The fvextra package

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

v1.4 from 2019/02/04

Abstract

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

Contents

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 ifthen and etoolbox packages are 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.

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}
```

‘quoted text’

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

`fontencoding` (string) (default: ⟨*document font encoding*⟩)
Set the font encoding inside fancyvrb commands and environments. Setting `fontencoding=none` resets to the default document font encoding.

`highlightcolor` (string) (default: `LightCyan`)
Set the color used for `highlightlines`, using a predefined color name from color or xcolor, or a color defined via `\definecolor`.

`highlightlines` (string) (default: ⟨*none*⟩)

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

The highlighting color can be customized with `highlightcolor`.

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

```
1 First line
2 Second line
3 Third line
4 Fourth line
5 Fifth line
```

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 `\FancyVerbHighlightLineFirst` as a starting point.

- `\FancyVerbHighlightLineFirst`: First line in a range.
- `\FancyVerbHighlightLineMiddle`: Inner lines in a range.
- `\FancyVerbHighlightLineLast`: Last line in a range.
- `\FancyVerbHighlightLineSingle`: Single highlighted lines.
- `\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 fvextra 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.

`linenos` (boolean) (default: `false`)

fancyvrb allows line numbers via the options `numbers=`⟨*position*⟩. This is essentially an alias for `numbers=left`. It primarily exists for better compatibility with the minted package.

`mathescape` (boolean) (default: `false`)
This causes everything between dollar signs `$...$` to be typeset as math. The caret `^` and underscore `_` have their normal math meanings.

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

Note that fvextra provides several patches that make math mode within verbatim as close to normal math mode as possible (section 9.3).

`numberfirstline` (boolean) (default: `false`)
When line numbering is used with `stepnumber` $\neq 1$, the first line may not always be numbered, depending on the line number of the first line. This causes the first line always to be numbered.

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

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

`numbers` (none | left | right | both) (default: `none`)
fvextra adds the `both` option for line numbering.

```
\begin{Verbatim}[numbers=both]
First line
Second line
Third line
Fourth line
\end{Verbatim}
```

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

`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 reim-

plemented `\SaveVerb`. It may be extended to environments (`\UseVerbatim`, etc.) in the future, if the relevant commands and environments are reimplemented.

`space` (macro) (default: `\textvisiblespace`, ␣)
Redefine the visible space character. Note that this is only used if `showspaces=true`. The color of the character may be set with `spacecolor`.

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

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

One␣␣two␣␣three

`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.

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

```
1 First line
  Second line
3 Third line
  Fourth line
```

`stepnumberoffsetvalues` (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. Using `firstnumber` to offset the numbering will change which lines are numbered and which line gets which number, but will not change which *numbers* appear. This option causes `firstnumber`

to be ignored in determining which line numbers are a multiple of `stepnumber`. `firstnumber` is still used in calculating the actual numbers that appear. As a result, the line numbers that appear will be a multiple of `stepnumber`, plus `firstnumber` minus 1.

This option gives the original behavior of fancyvrb when `firstnumber` is used with `stepnumber` $\neq 1$ (section 10.2).

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

```
   First line
5  Second line
   Third line
7  Fourth line
```

`tab` (macro) (default: fancyvrb's `\FancyVerbTab, ↦|`)

Redefine the visible tab character. Note that this is only used if `showtabs=true`. The color of the character may be set with `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 minted or pythontex packages.

fvextra patches fancyvrb 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 formatting 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 fancyvrb definitions.

4.2 Line and text formatting

`\FancyVerbFormatLine`
`\FancyVerbFormatText`

fancyvrb defines `\FancyVerbFormatLine`, which can be used to apply custom formatting to each individual line of text. By default, it takes a line as an argument and inserts it with no modification. This is equivalent to `\newcommand{\FancyVerbFormatLine}[1]{#1}`.[1]

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. fvextra leaves `\FancyVerbFormatLine` as applying to the entire line, and introduces a new command `\FancyVerbFormatText` that only applies to the text part of the line.[2] By default, `\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.

[1] The actual definition in fancyvrb is `\def\FancyVerbFormatLine#1{\FV@ObeyTabs{#1}}`. This is problematic because redefining the macro could easily eliminate `\FV@ObeyTabs`, which governs tab expansion. fvextra redefines the macro to `\def\FancyVerbFormatLine#1{#1}` and patches all parts of fancyvrb that use `\FancyVerbFormatLine` so that `\FV@ObeyTabs` is explicitly inserted at the appropriate points.

[2] When `breaklines=true`, each line is wrapped in a `\parbox`. `\FancyVerbFormatLine` is outside the `\parbox`, and `\FancyVerbFormatText` is inside.

```
\renewcommand{\FancyVerbFormatLine}[1]{%
  \fcolorbox{DarkBlue}{LightGray}{#1}}
\renewcommand{\FancyVerbFormatText}[1]{\textcolor{Green}{#1}}

\begin{Verbatim}[breaklines]
Some text that proceeds for a while and finally wraps onto another line
Some more text
\end{Verbatim}
```

```
Some text that proceeds for a while and finally wraps onto
↪  another line
Some more text
```

5 Reimplemented commands and environments

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 `\fvset{extra=false}` or by using `extra=false` in the optional arguments to a command or environment.

5.1 `\Verb`

`\Verb*[`⟨*options*⟩`]`⟨*delim char or {*⟩⟨*text*⟩⟨*delim char or }*⟩

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

`\Verb` is compatible with `breaklines` and the relevant line-breaking options.

Like the original fancyvrb implementation, the new `\Verb` can be starred (`*`) as a shortcut for `showspaces`, and accepts optional arguments.

Delimiters A repeated character like normal `\verb`, or a pair of curly braces `{...}`. If curly braces are used, then ⟨*text*⟩ cannot contain unpaired curly braces. Note that curly braces should be preferred when using `\Verb` inside other commands, and curly braces are *required* when `\Verb` is in a movable argument, such as in a `\section`. Non-ASCII characters now work as delimiters under pdfTeX with inputenc using UTF-8.[3] For example, `\Verb§verb§` now works as expected.

[3] 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.

Limitations inside other commands While the new `\Verb` does work inside arbitrary other commands, there are a few limitations.

- `#` and `%` cannot be used. If you need them, consider `\EscVerb` or perhaps `\SaveVerb` plus `\UseVerb`.
- Curly braces are only allowed in pairs.
- Multiple adjacent spaces will be collapsed into a single space.
- 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, `\Verb{\r \n}` becomes `\r\n`, but `\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.
- A single `^` is fine, but avoid `^^` because it will serve as an escape sequence for an ASCII command character.

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

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*[`⟨*options*⟩`]{`⟨*name*⟩`}`⟨*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*[`⟨*options*⟩`]{`⟨*name*⟩`}`

`\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*[`⟨*options*⟩`]{`⟨*backslash-escaped text*⟩`}`

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).

Delimiters 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.

[4] Allowing backslash escapes of letters would lead to ambiguity regarding spaces; see `\Verb`.

- Additional symbols or punctuation characters may require escaping if they are made `\active`, depending on their definitions.

Using in movable arguments `\EscVerb` works automatically in movable 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`.

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`.

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 `#`, `{`, `}`, `%`, `[`, `]`; the backslash `\` may be obtained via `\\` and the space via `\space`).[5]

For an alternative, see `breakbefore`. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` 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

[5] `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. fvextra 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.

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.

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

```
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCould⌋
↪  NeverFitOnOneLine'
```

`breakaftergroup` (boolean) (default: `true`)
When `breakafter` is used, group all adjacent identical characters together, and only allow a break after the last character. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` must both have the same setting.

`breakaftersymbolpre` (string) (default: `\,\footnotesize\ensuremath{_\rfloor}, ⌋`)
The symbol inserted pre-break for breaks inserted by `breakafter`.

`breakaftersymbolpost` (string) (default: ⟨*none*⟩)
The symbol inserted post-break for breaks inserted by `breakafter`.

`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.

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

```
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeve⌋
↪  rFitOnOneLine'
```

`breakanywheresymbolpre` (string) (default: `\,\footnotesize\ensuremath{_\rfloor}, ⌋`)
The symbol inserted pre-break for breaks inserted by `breakanywhere`.

`breakanywheresymbolpost` (string) (default: ⟨*none*⟩)
The symbol inserted post-break for breaks inserted by `breakanywhere`.

`breakautoindent` (boolean) (default: `true`)
When a line is broken, automatically indent the continuation lines to the indentation level of the first line. When `breakautoindent` and `breakindent` are used together, the indentations add. This indentation is combined with `breaksymbolindentleft` to give the total actual left indentation.

`breakbefore` (string) (default: ⟨*none*⟩)
Break lines before specified characters, not just at spaces, when `breaklines=true`. For example, `breakbefore=A` would allow breaks before capital A's. Special characters given to `breakbefore` should be backslash-escaped (usually `#`, `{`, `}`, `%`, `[`, `]`; the backslash `\` may be obtained via `\\` and the space via `\space`).[6]

For an alternative, see `breakafter`. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` 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.

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

```
some_string = 'SomeTextThatGoesOn⌋
↪  AndOnForSoLongThatItCouldNeverFitOnOneLine'
```

`breakbeforegroup` (boolean) (default: `true`)
When `breakbefore` is used, group all adjacent identical characters together, and only allow a break before the first character. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` must both have the same setting.

`breakbeforesymbolpre` (string) (default: `\,\footnotesize\ensuremath{_\rfloor},` ⌋)
The symbol inserted pre-break for breaks inserted by `breakbefore`.

`breakbeforesymbolpost` (string) (default: ⟨*none*⟩)
The symbol inserted post-break for breaks inserted by `breakbefore`.

`breakindent` (dimension) (default: ⟨*breakindentnchars*⟩)

[6] `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. fvextra defines special character escapes that are activated for `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. `breakbefore` is not catcode-sensitive.

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.

By default, automatic breaks occur at spaces. Use `breakanywhere` to enable breaking anywhere; use `breakbefore` and `breakafter` for more fine-tuned breaking.

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

...text.

