandafter\@gobble
    \fi
    ({{\FV@PDFStrEscVDetok@Scan#1``C \FV@<Sentinel>}}\FV@PDFStrEscVDetok@ScanGroup))
\endgroup
\FV@PDFStrEscVDetok@ScanToken
\gdef\FV@PDFStrEscVDetok@ScanToken#1{%
    \ifx\FV@Sentinel#1% 
    \expandafter\@gobble
    \else
    \expandafter\@firstofone
    \fi
    \detokenize{#1}``C \FV@<Sentinel>}}
\FV@PDFStrEscVDetok@ScanToken@i
\gdef\FV@PDFStrEscVDetok@ScanToken@i#1 #2\FV@<Sentinel>{%
    \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FVExtraPDFStringEscapeChar{#1}\FV@PDFStrEscVDetok@ScanToken%
    \FVExtraPDFStringEscapeChar{#2}\FV@PDFStrEscVDetok@ScanToken}
\FV@PDFStrEscVDetok@ScanTokenWithSpace
\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace#1{%
    \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenActiveSpace%
    \FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
\FV@PDFStrEscVDetok@ScanTokenActiveSpace
\gdef\FV@PDFStrEscVDetok@ScanTokenActiveSpace#1{%
    \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenActiveSpace%
    \FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
\FVExtraPDFStringEscapeChar
\gdef\FV@PDFStrEscVDetok@ScanTokenNoSpace#1#2``C{%
    \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenNoSpace@i#1\FV@<Sentinel>}}
\FV@PDFStrEscVDetok@ScanTokenWithSpace
\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace#1{%
    \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
\FV@PDFStrEscVDetok@ScanTokenWithSpace@i
\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1 #2\FV@<Sentinel>{%
    \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FVExtraPDFStringEscapeChar{#1}\FV@PDFStrEscVDetok@ScanToken%
    \FVExtraPDFStringEscapeChar{#2}\FV@PDFStrEscVDetok@ScanToken}%
\FV@PDFStrEscVDetok@ScanTokenWithSpace
\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace#1{%
    \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
\FV@PDFStrEscVDetok@ScanTokenWithSpace@i
\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1 #2\FV@<Sentinel>{%
    \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
\FVExtraPDFStringEscapeChar
\gdef\FV@PDFStrEscVDetok@ScanTokenNoSpace#1#2``C{%
    \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
    \else
    \expandafter\@secondoftwo
    \fi
    \FV@PDFStrEscVDetok@ScanTokenNoSpace{#1}\FV@PDFStrEscVDetok@ScanToken%
    \FV@PDFStrEscVDetok@ScanTokenNoSpace{#2}\FV@PDFStrEscVDetok@ScanToken}
\begingroup
\catcode`!\!=0\relax
\catcode`\=12!relax
\gdef\FV@PDFStrEscVDetok@ScanTokenActiveSpace{\040!FV@PDFStrEscVDetok@ScanToken}%
\catcode`!\!=0!relax
\endgroup

\texttt\gdef\FV@PDFStrEscVDetok@ScanTokenWithSpace@i#1#2\FV@<Sentinel>{%
\if\relax\detokenize{#2}\relax
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
\fi
\{\FV@PDFStrEscVDetok@ScanTokenEscSpace{#1}\%
\{\FV@PDFStrEscVDetok@ScanTokenCW{#1}{#2}\}%
\FV@PDFStrEscVDetok@ScanTokenEscSpace
This is modified to drop \#1, which will be the backslash, and use \040 for the space.
\gdef\FV@PDFStrEscVDetok@ScanTokenEscSpace#1\{\040!FV@PDFStrEscVDetok@ScanToken}{%
\catcode`\=12%
\gdef\FV@PDFStrEscVDetok@ScanTokenCW#1#2#3{%
\ifcat\noexpand#2a%
\expandafter\@firstoftwo%
\else%
\expandafter\@secondoftwo%
\fi%
\FVExtraPDFStringEscapeChars{#2} \FV@PDFStrEscVDetok@ScanToken#3%
\FVExtraPDFStringEscapeChars{#2}\FV@PDFStrEscVDetok@ScanToken#3}%
\endgroup

\FVExtraDetokenizeVArg
\texttt\gdef\FVExtraDetokenizeVArg#1#2{%
\begingroup
\catcode`\-=12%
\gdef\FV@PDFStrEscVDetok@ScanTokenCW#1#2#3{%
\ifcat\noexpand#2a%
\expandafter\@firstoftwo%
\else%
\expandafter\@secondoftwo%
\fi%
\FVExtraPDFStringEscapeChars{#2} \FV@PDFStrEscVDetok@ScanToken#3%
\FVExtraPDFStringEscapeChars{#2}\FV@PDFStrEscVDetok@ScanToken#3}%
\endgroup

Detokenization wrappers

\FVExtraDetokenizeVArg
\texttt\gdef\FVExtraDetokenizeVArg#1#2{%
\begingroup
\catcode`\``=\active
\let``=\FV@Sentinel
\edef\FV@Tmp{\FVExtraVerbatimDetokenize{#2}}% 
\expandafter\FV@DetokenizeVArg@i\expandafter{\FV@Tmp}{#1}}%
This is the same as \FVExtraDetonizeVArg, except it is intended to work with \FVExtraReadEscVArg by using \FVExtraEscapedVerbatimDetokenize.

\FVExtraDetonizeEscVArg
\FVExtraDetonizeREscVArg

12.4.6 Retokenizing detokenized arguments

\FVRetokVArg@Read
Read all tokens up to \active ^^C^^M, then save them in a macro for further use. This is used to read tokens inside $\textsc{scantokens}$ during retokenization. The $\textbf{\textbackslash{begingroup}}$ disables catcode modifications that will have been put in place for the reading process, including making $\textasciicircum\textasciicircum C$ and $\textasciicircum\textasciicircum M \textbackslash{active}$.

\FVExtraRetokenizeVArg
This retokenizes the detokenized output of something like \FVExtraVerbatimDetokenize or \FVExtraDetonizeVArg. \#1 is a macro that receives the output, \#2 sets catcodes but includes no \textbf{\textbackslash{begingroup}} or \textbf{\textbackslash{endgroup}}, and \#3 is the detokenized
characters. \texttt{\textbackslash FV@RetokVArg@Read} contains an \texttt{\textbackslash endgroup} that returns catcodes to their prior state.

This is a somewhat atypical use of \texttt{\textbackslash scantokens}. There is no \texttt{\textbackslash everyeof\{\textbackslash noexpand\}} to handle the end-of-file marker, and no \texttt{\textbackslash endlinechar=-1} to ignore the end-of-line token so that it does not become a space. Rather, the end-of-line \texttt{\textbackslash#M} is made \texttt{\textbackslash active} and used as a delimiter by \texttt{\textbackslash FV@RetokVArg@Read}, which reads characters under the new catcode regime, then stores them unexpanded in \texttt{\textbackslash FV@TmpRetoked}.

Inside \texttt{\textbackslash scantokens} is \texttt{\textbackslash#B\#3\textbackslash#C}. This becomes \texttt{\textbackslash#B\#3\textbackslash#C\textbackslash#M} once \texttt{\textbackslash scantokens} inserts the end-of-line token. \texttt{\textbackslash#B} is \texttt{\textbackslash let} to \texttt{\textbackslash FV@RetokVArg@Read}, rather than using \texttt{\textbackslash FV@RetokVArg@Read} directly, because \texttt{\textbackslash scantokens} acts as a \texttt{\textbackslash write} followed by \texttt{\textbackslash input}. That means that a command word like \texttt{\textbackslash FV@RetokVArg@Read} will have a space inserted after it, while an \texttt{\textbackslash active} character like \texttt{\textbackslash#B} will not. Using \texttt{\textbackslash#B} is a way to avoid needing to remove this space; it is simpler not to handle the scenario where \texttt{\textbackslash FV@RetokVArg@Read} introduces a space and the detokenized characters also start with a space. The \texttt{\textbackslash#C} is needed because trailing spaces on a line are automatically stripped, so a non-space character must be part of the delimiting token sequence.

\begin{verbatim}
\begingroup
\catcode`\#=\active
\catcode`\#=\active
\gdef\FVExtraRetokenizeVArg#1#2#3{%
  \begingroup
    #2%
    \catcode`\#=\active
    \catcode`\#=\active
    \catcode`\#=\active
    \let\#B\FV@RetokVArg@Read
    \let\#C\@empty
    \FV@DefEOLEmpty
    \scantokens{\#B\#3\#C}%
  \endgroup
\endgroup
\end{verbatim}

\subsection{Hooks}

These are hooks for extending \texttt{\textbackslash FV@FormattingPrep}. \texttt{\textbackslash FV@FormattingPrep} is inside a group, before the beginning of processing, so it is a good place to add extension code. These hooks are used for such things as tweaking math mode behavior and preparing for \texttt{\textbackslash breakbefore} and \texttt{\textbackslash breakafter}. The \texttt{\textbackslash PreHook} should typically be used, unless \texttt{\textbackslash fancyvrb}'s font settings, whitespace setup, and active character definitions are needed for extension code.

This is a hook for turning on Pygments-related features for packages like \texttt{\textbackslash minted} and \texttt{\textbackslash pythontex} (section 12.13). It needs to be the first thing in
Define versions of common escaped characters that reduce to raw characters. This is useful, for example, when working with text that is almost verbatim, but was captured in such a way that some escapes were unavoidable.

```
\def\#\FV@hashchar
\let\%\@percentchar
\let\{\@charlb
\let\}\@charrb
\let\$\FV@dollarchar
\let\&\FV@ampchar
\let\_\FV@underscorechar
\let\^\FV@caretchar
\let\\@backslashchar
\let\~\FV@tildechar
\let\~\FV@tildechar
\let\[\FV@leftsquarebracket
\let\]\FV@rightsquarebracket
\let\,\FV@commachar
\let\\%\@percentchar
\let\\{\@charlb
\let\\}\@charrb
\let\\$\FV@dollarchar
\let\\&\FV@ampchar
\let\\_\FV@underscorechar
\let\\^\FV@caretchar
\let\\\@backslashchar
\let\\~\FV@tildechar
\let\\~\FV@tildechar
\let\\[\FV@leftsquarebracket
\let\\]\FV@rightsquarebracket
\let\\,\FV@commachar
\}\%$ <-- highlighting
```

12.7 Inline-only options

Create `\fvinlineset` for inline-only options. Note that this only applies to new or reimplemented inline commands that use `\FV@UseInlineKeyValues`.

```
\FV@InlineKeyValues
935 \def\FV@InlineKeyValues{}

\fvinlineset
936 \def\fvinlineset#1{%
937 \expandafter\def\expandafter\FV@InlineKeyValues\expandafter{%
938 \FV@InlineKeyValues#1,}}

\FV@UseInlineKeyValues
939 \def\FV@UseInlineKeyValues{%
940 \expandafterfvset\expandafter{\FV@InlineKeyValues}}
```
12.8 Reimplementations

fvextra reimplements some fancyverb internals. The patches in section 12.10 fix bugs, handle edge cases, and extend existing functionality in logical ways, while leaving default fancyverb behavior largely unchanged. In contrast, reimplementations add features by changing existing behavior in significant ways. As a result, there is a boolean option `extra` that allows them to be disabled.

12.8.1 extra option

Boolean option that governs whether reimplemented commands and environments should be used, rather than the original definitions.

\begin{verbatim}
\newbool{FV@extra}
\define@booleankey{FV}{extra}{\booltrue{FV@extra}}{\boolfalse{FV@extra}}{\fvset{extra=true}}
\end{verbatim}

12.8.2 `\FancyVerbFormatInline`

This allows customization of inline verbatim material. It is the inline equivalent of `\FancyVerbFormatLine` and `\FancyVerbFormatText`.

\begin{verbatim}
\FancyVerbFormatInline
\end{verbatim}

12.8.3 `\Verb`

`\Verb` is reimplemented so that it functions as well as possible when used within other commands.

`\verb` cannot be used inside other commands. The original fancyverb implementation of `\Verb` does work inside other commands, but being inside other commands reduces its functionality since there is no attempt at retokenization. When used inside other commands, it essentially reduces to `\texttt`. `\Verb` also fails when the delimiting characters are active, since it assumes that the closing delimiting character will have catcode 12.

fvextra’s re-implemented `\Verb` uses `\scantokens` and careful consideration of catcodes to (mostly) remedy this. It also adds support for paired curly braces {...} as the delimiters for the verbatim argument, since this is often convenient when `\Verb` is used within another command. The original `\Verb` implementation is completely incompatible with curly braces being used as delimiters, so this doesn’t affect backward compatibility.

The re-implemented `\Verb` is constructed with `\FVExtraRobustCommand` so that it will function correctly after being in an expansion-only context, so long as the argument is delimited with curly braces.

\begin{verbatim}
\def\Verb{\FVExtraRobustCommand\RobustVerb\FVExtraUnexpandedReadStarOArgBVArg}
\end{verbatim}
\RobustVerb
949 \protected\def\RobustVerb{\FV@Command{}{Verb}}
950 \FVEextrapdfstringdefDisableCommands{%
951 \def\RobustVerb{}}
\FVC@Verb@FV  Save the original fancyverb definition of \FVC@Verb, so that the extra option can switch back to it.
952 \let\FVC@Verb@FV\FVC@Verb
\FVC@Verb  Redefine \FVC@Verb so that it will adjust based on extra.
953 \def\FVC@Verb{%
954 \begingroup
955 \FV@UseInlineKeyValues\FV@UseKeyValues
956 \ifFV@extra
957 \expandafter\endgroup\expandafter\FVC@Verb@Extra
958 \else
959 \expandafter\endgroup\expandafter\FVC@Verb@FV
960 \fi}
\FVC@Verb@Extra  fvextra reimplemention of \FVC@Verb.
959 When used after expansion, there is a check for valid delimiters, curly braces. If incorrect delimiters are used, and there are no following curly braces, then the reader macro \FVEextraUnexpandedReadStarOArgBVArg will give an error about unmatched braces. However, if incorrect delimiters are used, and there are following braces in a subsequent command, then this error will be triggered, preventing interference with the following command by the reader macro.
961 \def\FVC@Verb@Extra{%
962 \ifbool{FVExtraRobustCommandExpanded}%
963 {\@ifnextchar\bgroup
964 {\FVC@Verb@Extra@i}%
965 {\PackageError{fvextra}{\string\Verb\space delimiters must be paired curly braces in this context}%
966 {Use curly braces as delimiters}}}%
968 {\FVC@Verb@Extra@i}}
\FVC@Verb@Extra@i  \FVC@Verb@Extra@i
969 \def\FVC@Verb@Extra@i{%
970 \begingroup
971 \FVEextraReadVArg{%
972 \FV@UseInlineKeyValues\FV@UseKeyValues\FV@FormattingPrep
973 \FVEextraDetokenizeVArg{%
974 \FVEextraRetokenizeVArg{\FVC@Verb@Extra@ii}{\FV@CatCodes}}}{}
\FVC@Verb@Extra@ii  \FVC@Verb@Extra@ii
975 \def\FVC@Verb@Extra@ii{%
976 \ifFV@breaklines
977 \expandafter\@firstoftwo
978 \else
979 \expandafter\@secondoftwo
980 \fi
981 \FV@InsertBreaks{\FancyVerbFormatInline}{#1}%%
982 {\unbox(#1)}%
983 \endgroup}
12.8.4 \SaveVerb

This is reimplemented, following \Verb as a template, so that both \Verb and \SaveVerb are using the same reading and tokenization macros. This also adds support for \fvinlineset. Since the definition in fancyverb is

```
%\def\SaveVerb{\FV@Command{}{SaveVerb}}
%
```

only the internal macros need to be reimplemented.

```
\FVC@SaveVerb@FV
\FVC@SaveVerb
\FVC@SaveVerb@Extra
\FVC@SaveVerb@Extra@i
\FVC@SaveVerb@Extra@ii
```

12.8.5 \UseVerb

This adds support for \fvinlineset and line breaking. It also adds movable argument and PDF string support. A new option \texttt{retokenize} is defined that determines whether the typeset output is based on the \texttt{commandchars} and \texttt{codes} in place when \SaveVerb was used (default), or is retokenized under current \texttt{commandchars} and \texttt{codes}.