```
def f(x):
    return 'Some text ' +
    ↪  str(x)
```

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,
                 breakautoindent=false,
                 breaksymbolleft=\raisebox{0.8ex}{
                   \small\reflectbox{\carriagereturn}},
                 breaksymbolindentleft=0pt,
                 breaksymbolsepleft=0pt,
                 breaksymbolright=\small\carriagereturn,
                 breaksymbolindentright=0pt,
                 breaksymbolsepright=0pt]
def f(x):
    return 'Some text ' + str(x) + ' some more text ' +
    ↪  str(x) + ' even more text that goes on for a while'
\end{Verbatim}
```

```
def f(x):
    return 'Some text ' + str(x) + ' some more text ' +
↪ str(x) + ' even more text that goes on for a while'
```

Automatic line breaks will not work with `showspaces=true` unless you use `breakanywhere`, or use `breakbefore` or `breakafter` with `\space`. For example,

```
\begin{Verbatim}[breaklines, showspaces, breakafter=\space]
some_string = 'Some Text That Goes On And On For So Long That It Could Never Fit'
\end{Verbatim}
```

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

`breaksymbol` (string) (default: `breaksymbolleft`)
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: ⟨*breaksymbolindentleftnchars*⟩)
Alias for `breaksymbolindentleft`.

`breaksymbolindentnchars` (integer) (default: ⟨*breaksymbolindentleftnchars*⟩)
Alias for `breaksymbolindentleftnchars`.

`breaksymbolindentleft` (dimension) (default: ⟨*breaksymbolindentleftnchars*⟩)
The extra left indentation that is provided to make room for `breaksymbolleft`. This indentation is only applied when there is a `breaksymbolleft`.

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

`breaksymbolindentright` (dimension) (default: ⟨*breaksymbolindentrightnchars*⟩)
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: ⟨*breaksymbolsepleftnchars*⟩)
Alias for `breaksymbolsepleft`.

`breaksymbolsepnchars` (integer) (default: ⟨*breaksymbolsepleftnchars*⟩)
Alias for `breaksymbolsepleftnchars`.

`breaksymbolsepleft` (dimension) (default: ⟨*breaksymbolsepleftnchars*⟩)
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: ⟨*breaksymbolseprightnchars*⟩)
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, `\linewidth` will typically not be an exact integer multiple of the character width (assuming a fixed-width font), so the actual separation between the `breaksymbolright` and adjacent text will generally be larger than `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, `breaksymbolsepright` may be adjusted. (See the definition of `\FV@makeLineNumber` for implementation details.)

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

7.2 Line breaking and tab expansion

fancyvrb provides an `obeytabs` option that expands tabs based on tab stops rather than replacing them with a fixed number of spaces (see fancyvrb's `tabsize`). The fancyvrb implementation of tab expansion is not directly compatible with fvextra's line-breaking algorithm, but fvextra builds on the 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 `breakindent`, `breakautoindent`, and `breaksymbolindentleft` (alias `breaksymbolindent`).

A sample `Verbatim` environment that uses `obeytabs` with `breaklines` is shown below, with numbers beneath the environment indicating tab stops (`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 `breaksymbolindentleft` is equal to four monospace characters. (By default, `breakautoindent=true`, so the continuation line gets a tab plus `breaksymbolindentleft`.)

```
\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}
```

```
12345678123456781234567812345678123456781234567812345678123456781234567812345678
```

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 `breaksymbolindentleft=\temporarydimen` to the `Verbatim` environment to obtain the following:

```
        ↦First  ↦Second ↦Third  ↦And more text that goes on for a
             ↪  while until wrapping is needed
        ↦First  ↦Second ↦Third  ↦Forth
12345678123456781234567812345678123456781234567812345678123456781234567812345678
```

7.3 Advanced line breaking

7.3.1 A few notes on algorithms

`breakanywhere`, `breakbefore`, and `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 (`{...}`) and all math (`$...$`). 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 `commandchars` or `codes`. This means that math and macros that only take mandatory arguments (`{...}`) will function normally within otherwise verbatim text. However, macros that take optional arguments may not work because `[...]` is not treated specially, and thus break commands may be inserted within `[...]` depending on settings. Wrapping an entire macro, including its arguments, in a group will protect the optional argument: `{\`⟨*macro*⟩`[`⟨*oarg*⟩`]{`⟨*marg*⟩`}}`.

`breakbefore` and `breakafter` insert line breaking commands around specified characters. This process is catcode-independent; tokens are `\detokenize`d before they are checked against characters specified via `breakbefore` and `breakafter`.

7.3.2 Breaks within macro arguments

`\FancyVerbBreakStart`
`\FancyVerbBreakStop`

When `commandchars` or `codes` are used to include macros within verbatim content, the options `breakanywhere`, `breakbefore`, and `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 `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

```
\FancyVerbBreakStart ... \FancyVerbBreakStop
```

For example, suppose you have the macro

```
\newcommand{\mycmd}[1]{\_before:#1:after\_}
```

Then you would discover that line breaking does not occur:

```
\begin{Verbatim}[commandchars=\\\{\}, breaklines, breakafter=a]
\mycmd{1}\mycmd{2}\mycmd{3}\mycmd{4}\mycmd{5}
\end{Verbatim}
```

```
_before:1:after__before:2:after__before:3:after__before:4:after__before:5:after_
```

Now redefine the macro:

```
\renewcommand{\mycmd}[1]{\FancyVerbBreakStart\_before:#1:after\_\FancyVerbBreakStop}
```

This is the result:

```
\begin{Verbatim}[commandchars=\\\{\}, breaklines, breakafter=a]
\mycmd{1}\mycmd{2}\mycmd{3}\mycmd{4}\mycmd{5}
\end{Verbatim}
```

```
_before:1:after__before:2:after__before:3:after__before:4:a␣
↪  fter__before:5:after_
```

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

```
\let\originalmycmd\mycmd
\renewcommand{\mycmd}[1]{%
  \expandafter\FancyVerbBreakStart\originalmycmd{#1}\FancyVerbBreakStop}
```

Notice that in this case `\expandafter` is required, because `\FancyVerbBreakStart` does not perform 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 ... \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`.

7.3.3 Customizing break behavior

\FancyVerbBreakAnywhereBreak
\FancyVerbBreakBeforeBreak
\FancyVerbBreakAfterBreak

These macros govern the behavior of breaks introduced by `breakanywhere`, `breakbefore`, and `breakafter`. Breaks introduced by the default `breaklines`

when `showspaces=false` are standard breaks following spaces. No special commands are provided for working with them; the normal LaTeX commands for breaking should suffice.

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`:

```
\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.

`\VerbatimPygments{`⟨*literal_macro*⟩`}{`⟨*actual_macro*⟩`}`

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.

⟨*literal_macro*⟩ is the Pygments macro that literally appears in Pygments output; it corresponds to the Pygments `commandprefix`. For minted and pythontex, this is `\PYG`. ⟨*actual_macro*⟩ is the Pygments macro that should actually be used. For minted and pythontex, this is `\PYG`⟨*style*⟩. In the minted and pythontex approach, code is only highlighted once (`\PYG`), and then the style is changed by redefining the macro that literally appears (`\PYG`) to use the appropriate style macro (`\PYG`⟨*style*⟩).

`\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

fvextra modifies some fancyvrb behavior that is the result of bugs or omissions.

9.1 Visible spaces

The command `\FancyVerbSpace` defines the visible space when `showspaces=true`. The default 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 to use `\textvisiblespace`.

9.2 `obeytabs` with visible tabs and with tabs inside macro arguments

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

Various macros associated with `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 `commandchars`, etc.). In the fancyvrb implementation, using `obeytabs=true` when a tab is inside a group typically causes the entire line to vanish. fvextra patches this so that the tab is expanded and will be visible if `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}{        ⇥       ⇥Text after 1 space + 2 tabs}
\end{Verbatim}
```

```
␣⟶⟶Text␣after␣1␣space␣+␣2␣tabs
```

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}{      ⇥       ⇥2 leading tabs}
\textcolor{blue}{Text  ⇥       ⇥then 2 tabs}
\end{Verbatim}
```

```
⟶⟶2 leading tabs
Text⟶⟶then 2 tabs
```

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}
```

Verbatim $\displaystyle\frac{1}{x^2 + y^2}$ 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 fancyvrb, using a single quotation mark (`'`) in typeset math within verbatim material results in an error rather than a prime symbol ($'$).[7] fvextra redefines the behavior of the single quotation mark within math mode to fix this, so that it will become a proper prime.

The amsmath package provides a `\text` command for including normal text within math. With fancyvrb, `\text` does not behave normally when used in typeset math within verbatim material. fvextra redefines the backtick (`` ` ``) and the single quotation mark so that they function normally within `\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. fvextra also switches back to the default document font within `\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}[commandchars=\\\{\}, mathescape]
Verbatim $\displaystyle f'''(x) = \text{``Some quoted text---''}$
\end{Verbatim}
```

`Verbatim` $f'''(x) =$ “Some quoted text—”

9.4 Orphaned labels

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

9.5 `rulecolor` and `fillcolor`

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

[7] The single quotation mark is made active within verbatim material to prevent ligatures, via `\@noligs`. The default definition is incompatible with math mode.

9.6 Command lookahead tokenization

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

```
\def\FancyVerbTab{%
  \valign{%
    \vfil##\vfil\cr
    \hbox{$\scriptscriptstyle-$}\cr
    \hbox to 0pt{\hss$\scriptscriptstyle\rangle\mskip -.8mu$}\cr
    \hbox{$\scriptstyle\mskip -3mu\mid\mskip -1.4mu$}\cr}}
```

While this may be redefined directly, fvextra also defines a new option `tab`

\FancyVerbSpace

This defines the visible space character (␣) that is used when `showspaces=true`. The default definition (as patched by fvextra, section 9.1) is `\textvisiblespace`. While this may be redefined directly, fvextra also defines a new option `space`.

Version History

v1.4 (2019/02/04)

- Reimplemented `\Verb`. It now works as expected inside other commands (with a few limitations), including in movable arguments, and is compatible with `hyperref` for things like PDF bookmarks. It now supports `breaklines` and relevant line-breaking options.
- Reimplemented `\SaveVerb` and `\UseVerb` to be equivalent to the new `\Verb`. The new option `retokenize` allows saved verbatim material to be retokenized under new `commandchars` and `codes` when it is inserted with `\UseVerb`.
- New command `\EscVerb` works like the reimplemented `\Verb`, except that special characters can be escaped with a backslash. It works inside other commands without any limitations, including in movable arguments, and is compatible with `hyperref` for things like PDF bookmarks.

- Added `extra` option for switching between the reimplemented `\Verb`, `\SaveVerb`, `\UseVerb` and the original `fancyvrb` definitions. Reimplemented versions are used by default. This option will apply to any future reimplemented commands and environments.
- New command `\fvinlineset` only applies options to commands related to typesetting verbatim inline, like `\Verb`, `\SaveVerb`, `\UseVerb`. It only works with commands that are defined or reimplemented by `fvextra`. It overrides options from `\fvset`.
- Patched `fancyvrb` so that `\Verb` (either reimplemented version or original) can use characters like `%` for delimiters when used outside any commands.
- `obeytabs` now works with the `calc` package's redefined `\setcounter`. Since `minted` loads `calc`, this also fixes `minted` compatibility (`minted` #221).
- Added new option `fontencoding` (`minted` #208).
- `highlightlines` now works correctly with `frame` (#7).

v1.3.1 (2017/07/08)

- `beameroverlays` now works with `VerbatimOut`.

v1.3 (2017/07/08)

- Added `beameroverlays` option, which enables `beamer` overlays using the `<` and `>` characters.
- Added options `breakindentnchars`, `breaksymbolsepleftnchars` (alias `breaksymbolsepnchars`), `breaksymbolseprightnchars`, `breaksymbolindentleftnchars` (alias `breaksymbolindentnchars`), and `breaksymbolindentrightnchars`. These are identical to the pre-existing options without the `nchars` suffix, except that they allow indentation to be specified as an integer number of characters rather than as a dimension. As a result of these new options, `\settowidth` is no longer used in the preamble, resolving some font incompatibilities (#4).
- Clarified in the docs that `breaksymbolsepright` is a *minimum*, rather than exact, distance.

v1.2.1 (2016/09/02)

- The package is now compatible with classes and packages that redefine `\raggedright`.
- Fixed a bug that introduced extra space in inline contexts such as `\mintinline` when `breaklines=true` (#3).

v1.2 (2016/07/20)

- Added support for line breaking when working with Pygments for syntax highlighting.

- The default `highlightcolor` is now defined with `rgb` for compatibility with the `color` package. Fixed a bug in the conditional color definition when `color` and `xcolor` are not loaded before `fvextra`.

v1.1 (2016/07/14)

- The options `rulecolor` and `fillcolor` now accept color names directly; using `\color{<color_name>}` is no longer necessary, though it still works.
- Added `tabcolor` and `spacecolor` options for use with `showtabs` and `showspaces`.
- Added `highlightlines` option that takes a line number or range of line numbers and highlights the corresponding lines. Added `highlightcolor` option that controls hightlighting color.
- `obeytabs` no longer causes lines to vanish when tabs are inside macro arguments. Tabs and spaces inside a macro argument but otherwise at the beginning of a line are expanded correctly. Tabs inside a macro argument that are preceded by non-whitespace characters (not spaces or tabs) are expanded based on the starting position of the run of whitespace in which they occur.
- The line breaking options `breakanywhere`, `breakbefore`, and `breakafter` now work with multi-byte UTF-8 code points under pdfTeX with `inputenc`. They were already fully functional under XeTeX and LuaTeX.
- Added `curlyquotes` option, which essentially disables the `uquote` package.

v1.0 (2016/06/28)

- Initial release.

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.

```
\RequirePackage{ifthen}
\RequirePackage{etoolbox}
\RequirePackage{fancyvrb}
\RequirePackage{upquote}
\AtEndPreamble{%
  \ifx\encodingdefault\upquote@OTone
    \ifx\ttdefault\upquote@cmtt\else\RequirePackage{textcomp}\fi
```

```
  \else
    \RequirePackage{textcomp}
  \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@FVSpaceToken Macro with the same definition as fancyvrb's active space. Useful for `\ifx` comparisons with `\@ifnextchar` lookaheads.

```
\def\FV@FVSpaceToken{\FV@Space}
```

\FV@FVTabToken Macro with the same definition as fancyvrb's active tab. Useful for `\ifx` comparisons with `\@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`:

```
\def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&%
  \do\#\do\^\do\_\do\%\do\~}
```

If a command takes an argument delimited by a given symbol, but that symbol has been made `\active` and defined as `\outer` (perhaps it is being used as a short `\verb`), then changing the symbol's catcode is the only way to use it as a delimiter.

```
\def\FVExtraDoSpecials{%
  \do\ \do\!\do\"\do\#\do\$\do\%\do\&\do\'\do\(\do\)\do\*\do\+\do\,\do\-%
  \do\.\do\/\do\:\do\;\do\<\do\=\do\>\do\?\do\@\do\[\do\\\do\]\do\^\do\_%
  \do\`\do\{\do\|\do\}\do\~}
```

\FV@Special:<char> Create macros for all printable, non-alphanumeric ASCII characters. This is used in creating backslash escapes that can only be applied to ASCII symbols and punctuation; these macros serve as `\ifcsname` lookups for valid escapes.

```
\begingroup
\def\do#1{%
  \expandafter\global\expandafter
  \let\csname FV@Special:\expandafter\@gobble\detokenize{#1}\endcsname\relax}
\FVExtraDoSpecials
\endgroup
```

12.2.3 Sentinels

Sentinel macros are needed for scanning tokens.

There are two contexts in which sentinels may be needed. In delimited macro arguments, such as `\def\macro#1\sentinel{...}`, a sentinel is needed as the delimiter. Because the delimiting macro need not be defined, special delimiting macros need not be created for this case. The important thing is to ensure that the macro name is sufficiently unique to avoid collisions. Typically, using `\makeatletter` to allow something like `\@sentinel` will be sufficient. For added security, additional characters can be given catcode 11, to allow things like `\@sent!nel`.

The other context for sentinels is in scanning through a sequence of tokens that is delimited by a sentinel, and using `\ifx` comparisons to identify the sentinel and stop scanning. In this case, using an undefined macro is risky. Under normal conditions, the sequence of tokens could contain an undefined macro due to mistyping. In some 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 resistent to collisions is needed.

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

`\FV@Sentinel` This is the standard fvextra `\ifx` comparison sentinel. It expands to the control word `\FV@<Sentinel>`, which is very unlikely to be in any other macro since it requires that `@`, `<`, and `>` 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.

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

12.2.4 Active character definitions

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

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

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

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

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

```
\begingroup
\catcode`\^^M=\active%
\gdef\FV@DefEOLEmpty{\def^^M{}}%
\endgroup
```

`\FV@OuterDefSTXEmpty` 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.