70
FV@retokenize Whether \UseVerb uses saved verbatim with its original tokenization, or retokenizes under current commandchars and codes.

\UseVerb
\def\UseVerb{%
\FVExtraRobustCommand\RobustUseVerb\FVExtraUseVerbUnexpandedReadStarOArgMArg}

\RobustUseVerb
\protected\def\RobustUseVerb{
\FV@Command{}{UseVerb}}
\FVExtrapdfstringdefDisableCommands{%
\def\RobustUseVerb{}}

\FVC@UseVerb@FV
\let\FVC@UseVerb@FV\FVC@UseVerb
\def\FVC@UseVerb{%
\begingroup
\FV@UseInlineKeyValues\FV@UseKeyValues
\ifFV@extra
\expandafter\endgroup\expandafter\FVC@UseVerb@Extra
\else
\expandafter\endgroup\expandafter\FVC@UseVerb@FV
\fi}
\FV@UseVerb@Extra
\def\FV@UseVerb@Extra#1{%
@ifundefined{FV@SV@#1}{\FV@Error{Short verbatim text never saved to name `#1`}\FV@eha}{
\begingroup
\FV@UseInlineKeyValues\FV@UseKeyValues\FV@FormattingPrep
\ifbool{FV@retokenize}{
\expandafter\let\expandafter\FV@Tmp\csname FV@SVRaw@#1\endcsname
\expandafter\FV@UseVerb@Extra@Retok\expandafter{\FV@Tmp}}{
\expandafter\let\expandafter\FV@Tmp\csname FV@SV@#1\endcsname
\expandafter\FV@UseVerb@Extra\expandafter{\FV@Tmp}}}}
\FV@UseVerb@Extra@Retok
\def\FV@UseVerb@Extra@Retok#1{%
\FVExtraDetokenizeVArg{\FVExtraRetokenizeVArg{\FV@UseVerb@Extra}{\FV@CatCodes}}{#1}}
\FV@UseVerb@Extra
\def\FV@UseVerb@Extra#1{%
\ifFV@breaklines
\@firstoftwo
\else
\@secondoftwo
\fi
\FV@InsertBreaks{\FancyVerbFormatInline}{#1}
\mbox{#1}
\endgroup}

\FV@UseVerb@Extra@ReTok
\def\FV@UseVerb@Extra@ReTok#1{%
\FVExtraDetokenizeVArg{%
\FVExtraRetokenizeVArg{\FV@UseVerb@Extra}{\FV@CatCodes}}{#1}}

\FV@UseVerb@Extra
\def\FV@UseVerb@Extra#1{%\ifFV@breaklines
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
\fi
\FV@InsertBreaks{\FancyVerbFormatInline}{#1}
\mbox{#1}}

71
12.9 New commands and environments

12.9.1 \EscVerb

This is a variant of \Verb in which backslash escapes of the form \char are used for \char. Backslash escapes are only permitted for printable, non-alphanumeric ASCII characters. The argument is read under a normal catcode regime, so any characters that cannot be read under normal catcodes must always be escaped, and the argument must always be delimited by curly braces. This ensures that \EscVerb behaves identically whether or not it is used inside another command.

\EscVerb is constructed with \FVExtraRobustCommand so that it will function correctly after being in an expansion-only context.

\EscVerb Note that while the typeset mandatory argument will be read under normal catcodes, the reader macro for expansion is \FVExtraUnexpandedReadStarOArgBEscVArg. This reflects how the argument will be typeset.

\RobustEscVerb

\FVC@EscVerb Delimiting with curly braces is required, so that the command will always behave the same whether or not it has been through expansion.

\FVC@EscVerb@i

\FVC@EscVerb@ii

72
12.9.2 VerbEnv

Environment variant of \Verb. Depending on how this is used in the future, it may be worth improving error message and error recovery functionality, using techniques from fancyverb.

\VerbEnv

\def\VerbEnv{%
  \ifcsname @currenvir\endcsname
    \ifx@currenvir@empty
      \PackageError{fvextra}{VerbEnv is an environment}{VerbEnv is an environment}%
    \else
      \PackageError{fvextra}{VerbEnv is an environment}{VerbEnv is an environment}%
    \fi
  \fi
  \else
    \PackageError{fvextra}{VerbEnv is an environment}{VerbEnv is an environment}%
  \fi
  \VerbatimEnvironment
  \FVExtraReadOArgBeforeVEnv{\expandafter\VerbEnv@i\expandafter{\FV@EnvironName}}}

\def\VerbEnv@i#1#2{%
  \begingroup
    \let\do\@makeother\FVExtraDoSpecials
    \catcode`\ =\active
    \catcode`\^^I=\active
    \catcode`\^^M=\active
    \VerbEnv@ii{#1}{#2}}

\begingroup
  \catcode`\!=0
  \catcode`\!<1
  \catcode`\>=2
  \ifcatcode`\!=12
    \ifcatcode`\!=12
      \ifcatcode`\!=12
        \edef\VerbEnv@iii@Error{}% 
      \else
        \PackageError{fvextra}{Missing environment contents}{Missing environment contents}%
        \let\VerbEnv@iii@Error\VerbEnv@iii@Error% 
      \fi
      \VerbEnv@CheckLine##1\end{#1}##2\FV@Sentinel<% 
      \else
        \PackageError{fvextra}{Missing environment contents}{Missing environment contents}% 
        \let\VerbEnv@iii\VerbEnv@iii@Error% 
      \fi 
    \else
      \VerbEnv@CheckLine#3\end{#1}!FV@Sentinel1% 
      \VerbEnv@iii#1<##2<##3>% 
      \VerbEnv@iii#1#2#3\endgroup% 
    \fi
  \endgroup
  \def\VerbEnv@iii@Error#1#2#3{}
\endgroup
\def\VerbEnv@ii#1#2#3{%
  \begingroup
    \let\do\@makeother\FVExtraDoSpecials
    \catcode`\ =10\relax
    \catcode`\^^M=\active
    \VerbEnv@iv{#1}{#2}{#3}}
12.9.3 VerbatimWrite

This environment writes its contents to a file verbatim. Differences from fancyvrb's VerbatimOut:

- Multiple VerbatimWrite environments can write to the same file. The file is set via the writefilehandle option. This does mean that the user is responsible for creating a new file handle via \newwrite and then ideally invoking \closeout at the appropriate time.

- By default, text is really written verbatim. This is accomplished by a combination of setting catcodes to 12 (other) and \detokenize. This can be customized using the new writer option, which defines a macro that performs any processing on each line before writing it to file. By default, all fancyvrb options except for VerbatimWrite-specific options are ignored. This can be customized on a per-environment basis via environment optional arguments.

writefilehandle \Set file handle for VerbatimWrite.
\FancyVerbWriteFileHandle

\FV@SetWrite#1\FV@Sentinel{\let\FancyVerbWriteFileHandle\relax\if\expandafter\detokenize\expandafter{#2}\relax\let\FancyVerbWriteFileHandle#1\relax\fi\if\relax\PackageError{fvextra}{Missing or invalid file handle for write}{Need file handle from \string\newwrite}\fi}

writer \Define writer macro that processes each line before writing.
\FV@Writer

\FV@Writer\FancyVerbDefaultWriter\FancyVerbWriteFileHandle{\detokenize{#1}}
\fvset{writer=\FancyVerbDefaultWriter}

VerbatimWrite The environment implementation follows standard fancyvrb environment style.

A special write counter is used to track line numbers while avoiding incrementing the regular counter that is used for typeset code. Some macros do nothing with the default writer, but are needed to enable fancyvrb options when a custom writer is used in conjunction with optional environment arguments. These include \FancyVerbDefineActive, \FancyVerbFormatCom, and \FV@DefineTabOut.

\newcounter{FancyVerbWriteLine}
\def\VerbatimWrite{\FV@Environment{codes=,commandchars=none,commentchar=none,defineactive,%,gobble=0,formatcom=.firstline,lastline}%{VerbatimWrite}}
12.9.4 VerbatimBuffer

This environment stores its contents verbatim in a “buffer,” a sequence of numbered macros each of which contains one line of the environment. The “buffered” lines can then be looped over for further processing or later use.

By default, all fancyVerb options except for VerbatimBuffer-specific options are ignored. This can be customized on a per-environment basis via environment optional arguments.

afterbuffer Macro that is inserted after the last line of the environment is buffered, immediately before the environment ends.

\FV@afterbuffer
\define@key{FV}{afterbuffer}{\def\FV@afterbuffer{#1}}
\fvset{afterbuffer=}

FancyVerbBufferIndex Current index in buffer during buffering. This is given a \FancyVerb* macro name since it may be accessed by the user in defining custom bufferer.

\newcounter{FancyVerbBufferIndex}

bufferer This is the macro that adds lines to the buffer. The default is designed to create a truly verbatim buffer via \detokenize.

\FV@bufferer
\define@key{FV}{bufferer}{\let\FV@bufferer=#1\relax}
\def\FancyVerbDefaultBufferer#1{%
  \let\FV@bufferer=\relax
  \expandafter\xdef\csname fancyverb@line:arabic\endcsname{\endcsname{#1}}
  \def\FV@bufferer={\FancyVerbDefaultBufferer}}
\fvset{bufferer=\FancyVerbDefaultBufferer}

bufferlengthname Name of counter storing the length of the buffer.

\FV@bufferlengthname
\define@key{FV}{bufferlengthname}{\@ifcsname\csname#1\endcsname\fi}
\ifcsname c@\endcsname
\else
\newcounter{#1}
\fi
\def/FV@bufferlengthname{#1}\set{bufferlengthname=FancyVerbBufferLength}

\textbf{bufferlinename} Base name of buffer line macros. This is given a \FancyVerb* macro name since it may be accessed by the user in defining custom \texttt{bufferer}.
\define@key{FV}{bufferlinename}{\def/FancyVerbBufferLineName{#1}}\set{bufferlinename=FancyVerbBufferLine}

\textbf{buffername} Shortcut for setting \texttt{bufferlengthname} and \texttt{bufferlinename}.
\define@key{FV}{buffername}{\fvset{bufferlengthname=#1length,bufferlinename=#1line}}

\texttt{globalbuffer} Whether buffer macros and the buffer length counter are defined globally.
\newbool{FV@globalbuffer} \define@booleankey{FV}{globalbuffer}{{\booltrue{FV@globalbuffer}}}{{\boolfalse{FV@globalbuffer}}}\fvset{globalbuffer=false}

\texttt{VerbatimBuffer} The environment implementation follows standard \texttt{fancyvrb} environment style.
A special buffer counter is used to track line numbers while avoiding incrementing the regular counter that is used for typeset code. Some macros do nothing with the default \texttt{bufferer}, but are needed to enable \texttt{fancyvrb} options when a custom \texttt{bufferer} is used in conjunction with optional environment arguments. These include \FancyVerbDefineActive and \FancyVerbFormatCom. Since counters are global, the exact location of the \texttt{\setcounter} commands at the end of the environment relative to \texttt{\begingroup...\endgroup} is not important.

\newcounter{FancyVerbBufferLine}\newcounter{FV@oldbufferlength}\newbool{FV@globalbuffer@tmp} \let/FV@bufferlengthname@tmp\relax \let/FancyVerbBufferLineName@tmp\relax \let/FV@afterbuffer@tmp\relax \setcounter{FancyVerbBufferLine}{\expandafter\value\expandafter{\FV@bufferlengthname}} \let@c@FancyVerbLine@c@FancyVerbBufferLine \setcounter{FancyVerbBufferIndex}{\expandafter\value\expandafter{\FV@bufferlengthname}} \ifbool{FV@globalbuffer@tmp}{\global\booltrue{FV@globalbuffer@tmp}}{\global\boolfalse{FV@globalbuffer@tmp}\setcounter{FV@oldbufferlength}{\expandafter\value\expandafter{\FV@bufferlengthname}}} \setcounter{FV@oldbufferlength}{\expandafter\value\expandafter{\FV@oldbufferlengthname}}

\begin{verbatim}
\newbool{FV@globalbuffer}
\define@booleankey{FV}{globalbuffer}{{\booltrue{FV@globalbuffer}}}{{\boolfalse{FV@globalbuffer}}}
\fvset{globalbuffer=false}
\end{verbatim}

\begin{verbatim}
\newcounter{FancyVerbBufferLine}\newcounter{FV@oldbufferlength}\newbool{FV@globalbuffer@tmp} \let/FV@bufferlengthname@tmp\relax \let/FancyVerbBufferLineName@tmp\relax \let/FV@afterbuffer@tmp\relax \setcounter{FancyVerbBufferLine}{\expandafter\value\expandafter{\FV@bufferlengthname}} \let@c@FancyVerbLine@c@FancyVerbBufferLine \setcounter{FancyVerbBufferIndex}{\expandafter\value\expandafter{\FV@bufferlengthname}} \ifbool{FV@globalbuffer@tmp}{\global\booltrue{FV@globalbuffer@tmp}}{\global\boolfalse{FV@globalbuffer@tmp}\setcounter{FV@oldbufferlength}{\expandafter\value\expandafter{\FV@bufferlengthname}}} \setcounter{FV@oldbufferlength}{\expandafter\value\expandafter{\FV@oldbufferlengthname}}
\end{verbatim}
\VerbatimInsertBuffer \textbf{This inserts an existing buffer created by VerbatimBuffer as a Verbatim environment. It customizes Verbatim internals to function with a buffer in a command context.}

\newcommand{\VerbatimInsertBuffer}[1][]{\begingroup
\def\FV@KeyValues{#1}
\def\FV@Scan{\FV@CatCodes
\xdef\FV@EnvironName{Verbatim}\ifnum\expandafter\value\expandafter{\FV@bufferlengthname}=\z@\relax
\PackageError{fvextra}{Buffer length counter \FV@bufferlengthname\space is invalid or zero}{\relax
\fi
\let\FV@GetLine\relax
\let\FVOGetLine\relax
\let\FVOCheckScan\relax
\setcounter{FancyVerbBufferIndex}{1}%
\def\VerbatimClearBuffer\def\FV@Line##1{\% 
\FVExtraRetokenizeVArg{\def\FV@Line}{##1}}
\def\FancyVerbGetLine{% 
\ifnum\value{FancyVerbBufferIndex}>\expandafter\value\expandafter{\FV@bufferlengthname}\relax 
\global\let\FV@EnvironName\relax 
\let\next\relax 
\else 
\ifcsname\FancyVerbBufferLineName\arabic{FancyVerbBufferIndex}\endcsname 
\expandafter\let\expandafter\FV@Line\csname\FancyVerbBufferLineName\arabic{FancyVerbBufferIndex}\endcsname 
\expandafter\VerbatimInsertBuffer\def\FV@Line\expandafter{\FV@Line\next}\% 
\stepcounter{FancyVerbBufferIndex}\% 
\else 
\def\next{% 
\PackageError{fvextra}{Buffer with line macro named \FancyVerbBufferLineName\arabic{FancyVerbBufferIndex} does not exist}{Check bufferlinename, bufferlengthname, and globalbuffer settings}\% 
} \% 
\fi 
\fi \next\%}
\FV@Verbatim 
\FVE@Verbatim 
\setcounter{FancyVerbBufferIndex}{0}%
\endgroup
\@doendpe}

12.9.6 \VerbatimClearBuffer

Clear an existing buffer.
\VerbatimClearBuffer
\newcommand{\VerbatimClearBuffer}{\% 
\begingroup 
\def\FV@KeyValues{#1}\% 
\FV@UseKeyValues 
\setcounter{FancyVerbBufferIndex}{0}\% 
\endgroup}

79
12.10 Patches

12.10.1 Delimiting characters for verbatim commands

Unlike \verb, fancyvrb’s commands like \Verb cannot take arguments delimited by characters like # and % due to the way that starred commands and optional arguments are implemented. The relevant macros are redefined to make this possible.

fancyvrb’s \Verb is actually implemented in \FVCommand which allows versions of commands with customized options:

\FVCommand{(customized_options)}{(base_command_name)}

\Verb is then defined as \def\Verb{\FVCommand{Verb}}. The definition of \FVCommand (and \FVCommand which it uses internally) involves looking ahead for a star * (\@ifstar) and for a left square bracket [ that delimits an optional argument (\@ifnextchar). As a result, the next character is tokenized under the current, normal catcode regime. This prevents \Verb from being able to use delimiting characters like # and % that work with \verb.