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

`\FV@OuterDefETXEmpty` 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.

```
\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.

`\ifFV@pdfTeXinputenc` 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 `\ifx` 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

```
\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.

```
\newif\ifFV@pdfTeXinputenc
\FV@pdfTeXinputencfalse
\ifcsname pdfmatch\endcsname
\ifx\pdfmatch\relax
\else
  \@ifpackageloaded{inputenc}%
   {\ifcsname inputencodingname\endcsname
    \ifx\inputencodingname\relax
    \else
      \FV@pdfTeXinputenctrue
    \fi\fi}
   {}%
\fi\fi
```

Define UTF macros conditionally:

```
\ifFV@pdfTeXinputenc
```

\FV@U8:<byte> 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
```

\FV@UTF@two@octets
\FV@UTF@three@octets
\FV@UTF@four@octets

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.

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

End conditional creation of `UTF` macros:

```
\fi
```

12.4 Reading and processing command arguments

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 fancyvrb commands like `\Verb`. They may also be used in other packages; minted and pythontex use them for handling inline code.

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

12.4.1 Tokenization and lookahead

`\FVExtra@ifnextcharVArg`

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` }%
  \let\do\@makeother\FVExtraDoSpecials
  \catcode`\ =\FV@TmpSpaceCat\relax
  \catcode`\{=1
  \catcode`\}=2
  \@ifnextchar#1{\endgroup#2}{\endgroup#3}}
```

`\FVExtra@ifstarVArg`

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*⟩.

```
\newcommand{\FVExtraReadOArgBeforeVArg}[2][]{%
  \FVExtra@ifnextcharVArg[%
   {\FV@ReadOArgContinue{#2}}%
   {\FV@ReadOArgContinue{#2}[#1]}}
```

\FVExtraReadOArgBeforeVEnv Read an optional argument that comes before the contents of a verbatim environment, after the \begin{⟨*environment*⟩} but before the start of the next line where the verbatim content begins. 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 \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. See the definition of \FV@BeginScanning in fancyvrb for an example of doing this:

```
\begingroup
\catcode`\^^M=\active
  \gdef\FV@BeginScanning#1^^M{%
    \def\@tempa{#1}\ifx\@tempa\@empty\else\FV@BadBeginError\fi%
    \FV@GetLine}%
\endgroup
```

\@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.

```
\newcommand{\FVExtraReadOArgBeforeVEnv}[2][]{%
  \begingroup
  \catcode`\^^M=\active
  \@ifnextchar[%
```

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

\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).

```
\def\FVExtraReadVArg#1{%
  \begingroup
  \ifFV@pdfTeXinputenc
    \ifdefstring{\inputencodingname}{utf8}%
     {\let\FV@ReadVArg@Char\FV@ReadVArg@Char@UTF}%
     {}%
  \fi
  \edef\FV@TmpSpaceCat{\the\catcode` }%
  \let\do\@makeother\FVExtraDoSpecials
  \catcode`\^^B=\active
  \FV@OuterDefSTXEmpty
  \catcode`\^^C=\active
  \FV@OuterDefETXEmpty
  \catcode`\^^M=\active
  \FV@OuterDefEOLEmpty
  \begingroup
  \catcode`\ =\FV@TmpSpaceCat\relax
  \catcode`\{=1
  \catcode`\}=2
  \@ifnextchar\bgroup
   {\endgroup
    \catcode`\{=1
    \catcode`\}=2
    \catcode`\ =\active
```

```
    \catcode`\^^I=\active
    \FV@ReadVArg@Group{#1}\FV@EOL}%
  {\endgroup
    \catcode`\ =\active
    \catcode`\^^I=\active
    \FV@ReadVArg@Char{#1}\FV@EOL}}
```

`\FV@ReadVArg@Group` 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.)

```
\def\FV@ReadVArg@Group#1#2#3{%
  \endgroup
  #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 singler 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.

```
\def\FV@ReadVArg@Char#1#2#3{%
  \expandafter\expandafter\expandafter
  \if\expandafter\expandafter\expandafter\relax\expandafter\@gobble\detokenize{#3}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\PackageError{fvextra}%
    {Verbatim delimiters must be single characters, not commands}%
    {Try a different delimiter}}%
  \def\FV@ReadVArg@Char@i##1##2##3#3{%
    \endgroup
    ##1{##3}}%
  \FV@ReadVArg@Char@i{#1}\FV@EOL}%
```

Alternate implementation for pdfTeX with inputenc using UTF-8

Start conditional creation of macros:

```
\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.

```
\def\FV@ReadVArg@Char@UTF#1#2#3{%
  \expandafter\expandafter\expandafter
  \if\expandafter\expandafter\expandafter\relax\expandafter\@gobble\detokenize{#3}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\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
  {\def\FV@UTF@octets@after##1{\FV@ReadVArg@Char@UTF@i{#1}{##1}}%
   \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{%
    \endgroup
    ##1{##3}}%
  \FV@ReadVArg@Char@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 processing, and is `\protected`. `\reader` is an expandable macro that reads all of `\robustcmd`'s arguments, then wraps them in `\FVExtraAlwaysUnexpanded`. When `\FVExtraAlwaysUnexpanded{`⟨*args*⟩`}` is expanded, the result is always `\FVExtraAlwaysUnexpanded{`⟨*args*⟩`}`. `\FVExtraRobustCommand` is `\protected` and manages everything in a context-sensitive manner.

- In a normal context, `\FVExtraRobustCommand` reads two arguments, which will be `\robustcmd` and `\reader`. It detects that `\reader` has not expanded to `\FVExtraAlwaysUnexpanded{`⟨*args*⟩`}`, so it discards `\reader` and reinserts `\robustcmd` so that it can operate normally.
- In an expansion-only context, neither `\FVExtraRobustCommand` nor `\robustcmd` will expand, because both are `\protected`. `\reader` will read `\robustcmd`'s arguments and protect them with `\FVExtraAlwaysUnexpanded`. When this is used later in a normal context, `\FVExtraRobustCommand` reads two arguments, which will be `\robustcmd` and `\FVExtraAlwaysUnexpanded`. It detects that `\reader` did expand, so it discards `\FVExtraAlwaysUnexpanded` and reads its argument to discard the wrapping braces. Then it reinserts `\robustcmd`⟨*args*⟩ so that everything can proceed as if expansion had not occurred.

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

`\FVExtraAlwaysUnexpanded` Always expands to itself, thanks to `\unexpanded`.

```
\long\def\FVExtraAlwaysUnexpanded#1{%
  \unexpanded{\FVExtraAlwaysUnexpanded{#1}}}
\FVExtrapdfstringdefDisableCommands{%
  \long\def\FVExtraAlwaysUnexpanded#1{#1}}
```

`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{%
  \ifx#2\FVExtraAlwaysUnexpanded
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\booltrue{FVExtraRobustCommandExpanded}\FV@RobustCommand@i{#1}}%
  {\boolfalse{FVExtraRobustCommandExpanded}#1}}
\FVExtrapdfstringdefDisableCommands{%
  \def\FVExtraRobustCommand{}}
```

\FV@RobustCommand@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\FV@RobustCommand@i#1#2{\expandafter#1\@gobble#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#{`. This reads everything up to, but not including, the next brace group.

```
\def\FVExtraUnexpandedReadStarOArgMArg#1#{%
  \FV@UnexpandedReadStarOArgMArg@i{#1}}
```

\FV@UnexpandedReadStarOArgMArg@i

```
\def\FV@UnexpandedReadStarOArgMArg@i#1#2{%
  \FVExtraAlwaysUnexpanded{\FVExtraUnexpandedReadStarOArgMArg#1{#2}}}
\FVExtrapdfstringdefDisableCommands{%
  \makeatletter
  \def\FV@UnexpandedReadStarOArgMArg@i#1#2{#2}%
  \makeatother}
```

raUseVerbUnexpandedReadStarOArgMArg

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.

```
\def\FVExtraUseVerbUnexpandedReadStarOArgMArg#1#{%
  \FV@UseVerbUnexpandedReadStarOArgMArg@i{#1}}
```

seVerbUnexpandedReadStarOArgMArg@i

```
\def\FV@UseVerbUnexpandedReadStarOArgMArg@i#1#2{%
  \FVExtraAlwaysUnexpanded{\FVExtraUseVerbUnexpandedReadStarOArgMArg#1{#2}}}
\FVExtrapdfstringdefDisableCommands{%
  \makeatletter
  \def\FV@UseVerbUnexpandedReadStarOArgMArg@i#1#2{%
    \ifcsname FV@SVRaw@#2\endcsname
      \expandafter\expandafter\expandafter\FVExtraPDFStringVerbatimDetokenize
      \expandafter\expandafter\expandafter{\csname FV@SVRaw@#2\endcsname}%
    \fi}%
  \makeatother}
```

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

```
\def\FVExtraUnexpandedReadStarOArgBVArg#1#{%
  \FV@UnexpandedReadStarOArgBVArg@i{#1}}
```

\FV@UnexpandedReadStarOArgBVArg@i

```
\def\FV@UnexpandedReadStarOArgBVArg@i#1#2{%
  \FVExtraAlwaysUnexpanded{\FVExtraUnexpandedReadStarOArgBVArg#1{#2}}}
\FVExtrapdfstringdefDisableCommands{%
  \makeatletter
  \def\FV@UnexpandedReadStarOArgBVArg@i#1#2{%
    \FVExtraPDFStringVerbatimDetokenize{#2}}%
  \makeatother}
```

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

```
\def\FVExtraUnexpandedReadStarOArgBEscVArg#1#{%
  \FV@UnexpandedReadStarOArgBEscVArg@i{#1}}
```

V@UnexpandedReadStarOArgBEscVArg@i

```
\def\FV@UnexpandedReadStarOArgBEscVArg@i#1#2{%
  \FVExtraAlwaysUnexpanded{\FVExtraUnexpandedReadStarOArgBEscVArg#1{#2}}}
\FVExtrapdfstringdefDisableCommands{%
  \makeatletter
  \def\FV@UnexpandedReadStarOArgBEscVArg@i#1#2{%
    \FVExtraPDFStringEscapedVerbatimDetokenize{#2}}%
  \makeatother}
```

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.