\FVCommand and \FVCommand are redefined so that this lookahead tokenizes under a typical verbatim catcode regime (with one exception that is explained below). This enables \verb-style delimiters. This does not account for any custom catcode changes introduced by \fvset, customized commands, or optional arguments. However, delimiting characters should never need custom catcodes, and both the fancyvrb definition of \Verb (when not used inside another macro) as well as the fvextra reimplementation (in all cases) handle the possibility of delimiters with valid but non-typical catcodes. Other, non-verbatim commands that use \FVCommand, such as \UseVerb, are not affected by the patch.

The catcode regime for lookahead has one exception to a typical verbatim catcode regime: The curly braces {} retain their normal codes. This allows the fvextra reimplementation of \Verb to use a pair of curly braces as delimiters, which can be convenient when \Verb is used within another command. Since the original fancyvrb implementation of \Verb with unpatched \FVCommand is incompatible with curly braces being used as delimiters in any form, this does not affect any pre-existing fancyvrb functionality.

\FVCommand

1357 \def\FVCommand#1#2{%
1358 \FVExtra@ifstarVArg
1359 \{\def\FVKeyValues{#1,showspaces,showtabs}\FVCommand{#2}}%
1360 \{\def\FVKeyValues{#1}\FVCommand{#2}}%

\FVCommand

1361 \def\FVCommand#1{%
1362 \FVExtra@ifnextcharVArg%
1363 \{\FVGetKeyValues{\@nameuse{FVC\#1}}%
1364 \{\@nameuse{FVC\#1}}%

12.10.2 \CustomVerbatimCommand compatibility with \FVExtraRobustCommand

\CustomVerbatimCommand #1 is \newcommand or \renewcommand, #2 is the (re)new command, #3 is the base fancyvrb command, #4 is options.
12.10.3 Visible spaces

The default definition of visible spaces (showspaces=true) could allow font commands to escape under some circumstances, depending on how it is used:

%{
\catcode`\ =12
\gdef\FancyVerbSpace{\tt}
}%

\textvisiblespace is not an alternative because it does not have the correct width. The redefinition follows https://tex.stackexchange.com/a/120231/10742.

12.10.4 obeytabs with visible tabs and with tabs inside macro arguments

\FV$\texttt{TrueTab}$ governs tab appearance when obeytabs=true and showtabs=true. It is redefined so that symbols with flexible width, such as $\texttt{\rightarrowfill}$, will work as expected. In the original fancyverb definition, $\verb|\kern@tempdim\hbox to\z|$. The $\verb|\kern|$ is removed and instead the $\verb|\hbox|$ is given the width $\verb|\@tempdim|$. $\verb|\FV$\texttt{TrueTab}$ and related macros are also modified so that they function for tabs inside macro arguments when obeytabs=true (inside curly braces {}) with their normal meaning, when using commandchars, etc.). The fancyverb implementation of tab expansion assumes that tabs are never inside a group; when a group that contains a tab is present, the entire line typically vanishes. The new implementation keeps the fancyverb behavior exactly for tabs outside groups; they are perfectly expanded to tab stops. Tabs inside groups cannot be perfectly expanded to tab stops, at least not using the fancyverb approach. Instead, when fvextra encounters a
run of whitespace characters (tabs and possibly spaces), it makes the assumption that the nearest tab stop was at the beginning of the run. This gives the correct behavior if the whitespace characters are leading indentation that happens to be within a macro. Otherwise, it will typically not give correct tab expansion—but at least the entire line will not be discarded, and the run of whitespace will be represented, even if imperfectly.

A general solution to tab expansion may be possible, but will almost certainly require multiple compiles, perhaps even one compile (or more) per tab. The `zref` package provides a \`zsaveposx` macro that stores the current x position on the page for subsequent compiles. This macro, or a similar macro from another package, could be used to establish a reference point at the beginning of each line. Then each run of whitespace that contains a tab could have a reference point established at its start, and tabs could be expanded based on the distance between the start of the run and the start of the line. Such an approach would allow the first run of whitespace to measure its distance from the start of the line on the 2nd compile (once both reference points were established), so it would be able expand the first run of whitespace correctly on the 3rd compile. That would allow a second run of whitespace to definitely establish its starting point on the 3rd compile, which would allow it to expand correctly on the 4th compile. And so on. Thus, while it should be possible to perform completely correct tab expansion with such an approach, it will in general require at least 4 compiles to do better than the current approach. Furthermore, the sketch of the algorithm provided so far does not include any complications introduced by line breaking. In the current approach, it is necessary to determine how each tab would be expanded in the absence of line breaking, save all tab widths, and then expand using saved widths during the actual typesetting with line breaking.

\begin{verbatim}
\newcounter{FV@TrueTabGroupLevel}
\FV@@ObeyTabs
\end{verbatim}

The `fancyvrb` macro responsible for tab expansion is modified so that it can handle tabs inside groups, even if imperfectly. We need to use a special version of the space, \`FV@Space@ObeyTabs`, that within a group will capture all following spaces or tabs and then insert them with tab expansion based on the beginning of the run of whitespace. We need to record the current group level, but then increment it by 1 because all comparisons will be performed within the \`hbox{...}`. The \`FV@TmpCurrentGroupLevel` is needed for compatibility with the `calc` package, which redefines \`setcounter`.}

\begin{verbatim}
\def\FV@ObeyTabs#1{\let\FV@Space@Orig\FV@Space
\let\FV@Space\FV@Space@ObeyTabs
\edef\FV@TmpCurrentGroupLevel{\the\currentgrouplevel}\
\setcounter{FV@TrueTabGroupLevel}{\FV@TmpCurrentGroupLevel}\
\addtocounter{FV@TrueTabGroupLevel}{1}\
\setbox\FV@TabBox=\hbox{#1}\box\FV@TabBox
\let\FV@Space\FV@Space@Orig}
\end{verbatim}

\FV@TrueTab Version that follows `fancyvrb` if not in a group and takes another approach otherwise.
When linebreaking is in use, the `fancyvrb` tab expansion algorithm cannot be used directly, since it involves `\hbox`, which doesn’t allow for line breaks. In those cases, tab widths will be calculated for the case without breaks and saved, and then saved widths will be used in the actual typesetting. This macro is `\let` to width-saving code in those cases.

\FV@TrueTabCounter

Counter for tracking saved tabs.

\FV@TrueTabSaveWidth

Save the current tab width, then increment the tab counter. `\@tempdima` will hold the current tab width.

\FV@TrueTab@NoGroup

This follows the `fancyvrb` approach exactly, except for the `\hbox` to `\@tempdima` adjustment and the addition of `\FV@TrueTabSaveWidth`.

\FV@ObeyTabs@Whitespace@Tab

In a group where runs of whitespace characters are collected, we need to keep track of whether a tab has been found, so we can avoid expansion and the associated `\hbox` for spaces without tabs.