\FVExtraPDFStringEscapeChar 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.

```
\def\FVExtraPDFStringEscapeChar#1{%
  \ifcsname FV@PDFStringEscapeChar@#1\endcsname
    \csname FV@PDFStringEscapeChar@#1\endcsname
  \else
    #1%
  \fi}
\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&
!catcode`!\=12!relax&
!expandafter!gdef!csname FV@PDFStringEscapeChar@\!endcsname{\\}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@%!endcsname{\%}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@(!endcsname{\(}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@)!endcsname{\)}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@^^J!endcsname{\n}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@^^M!endcsname{\r}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@^^I!endcsname{\t}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@^^H!endcsname{\b}&
!expandafter!gdef!csname FV@PDFStringEscapeChar@^^L!endcsname{\f}&
!catcode`!\=0!relax&
\endgroup
```

`\FVExtraPDFStringEscapeChars`

```
\def\FVExtraPDFStringEscapeChars#1{%
  \FV@PDFStringEscapeChars#1\FV@Sentinel}
```

`\FV@PDFStringEscapeChars`

```
\def\FV@PDFStringEscapeChars#1{%
  \ifx#1\FV@Sentinel
  \else
    \FVExtraPDFStringEscapeChar{#1}%
    \expandafter\FV@PDFStringEscapeChars
  \fi}%
```

12.4.5 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

\FVExtraVerbatimDetokenize

Detokenize tokens as if their original source was tokenized verbatim, rather than under any other catcode regime that may actually have been in place. This recovers the original source when tokenization was verbatim. Otherwise, it recovers the closest approximation of the source that is possible given information loss during tokenization (for example, adjacent space characters may be merged into a single space token). This is useful in constructing nearly verbatim commands that can be used inside other commands. It functions in an expansion-only context ("fully expandable," works in `\edef`).

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 `^^C` before using `\FVExtraVerbatimDetokenize`:

```
\begingroup
\catcode`\^^C=\active
\def^^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{\macroA\macroB{}\csname name\endcsname123}
```

yields

```
\macroA \macroB {}\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{\macroA\macroB{}\csname name\endcsname123}
```

should yield

```
\macroA\macroB{}\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`.

```
\gdef\FVExtraVerbatimDetokenize#1{%
  \FV@VDetok@Scan{}#1^^C \FV@<Sentinel>}
```

`\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).

- `\FV@VDetok@Scan` searches for spaces to replace. After any spaces in ⟨*tokens*⟩ have been handled, the space in `^^C␣\FV@<Sentinel>` triggers space processing. When `\FV@VDetok@Scan` detects the sentinel macro `\FV@<Sentinel>`, scanning stops.
- The `{}` protects the beginning of ⟨*tokens*⟩, so that if ⟨*tokens*⟩ is a group, its braces won't be gobbled. Later, the inserted `{}` must be stripped so that it does not become part the processed ⟨*tokens*⟩.
- `^^C` is a convenient separator between ⟨*tokens*⟩ and the rest of the sentinel sequence.
  - Since `\FV@VDetok@Scan` has delimited arguments, a leading catcode 10 space in ⟨*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 `^^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 `^^C␣\FV@<Sentinel>` may have to be inserted into processing multiple times (for example, in recursive handling of groups).

– `\FVExtraReadVArg` defines `^^C` as `\outer`, so any verbatim argument read by it is guaranteed not to contain `^^C`. This is in contrast to `\active` ASCII symbols and to two-character sequences `<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 `^^C` that would clash (thanks, `ripgrep`!). As a control character, it should not be in common use except as a sentinel or for similar special purposes.

If ⟨*tokens*⟩ is empty or contains no spaces, then `#1` will contain `{}`⟨*tokens*⟩`^^C` and `#2` will be empty. Otherwise, `#1` will contain `{}`⟨*tokens_to_space*⟩ and `#2` will contain ⟨*tokens_after_space*⟩`^^C`␣.

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

Note that `\FV@<Sentinel>` is removed during each space search, and thus must be reinserted in `\FV@VDetok@ScanCont`. It would be possible to use the macro signature `#1 #2` instead of `#1 #2\FV@<Sentinel>`, and then do an `\ifx` test on `#2` for `\FV@<Sentinel>`. However, that is problematic, because `#2` may contain an arbitrary sequence of arbitrary tokens, so it cannot be used safely without `\detokenize`.

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

`\FV@VDetok@ScanEnd` 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, `\@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>}`.

```
\gdef\FV@VDetok@ScanEnd#1^^C{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
```

```
{\expandafter\FV@VDetok@ScanGroup\@gobble#1{\FV@<Sentinel>}}}
```

\FV@VDetok@ScanCont

Continue scanning after removing a space in `\FV@VDetok@Scan`.

`#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 `\@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.

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

\FV@VDetok@ScanGroup

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

\FV@VDetok@ScanGroup@i

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@ScanGroup@i#1{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FV@VDetok@ScanEmptyGroup}%
  {\FV@VDetok@ScanGroup@ii{}#1\FV@<Sentinel>^^C}}
```

`\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)
\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␣\FV@<Sentinel>` is needed, though. Then group scanning must continue.

```
\begingroup
\catcode`\(=1
\catcode`\)=2
\catcode`\{=12
\catcode`\}=12
\gdef\FV@VDetok@ScanGroup@ii#1\FV@<Sentinel>#2^^C(%
  \if\relax\detokenize(#2)\relax
    \expandafter\@firstofone
  \else
    \expandafter\@gobble
  \fi
  ({\FV@VDetok@Scan#1^^C \FV@<Sentinel>}\FV@VDetok@ScanGroup))
\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
  {\expandafter\FV@VDetok@ScanToken@i\detokenize{#1}^^C \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 ⟨*detokenized*⟩`^^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
  {\FV@VDetok@ScanTokenNoSpace#1}%
  {\FV@VDetok@ScanTokenWithSpace{#1}}}
```

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

```
\gdef\FV@VDetok@ScanTokenNoSpace#1^^C{#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
  {\FV@VDetok@ScanTokenActiveSpace}%
  {\FV@VDetok@ScanTokenWithSpace@i#1\FV@<Sentinel>}}
```

`\FV@VDetok@ScanTokenActiveSpace`

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

`\FV@VDetok@ScanTokenWithSpace@i` 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.

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

```
    \expandafter\@secondoftwo
  \fi
  {\FV@VDetok@ScanTokenEscSpace{#1}}%
  {\FV@VDetok@ScanTokenCW{#1#2}}}
```

\FV@VDetok@ScanTokenEscSpace

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

\FV@VDetok@ScanTokenCW 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 \FV@VDetok@ScanToken#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\FV@VDetok@ScanTokenCW#1#2{%
\ifcat\noexpand#2a%
\expandafter\@firstoftwo%
\else%
\expandafter\@secondoftwo%
\fi%
{#1 \FV@VDetok@ScanToken#2}%
{#1\FV@VDetok@ScanToken#2}}%
\endgroup
```

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

FVExtraPDFStringVerbatimDetokenize This is identical to \FVExtraVerbatimDetokenize, 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\FVExtraPDFStringVerbatimDetokenize#1{%
  \FV@PDFStrVDetok@Scan{}#1^^C \FV@<Sentinel>}
```

\FV@PDFStrVDetok@Scan

```
\gdef\FV@PDFStrVDetok@Scan#1 #2\FV@<Sentinel>{%
  \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
```

```
  \fi
  {\FV@PDFStrVDetok@ScanEnd#1}%
  {\FV@PDFStrVDetok@ScanCont{#1}{#2}}}

\FV@PDFStrVDetok@ScanEnd
\gdef\FV@PDFStrVDetok@ScanEnd#1^^C{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\expandafter\FV@PDFStrVDetok@ScanGroup\@gobble#1{\FV@<Sentinel>}}}

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

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

\FV@PDFStrVDetok@ScanGroup@i
\gdef\FV@PDFStrVDetok@ScanGroup@i#1{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FV@PDFStrVDetok@ScanEmptyGroup}%
  {\FV@PDFStrVDetok@ScanGroup@ii{}#1\FV@<Sentinel>^^C}}

\FV@PDFStrVDetok@ScanEmptyGroup
\begingroup
\catcode`\(=1
\catcode`\)=2
\catcode`\{=12
\catcode`\}=12
\gdef\FV@PDFStrVDetok@ScanEmptyGroup({}\FV@PDFStrVDetok@ScanGroup)
\endgroup
```

\FV@PDFStrVDetok@ScanGroup@ii

```
\begingroup
\catcode`\(=1
\catcode`\)=2
\catcode`\{=12
\catcode`\}=12
\gdef\FV@PDFStrVDetok@ScanGroup@ii#1\FV@<Sentinel>#2^^C(%
  \if\relax\detokenize(#2)\relax
    \expandafter\@firstofone
  \else
    \expandafter\@gobble
  \fi
  ({\FV@PDFStrVDetok@Scan#1^^C \FV@<Sentinel>}\FV@PDFStrVDetok@ScanGroup))
\endgroup
```

\FV@PDFStrVDetok@ScanToken

```
\gdef\FV@PDFStrVDetok@ScanToken#1{%
  \ifx\FV@Sentinel#1%
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\expandafter\FV@PDFStrVDetok@ScanToken@i\detokenize{#1}^^C \FV@<Sentinel>}}
```

\FV@PDFStrVDetok@ScanToken@i

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

\FV@PDFStrVDetok@ScanTokenNoSpace This is modified to use \FVExtraPDFStringEscapeChars.

```
\gdef\FV@PDFStrVDetok@ScanTokenNoSpace#1^^C{%
  \FVExtraPDFStringEscapeChars{#1}\FV@PDFStrVDetok@ScanToken}
```

FV@PDFStrVDetok@ScanTokenWithSpace

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

@PDFStrVDetok@ScanTokenActiveSpace This is modified to use \040 rather than a catcode 12 space.

```
\begingroup
\catcode`\!=0\relax
\catcode`\\=12!relax
!gdef!FV@PDFStrVDetok@ScanTokenActiveSpace{\040!FV@PDFStrVDetok@ScanToken}%
!catcode`!\=0!relax
\endgroup
```

\FV@PDFStrVDetok@ScanTokenWithSpace@i

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.