\FV@TrueTab@Group

If in a group, a tab should start collecting whitespace characters for later tab expansion, beginning with itself. The collected whitespace will use `\FV@FVTabToken` and `\FV@FVSpaceToken` so that any `\ifx` comparisons performed later will behave as expected. This shouldn’t be strictly necessary, because `\FancyVerbBreakStart` operates with saved tab widths rather than using the tab expansion code directly. But it is safer in case any other unanticipated scanning is going on.
Space treatment, like tab treatment, now depends on whether we are in a group, because in a group we want to collect all runs of whitespace and then expand any tabs.

```
\def\FV@Space@ObeyTabs{%
  \ifnum\value{FV@TrueTabGroupLevel}=\the\currentgrouplevel\relax
    \expandafter\FV@Space@ObeyTabs@NoGroup
  \else
    \expandafter\FV@Space@ObeyTabs@Group
  \fi}
\FV@Space@ObeyTabs@NoGroup
Fall back to normal space.
\def\FV@Space@ObeyTabs@NoGroup{\FV@Space@Orig}
\FV@Space@ObeyTabs@Group
Make a note that no tabs have yet been encountered, store the current space, then scan for following whitespace.
\def\FV@Space@ObeyTabs@Group{%
  \boolfalse{FV@ObeyTabs@Whitespace@Tab}%
  \gdef\FV@TmpWhitespace{\FV@FVSpaceToken}%
  \FV@ObeyTabs@ScanWhitespace
\FV@ObeyTabs@ScanWhitespace
Collect whitespace until the end of the run, then process it. Proper lookahead comparison requires \FV@FVSpaceToken and \FV@FVTabToken.
\def\FV@ObeyTabs@ScanWhitespace{%
  \@ifnextchar\FV@FVSpaceToken{%\FV@TrueTab@CaptureWhitespace@Space}{
    \ifx\@let@token\FV@FVTabToken
      \expandafter\FV@TrueTab@CaptureWhitespace@Tab
    \else
      \expandafter\FV@ObeyTabs@ResolveWhitespace
    \fi}
\FV@TrueTab@CaptureWhitespace@Space
\def\FV@TrueTab@CaptureWhitespace@Space#1{%
  \g@addto@macro\FV@TmpWhitespace{\FV@FVSpaceToken}%
  \FV@ObeyTabs@ScanWhitespace
\FV@TrueTab@CaptureWhitespace@Tab
\def\FV@TrueTab@CaptureWhitespace@Tab#1{%
  \booltrue{FV@ObeyTabs@Whitespace@Tab}%
  \g@addto@macro\FV@TmpWhitespace{\FV@FVTabToken}%
  \FV@ObeyTabs@ScanWhitespace
\FV@ObeyTabs@ResolveWhitespace
Yet another tab definition, this one for use in the actual expansion of tabs in whitespace. This uses the \texttt{fancyverb} algorithm, but only over a restricted region known to contain no groups.
\newbox\FV@TabBox@Group
\def\FV@TrueTab@Group@Expand{%
  \egroup
  \@tempdima=\FV@ObeyTabSize sp\relax
  \@tempcnta=\wd\FV@TabBox@Group
  \advance\@tempcnta\FV@@ObeyTabSize\relax
  \divide\@tempdima\@tempcnta
  \multiply\@tempdima\@tempcnta
  \advance\@tempdima-\wd\FV@TabBox@Group
  \FV@TrueTabSaveWidth
  \setbox\FV@TabBox@Group\hbox\bgroup
  \unhbox\FV@TabBox@Group\hbox to\@tempdima{\hss\FV@TabChar}}
\FV@ObeyTabs@ResolveWhitespace Need to make sure the right definitions of the space and tab are in play here. Only do tab expansion, with the associated \hbox, if a tab is indeed present.

\begin{verbatim}
1466 \def\FV@ObeyTabs@ResolveWhitespace{%
1467 \let\FV@Space\FV@Space@Orig
1468 \let\FV@Tab\FV@TrueTab@Group@Expand
1469 \expandafter\FV@ObeyTabs@ResolveWhitespace@i\expandafter{\FV@TmpWhitespace}%
1470 \let\FV@Space\FV@Space@ObeyTabs
1471 \let\FV@Tab\FV@TrueTab}
1472 \def\FV@ObeyTabs@ResolveWhitespace@i#1{%
1473 \ifbool{FV@ObeyTabs@Whitespace@Tab}\
1474 {\setbox\FV@TabBox@Group=\hbox{#1}\box\FV@TabBox@Group}%
1475 \{#1\}}
\end{verbatim}

12.10.5 Spacing in math mode

\FancyVerbMathSpace \FV@Space is defined as either a non-breaking space or a visible representation of a space, depending on the option showspaces. Neither option is desirable when typeset math is included within verbatim content, because spaces will not be discarded as in normal math mode. Define a space for math mode.

\begin{verbatim}
1476 \def\FancyVerbMathSpace{ }
\end{verbatim}

\FV@SetupMathSpace Define a macro that will activate math spaces, then add it to an \fextra hook.

\begin{verbatim}
1477 \def\FV@SetupMathSpace{%
1478 \everymath\expandafter{\the\everymath\let\FV@Space\FancyVerbMathSpace}}
1479 \g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathSpace}
\end{verbatim}

12.10.6 Fonts and symbols in math mode

The single quote (’) does not become ‘\prime when typeset math is included within verbatim content, due to the definition of the character in \@noligs. This patch adds a new definition of the character in math mode, inspired by http://tex.stackexchange.com/q/223876/10742. It also redefines other characters in \@noligs to behave normally within math mode and switches the default font within math mode, so that amsmath’s \text will work as expected.

\FV@pr@m@s Define a version of \pr@m@s from \latex\.txt that works with active ‘. In verbatim contexts, ‘ is made active by \@noligs.

\begin{verbatim}
1480 \begingroup
1481 \catcode`\`\active
1482 \catcode`\^=7
1483 \edef\FV@pr@m@s{%
1484 \ifx\@let@token\expandafter\pr@@@s
1485 \else\pr@@@t
1486 \fi
1487 \egroup}
1488 \expandafter\expandafter\expandafter\pr@@@t
1489 \expandafter\expandafter\expandafter\pr@@@s
1490 \else
1491 \fi
1492 \fi}
1493 \endgroup
\end{verbatim}
\FV@SetupMathFont Set the font back to default from the verbatim font.
\FV@SetupMathFont{\everymath\expandafter{\the\everymath\fontfamily{\familydefault}\selectfont}}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathFont}

\FV@SetupMathLigs Make all characters in \@noligsshould behave normally, and switch to \FV@pr@m@s. The relevant definition from \texttt{latex.ltx}:
\verbatim@nolig@list{\do`\do<\do>\do,\do\do-}
\FV@SetupMathLigs{\everymath\expandafter{\the\everymath\let\pr@m@s\FV@pr@m@s
\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{\ifmmode\active@math@prime\else`i}\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{`}\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{<}\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{>}\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{,}\begingroup\lccode`\do`\do`\lowercase{\endgroup\def\do`}{-}}\}}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathLigs}

12.10.7 Ophaned label
\FV@BeginListFrame@Lines When \texttt{frame=lines} is used with a label, the label can be orphaned. This overwrites the default definition to add \texttt{\penalty\@M}. The fix is attributed to \url{http://tex.stackexchange.com/a/168021/10742}.
\FV@BeginListFrame@Lines{\begingroup\lineskip\z@skip\FV@SingleFrameLine{\z@}\kern-0.5\baselineskip\relax\baselineskip\z@skip\kern\FV@FrameSep\relax\penalty\@M\endgroup}

12.10.8 rulecolor and fillcolor
The \texttt{rulecolor} and \texttt{fillcolor} options are redefined so that they accept color names directly, rather than requiring \texttt{\color\{\textit{color_name}\}}. The definitions still allow the old usage.
\define@key{FV}{rulecolor}{\ifstrequal{#1}{none}{\let\FancyVerbRuleColor\relax}{\let\FancyVerbRuleColor\relax}{\def\@tempa{#1}}{\ifstrequal{#1}{none}{\let\FancyVerbRuleColor\relax}}{\def\@tempa{#1}}}
12.11 Extensions

12.11.1 New options requiring minimal implementation

linenos fancyvrb allows line numbers via the options numbers=left and numbers=right. This creates a linenos key that is essentially an alias for numbers=left.

\define@booleankey{FV}{linenos}{\@nameuse{FV@Numbers@left}}{\@nameuse{FV@Numbers@none}}

Redefine \FancyVerbTab.

\define@key{FV}{tab}{\def{\FancyVerbTab{#1}}}

\define@key{FV}{tabcolor}{\ifstrempty{#1}{\let{\FV@TabColor}{\relax}}{\ifstrequal{#1}{none}{\let{\FV@TabColor}{\relax}}{\def{\FV@TabColor{\textcolor{#1}}}}}}

\define@booleankey{FV}{showtabs}{\def{\FV@TabChar{\FV@TabColor{\FancyVerbTab}}}}{\let{\FV@TabChar}{\relax}}

\fvset{tabcolor=none, showtabs=false}

shospaces Reimplement shospaces with a bool to work with new space options.
FV@shospaces
\newbool{FV@showspaces}
\define@booleankey{FV}{showspaces}\
{\booltrue{FV@showspaces}}\
{\boolefalse{FV@showspaces}}
\fvset{showspaces=false}

space Redefine \FancyVerbSpace, which is the visible space.
\define@key{FV}{space}{\def\FancyVerbSpace{#1}}
\fvset{spacecolor=none}

spacecolor Set space color, or allow it to adjust to surroundings (the default fancyvrb behavior).
This involves re-creating the showspaces option to add \FV@SpaceColor.
\define@key{FV}{spacecolor}\
{\ifstrempty{#1}\
{\let\FV@SpaceColor\relax}\
{\ifstrequal{#1}{none}\
{\let\FV@SpaceColor\relax}\
{\def\FV@SpaceColor{\textcolor{#1}}}}}
\fvset{spacecolor=none}

spacebreak Line break for spaces that is inserted when spaces are visible (showspaces=true) or
\FancyVerbSpaceBreak when breaks around spaces are handled specially (breakcollapsespaces=false).
Not used for regular spaces under default conditions.
\define@key{FV}{spacebreak}{\def\FancyVerbSpaceBreak{#1}}
\fvset{spacebreak=\discretionary{}{}{}}

breakcollapsespaces When a line break occurs within a sequence of regular space characters
\FV@DefFVSpace (showspaces=false), collapse the spaces into a single space and then replace
it with the break. When this is true, a sequence of spaces will cause at most
a single line break, and the first character on the wrapped line after the break
will be a non-space character. When this is false, a sequence of spaces may
result in multiple line breaks. Each wrapped line besides the last will contain only
spaces. The final wrapped line may contain leading spaces before any non-space
character(s).
\newbool{FV@breakcollapsespaces}
\define@booleankey{FV}{breakcollapsespaces}\
{\booltrue{FV@breakcollapsespaces}}\
{\boolefalse{FV@breakcollapsespaces}}
\fvset{breakcollapsespaces=true}

\FV@DefFVSpace Redefine \FV@Space based on fextra options that affect spaces.
This must be added to \FV@FormattingPrep@PreHook, but only after breakbefore
and breakafter macros are defined. Hence the \AtEndOfPackage.
\def\FV@DefFVSpace{\
{\ifboolexpr{\booland{\FV@showspaces}{\breakcollapsespaces}}\
{\ifcsname FV@BreakBefore@Token\FV@SpaceCatTen\endcsname\
{\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}}}\
{\else\ifcsname FV@BreakAfter@Token\FV@SpaceCatTen\endcsname\
{\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}}}\
{\else\
{\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}\FancyVerbSpaceBreak}}}}}\
{\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}}}\
{
\FV@DefFVSpaceBreak}%

88
mathescape Give $, &, ^, and _ their normal catcodes to allow normal typeset math.

\define@booleankey{FV}{mathescape}\
{\let\FancyVerbMathEscape\FV@MathEscape}\
{\let\FancyVerbMathEscape\relax}\
\def\FV@MathEscape{\catcode`$=3\catcode`&=4\catcode`\^=7\catcode`_=8\relax}\
\FV@AddToHook\FV@FormattingPrep@PreHook\FancyVerbMathEscape\
\fvset{mathescape=false}

beameroverlays Give < and > their normal catcodes (not \active), so that beamer overlays will work. This modifies \@noligs because that is the only way to prevent the settings from being overwritten later. This could have used \FV@CatCodesHook, but then it would have had to compare \@noligs to \relax to avoid issues when \let\@noligs\relax in VerbatimOut.

\define@booleankey{FV}{beameroverlays}\
{\let\FancyVerbBeamerOverlays\FV@BeamerOverlays}\
{\let\FancyVerbBeamerOverlays\relax}\
\def\FV@BeamerOverlays{\expandafter\def\expandafter\@noligs\expandafter{\@noligs\begingroup\lccode`~\~\lowercase{\endgroup\def~}{})}}\
\g@addto@macro\FV@FormattingPrep@PreHook\FancyVerbBeamerOverlays\
\fvset{beameroverlays=false}

curlyquotes Let ` and ' produce curly quotation marks ‘ and ’ rather than the backtick and typewriter single quotation mark produced by default via upquote.

\newbool{FV@CurlyQuotes}\
\define@booleankey{FV}{curlyquotes}\
{\booltrue{FV@CurlyQuotes}}\
{\boolfalse{FV@CurlyQuotes}}\
\def\FancyVerbCurlyQuotes{\ifbool{FV@CurlyQuotes}{\expandafter\def\expandafter\@noligs\expandafter{\@noligs\begingroup\lccode`~\~\lowercase{\endgroup\def~}{}}}{}}\
\g@addto@macro\FV@FormattingPrep@PreHook\FancyVerbCurlyQuotes\
\fvset{curlyquotes=false}
fontencoding Add option for font encoding.

Add option for font encoding.

\define@key{FV}{fontencoding}{
  \ifstrempty{#1}{\relax}
  \ifstrequal{#1}{none}{\relax}{\fontencoding{#1}}}
\expandafter\def\expandafter\FV@SetupFont\expandafter{\FV@FontEncoding\FV@SetupFont}
\fvset{fontencoding=none}

12.11.2 Formatting with \FancyVerbFormatLine, \FancyVerbFormatText, and \FancyVerbHighlightLine

\texttt{fancyvrb} defines \texttt{\FancyVerbFormatLine}, which defines the formatting for each line. The introduction of line breaks introduces an issue for \texttt{\FancyVerbFormatLine}. Does it format the entire line, including any whitespace in the margins or behind line break symbols (that is, is it outside the \texttt{\parbox} in which the entire line is wrapped when breaking is active)? Or does it only format the text part of the line, only affecting the actual characters (inside the \texttt{\parbox})? Since both might be desirable, \texttt{\FancyVerbFormatLine} is assigned to the entire line, and a new macro \texttt{\FancyVerbFormatText} is assigned to the text, within the \texttt{\parbox}.

An additional complication is that the \texttt{fancyvrb} documentation says that the default value is \texttt{\def\FancyVerbFormatLine#1{#1}}. But the actual default is \texttt{\def\FancyVerbFormatLine#1{\FV@ObeyTabs{#1}}}.

\texttt{\FV@ObeyTabs} needs to operate directly on the line to handle tabs. As a result, all \texttt{fancyvrb} commands that involve \texttt{\FancyVerbFormatLine} are patched, so that \texttt{\def\FancyVerbFormatLine#1{#1}}.

An additional macro \texttt{\FancyVerbHighlightLine} is added between \texttt{\FancyVerbFormatLine} and \texttt{\FancyVerbFormatText}. This is used to highlight selected lines (section 12.11.4). It is inside \texttt{\FancyVerbHighlightLine} so that if \texttt{\FancyVerbHighlightLine} is used to provide a background color, \texttt{\FancyVerbHighlightLine} can override it.

\texttt{\FancyVerbFormatLine} Format the entire line, following the definition given in the \texttt{fancyvrb} documentation. Because this is formatting the entire line, using boxes works with line breaking.

\texttt{\FancyVerbFormatText} Format only the text part of the line. Because this is inside all of the line breaking commands, using boxes here can conflict with line breaking.

\texttt{\FV@ListProcessLine@NoBreak} Redefined \texttt{\FV@ListProcessLine} in which \texttt{\FancyVerbFormatText} is added and tab handling is explicit. The \texttt{@NoBreak} suffix is added because \texttt{\FV@ListProcessLine} will be \texttt{\let} to either this macro or to \texttt{\FV@ListProcessLine@Break} depending on whether line breaking is enabled.

\texttt{\FV@ListProcessLine@NoBreak} Redefined \texttt{\FV@ListProcessLine} in which \texttt{\FancyVerbFormatText} is added and tab handling is explicit. The \texttt{@NoBreak} suffix is added because \texttt{\FV@ListProcessLine} will be \texttt{\let} to either this macro or to \texttt{\FV@ListProcessLine@Break} depending on whether line breaking is enabled.
12.11.3 Line numbering

Add several new line numbering options. \texttt{numberfirstline} always numbers the first line, regardless of \texttt{stepnumber}. \texttt{stepnumberfromfirst} numbers the first line, and then every line that differs from its number by a multiple of \texttt{stepnumber}. \texttt{stepnumberoffsetvalues} determines whether line number are always an exact multiple of \texttt{stepnumber} (the new default behavior) or whether there is an offset when \texttt{firstnumber} \neq 1 (the old default behavior). A new option \texttt{numbers=both} is created to allow line numbers on both left and right simultaneously.

\fvset{numberfirstline=false}
\fvset{stepnumberfromfirst=false}
\fvset{stepnumberoffsetvalues=false}
Redefine `fancyvrb` macro to account for `numberfirstline`, `stepnumberfromfirst`, and `stepnumberoffsetvalues`. The `\let\FancyVerbStartNum\@ne` is needed to account for the case where `firstline` is never set, and defaults to zero (`\z@`).

```latex
\FV@Numbers@left
\def\FV@Numbers@left{%  
\let\FV@RightListNumber\relax
\def\FV@LeftListNumber{%  
\ifx\FancyVerbStartNum\z@
  \let\FancyVerbStartNum\@ne
\fi
\ifbool{FV@StepNumberFromFirst}{%  
  \@tempcnta=\FV@CodeLineNo
  \@tempcntb=\FancyVerbStartNum
  \advance\@tempcntb\FV@StepNumber
  \divide\@tempcntb\FV@StepNumber
  \multiply\@tempcntb\FV@StepNumber
  \advance\@tempcnta\@tempcntb
  \advance\@tempcnta-\FancyVerbStartNum
  \@tempcntb=\@tempcnta
  \ifbool{FV@StepNumberOffsetValues}{%  
    \@tempcnta=\FV@CodeLineNo
    \@tempcntb=\FV@CodeLineNo
    \ifbool{FV@NumberFirstLine}{%  
      \ifnum\FV@CodeLineNo=\FancyVerbStartNum
        \hbox to\z@{(\hss\theFancyVerbLine\kern\FV@NumberSep}%
      \fi
    \}{}
  \}{}
}\FV@Numbers@right
\def\FV@Numbers@right{%  
\let\FV@LeftListNumber\relax
\def\FV@RightListNumber{%  
\ifx\FancyVerbStartNum\z@
  \let\FancyVerbStartNum\@ne
\fi
\ifbool{FV@StepNumberFromFirst}{%  
  \@tempcnta=\FV@CodeLineNo
  \@tempcntb=\FancyVerbStartNum
  \advance\@tempcntb\FV@StepNumber
  \divide\@tempcntb\FV@StepNumber
  \multiply\@tempcntb\FV@StepNumber
  \ifnum\@tempcnta=\@tempcntb
    \if@FV@NumberBlankLines
      \hbox to\z@{(\hss\theFancyVerbLine\kern\FV@NumberSep}%
    \else
      \@tempcntb=\@tempcnta
      \ifbool{FV@StepNumberOffsetValues}{%  
        \@tempcnta=\FV@CodeLineNo
        \ifbool{FV@NumberFirstLine}{%  
          \ifnum\FV@CodeLineNo=\FancyVerbStartNum
            \hbox to\z@{(\hss\theFancyVerbLine\kern\FV@NumberSep}%
          \fi
        \}{}
      \}{}
    \}{}
\}{}
```

\FV@Numbers@right  Redefine `fancyvrb` macro to account for `numberfirstline`, `stepnumberfromfirst`, and `stepnumberoffsetvalues`. The `\let\FancyVerbStartNum\@ne` is needed to account for the case where `firstline` is never set, and defaults to zero (`\z@`).

```latex
\def\FV@Numbers@right{%  
\let\FV@LeftListNumber\relax
\def\FV@RightListNumber{%  
\ifx\FancyVerbStartNum\z@
  \let\FancyVerbStartNum\@ne
\fi
\ifbool{FV@StepNumberFromFirst}{%  
  \@tempcnta=\FV@CodeLineNo
  \@tempcntb=\FancyVerbStartNum
  \advance\@tempcntb\FV@StepNumber
  \divide\@tempcntb\FV@StepNumber
  \multiply\@tempcntb\FV@StepNumber
  \ifnum\@tempcnta=\@tempcntb
    \if@FV@NumberBlankLines
      \hbox to\z@{(\hss\theFancyVerbLine\kern\FV@NumberSep}%
    \else
      \@tempcntb=\@tempcnta
      \ifbool{FV@StepNumberOffsetValues}{%  
        \@tempcnta=\FV@CodeLineNo
        \ifbool{FV@NumberFirstLine}{%  
          \ifnum\FV@CodeLineNo=\FancyVerbStartNum
            \hbox to\z@{(\hss\theFancyVerbLine\kern\FV@NumberSep}%
          \fi
        \}{}
      \}{}
    \}{}
```

92
\divide\@tempcntb\FV@StepNumber
\multiply\@tempcntb\FV@StepNumber
\advance\@tempcnta\@tempcntb
\advance\@tempcnta-\FancyVerbStartNum
\@tempcntb=\@tempcnta}\
{\ifbool{FV@StepNumberOffsetValues}\
{\@tempcnta=\FV@CodeLineNo\
\@tempcntb=\FV@CodeLineNo}\
{\@tempcnta=c\@FancyVerbLine\
\@tempcntb=c\@FancyVerbLine}}\
\divide\@tempcntb\FV@StepNumber
\multiply\@tempcntb\FV@StepNumber
\ifnum\@tempcnta=\@tempcntb\
{\if@FV@NumberBlankLines\hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}\%\
\else\%\
\fi\%\
\else\%\
\if\FV@Line\empty\%\
\else\%\
\fi\%\
\fi}\
\else\%\
\ifbool{FV@NumberFirstLine}{{%\
\ifnum\FV@CodeLineNo=\FancyVerbStartNum\%\
\hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}\%\
\fi}{}%\
\fi}\
\fi\%
\FV@Numbers@both

Define a new macro to allow numbers=both. This copies the definitions of \FV@LeftListNumber and \FV@RightListNumber from \FV@Numbers@left and \FV@Numbers@right, without the \relax's.
\def\FV@Numbers@both{%\
\def\FV@LeftListNumber{%\
\ifx\FancyVerbStartNum\z@\%\
\let\FancyVerbStartNum\one\%\
\fi\%\
\ifbool{FV@StepNumberFromFirst}%\
{\@tempcnta=\FV@CodeLineNo\%\
\@tempcntb=\FancyVerbStartNum\%\
\advance\@tempcntb\FV@StepNumber\%\
\divide\@tempcntb\FV@StepNumber\%\
\multiply\@tempcntb\FV@StepNumber\%\
\advance\@tempcnta-\FancyVerbStartNum\%\
\@tempcntb=\@tempcnta}}\%
\ifbool{FV@StepNumberOffsetValues}\
{\@tempcnta=\FV@CodeLineNo\%\
\@tempcntb=\FV@CodeLineNo}\
{\@tempcnta=c\@FancyVerbLine\
\@tempcntb=c\@FancyVerbLine}}\%
\divide\@tempcntb\FV@StepNumber\%\
\multiply\@tempcntb\FV@StepNumber\%\
\ifnum\@tempcnta=\@tempcntb\%}
\if@FV@NumberBlankLines
  \hbox to\z@{\hss theFancyVerbLine \kern \FV@NumberSep}\% \\
\else \\
  \fx\FV@Line\empty \\
\else \\
  \hbox to\z@{\hss theFancyVerbLine \kern \FV@NumberSep}\% \\
\fi \\
\fi \\
\else \\
  \ifbool{FV@NumberFirstLine}{% \\
    \ifnum \FV@CodeLineNo = \FancyVerbStartNum \\
      \hbox to\z@{\hss theFancyVerbLine \kern \FV@NumberSep}\% \\
    \fi}{}% \\
  \fi \% \\
  \def\FV@RightListNumber{% \\
  \ifbool{FV@NumberFirstLine}{% \\
    \ifnum \FV@CodeLineNo = \FancyVerbStartNum \\
      \hbox to\z@{\hss theFancyVerbLine \kern \FV@NumberSep}\% \\
    \fi}{}% \\
  \fi \% \\
  \let \FancyVerbStartNum \@ne \\
  \ifbool{FV@StepNumberFromFirst}{% \\
    \@tempcnta = \FV@CodeLineNo \\
    \@tempcntb = \FancyVerbStartNum \\
    \advance \@tempcntb \FV@StepNumber \\
    \divide \@tempcntb \FV@StepNumber \\
    \multiply \@tempcntb \FV@StepNumber \\
    \advance \@tempcnta \@tempcntb \\
    \advance \@tempcnta - \FancyVerbStartNum \\
    \@tempcntb = \@tempcnta}% \\
  \else \\
    \ifbool{FV@StepNumberOffsetValues}{% \\
      \@tempcnta = \FV@CodeLineNo \\
      \@tempcntb = \FV@CodeLineNo}% \\
    \else \\
      \@tempcnta = \c@FancyVerbLine \\
      \@tempcntb = \c@FancyVerbLine}% \\
    \fi \\
    \divide \@tempcntb \FV@StepNumber \\
    \multiply \@tempcntb \FV@StepNumber \\
    \ifnum \@tempcnta = \@tempcntb \\
      \if@FV@NumberBlankLines \\
        \hbox to\z@{\kern \FV@NumberSep \theFancyVerbLine \hss}\% \\
      \else \\
        \fx\FV@Line\empty \\
      \else \\
        \hbox to\z@{\kern \FV@NumberSep \theFancyVerbLine \hss}\% \\
      \fi \\
    \fi \\
    \else \\
    \fi \\
  \fi \\
  \fi \\
  \fi \\
\}
12.11.4 Line highlighting or emphasis

This adds an option `highlightlines` that allows specific lines, or lines within a range, to be highlighted or otherwise emphasized.

highlightlines
\FV@HighlightLinesList
\definekey{FV}{highlightlines}{\def\FV@HighlightLinesList{#1}}%
\fsset{highlightlines=}

highlightcolor
Define color for highlighting. The default is LightCyan. A good alternative for a brighter color would be LemonChiffon.
\definekey{FV}{highlightcolor}{\def\FancyVerbHighlightColor{#1}}%
\let\FancyVerbHighlightColor\@empty
\ifcsname definecolor\endcsname
\ifx\definecolor\relax
\definecolor{FancyVerbHighlightColor}{rgb}{0.878, 1, 1}
\fsset{highlightcolor=FancyVerbHighlightColor}
\fi\fi
\AtBeginDocument{%
\ifx\FancyVerbHighlightColor\@empty
\ifcsname definecolor\endcsname
\ifx\definecolor\relax
\definecolor{FancyVerbHighlightColor}{rgb}{0.878, 1, 1}
\fsset{highlightcolor=FancyVerbHighlightColor}
\fi\fi
\fi}

\FancyVerbHighlightLine
This is the entry macro into line highlighting. By default it should do nothing. It is always invoked between \FancyVerbFormatLine and \FancyVerbFormatText, so that it can provide a background color (won’t interfere with line breaking) and can override any formatting provided by \FancyVerbFormatLine. It is \let to \FV@HighlightLine when highlighting is active.
\def\FancyVerbHighlightLine#1{#1}

\FV@HighlightLine
This determines whether highlighting should be performed, and if so, which macro should be invoked.
\def\FV@HighlightLine#1{%
\@tempcnta=\c@FancyVerbLine
\@tempcntb=\c@FancyVerbLine
\ifcsname FV@HighlightLine:\number\@tempcnta\endcsname
\advance\@tempcntb\m@ne
\ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
\let\FV@HighlightLine@Next=FancyVerbHighlightLineMiddle
\else
\let\FV@HighlightLine@Next=FancyVerbHighlightLineLast
\fi
\fi\fi
A normal line that is not highlighted or otherwise emphasized. This could be redefined to de-emphasize the line.

\def\FancyVerbHighlightLineNormal#1{#1}

\FV@TmpLength
\newlength{\FV@TmpLength}

\FancyVerbHighlightLineFirst The first line in a multi-line range.

\fboxsep is set to zero so as to avoid indenting the line or changing inter-line spacing. It is restored to its original value inside to prevent any undesired effects. The \strut is needed to get the highlighting to be the appropriate height. The \rlap and \hspace make the \colorbox expand to the full \linewidth. Note that if \fboxsep \neq 0, then we would want to use \dimexpr\linewidth-2\fboxsep or add \hspace{-2\fboxsep} at the end.

If this macro is customized so that the text cannot take up the full \linewidth, then adjustments may need to be made here or in the line breaking code to make sure that line breaking takes place at the appropriate location.

\def\FancyVerbHighlightLineFirst#1{%
\setlength{\FV@TmpLength}{\fboxsep}%
\setlength{\fboxsep}{0pt}%
\colorbox{\FancyVerbHighlightColor}{%}
\setlength{\fboxsep}{\FV@TmpLength}%
\rlap{\strut#1}%
\hspace{\linewidth}%
\ifx\FV@RightListFrame\relax\else
\hspace{-\FV@FrameSep}%
\hspace{-\FV@FrameRule}%
\fi
\ifx\FV@LeftListFrame\relax\else
\hspace{-\FV@FrameSep}%
\hspace{-\FV@FrameRule}%
\fi
\hss
}%

\FancyVerbHighlightLineMiddle A middle line in a multi-line range.

\FancyVerbHighlightLineLast The last line in a multi-line range.
12.12 Line breaking

The following code adds automatic line breaking functionality to \texttt{fancyvrb}'s \texttt{Verbatim} environment. Automatic breaks may be inserted after spaces, or before or after specified characters. Breaking before or after specified characters involves scanning each line token by token to insert \texttt{discretionary} at all potential break locations.

12.12.1 Options and associated macros

Begin by defining keys, with associated macros, bools, and dimens.

\FV@SetToWidthNChars Set a dimen to the width of a given number of characters. This is used in setting several indentation-related dimensions.
1922 \newcount\FV@LoopCount
1923 \newbox\FV@NCharsBox
1924 \def\FV@SetToWidthNChars#1#2{%
1925 \FV@LoopCount=#2\relax
1926 \ifnum\FV@LoopCount>0
1927 \def\FV@NChars{}%
1928 \loop
1929 \ifnum\FV@LoopCount>0
1930 \expandafter\def\expandafter\FV@NChars\expandafter{\FV@NChars x}%
1931 \fi
1932 \advance\FV@LoopCount by -1
1933 \ifnum\FV@LoopCount>0
1934 \repeat
1935 \setbox\FV@NCharsBox\hbox{\FV@NChars}%
1936 \ifcase=\wd\FV@NCharsBox
1937 \else
1938 \relax
1939 \fi
1940 }

\FV@breaklines Turn line breaking on or off. The \FV@ListProcessLine from fancyvrb is \let to a (patched) version of the original or a version that supports line breaks.
1941 \newbool{FV@breaklines}
1942 \define@booleankey{FV}{breaklines}{\booltrue{FV@breaklines} %
1943 \let\FV@ListProcessLine\FV@ListProcessLine@Break}%
1944 {\boolfalse{FV@breaklines} %
1945 \let\FV@ListProcessLine\FV@ListProcessLine@NoBreak}
1946 \AtEndOfPackage{\fvset{breaklines=false}}

\FV@BreakLinesLuaTeXHook Fix hyphen handling under LuaTeX. \automatichyphenmode=2 would work for environments, but doesn’t seem to work inline. Instead, the active hyphen is redefined to \mbox{-}.

\FV@BreakLinesIndentationHook A hook for performing on-the-fly indentation calculations when breaklines=true.

This is needed before \@noligs is ever used, so it is placed in \FV@FormattingPrep@PreHook.
1948 \def\FV@BreakLinesLuaTeXHook{%
1949 \expandafter\def\expandafter\@noligs\expandafter{\@noligs
1950 \begingroup\lccode`~=`\-
\lowercase{\endgroup\def~}{\leavevmode\kern\z@\mbox{-}}}%
1951 \ifcsname directlua\endcsname
1952 \ifx\directlua\relax
1953 \else
1954 \FV@AddToHook\FV@FormattingPrep@PreHook\FV@BreakLinesLuaTeXHook
1955 \fi
1956 \fi

\FV@BreakLinesIndentationHook This is used for all \*NChars related indentation. It is important to use \FV@FormattingPrep@PostHook because it is always invoked after any font-related settings.
1957 \def\FV@BreakLinesIndentationHook{}
1958 \g@addto@macro\FV@FormattingPrep@PostHook{%
1959 \if\FV@breaklines
1960 \FV@BreakLinesIndentationHook
1961 \fi}
\FV@BreakIndent\breakindent\newdimen\FV@BreakIndent\newcount\FV@BreakIndentNChars
\define@key{FV}{breakindent}{\FV@BreakIndent=#1\relax}
\define@key{FV}{breakindentnchars}{\FV@BreakIndentNChars=0\relax}
\g@addto@macro\FV@BreakLinesIndentationHook{%\ifnum\FV@BreakIndentNChars>0
\FV@SetToWidthNChars{\FV@BreakIndent}{\FV@BreakIndentNChars}\fi}
\fvset{breakindentnchars=0}
\FV@breakautoindent\breakautoindent\newbool{FV@breakautoindent}
\define@booleankey{FV}{breakautoindent}{\booltrue{FV@breakautoindent}}{\boolfalse{FV@breakautoindent}}
\fvset{breakautoindent=true}
\FancyVerbBreakSymbolLeft\breaksymbolleft\newdimen\FV@BreakSymbolSepLeft\newcount\FV@BreakSymbolSepLeftNChars
\define@key{FV}{breaksymbolsepleft}{\FV@BreakSymbolSepLeft=#1\relax}
\define@key{FV}{breaksymbolsep}{\fvset{breaksymbolsepleft=#1}}
\define@key{FV}{breaksymbolsepleftnchars}{\FV@BreakSymbolSepLeftNChars=#1\relax}
\define@key{FV}{breaksymbolsepnchars}{\fvset{breaksymbolsepleftnchars=#1}}
\g@addto@macro\FV@BreakLinesIndentationHook{%\ifnum\FV@BreakSymbolSepLeftNChars>0
\FV@SetToWidthNChars{\FV@BreakSymbolSepLeft}{\FV@BreakSymbolSepLeftNChars}\fi}
\fvset{breaksymbolsepleftnchars=2}
\FV@BreakSymbolSepRight\breaksymbolright\newdimen\FV@BreakSymbolSepRight\newcount\FV@BreakSymbolSepRightNChars
\define@key{FV}{breaksymbolsepright}{\FV@BreakSymbolSepRight=#1\relax}
\define@key{FV}{breaksymbolsep}{\fvset{breaksymbolsepright=#1}}
\define@key{FV}{breaksymbolseprightnchars}{\FV@BreakSymbolSepRightNChars=#1\relax}
\g@addto@macro\FV@BreakLinesIndentationHook{%\ifnum\FV@BreakSymbolSepRightNChars>0
\FV@SetToWidthNChars{\FV@BreakSymbolSepRight}{\FV@BreakSymbolSepRightNChars}\fi}
\fvset{breaksymbolseprightnchars=2}
We need macros that contain the logic for typesetting the break symbols. By default, the symbol macros contain everything regarding the symbol and its typesetting, while these macros contain pure logic. The symbols should be wrapped in braces so that formatting commands (for example, \texttt{\tiny}) don’t escape.

The left break symbol should only appear with continuation lines. Note that \texttt{linenumber} here refers to local line numbering for the broken line, not line numbering for all lines in the environment being typeset.

We need a counter for keeping track of the local line number for the last segment of a broken line, so that we can avoid putting a right continuation symbol there. A line that is broken will ultimately be processed twice when there is a right continuation symbol, once to determine the local line numbering, and then again for actual insertion into the document.
FV@SetLineBreakLast Store the local line number for the last continuation line.
2033 \newcommand{\FV@SetLineBreakLast}{%
2034 \setcounter{FancyVerbLineBreakLast}{\value{linenumber}}}%

FancyVerbBreakSymbolRightLogic Only insert a right break symbol if not on the last continuation line.
2035 \newcommand{\FancyVerbBreakSymbolRightLogic}{[1]{%}%
2036 \ifnum\value{linenumber}=\value{FancyVerbLineBreakLast}\relax\else{#1}\fi}%

\FancyVerbBreakStart Macro that starts fine-tuned breaking (breakanywhere, breakbefore, breakafter) by examining a line token-by-token. Initially \let to \relax; later \let to \FV@Break as appropriate.
2037 \let\FancyVerbBreakStart\relax

\FancyVerbBreakStop Macro that stops the fine-tuned breaking region started by \FancyVerbBreakStart. Initially \let to \relax; later \let to \FV@EndBreak as appropriate.
2038 \let\FancyVerbBreakStop\relax

\FV@Break@DefaultToken Macro that controls default token handling between \FancyVerbBreakStart and \FancyVerbBreakStop. Initially \let to \FV@Break@NBToken, which does not insert breaks. Later \let to \FV@Break@AnyToken or \FV@Break@BeforeAfterToken if breakanywhere or breakbefore/breakafter are in use.
2039 \let\FV@Break@DefaultToken\FV@Break@NBToken

FV@breakanywhere Allow line breaking (almost) anywhere. Set \FV@Break and \FV@EndBreak to be used, and \let \FV@Break@DefaultToken to the appropriate macro.
2040 \newbool{FV@breakanywhere}
2041 \define@booleankey{FV}{breakanywhere}{}{\booltrue{FV@breakanywhere}}%
2042 \let\FV@BreakBefore\@empty
2043 \let\FV@BreakBefore\relax
2044 \let\FV@BreakBefore\relax
2045 \let\FV@BreakBefore\relax
2046 \let\FV@BreakBefore\relax
2047 \let\FV@BreakBefore\relax
2048 \let\FV@BreakBefore\relax
2049 \let\FV@BreakBefore\relax
2050 \fvset{breakanywhere=false}

\FV@BreakBefore Allow line breaking (almost) anywhere, but only before specified characters.
2051 \define@key{FV}{breakbefore}{}{\ifstrempty{#1}}%
2052 \let\FV@BreakBefore\@empty
2053 \let\FV@BreakBefore\relax
2054 \let\FV@BreakBefore\relax
2055 \let\FV@BreakBefore\relax
2056 \let\FV@BreakBefore\relax
2057 \let\FV@BreakBefore\relax
2058 \let\FV@BreakBefore\relax
2059 \let\FV@BreakBefore\relax
2060 \let\FV@BreakBefore\relax
2061 \fvset{breakbefore={}}
FV@breakbeforeinrun Determine whether breaking before specified characters is always allowed before each individual character, or is only allowed before the first in a run of identical characters.

\newbool{FV@breakbeforeinrun} \define@booleankey{FV}{breakbeforeinrun} \{\booltrue{FV@breakbeforeinrun}\} \{\boolfalse{FV@breakbeforeinrun}\} \fvset{breakbeforeinrun=false}

\FV@BreakBeforePrep We need a way to break before characters if and only if they have been specified as breaking characters. It would be possible to do that via a nested conditional, but that would be messy. It is much simpler to create an empty macro whose name contains the character, and test for the existence of this macro. This needs to be done inside a \begingroup...\endgroup so that the macros do not have to be cleaned up manually. A good place to do this is in \FV@FormattingPrep, which is inside a group and before processing starts. The macro is added to \FV@FormattingPrep@PreHook, which contains \fvextra extensions to \FV@FormattingPrep, after \FV@BreakAfterPrep is defined below.

The procedure here is a bit roundabout. We need to use \FV@EscChars to handle character escapes, but the character redefinitions need to be kept local, requiring that we work within a \begingroup...\endgroup. So we loop through the breaking tokens and assemble a macro that will itself define character macros. Only this defining macro is declared global, and it contains \expanded characters so that there is no longer any dependence on \FV@EscChars.

\FV@BreakBeforePrep@PygmentsHook allows additional break preparation for Pygments-based packages such as \minted and \pythontex. When Pygments highlights code, it converts some characters into macros; they do not appear literally. As a result, for breaking to occur correctly, breaking macros need to be created for these character macros and not only for the literal characters themselves.

A pdfTeX-compatible version for working with UTF-8 is defined later, and \FV@BreakBeforePrep is \let to it under pdfTeX as necessary.
\FV@BreakAfter Allow line breaking (almost) anywhere, but only after specified characters.

\define@key{FV}{breakafter}{% 
  \ifstrempty{#1}% \let\FV@BreakAfter@empty \let\FancyVerbBreakStart\relax \let\FancyVerbBreakStop\relax \let\FV@Break@DefaultToken\FV@Break@NBToken}% \def\FV@BreakAfter{#1}% \let\FancyVerbBreakStart\FV@Break \let\FancyVerbBreakStop\FV@EndBreak \let\FV@Break@DefaultToken\FV@Break@beforeAfterToken}% \fvsset{breakafter={}}

\FV@breakafterinrun Determine whether breaking after specified characters is always allowed after each individual character, or is only allowed after the last in a run of identical characters.

\newbool{FV@breakafterinrun} \define@booleankey{FV}{breakafterinrun}{\booltrue{FV@breakafterinrun}}{\boolfalse{FV@breakafterinrun}} \fvsset{breakafterinrun=false}

\FV@BreakAfterPrep This is the breakafter equivalent of \FV@BreakBeforePrep. It is also used within \FV@FormattingPrep. The order of \FV@BreakBeforePrep and \FV@BreakAfterPrep must always be second, because it checks for conflicts with breakbefore.

A pdfTeX-compatible version for working with UTF-8 is defined later, and \FV@BreakAfterPrep is \let to it under pdfTeX as necessary.

\def\FV@BreakAfterPrep{% \ifx\FV@BreakAfter@empty\relax \else \gdef\FV@BreakAfter@Def{}\begingroup \def\FV@BreakAfter@Process##1##2\FV@Undefined{% \expandafter\FV@BreakAfter@Process@i\expandafter{##1}% \expandafter\ifx\expandafter\relax\detokenize{##2}\relax \else \FV@BreakAfter@Process##2\FV@Undefined\fi }\def\FV@BreakAfter@Process@i##1{% \ifcsname FV@BreakBefore@Token\detokenize{##1}\endcsname \ifbool{FV@breakbeforeinrun}{}{\PackageError{fvextra}{Conflicting breakbeforeinrun and breakafterinrun for "\detokenize{##1}"}{Conflicting breakbeforeinrun and breakafterinrun for "\detokenize{##1}"}}}%\fvsset{breakafterinrun=false}%\fi \fi \def\FV@BreakAfter@Process@i##1{% \ifsname FV@BreakBefore@Token\detokenize{##1}\endsname \fbool{FV@breakbeforeinrun}% {\fbool{FV@breakafterinrun}% {\PackageError{fvextra}{Conflicting breakbeforeinrun and breakafterinrun for "\detokenize{##1}"}{Conflicting breakbeforeinrun and breakafterinrun for "\detokenize{##1}"}}}%{\fbool{FV@breakafterinrun}%}
Now that \texttt{\FV@BreakBeforePrep} and \texttt{\FV@BreakAfterPrep} are defined, add them to \texttt{\FV@FormattingPrep@PreHook}, which is the \texttt{fvextra} extension to \texttt{\FV@FormattingPrep}. The ordering here is important, since \texttt{\FV@BreakAfterPrep} contains compatibility checks with \texttt{\FV@BreakBeforePrep}, and thus must be used after it. Also, we have to check for the pdfTeX engine with \texttt{inputenc} using UTF-8, and use the UTF macros instead when that is the case.

\begin{verbatim}
\@addto@macro{\FV@FormattingPrep@PreHook}{%  
\if\FV@pdfTeXinputenc  \ifdefstring{\inputencodingname}{utf8}{  
\let{\FV@BreakBeforePrep}{\FV@BreakBeforePrep@UTF}  
\let{\FV@BreakAfterPrep}{\FV@BreakAfterPrep@UTF}  
}{}  
\fi  
\FV@BreakBeforePrep\FV@BreakAfterPrep}
\end{verbatim}

\texttt{\FancyVerbBreakAnywhereSymbolPre} The pre-break symbol for breaks introduced by \texttt{breakanywhere}. That is, the symbol before breaks that occur between characters, rather than at spaces.

\begin{verbatim}
\define@key{FV}{breakanywheresymbolpre}{%  \ifdefstrempty{#1}{}{\def{\FancyVerbBreakAnywhereSymbolPre}{\hbox{#1}}}  
\fvset{breakanywheresymbolpre={}}
\end{verbatim}

\texttt{\FancyVerbBreakAnywhereSymbolPost} The post-break symbol for breaks introduced by \texttt{breakanywhere}.

\begin{verbatim}
\define@key{FV}{breakanywheresymbolpost}{%  \ifdefstrempty{#1}{}{\def{\FancyVerbBreakAnywhereSymbolPost}{\hbox{#1}}}  
\fvset{breakanywheresymbolpost={}}
\end{verbatim}

\texttt{\FancyVerbBreakBeforeSymbolPre} The pre-break symbol for breaks introduced by \texttt{breakbefore}.

\begin{verbatim}
\define@key{FV}{breakbeforesymbolpre}{%  \ifdefstrempty{#1}{}{\def{\FancyVerbBreakBeforeSymbolPre}{\hbox{#1}}}  
\fvset{breakbeforesymbolpre={}}
\end{verbatim}
The post-break symbol for breaks introduced by `breakbefore`.
\begin{verbatim}
define@key{FV}{breakbeforesymbolpost}{{
  \ifstrempty{#1}{{
    \def\FancyVerbBreakBeforeSymbolPost{}}
  }{
    \def\FancyVerbBreakBeforeSymbolPost{\hbox{#1}}}}
\end{verbatim}
\fvset{breakbeforesymbolpost={}}

The pre-break symbol for breaks introduced by `breakafter`.
\begin{verbatim}
define@key{FV}{breakaftersymbolpre}{{
  \ifstrempty{#1}{{
    \def\FancyVerbBreakAfterSymbolPre{}}
  }{
    \def\FancyVerbBreakAfterSymbolPre{\hbox{#1}}}}
\end{verbatim}
\fvset{breakaftersymbolpre={\,ootnotesize\ensuremath{_floor}}}

The post-break symbol for breaks introduced by `breakafter`.
\begin{verbatim}
define@key{FV}{breakaftersymbolpost}{{
  \ifstrempty{#1}{{
    \def\FancyVerbBreakAfterSymbolPost{}}
  }{
    \def\FancyVerbBreakAfterSymbolPost{\hbox{#1}}}}
\end{verbatim}
\fvset{breakaftersymbolpost={}}

The macro governing breaking for `breakanywhere=true`.
\begin{verbatim}
newcommand{\FancyVerbBreakAnywhereBreak}{%\discretionary{\FancyVerbBreakAnywhereSymbolPre}{\FancyVerbBreakAnywhereSymbolPost}{}%}
\end{verbatim}

The macro governing breaking for `breakbefore=true`.
\begin{verbatim}
newcommand{\FancyVerbBreakBeforeBreak}{%\discretionary{\FancyVerbBreakBeforeSymbolPre}{\FancyVerbBreakBeforeSymbolPost}{}%}
\end{verbatim}

The macro governing breaking for `breakafter=true`.
\begin{verbatim}
newcommand{\FancyVerbBreakAfterBreak}{%\discretionary{\FancyVerbBreakAfterSymbolPre}{\FancyVerbBreakAfterSymbolPost}{}%}
\end{verbatim}

When inserting breaks, insert breaks within groups (typically `{...}`) instead of skipping over them. This isn’t the default because it is incompatible with many macros since it inserts breaks into all arguments. For those cases, redefining macros to use `\FancyVerbBreakStart...\FancyVerbBreakStop` to insert breaks is better.
\begin{verbatim}
newbool{\FV@breaknonspaceingroup}
define@booleankey{FV}{breaknonspaceingroup}{{\booltrue{\FV@breaknonspaceingroup}}}{\boolfalse{\FV@breaknonspaceingroup}}
\end{verbatim}
\fvset{breaknonspaceingroup=false}

### 12.12.2 Line breaking implementation

**Helper macros**

`\FV@LineBox` A box for saving a line of text, so that its dimensions may be determined and thus we may figure out if it needs line breaking.
\begin{verbatim}
\newsavebox{\FV@LineBox}
\end{verbatim}
\FV@LineIndentBox A box for saving the indentation of code, so that its dimensions may be determined for use in auto-indentation of continuation lines.
\newsavebox{\FV@LineIndentBox}

\FV@LineIndentChars A macro for storing the indentation characters, if any, of a given line. For use in auto-indentation of continuation lines
\let\FV@LineIndentChars@empty

\FV@GetLineIndent A macro that takes a line and determines the indentation, storing the indentation chars in \FV@LineIndentChars.
\def\FV@GetLineIndent{% \ifnextchar\FV@Sentinel \FV@GetLineIndent@End\else\ifx\@let@token\FV@FVSpaceToken \let\FV@Next\FV@GetLineIndent@Whitespace \else\ifx\@let@token\FV@FVTabToken \let\FV@Next\FV@GetLineIndent@Whitespace \else\ifcsname FV@PYG@Redefed\endcsname \ifx\@let@token\FV@PYG@Redefed \let\FV@Next\FV@GetLineIndent@Pygments \else \let\FV@Next\FV@GetLineIndent@End \fi \else \let\FV@Next\FV@GetLineIndent@End \fi\fi\fi\fi\FV@Next}\FV@GetLineIndent@End\FV@Sentinel{}\FV@GetLineIndent@Whitespace\expandafter\def\expandafter\FV@LineIndentChars\expandafter{\FV@LineIndentChars\@empty}\FV@GetLineIndent\FV@GetLineIndent@Pygments\FV@GetLineIndent@End\FV@Next\FV@GetLineIndent@End\FV@Next\FV@GetLineIndent@End\FV@Next

\FV@GetLineIndent@End\FV@Sentinel{}
\FV@GetLineIndent@Whitespace\expandafter\def\expandafter\FV@LineIndentChars\expandafter{\FV@LineIndentChars\@empty}\FV@GetLineIndent\FV@GetLineIndent@Pygments\FV@GetLineIndent@End\FV@Next\FV@GetLineIndent@End\FV@Next\FV@GetLineIndent@End\FV@Next

Tab expansion

The fancyvrb option obeytabs uses a clever algorithm involving boxing and unboxing to expand tabs based on tab stops rather than a fixed number of equivalent space characters. (See the definitions of \FV@@ObeyTabs and \FV@TrueTab in section 12.10.4.) Unfortunately, since this involves \hbox, it interferes with the line breaking algorithm, and an alternative is required.

There are probably many ways tab expansion could be performed while still allowing line breaks. The current approach has been chosen because it is relatively straightforward and yields identical results to the case without line breaks. Line breaking involves saving a line in a box, and determining whether the box is too wide. During this process, if obeytabs=true, \FV@TrueTabSaveWidth, which is inside \FV@TrueTab, is \let to a version that saves the width of every tab in a macro. When a line is broken, all tabs within it will then use a variant of \FV@TrueTab that sequentially retrieves the saved widths. This maintains the exact behavior of the case without line breaks.

Note that the special version of \FV@TrueTab is based on the fvextra patched version of \FV@TrueTab, not on the original \FV@TrueTab defined in fancyvrb.
Version of \FV@TrueTab that uses pre-computed tab widths.

\begin{verbatim}
2224  \def\FV@TrueTab@UseWidth{% 
2225    \@tempdima=\csname FV@TrueTab:Width\arabic{FV@TrueTabCounter}\endcsname sp\relax 
2226    \stepcounter{FV@TrueTabCounter}% 
2227    \hbox to\@tempdima{\hss\FV@TabChar}}
\end{verbatim}

Line scanning and break insertion macros

The strategy here is to scan through text token by token, inserting potential breaks at appropriate points. The final text with breaks inserted is stored in \FV@BreakBuffer, which is ultimately passed on to a wrapper macro like \FancyVerbFormatText or \FancyVerbFormatInline.

If user macros insert breaks via \FancyVerbBreakStart...\FancyVerbBreakStop, this invokes an additional scanning/insertion pass within each macro after expansion. The scanning/insertion only applies to the part of the expanded macros wrapped in \FancyVerbBreakStart...\FancyVerbBreakStop. At the time this occurs, during macro processing, text will already be wrapped in a wrapper macro like \FancyVerbFormatText or \FancyVerbFormatInline. That is, the built-in break insertion occurs before any typesetting, but user macro break insertion occurs during typesetting.

Token comparison is currently based on \texttt{\ifx}. This is sufficient for verbatim text but a comparison based on \texttt{\detokenize} might be better for cases when \texttt{commandchars} is in use. For example, with \texttt{commandchars} characters other than the curly braces \{\} might be the group tokens.

It would be possible to insert each token/group into the document immediately after it is scanned, instead of accumulating them in a “buffer.” But that would interfere with macros. Even in the current approach, macros that take optional arguments are problematic, since with some settings breaks will interfere with optional arguments.\footnote{Through a suitable definition that tracks the current state and looks for square brackets, this might be circumvented. Then again, in verbatim contexts, macro use should be minimal, so the restriction to macros without optional arguments should generally not be an issue.}

The last token is tracked with \texttt{\FV@LastToken}, to allow lookahead when breaking by groups of identical characters. \texttt{\FV@LastToken} is \texttt{\let} to \texttt{\FV@Undefined} any time the last token was something that shouldn’t be compared against (for example, a non-empty group), and it is not reset whenever the last token may be ignored (for example, \{\}). When setting \texttt{\FV@LastToken}, it is vital always to use \texttt{\let\FV@LastToken=\ldots} so that \texttt{\let\FV@LastToken==} will work (so that the equals sign = won’t break things).

\texttt{\FV@BreakBufferDepth} Track buffer depth while inserting breaks. Some macros and command sequences require recursive processing. For example, groups \{\ldots\} (with \texttt{commandchars} and \texttt{breaknonspaceingroup}), math, and nested \texttt{\FancyVerbBreakStart...\FancyVerbBreakStop}. Depth starts at zero. The current buffer at depth \texttt{n} is always \texttt{\FV@BreakBuffer}, with other buffers \texttt{\FV@BreakBuffer<n>} etc. named via \texttt{\csname} to allow for the integer.

\begin{verbatim}
2228  \newcounter{FV@BreakBufferDepth}
\end{verbatim}

\texttt{\FV@BreakBuffer@Append} Append to \texttt{\FV@BreakBuffer}.

\begin{verbatim}
2229  \def\FV@BreakBuffer@Append#1{%
\end{verbatim}
\FV@BreakBufferStart

Create a new buffer, either at the beginning of scanning or during recursion. The
single mandatory argument is the macro for handling tokens, which is \let to
\FV@Break@Token. An intermediate \FV@BreakBufferStart@i is used to optimize \ifx comparisons for \FV@BreakBufferStart during scanning.

For recursion, \FV@BreakBuffer<n> and \FV@Break@Token<n> store the state
(buffer and token handling macro) immediately prior to recursion with depth <n>.

\FV@BreakBufferStart{%
\FV@BreakBufferStart@i{%
\ifnum\value{FV@BreakBufferDepth}>0\relax
\expandafter\let\csname FV@BreakBuffer\arabic{FV@BreakBufferDepth}\endcsname
\FV@BreakBuffer
\expandafter\let\csname FV@Break@Token\arabic{FV@BreakBufferDepth}\endcsname
\FV@Break@Scan
\fi
\def\FV@BreakBuffer{}%
\let\FV@Break@Token=#1%
\stepcounter{FV@BreakBufferDepth}%
\let\FV@LastToken=\FV@Undefined
\FV@Break@Scan
\}
\FV@InsertBreaks This inserts breaks within text (#2) and stores the result in \FV@BreakBuffer. Then it invokes a macro (#1) on the result. That allows \FancyVerbFormatInline and \FancyVerbFormatText to operate on the final text (with breaks) directly, rather than being given text without breaks or text wrapped with macros that will (potentially recursively) insert breaks. (Breaks inserted by user macros are not yet present, though, since they are only inserted—potentially recursively—during macro processing.)

The initial \ifx skips break insertion when break insertion is turned off (\FancyVerbBreakStart is \relax).

The current definition of \FV@Break@Token is swapped for a UTF-8 compatible one under pdfTeX when necessary. In what follows, the default macros are defined after \FV@Break, since they make the algorithms simpler to understand. The more complex UTF variants are defined afterward.

\def\FV@InsertBreaks#1#2{% 
\ifx\FancyVerbBreakStart\relax 
\else 
\fi
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
\fi
{#1{#2}}% 
{\if\FV@pdfTeXinputenc 
  \ifdefstring{\inputencodingname}{utf8} 
  {\if\FV@Break@DefaultToken\FV@Break@AnyToken 
    \let\FV@Break@DefaultToken\FV@Break@AnyToken\UTF 
  \else 
    \ifdefstring{\FV@Break@DefaultToken}{\FV@Break@BeforeAfterToken} 
    \let\FV@Break@DefaultToken\FV@Break@BeforeAfterToken\UTF 
  \fi
} 
\fi
\setcounter{FV@BreakBufferDepth}{0} 
\boolfalse{FV@UserMacroBreaks}%
\FV@Break The entry macro for break insertion. Whatever is delimited (after expansion) by \FV@Break...\FV@EndBreak will be scanned token by token/group by group, and accumulated (with any added breaks) in \FV@BreakBuffer. After scanning is complete, \FV@BreakBuffer will be inserted.
\FV@EndBreak
\FV@Break@Scan Look ahead via \@ifnextchar. Don’t do anything if we’re at the end of the region to be scanned. Otherwise, invoke a macro to deal with what’s next based on whether it is math, or a group, or something else.

This and some following macros are defined inside of groups to ensure proper catcodes. The check against \FV@BreakBufferStart should typically not be necessary; it is included for completeness and to allow for future extensions and customization. \FV@BreakBufferStart is only inserted raw (rather than wrapped in \FancyVerbBreakStart) in token processing macros, where it initiates (or restarts) scanning and is not itself scanned.

\FV@Break@Math Grab an entire math span, and insert it into \FV@BreakBuffer. Due to grouping, this works even when math contains things like \text{\$x\$}. After dealing with the math span, continue scanning.
Grab the group, and insert it into \FVBreakBuffer (as a group) before continuing scanning.

Append token to buffer while adding no breaks (NB) and reset last token.

Deal with breaking around any token. This doesn’t break macros with _mandatory_ arguments, because \FancyVerbBreakAnywhereBreak is inserted _before_ the token. Groups themselves are added without any special handling. So a macro would end up right next to its original arguments, without anything being inserted. Optional arguments will cause this approach to fail; there is currently no attempt to identify them, since that is a much harder problem.

If it is ever necessary, it would be possible to create a more sophisticated version involving catcode checks via %ifcat. Something like this:

\begingroup
\catcode`\a=11\%
\catcode`\+=12\%
\gdef\FVBreak...
% \ifcat\noexpand#1a%
% \FVBreakBuffer@Append...
% \else
% ...
%\endgroup
%
\def\FVBreakAnyToken#1{%
\ifx\FVBreakSpaceToken#1\relax
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
%\fi
\def\FVBreakAnyToken#1{%
\ifx\FVBreakSpaceToken#1\relax
\expandafter\@firstoftwo
\else
\expandafter\@secondoftwo
%\fi

Deal with breaking around only specified tokens. This is a bit trickier. We only break if a macro corresponding to the token exists. We also need to check whether the specified token should be grouped, that is, whether breaks are allowed between identical characters. All of this has to be written carefully so that nothing is accidentally inserted into the stream for future scanning.

Dealing with tokens followed by empty groups (for example, `\x{}`) is particularly challenging when we want to avoid breaks between identical characters. When a token is followed by a group, we need to save the current token for later reference (`\x` in the example), then capture and save the following group, and then—only if the group was empty—see if the following token is identical to the old saved token.

The `\csname @let@token\endcsname` prevents issues if `\@let@token` is ever `\else` or `\fi`. 
\FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak#1}\%
\fi
\let\FV@Break@Next\FV@Break@Scan
\let\FV@LastToken=#1\%
\fi}\
{\ifx#1\FV@LastToken\relax
\ifsname\FV@BreakAfter@Token\detokenize{#1}\endsname
\let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
\def\FV@RescanToken{#1}\%
\else\n\FV@BreakBuffer@Append{#1}\%
\let\FV@Break@Next\FV@Break@Scan
\let\FV@LastToken=#1\%
\fi
\else
\ifsname\FV@BreakAfter@Token\detokenize{#1}\endsname
\ifx#1\FV@FVSpaceToken\n\FV@BreakBuffer@Append{\FancyVerbBreak}\%
\else\n\FV@BreakBuffer@Append{\FancyVerbBeforeBreak}\%
\fi
\fi
\let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
\def\FV@RescanToken{#1}\%
\else
\ifx#1\FV@FVSpaceToken\n\FV@BreakBuffer@Append{\FancyVerbBreak}\%
\else\n\FV@BreakBuffer@Append{\FancyVerbBeforeBreak}\%
\fi
\fi
\let\FV@Break@Next\FV@Break@Scan
\let\FV@LastToken=#1\%
\fi
\fi}\%
\FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan{\
\expandafter\FV@Break@AfterTokenBreak\FV@RescanToken}
\def\FV@Break@AfterTokenBreak#1{\
\ifbool{FV@breakafterinrun}{\ifx#1\FV@FVSpaceToken\
\FV@BreakBuffer@Append{#1\FancyVerbSpaceBreak}\
\else\n\FV@BreakBuffer@Append{#1\FancyVerbBreakAfterBreak}\
\fi
\fi}
\let\FV@LastToken=#1\%
\expandafter\FV@Break@AfterTokenBreak@i{#1}
\expandafter\@secondoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
\expandafter\@firstoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
\def\FV@Break@AfterTokenBreak@i#1{\
\ifbool{FV@breakafterinrun}{\ifx#1\FV@FVSpaceToken\
\FV@BreakBuffer@Append{#1\FancyVerbSpaceBreak}\%
\else\n\FV@BreakBuffer@Append{#1\FancyVerbBreakAfterBreak}\%
\fi
\fi}
\let\FV@LastToken=#1\%
\expandafter\FV@Break@AfterTokenBreak@i{#1}
\expandafter\@secondoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
\expandafter\@firstoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
\def\FV@Break@AfterTokenBreak@i#1{\
\ifbool{FV@breakafterinrun}{\ifx#1\FV@FVSpaceToken\
\FV@BreakBuffer@Append{#1\FancyVerbSpaceBreak}\%
\else\n\FV@BreakBuffer@Append{#1\FancyVerbBreakAfterBreak}\%
\fi
\fi}
\let\FV@LastToken=#1\%
\expandafter\FV@Break@AfterTokenBreak@i{#1}
\expandafter\@secondoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
\expandafter\@firstoftwo
{\FV@BreakBuffer@Append{#1}\%
\FV@Break@Scan}{\FV@Break@AfterTokenBreak@i{#1}}
Line scanning and break insertion macros for pdfTeX with UTF-8

The macros above work with the XeTeX and LuaTeX engines and are also fine for pdfTeX with 8-bit character encodings. Unfortunately, pdfTeX works with multi-byte UTF-8 code points at the byte level, making things significantly trickier. The code below re-implements the macros in a manner compatible with the inputenc package with option utf8. Note that there is no attempt for compatibility with utf8x; utf8 has been significantly improved in recent years and should be sufficient in the vast majority of cases. And implementing variants for utf8 was already sufficiently painful.

Create macros conditionally:

\if\FV@pdfTeXinputenc\FV@BreakBeforePrep@UTF\fi

We need UTF variants of the \texttt{breakbefore} and \texttt{breakafter} prep macros. These are only ever used with inputenc with UTF-8. There is no need for encoding checks
here; checks are performed in `\FV@FormattingPrep@PreHook` (checks are inserted into it after the non-UTF macro definitions).