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

\FV@PDFStrVDetok@ScanTokenEscSpace

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

```
\begingroup
\catcode`\!=0\relax
\catcode`\\=12!relax
!gdef!FV@PDFStrVDetok@ScanTokenEscSpace#1{%
  !FVExtraPDFStringEscapeChar{#1}\040!FV@PDFStrVDetok@ScanToken}%
!catcode`!\=0!relax
\endgroup
```

\FV@PDFStrVDetok@ScanTokenCW

This is modified to add `\FVExtraPDFStringEscapeChars`.

```
\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}}
\endgroup
```

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

\FVExtraEscapedVerbatimDetokenize

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 backslash 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 backslash \ 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 backslash 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>}}}
```

`\FV@EscVDetok@ScanCont`

```
\begingroup
\catcode`\ =12%
\gdef\FV@EscVDetok@ScanCont#1#2{%
\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>}%
\endgroup
```

```
\FV@EscVDetok@ScanGroup
570 \gdef\FV@EscVDetok@ScanGroup#1#{%
571   \FV@EscVDetok@ScanToken#1\FV@Sentinel
572   \FV@EscVDetok@ScanGroup@i}

\FV@EscVDetok@ScanGroup@i
573 \gdef\FV@EscVDetok@ScanGroup@i#1{%
574   \if\relax\detokenize{#1}\relax
575     \expandafter\@firstoftwo
576   \else
577     \expandafter\@secondoftwo
578   \fi
579   {\FV@EscVDetok@ScanEmptyGroup}%
580   {\FV@EscVDetok@ScanGroup@ii{}#1\FV@<Sentinel>^^C}}

\FV@EscVDetok@ScanEmptyGroup
581 \begingroup
582 \catcode`\(=1
583 \catcode`\)=2
584 \catcode`\{=12
585 \catcode`\}=12
586 \gdef\FV@EscVDetok@ScanEmptyGroup({}\FV@EscVDetok@ScanGroup)
587 \endgroup

\FV@EscVDetok@ScanGroup@ii
588 \begingroup
589 \catcode`\(=1
590 \catcode`\)=2
591 \catcode`\{=12
592 \catcode`\}=12
593 \gdef\FV@EscVDetok@ScanGroup@ii#1\FV@<Sentinel>#2^^C(%
594   \if\relax\detokenize(#2)\relax
595     \expandafter\@firstofone
596   \else
597     \expandafter\@gobble
598   \fi
599   ({\FV@EscVDetok@Scan#1^^C \FV@<Sentinel>}\FV@EscVDetok@ScanGroup))
600 \endgroup

\FV@EscVDetok@ScanToken
601 \gdef\FV@EscVDetok@ScanToken#1{%
602   \ifx\FV@Sentinel#1%
603     \expandafter\@gobble
604   \else
605     \expandafter\@firstofone
606   \fi
607   {\expandafter\FV@EscVDetok@ScanToken@i\detokenize{#1}^^C \FV@<Sentinel>}}

\FV@EscVDetok@ScanToken@i
```

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

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 `\FV@REscVDetok@<name>`. They are used in `\FVExtraDetokenizeREscVArg`. The alternate `\FV@REscVDetok@<name>` macros replace invalid escape sequences with the undefined `\FV@<InvalidEscape>`, which is later scanned for with a delimited macro.

`\FV@EscVDetok@ScanTokenNoSpace` This was modified from `\FV@VDetok@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}}
```

`\FV@REscVDetok@ScanTokenNoSpace`

```
\gdef\FV@REscVDetok@ScanTokenNoSpace#1#2^^C{%
  \if\relax\detokenize{#2}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {#1\FV@EscVDetok@ScanToken}%
  {\ifcsname FV@Special:\detokenize{#2}\endcsname#2\else\noexpand\FV@<InvalidEscape>\fi
   \FV@EscVDetok@ScanToken}}
```

`\FV@EscVDetok@ScanTokenWithSpace`

```
\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>}}
```

\FV@EscVDetok@ScanTokenActiveSpace

```
\begingroup
\catcode`\ =12%
\gdef\FV@EscVDetok@ScanTokenActiveSpace{ \FV@EscVDetok@ScanToken}%
\endgroup
```

\FV@EscVDetok@ScanTokenWithSpace@i 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@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}}}
```

FV@REscVDetok@ScanTokenWithSpace@i

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

\FV@EscVDetok@ScanTokenEscSpace This is modified to drop #1, which will be the backslash.

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

```
\gdef\FVExtraPDFStringEscapedVerbatimDetokenize#1{%
  \FV@PDFStrEscVDetok@Scan{}#1^^C \FV@<Sentinel>}
```

\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

```
\gdef\FV@PDFStrEscVDetok@ScanEnd#1^^C{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\expandafter\FV@PDFStrEscVDetok@ScanGroup\@gobble#1{\FV@<Sentinel>}}}
```

\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 preceeding or following token could be an escaped space.

```
\begingroup
\catcode`\!=0\relax
\catcode`\\=12!relax
!gdef!FV@PDFStrEscVDetok@ScanCont#1#2{%
  !if!relax!detokenize{#1}!relax
    !expandafter!@gobble
  !else
    !expandafter!@firstofone
  !fi
  {!expandafter!FV@PDFStrEscVDetok@ScanGroup!@gobble#1{!FV@<Sentinel>}}%
  \040%<-space
```

```
    !FV@PDFStrEscVDetok@Scan{}#2!FV@<Sentinel>}%
!catcode`!\=0!relax
\endgroup

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

 \FV@PDFStrEscVDetok@ScanGroup@i
\gdef\FV@PDFStrEscVDetok@ScanGroup@i#1{%
  \if\relax\detokenize{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FV@PDFStrEscVDetok@ScanEmptyGroup}%
  {\FV@PDFStrEscVDetok@ScanGroup@ii{}#1\FV@<Sentinel>^^C}}

\FV@PDFStrEscVDetok@ScanEmptyGroup
\begingroup
\catcode`\(=1
\catcode`\)=2
\catcode`\{=12
\catcode`\}=12
\gdef\FV@PDFStrEscVDetok@ScanEmptyGroup({}\FV@PDFStrEscVDetok@ScanGroup)
\endgroup

 \FV@PDFStrEscVDetok@ScanGroup@ii
\begingroup
\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
  {\expandafter\FV@PDFStrEscVDetok@ScanToken@i\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
  {\FV@PDFStrEscVDetok@ScanTokenNoSpace#1}%
  {\FV@PDFStrEscVDetok@ScanTokenWithSpace{#1}}}
```

`V@PDFStrEscVDetok@ScanTokenNoSpace` This was modifed to add `\FVExtraPDFStringEscapeChar`

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

`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>}}
```

`DFStrEscVDetok@ScanTokenActiveSpace` This is modified to use `\040` for the space.

```
\begingroup
\catcode`\!=0\relax
\catcode`\\=12!relax
!gdef!FV@PDFStrEscVDetok@ScanTokenActiveSpace{\040!FV@PDFStrEscVDetok@ScanToken}%
!catcode`!\=0!relax
\endgroup
```

`DFStrEscVDetok@ScanTokenWithSpace@i`

```
\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.

```
\begingroup
\catcode`\!=0\relax
\catcode`\\=12!relax
!gdef!FV@PDFStrEscVDetok@ScanTokenEscSpace#1{\040!FV@PDFStrEscVDetok@ScanToken}
!catcode`!\=0!relax
\endgroup
```

\FV@PDFStrEscVDetok@ScanTokenCW This is modified to use \FVExtraPDFStringEscapeChars.

```
\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 Detokenize a verbatim argument read by \FVExtraReadVArg. This is a wrapper around \FVExtraVerbatimDetokenize that adds some additional safety by ensuring ^^C is \active with an appropriate definition, at the cost of not working in an expansion-only context. This tradeoff isn't an issue when working with \FVExtraReadVArg, because it has the same expansion limitations.

```
\gdef\FVExtraDetokenizeVArg#1#2{%
  \begingroup
  \catcode`\^^C=\active
  \let^^C\FV@Sentinel
  \edef\FV@Tmp{\FVExtraVerbatimDetokenize{#2}}%
  \expandafter\FV@DetokenizeVArg@i\expandafter{\FV@Tmp}{#1}}
\gdef\FV@DetokenizeVArg@i#1#2{%
  \endgroup
  #2{#1}}
```

\FVExtraDetokenizeEscVArg This is the same as \FVExtraDetokenizeVArg, except it is intended to work with \FVExtraReadEscVArg by using \FVExtraEscapedVerbatimDetokenize.

```
\gdef\FVExtraDetokenizeEscVArg#1#2{%
  \begingroup
  \catcode`\^^C=\active
  \let^^C\FV@Sentinel
  \edef\FV@Tmp{\FVExtraEscapedVerbatimDetokenize{#2}}%
  \expandafter\FV@DetokenizeVArg@i\expandafter{\FV@Tmp}{#1}}
```

\FVExtraDetokenizeREscVArg

```
\gdef\FVExtraDetokenizeREscVArg#1#2{%
  \begingroup
  \catcode`\^^C=\active
  \let^^C\FV@Sentinel
  \let\FV@EscVDetok@ScanTokenNoSpace\FV@REscVDetok@ScanTokenNoSpace
  \let\FV@EscVDetok@ScanTokenWithSpace@i\FV@REscVDetok@ScanTokenWithSpace@i
  \edef\FV@Tmp{\FVExtraEscapedVerbatimDetokenize{#2}}%
  \expandafter\FV@DetokenizeREscVArg@InvalidEscapeCheck\FV@Tmp\FV@<InvalidEscape>\FV@<Sentinel>
  \expandafter\FV@DetokenizeVArg@i\expandafter{\FV@Tmp}{#1}}
\gdef\FV@DetokenizeREscVArg@InvalidEscapeCheck#1\FV@<InvalidEscape>#2\FV@<Sentinel>{%
  \if\relax\detokenize{#2}\relax
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\PackageError{fvextra}%
   {Invalid backslash escape; only escape ASCII symbols and punctuation}%
   {Only use \@backslashchar <char> for ASCII symbols and punctuation}}}
```

End catcodes for this subsection:

```
\endgroup
```

12.4.6 Retokenizing detokenized arguments

\FV@RetokVArg@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 `\scantokens` during retokenization. The `\endgroup` disables catcode modifications that will have been put in place for the reading process, including making `^^C` and `^^M` `\active`.

```
\begingroup
\catcode`\^^C=\active%
\catcode`\^^M=\active%
\gdef\FV@RetokVArg@Read#1^^C^^M{%
  \endgroup%
  \def\FV@TmpRetoked{#1}}%
\endgroup
```

\FVExtraRetokenizeVArg

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

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

Inside `\scantokens` is `^^B#3^^C`. This becomes `^^B#3^^C^^M` once `\scantokens` inserts the end-of-line token. `^^B` is `\let` to `\FV@RetokVArg@Read`, rather than using `\FV@RetokVArg@Read` directly, because `\scantokens` acts as a `\write` followed by `\input`. That means that a command word like `\FV@RetokVArg@Read` will have a space inserted after it, while an `\active` character like `^^B` will not. Using `^^B` is a way to avoid needing to remove this space; it is simpler not to handle the scenario where `\FV@RetokVArg@Read` introduces a space and the detokenized characters also start with a space. The `^^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.

```
\begingroup
\catcode`\^^B=\active
\catcode`\^^C=\active
\gdef\FVExtraRetokenizeVArg#1#2#3{%
  \begingroup
  #2%
  \catcode`\^^B=\active
  \catcode`\^^C=\active
  \catcode`\^^M=\active
  \let^^B\FV@RetokVArg@Read
  \let^^C\@empty
  \FV@DefEOLEmpty
  \scantokens{^^B#3^^C}%
  \expandafter\FV@RetokenizeVArg@i\expandafter{\FV@TmpRetoked}{#1}}%
\gdef\FV@RetokenizeVArg@i#1#2{%
  #2{#1}}
\endgroup
```

12.5 Hooks

`\FV@FormattingPrep@PreHook`
`\FV@FormattingPrep@PostHook`

These are hooks for extending `\FV@FormattingPrep`. `\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 `breakbefore` and `breakafter`. The `PreHook` should typically be used, unless fancyvrb's font settings, whitespace setup, and active character definitions are needed for extension code.

```
\let\FV@FormattingPrep@PreHook\@empty
\let\FV@FormattingPrep@PostHook\@empty
\expandafter\def\expandafter\FV@FormattingPrep\expandafter{%
  \expandafter\FV@FormattingPrep@PreHook\FV@FormattingPrep\FV@FormattingPrep@PostHook}
```

`\FV@PygmentsHook`

This is a hook for turning on Pygments-related features for packages like minted and pythontex (section 12.13). It needs to be the first thing in `\FV@FormattingPrep@PreHook`, since it will potentially affect some of the later things in the hook. It is activated by `\VerbatimPygments`.

```
\let\FV@PygmentsHook\relax
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@PygmentsHook}
```

12.6 Escaped characters

\FV@EscChars

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.

```
\edef\FV@hashchar{\string#}
\edef\FV@dollarchar{\string$}
\edef\FV@ampchar{\string&}
\edef\FV@underscorechar{\string_}
\edef\FV@tildechar{\string~}
\edef\FV@leftsquarebracket{\string[}
\edef\FV@rightsquarebracket{\string]}
\newcommand{\FV@EscChars}{%
  \let\#\FV@hashchar
  \let\%\@percentchar
  \let\{\@charlb
  \let\}\@charrb
  \let\$\FV@dollarchar
  \let\&\FV@ampchar
  \let\_\FV@underscorechar
  \let\\\@backslashchar
  \let~\FV@tildechar
  \let\~\FV@tildechar
  \let\[\FV@leftsquarebracket
  \let\]\FV@rightsquarebracket
} %$ <- 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

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

\fvinlineset

```
\def\fvinlineset#1{%
  \expandafter\def\expandafter\FV@InlineKeyValues\expandafter{%
    \FV@InlineKeyValues#1,}}
```

\FV@UseInlineKeyValues

```
\def\FV@UseInlineKeyValues{%
  \expandafter\fvset\expandafter{\FV@InlineKeyValues}}
```

12.8 Reimplementations

fvextra reimplements some fancyvrb internals. The patches in section 12.10 fix bugs, handle edge cases, and extend existing functionality in logical ways, while leaving default fancyvrb 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.

FV@extra

```
\newbool{FV@extra}
```

extra

```
\define@booleankey{FV}{extra}%
 {\booltrue{FV@extra}}%
 {\boolfalse{FV@extra}}
\fvset{extra=true}
```

12.8.2 `\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 fancyvrb 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.

\Verb

```
\def\Verb{%
  \FVExtraRobustCommand\RobustVerb\FVExtraUnexpandedReadStarOArgBVArg}
```

\RobustVerb

```
\protected\def\RobustVerb{\FV@Command{}{Verb}}
\FVExtrapdfstringdefDisableCommands{%
  \def\RobustVerb{}}
```

\FVC@Verb@FV Save the original fancyvrb definition of \FVC@Verb, so that the extra option can switch back to it.

```
\let\FVC@Verb@FV\FVC@Verb
```

\FVC@Verb Redefine \FVC@Verb so that it will adjust based on extra.

```
\def\FVC@Verb{%
  \begingroup
  \FV@UseInlineKeyValues\FV@UseKeyValues
  \ifFV@extra
    \expandafter\endgroup\expandafter\FVC@Verb@Extra
  \else
    \expandafter\endgroup\expandafter\FVC@Verb@FV
  \fi}
```

\FVC@Verb@Extra fvextra reimplementation of \FVC@Verb.

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 \FVExtraUnexpandedReadStarOArgBVArg 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.

```
\def\FVC@Verb@Extra{%
  \ifbool{FVExtraRobustCommandExpanded}%
   {\@ifnextchar\bgroup
     {\FVC@Verb@Extra@i}%
     {\PackageError{fvextra}%
       {\string\Verb\space delimiters must be paired curly braces in this context}%
       {Use curly braces as delimiters}}}%
   {\FVC@Verb@Extra@i}}
```

\FVC@Verb@Extra@i

```
\def\FVC@Verb@Extra@i{%
  \begingroup
  \FVExtraReadVArg{%
    \FV@UseInlineKeyValues\FV@UseKeyValues\FV@FormattingPrep
    \FVExtraDetokenizeVArg{%
      \FVExtraRetokenizeVArg{\FVC@Verb@Extra@ii}{\FV@CatCodes}}}}
```

\FVC@Verb@Extra@ii

```
\def\FVC@Verb@Extra@ii#1{%
  \ifFV@BreakLines
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FancyVerbBreakStart#1\FancyVerbBreakStop}%
  {\mbox{#1}}%
  \endgroup}
```

12.8.3 \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 fancyvrb is

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

only the internal macros need to be reimplemented.

`\FVC@SaveVerb@FV`

```
\let\FVC@SaveVerb@FV\FVC@SaveVerb
```

`\FVC@SaveVerb`

```
\def\FVC@SaveVerb{%
  \begingroup
  \FV@UseInlineKeyValues\FV@UseKeyValues
  \ifFV@extra
    \expandafter\endgroup\expandafter\FVC@SaveVerb@Extra
  \else
    \expandafter\endgroup\expandafter\FVC@SaveVerb@FV
  \fi}
```

`\FVC@SaveVerb@Extra` In addition to following the `\Verb` implementation, this saves a raw version of the text to allow `retokenize` with `\UseVerb`. The raw version is also used for conversion to a PDF string if that is needed.