```tex
\def\FV@BreakBeforePrep@UTF{% 
\ifx\FV@BreakBefore\@empty\relax
\else
  \gdef\FV@BreakBefore@Def{}%
  \begingroup
  \def\FV@BreakBefore@Process##1{% 
    \ifcsname FV@U8:\detokenize{##1}\endcsname 
    \let\FV@UTF@octets@after\FV@BreakBefore@Process@ii 
  \else
    \ifx##1\FV@Undefined 
      \let\FV@Break@Next\@gobble 
    \else 
      \let\FV@Break@Next\FV@BreakBefore@Process@i 
    \fi
  \fi
  \FV@Break@Next##1% 
}
\def\FV@BreakBefore@Process@i##1{% 
  \expandafter\FV@BreakBefore@Process@ii\expandafter{##1}}%
\def\FV@BreakBefore@Process@ii##1{% 
  \g@addto@macro\FV@BreakBefore@Def{\@namedef{FV@BreakBefore@Token\detokenize{##1}}{}}}%
\FV@BreakBefore@Process 
}%
\FV@EscChars 
\expandafter\FV@BreakBefore@Process\FV@BreakBefore\FV@Undefined 
\endgroup 
\FV@BreakBefore@Def 
\FV@BreakBeforePrep@PygmentsHook 
}\fi
\fi
}

\FV@BreakAfterPrep@UTF 
\def\FV@BreakAfterPrep@UTF{% 
\ifx\FV@BreakAfter\@empty\relax
\else 
  \gdef\FV@BreakAfter@Def{}%
  \begingroup
  \def\FV@BreakAfter@Process##1{% 
    \ifcsname FV@U8:\detokenize{##1}\endcsname 
    \let\FV@UTF@octets@after\FV@BreakAfter@Process@ii 
  \else
    \ifx##1\FV@Undefined 
      \let\FV@Break@Next\@gobble 
    \else 
      \let\FV@Break@Next\FV@BreakAfter@Process@i 
    \fi
  \fi
  \FV@Break@Next##1% 
}
\def\FV@BreakAfter@Process@i##1{% 
  \expandafter\FV@BreakAfter@Process@ii\expandafter{##1}}%
\def\FV@BreakAfter@Process@ii##1{% 
  \g@addto@macro\FV@BreakAfter@Def{\@namedef{FV@BreakAfter@Token\detokenize{##1}}{}}}%
\FV@BreakAfter@Process 
}%
\FV@EscChars 
\expandafter\FV@BreakAfter@Process\FV@BreakAfter\FV@Undefined 
\endgroup 
\FV@BreakAfter@Def 
\FV@BreakAfterPrep@PygmentsHook 
}\fi
\fi
}
```

115
Instead of just adding each token to `\FV@BreakBuffer` with a preceding break, also check for multi-byte code points and capture the remaining bytes when they are encountered.

```latex
\def\FV@Break@AnyToken@UTF#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\expandafter\let\expandafter\FV@Break@Next\expandafter\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@UTF@octets@after\FV@Break@AnyToken@UTF@i
  \else
    \let\FV@Break@Next\FV@Break@AnyToken@UTF@i
  \fi
  \FV@Break@Next{#1}%
}
```

```latex
\def\FV@Break@AnyToken@UTF@i#1{%
  \def\FV@CurrentToken{#1}%
  \ifx\FV@CurrentToken\FV@ActiveSpaceToken\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  \let\FV@LastToken\FV@CurrentToken
  \FV@BreakBuffer@Append{#1}\FV@Break@Scan
  \ifx\FV@LastToken\FV@ActiveSpaceToken
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
} %}
```

```latex
\FV@Break@AnyToken@UTF

Instead of just adding each token to `\FV@BreakBuffer` with a preceding break, also check for multi-byte code points and capture the remaining bytes when they are encountered.

```latex
\def\FV@Break@AnyToken@UTF#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\expandafter\let\expandafter\FV@Break@Next\expandafter\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@UTF@octets@after\FV@Break@AnyToken@UTF@i
  \else
    \let\FV@Break@Next\FV@Break@AnyToken@UTF@i
  \fi
  \FV@Break@Next{#1}%
}
```

```latex
\def\FV@Break@AnyToken@UTF@i#1{%
  \def\FV@CurrentToken{#1}%
  \ifx\FV@CurrentToken\FV@ActiveSpaceToken\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  \let\FV@LastToken\FV@CurrentToken
  \FV@BreakBuffer@Append{#1}\FV@Break@Scan
  \ifx\FV@LastToken\FV@ActiveSpaceToken
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
} %}
```
Due to the way that the flow works, \#1 will sometimes be a single byte and sometimes be a multi-byte UTF-8 code point. As a result, it is vital use use `\detokenize` in the UTF-8 leading byte checks; `\string` would only deal with the first byte. It is also important to keep track of the distinction between `\FV@Break@Next#1` and `\FV@Break@Next{#1}`. In some cases, a multi-byte sequence is being passed on as a single argument, so it must be enclosed in curly braces; in other cases, it is being re-inserted into the scanning stream and curly braces must be avoided lest they be interpreted as part of the original text.
\fi
\let\FV@Break@Next\FV@Break@Scan
\def\FV@LastToken{\relax}%
\fi}%
{\ifx\FV@CurrentToken\FV@LastToken\relax
  {\ifcsname FV@BreakAfter@Token\detokenize{\relax}\endcsname
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
    \def\FV@RescanToken{\relax}%
  \else
    \FV@BreakBuffer@Append{\relax}
    \let\FV@Break@Next\FV@Break@Scan
    \def\FV@LastToken{\relax}%
  \fi
  \else
    \ifcsname FV@BreakAfter@Token\detokenize{\relax}\endcsname
      \ifx\FV@CurrentToken\FV@ActiveSpaceToken
        \FV@BreakBuffer@Append{\FancyVerbSpaceBreak\relax}
      \else
        \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak\relax}
      \fi
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
    \def\FV@RescanToken{\relax}%
  \else
    \ifx\FV@CurrentToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak\relax}
    \fi
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\fi}
\FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF{%
  \expandafter\FV@Break@AfterTokenBreak@UTF\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakAfterBreak\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF%i{%
  \expandafter\FV@Break@AfterTokenBreak@UTF%i\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF%i#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak%i\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak%i\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak%i\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF@i{%
  \expandafter\FV@Break@AfterTokenBreak@UTF@i\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF@i#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak@i\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakAfterBreak@i\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak@i\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF@ii{%
  \expandafter\FV@Break@AfterTokenBreak@UTF@ii\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF@ii#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak@ii\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakAfterBreak@ii\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak@ii\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF@iii{%
  \expandafter\FV@Break@AfterTokenBreak@UTF@iii\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF@iii#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak@iii\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakAfterBreak@iii\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak@iii\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF@iv{%
  \expandafter\FV@Break@AfterTokenBreak@UTF@iv\expandafter{\FV@RescanToken}
  \edef\FV@Break@AfterTokenBreak@UTF@iv#1{%
  \ifbool{FV@breakafterinrun}{%
    \ifx\FV@LastToken\FV@ActiveSpaceToken
      \FV@BreakBuffer@Append{\FancyVerbSpaceBreak@iv\relax}
    \else
      \FV@BreakBuffer@Append{\FancyVerbBreakAfterBreak@iv\relax}
    \fi
  \else
    \FV@BreakBuffer@Append{\FancyVerbBreakBeforeBreak@iv\relax}
  \fi
  \let\FV@Break@Next\FV@Break@Scan
  \def\FV@LastToken{\relax}%
  \fi}%
End the conditional creation of the pdfTeX UTF macros:

Line processing before scanning

The \texttt{lineno} package is used for formatting wrapped lines and inserting break symbols. We need a version of \texttt{lineno}'s \texttt{makeLineNumber} that is adapted for our purposes. This is adapted directly from the example \texttt{makeLineNumber} that is given in the \texttt{lineno} documentation under the discussion of internal line numbers. The \texttt{FV@SetLineBreakLast} is needed to determine the internal line number of the last segment of the broken line, so that we can disable the right-hand break symbol on this segment. When a right-hand break symbol is in use, a line of code will be processed twice: once to determine the last internal line number, and once to use this information only to insert right-hand break symbols on the appropriate lines. During the second run, \texttt{FV@SetLineBreakLast} is disabled by \texttt{letting} it to \texttt{relax}.

\begin{verbatim}
\def\FV@makeLineNumber{%
  \hss
  \FancyVerbBreakSymbolLeftLogic{\FancyVerbBreakSymbolLeft}%
  \hbox to \FV@BreakSymbolSepLeft{\hfill}%
  \rlap{\hskip\linewidth
    \hbox to \FV@BreakSymbolSepRight{\hfill}%
    \FancyVerbBreakSymbolRightLogic{\FancyVerbBreakSymbolRight}%
    \FV@SetLineBreakLast}
%
}\end{verbatim}

We need a copy of the default \texttt{\raggedright} to ensure that everything works with classes or packages that use a special definition.

\begin{verbatim}
\def\FV@RaggedRight{%
  \let\\@centercr
  \@rightskip\@flushglue
  \leftskip\z@skip
  \parindent\z@}
\end{verbatim}

This is the effective line width within a broken line.

\begin{verbatim}
\def\FV@LineWidth{\newdimen\FV@LineWidth}
\end{verbatim}

This is the macro that does most of the work. It was inspired by Marco Daniel’s code at \url{http://tex.stackexchange.com/a/112573/10742}.

This macro is invoked when a line is too long. We modify \texttt{FV@LineWidth} to take into account \texttt{breakindent} and \texttt{breakautoindent}, and insert \texttt{hboxes} to fill the empty space. We also account for \texttt{breaksymbolindentleft} and \texttt{breaksymbolindentright}, but only when there are actually break symbols. The code is placed in a \texttt{parbox}. Break symbols are inserted via \texttt{lineno}'s \texttt{internallinenumbers*}, which does internal line numbers without continuity between environments (the \texttt{linenumber} counter is automatically reset). The
beginning of the line has negative \hspace inserted to pull it out to the
correct starting position. \strut is used to maintain correct line heights. The
\parbox is followed by an empty \hbox that takes up the space needed for a
right-hand break symbol (if any). \FV@BreakByTokenAnywhereHook is a hook for
using breakbytokenanywhere when working with Pygments. Since it is within
internallinenumbers*, its effects do not escape.
\begin{Verbatim}
def\FV@SaveLineBox#1{%  
  \savebox{\FV@LineBox}{%  
    \advance\FV@LineWidth by -\FV@BreakIndent  
    \hbox to \FV@BreakIndent{\hfill}%  
    \ifbool{FV@breakautoindent}%  
      \let\FV@LineIndentChars\@empty  
      \FV@GetLineIndent#1\FV@Sentinel  
      \savebox{\FV@LineIndentBox}{\FV@LineIndentChars}%  
    \hbox to \wd{\FV@LineIndentBox}{\hfill}%  
    \advancemacro{\FV@LineWidth by -\wd{\FV@LineIndentBox}}%  
    \setcounter{FV@TrueTabCounter}{0}}%  
  \ifdefempty{\FancyVerbBreakSymbolLeft}{}%  
    \hbox to \FV@BreakSymbolIndentLeft{\hfill}%  
    \ifbool{FV@breakautoindent}%  
      \hspace*{-\wd{\FV@LineIndentBox}}%  
    \let\FV@BreakByTokenAnywhereHook\relax  
    \FV@ListProcessLine@Break\FV@ListProcessLine@Break
\end{Verbatim}

This macro is based on the original \FV@ListProcessLine and follows it as closely
as possible. \FV@LineWidth is reduced by \FV@FrameSep and \FV@FrameRule so
that text will not overrun frames. This is done conditionally based on which frames
are in use. We save the current line in a box, and only do special things if the box
is too wide. For uniformity, all text is placed in a \parbox, even if it doesn’t need
to be wrapped.

If a line is too wide, then it is passed to \FV@SaveLineBox. If there is no
right-hand break symbol, then the saved result in \FV@LineBox may be used
immediately. If there is a right-hand break symbol, then the line must be processed
a second time, so that the right-hand break symbol may be removed from the final segment of the broken line (since it does not continue). During the first use of \texttt{\FV@SaveLineBox}, the counter \texttt{FancyVerbLineBreakLast} is set to the internal line number of the last segment of the broken line. During the second use of \texttt{\FV@SaveLineBox}, we disable this (\texttt{\let\FV@SetLineBreakLast\relax}) so that the value of \texttt{FancyVerbLineBreakLast} remains fixed and thus may be used to determine when a right-hand break symbol should be inserted.

```latex
\def\FV@ListProcessLine@Break#1{%
\hbox to \hsize{%\kern\leftmargin\hbox to \linewidth{%\ifx\FV@RightListFrame\relax\else\advance\FV@LineWidth by \FV@FrameSep\fi\ifx\FV@LeftListFrame\relax\else\advance\FV@LineWidth by \FV@FrameRule\fi\ifx\FV@Tab\FV@TrueTab\let\FV@TrueTabSaveWidth\FV@TrueTabSaveWidth@Save\setcounter{FV@TrueTabCounter}{0}\fi\sbox{\FV@LineBox}{%\let\FancyVerbBreakStart\relax\let\FancyVerbBreakStop\relax\FancyVerbFormatLine{%%\FancyVerbHighlightLine %<-- Default definition using \rlap breaks breaking{\FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}%\ifx\FV@Tab\FV@TrueTab\let\FV@TrueTab\FV@TrueTab@UseWidth\setcounter{FV@TrueTabCounter}{0}\fi\FV@SaveLineBox{#1}%\ifdefempty{\FancyVerbBreakSymbolRight}{}{\let\FV@SetLineBreakLast\relax\setcounter{FV@TrueTabCounter}{0}\FV@SaveLineBox{#1}}%\FV@LeftListNumber\FV@LeftListFrame\FancyVerbFormatLine{%\FancyVerbHighlightLine{%\usebox{\FV@LineBox}}}%\FV@RightListFrame\FV@RightListNumber\ifx\FV@Tab\FV@TrueTab@UseWidth\let\FV@Tab\FV@TrueTab\fi\%\FV@ListProcessLine@Break\%\hbox to \hsize{%\kern\leftmargin\hbox to \linewidth{%\ifx\FV@RightListFrame\relax\else\advance\FV@LineWidth by \FV@FrameSep\fi\ifx\FV@LeftListFrame\relax\else\advance\FV@LineWidth by \FV@FrameRule\fi\ifx\FV@Tab\FV@TrueTab\let\FV@TrueTabSaveWidth\FV@TrueTabSaveWidth@Save\setcounter{FV@TrueTabCounter}{0}\fi\sbox{\FV@LineBox}{%\let\FancyVerbBreakStart\relax\let\FancyVerbBreakStop\relax\FancyVerbFormatLine{%\FancyVerbHighlightLine %<-- Default definition using \rlap breaks breaking{\FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}%\ifx\FV@Tab\FV@TrueTab\let\FV@TrueTab\FV@TrueTab@UseWidth\setcounter{FV@TrueTabCounter}{0}\fi\FV@SaveLineBox{#1}%\ifdefempty{\FancyVerbBreakSymbolRight}{}{\let\FV@SetLineBreakLast\relax\setcounter{FV@TrueTabCounter}{0}\FV@SaveLineBox{#1}}%\FV@LeftListNumber\FV@LeftListFrame\FancyVerbFormatLine{%\FancyVerbHighlightLine{%\usebox{\FV@LineBox}}}%\FV@RightListFrame\FV@RightListNumber\ifx\FV@Tab\FV@TrueTab@UseWidth\let\FV@Tab\FV@TrueTab\fi
```

122
12.13 Pygments compatibility

This section makes line breaking compatible with Pygments, which is used by several packages including minted and pythontex for syntax highlighting. A few additional line breaking options are also defined for working with Pygments.

Pygments converts some characters into macros to ensure that they appear literally. As a result, \texttt{breakbefore} and \texttt{breakafter} would fail for these characters. This macro checks for the existence of breaking macros for these characters, and creates breaking macros for the corresponding Pygments character macros as necessary.

The argument that the macro receives is the detokenized name of the main Pygments macro, with the trailing space that detokenization produces stripped. All macro names must end with a space, because the breaking algorithm uses detokenization on each token when checking for breaking macros, and this will produce a trailing space.

\begin{verbatim}
\def\FV@BreakBeforePrep@Pygments#1{\ifcsname FV@BreakBefore@Token\#1Zbs\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zus\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zob\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zcb\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zca\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zlt\endcsname\fi\ifcsname FV@BreakBefore@Token\#1Zgt\endcsname\fi}
\end{verbatim}
\ifcsname FV@BreakBefore@Token\textbackslash hashchar@endcsname
\@namedef{FV@BreakBefore@Token#1Zsh }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash percentchar@endcsname
\@namedef{FV@BreakBefore@Token#1Zpc }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash dollarchar@endcsname
\@namedef{FV@BreakBefore@Token#1Zdl }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{-}\endcsname
\@namedef{FV@BreakBefore@Token#1Zhy }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{'}\endcsname
\@namedef{FV@BreakBefore@Token#1Zsq }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{"}\endcsname
\@namedef{FV@BreakBefore@Token#1Zdq }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{^}\endcsname
\@namedef{FV@BreakBefore@Token#1Zca }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{\&}\endcsname
\@namedef{FV@BreakBefore@Token#1Zam }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{<}\endcsname
\@namedef{FV@BreakBefore@Token#1Zat }{}%
\fi
\ifcsname FV@BreakBefore@Token\textbackslash{>}\endcsname
\@namedef{FV@BreakBefore@Token#1Zrb }{}%
\fi
\fi

\FV@BreakAfterPrep@Pygments
\def\FV@BreakAfterPrep@Pygments#1{%}
\ifcsname FV@BreakAfter@Token\textbackslash\textbackslash@endcsname
\@namedef{FV@BreakAfter@Token#1Zbs }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslashunderscorechar@endcsname
\@namedef{FV@BreakAfter@Token#1Zus }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textbackslash}@endcsname
\@namedef{FV@BreakAfter@Token#1Zob }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textbackslash}\endcsname
\@namedef{FV@BreakAfter@Token#1Zcb }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\ifcsname FV@BreakAfter@Token\textbackslash{\textasciitilde}\endcsname
\@namedef{FV@BreakAfter@Token#1Zti }{}%
\fi
\fi

\FV@BreakAfterPrep@Pygments
When Pygments is used, do not allow breaks within Pygments tokens. So, for example, breaks would not be allowed within a string, but could occur before or after it. This has no affect when Pygments is not in use, and is only intended for minted, pythontex, and similar packages.

breakbytoken

breakbytokenanywhere

breakbytoken prevents breaks within tokens. Breaks outside of tokens may still occur at spaces. This option also enables breaks between immediately adjacent tokens that are not separated by spaces. Its definition is tied in with breakbytoken so that breakbytoken may be used as a check for whether either option is in use; essentially, breakbytokenanywhere is treated as a special case of breakbytoken.
This is the break introduced when \texttt{breakbytokenanywhere=true}. Alternatives would be \texttt{\discretionary{}{}{}} or \texttt{\linebreak[0]}. \FancyVerbBreakByTokenAnywhereBreak

\VerbatimPygments This is the command that activates Pygments features. It must be invoked before \texttt{\begin{Verbatim}}, etc., but inside a \texttt{\begingroup...\endgroup} so that its effects do not escape into the rest of the document (for example, within the beginning of an environment. It takes two arguments: The Pygments macro that literally appears (\PYG for minted and pythonex), and the Pygments macro that should actually be used (\PYG\texttt{\textit{style\_name}} for minted and pythonex). The two are distinguished because it can be convenient to highlight everything using the same literal macro name, and then \texttt{\let} it to appropriate values to change styles, rather than redoing all highlighting to change styles. It modifies \texttt{\FV@PygmentsHook}, which is at the beginning of \texttt{\FV@FormattingPrep@PreHook}, to make the actual changes at the appropriate time.

\def\VerbatimPygments#1#2{% 
\def\FV@PygmentsHook{\FV@VerbatimPygments{#1}{#2}}
\FV@VerbatimPygments

This does all the actual work. Again, \texttt{#1} is the Pygments macro that literally appears, and \texttt{#2} is the macro that is actually to be used.

The \texttt{\breakbefore} and \texttt{\breakafter} hooks are redefined. This requires some trickery to get the detokenized name of the main Pygments macro without the trailing space that detokenization of a macro name produces.

In the non-\texttt{breakbytoken} case, \texttt{#1} is redefined to use \texttt{#2} internally, bringing in \texttt{\FancyVerbBreakStart} and \texttt{\FancyVerbBreakStop} to allow line breaks.

In the \texttt{breakbytoken} cases, an \texttt{\hbox} is used to prevent breaks within the macro (breaks could occur at spaces even without \texttt{\FancyVerbBreakStart}). The \texttt{\breakbytoken} case is similar but a little tricky. \texttt{\FV@BreakByTokenAnywhereHook}, which is inside \texttt{\FV@SaveLineBox} where line breaking occurs, is used to define \texttt{\FV@BreakByTokenAnywhereBreak} so that it will “do nothing” the first time it is used and on subsequent invocations become \texttt{\FancyVerbBreakByTokenAnywhereBreak}. Because the hook is within the \texttt{internallinenumbers*} environment, the redefinition doesn’t escape, and the default global definition of \texttt{\FV@BreakByTokenAnywhereBreak} as \texttt{\relax} is not affected. We don’t want the actual break to appear before the first Pygments macro in case it might cause a spurious break after leading whitespace. But we must have breaks before Pygments macros because otherwise lookahead would be necessary.

An intermediate variable \texttt{\FV@PYG} is defined to avoid problems in case \texttt{#1=#2}. There is also a check for a non-existant \texttt{#2} (\PYG\texttt{\textit{style\_name}} may not be created until a later compile in the pythonex case); if \texttt{#2} does not exist, fall back to \texttt{#1}. For the existence check, \texttt{\ifx...\relax} must be used instead of \texttt{\ifcsname...\endcsname}, because \texttt{#2} will be a macro, and will typically be created with \texttt{\csname...\endcsname} which will \texttt{\let} the macro to \texttt{\relax} if it doesn’t already exist.

\texttt{\FV@PYG@Redefed} is \texttt{\let} to the Pygments macro that appears literally (after redefinition), so that it can be detected elsewhere to allow for special processing, such as in \texttt{\breakautoindent}.

\def\FV@VerbatimPygments#1#2{%}
\edef\FV@PYG@Literal{\expandafter\FV@DetokMacro@StripSpace\detokenize{#1}}\edef\FV@BreakBeforePrep@PygmentsHook{\expandafter\FV@BreakBeforePrep@Pygments\expandafter{\FV@PYG@Literal}}\edef\FV@BreakAfterPrep@PygmentsHook{\expandafter\FV@BreakAfterPrep@Pygments\expandafter{\FV@PYG@Literal}}\ifx#2\relax\let\FV@PYG=#1\relax\else\let\FV@PYG=#2\relax\fi\ifbool{FV@breakbytoken}{\ifbool{FV@breakbytokenanywhere}{\def\FV@BreakByTokenAnywhereHook{\def\FV@BreakByTokenAnywhereBreak{\let\FV@BreakByTokenAnywhereBreak\FancyVerbBreakByTokenAnywhereBreak}}\def#1##1##2{\FV@BreakByTokenAnywhereBreak\leavevmode\hbox{\FV@PYG{##1}{##2}}}}\def#1##1##2{\leavevmode\hbox{\FV@PYG{##1}{\FancyVerbBreakStart##2\FancyVerbBreakStop}}}}\let\FV@PYG@Redefed=#1\relax\let\FV@BreakByTokenAnywhereBreak\relax\def\FV@DetokMacro@StripSpace#1 {#1}