```
\def\FVC@SaveVerb@Extra#1{%
  \@namedef{FV@SV@#1}{}%
  \@namedef{FV@SVRaw@#1}{}%
  \begingroup
  \FVExtraReadVArg{%
    \FVC@SaveVerb@Extra@i{#1}}}
```

`\FVC@SaveVerb@Extra@i`

```
\def\FVC@SaveVerb@Extra@i#1#2{%
  \FV@UseInlineKeyValues\FV@UseKeyValues\FV@FormattingPrep
  \FVExtraDetokenizeVArg{%
    \FVExtraRetokenizeVArg{\FVC@SaveVerb@Extra@ii{#1}{#2}}{\FV@CatCodes}}{#2}}
```

`\FVC@SaveVerb@Extra@ii`

```
\def\FVC@SaveVerb@Extra@ii#1#2#3{%
  \global\let\FV@AfterSave\FancyVerbAfterSave
  \endgroup
  \@namedef{FV@SV@#1}{#3}%
  \@namedef{FV@SVRaw@#1}{#2}%
  \FV@AfterSave}%
```

12.8.4 \UseVerb

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

FV@retokenize
retokenize

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

```
\newbool{FV@retokenize}
\define@booleankey{FV}{retokenize}%
 {\booltrue{FV@retokenize}}{\boolfalse{FV@retokenize}}
```

\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
```

\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}
```

\FVC@UseVerb@Extra

```
\def\FVC@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
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FancyVerbBreakStart#1\FancyVerbBreakStop}%
  {\mbox{#1}}%
  \endgroup}
```

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.

```
\def\EscVerb{%
  \FVExtraRobustCommand\RobustEscVerb\FVExtraUnexpandedReadStarOArgBEscVArg}
```

\RobustEscVerb

```
\protected\def\RobustEscVerb{\FV@Command{}{EscVerb}}
\FVExtrapdfstringdefDisableCommands{%
  \def\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.

```
\def\FVC@EscVerb{%
  \@ifnextchar\bgroup
   {\FVC@EscVerb@i}%
   {\PackageError{fvextra}%
    {Invalid argument; argument must be delimited by paired curly braces}%
    {Delimit argument with curly braces}}}
```

\FVC@EscVerb@i

```
\def\FVC@EscVerb@i#1{%
  \begingroup
  \FV@UseInlineKeyValues\FV@UseKeyValues\FV@FormattingPrep
  \FVExtraDetokenizeREscVArg{%
    \FVExtraRetokenizeVArg{\FVC@EscVerb@ii}{\FV@CatCodes}}{#1}}
```

\FVC@EscVerb@ii

```
\def\FVC@EscVerb@ii#1{%
  \ifFV@BreakLines
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi
  {\FancyVerbBreakStart#1\FancyVerbBreakStop}%
  {\mbox{#1}}%
  \endgroup}
```

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 `\FVC@Verb`. This is invoked by a helper macro `\FV@Command` which allows versions of commands with customized options:

`\FV@Command{`⟨*customized_options*⟩`}{`⟨*base_command_name*⟩`}`

`\Verb` is then defined as `\def\Verb{\FV@Command{}{Verb}}`. The definition of `\FV@Command` (and `\FV@@Command` 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`.

`\FV@Command` and `\FV@@Command` 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 `\FV@Command`, 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 `\FV@Command` is incompatible with curly braces being used as delimiters in any form, this does not affect any pre-existing fancyvrb functionality.

`\FV@Command`

```
\def\FV@Command#1#2{%
  \FVExtra@ifstarVArg
   {\def\FV@KeyValues{#1,showspaces}\FV@@Command{#2}}%
   {\def\FV@KeyValues{#1}\FV@@Command{#2}}}
```

`\FV@@Command`

```
\def\FV@@Command#1{%
  \FVExtra@ifnextcharVArg[%
    {\FV@GetKeyValues{\@nameuse{FVC@#1}}}%
    {\@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.

```
\def\@CustomVerbatimCommand#1#2#3#4{%
  \begingroup\fvset{#4}\endgroup
  \@ifundefined{FVC@#3}%
   {\FV@Error{Command `\string#3' is not a FancyVerb command.}\@eha}%
   {\ifcsname Robust#3\endcsname
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi
    {\expandafter\let\expandafter\@tempa\csname #3\endcsname
     \def\@tempb##1##2##3{%
       \expandafter\def\expandafter\@tempc\expandafter{%
         \csname Robust\expandafter\@gobble\string#2\endcsname}%
       \def\@tempd####1{%
         #1{#2}{##1####1##3}}%
       \expandafter\@tempd\@tempc
       \expandafter\protected\expandafter\def\@tempc{\FV@Command{#4}{#3}}}%
     \expandafter\@tempb\@tempa}%
    {#1{#2}{\FV@Command{#4}{#3}}}}}
```

12.10.3 Visible spaces

`\FancyVerbSpace` 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 }}
```

The command is redefined in more robust and standard LaTeX form.

```
\def\FancyVerbSpace{\textvisiblespace}
```

12.10.4 obeytabs with visible tabs and with tabs inside macro arguments

`\FV@TrueTab` governs tab appearance when `obeytabs=true` and `showtabs=true`. It is redefined so that symbols with flexible width, such as `\rightarrowfill`, will work as expected. In the original fancyvrb definition, `\kern\@tempdima\hbox to\z@{...}`. The `\kern` is removed and instead the `\hbox` is given the width `\@tempdima`.

`\FV@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 fancyvrb 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 fancyvrb 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 fancyvrb 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.

FV@TrueTabGroupLevel Counter for keeping track of the group level (\currentgrouplevel) at the very beginning of a line, inside \FancyVerbFormatLine but outside \FancyVerbFormatText, which is where the tab expansion macro is invoked. This allows us to determine whether we are in a group, and expand tabs accordingly.

```
\newcounter{FV@TrueTabGroupLevel}
```

\FV@@ObeyTabs 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.

```
\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}
```

\FV@TrueTab Version that follows fancyvrb if not in a group and takes another approach otherwise.

```
\def\FV@TrueTab{%
  \ifnum\value{FV@TrueTabGroupLevel}=\the\currentgrouplevel\relax
    \expandafter\FV@TrueTab@NoGroup
  \else
    \expandafter\FV@TrueTab@Group
  \fi}
```

\FV@TrueTabSaveWidth 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.

```
\let\FV@TrueTabSaveWidth\relax
```

FV@TrueTabCounter Counter for tracking saved tabs.

```
\newcounter{FV@TrueTabCounter}
```

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

```
\def\FV@TrueTabSaveWidth@Save{%
  \expandafter\xdef\csname FV@TrueTab:Width\arabic{FV@TrueTabCounter}\endcsname{%
    \number\@tempdima}%
  \stepcounter{FV@TrueTabCounter}}
```

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

```
\def\FV@TrueTab@NoGroup{%
  \egroup
  \@tempdima=\FV@ObeyTabSize sp\relax
  \@tempcnta=\wd\FV@TabBox
  \advance\@tempcnta\FV@@ObeyTabSize\relax
  \divide\@tempcnta\@tempdima
  \multiply\@tempdima\@tempcnta
  \advance\@tempdima-\wd\FV@TabBox
  \FV@TrueTabSaveWidth
  \setbox\FV@TabBox=\hbox\bgroup
    \unhbox\FV@TabBox\hbox to\@tempdima{\hss\FV@TabChar}}
```

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.

```
\newboolean{FV@ObeyTabs@Whitespace@Tab}
```

\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.

```
\def\FV@TrueTab@Group{%
  \booltrue{FV@ObeyTabs@Whitespace@Tab}%
  \gdef\FV@TmpWhitespace{\FV@FVTabToken}%
  \FV@ObeyTabs@ScanWhitespace}
```

\FV@Space@ObeyTabs 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}}
\def\FV@TrueTab@CaptureWhitespace@Space#1{%
  \g@addto@macro\FV@TmpWhitespace{\FV@FVSpaceToken}%
  \FV@ObeyTabs@ScanWhitespace}
\def\FV@TrueTab@CaptureWhitespace@Tab#1{%
  \booltrue{FV@ObeyTabs@Whitespace@Tab}%
  \g@addto@macro\FV@TmpWhitespace{\FV@FVTabToken}%
  \FV@ObeyTabs@ScanWhitespace}
```

\FV@TrueTab@Group@Expand Yet another tab definition, this one for use in the actual expansion of tabs in whitespace. This uses the fancyvrb 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\@tempcnta\@tempdima
  \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.

```
\def\FV@ObeyTabs@ResolveWhitespace{%
  \let\FV@Space\FV@Space@Orig
  \let\FV@Tab\FV@TrueTab@Group@Expand
  \expandafter\FV@ObeyTabs@ResolveWhitespace@i\expandafter{\FV@TmpWhitespace}%
  \let\FV@Space\FV@Space@ObeyTabs
  \let\FV@Tab\FV@TrueTab}
\def\FV@ObeyTabs@ResolveWhitespace@i#1{%
  \ifbool{FV@ObeyTabs@Whitespace@Tab}%
   {\setbox\FV@TabBox@Group=\hbox{#1}\box\FV@TabBox@Group}%
   {#1}}
```

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.

```
\def\FancyVerbMathSpace{ }
```

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

```
\def\FV@SetupMathSpace{%
  \everymath\expandafter{\the\everymath\let\FV@Space\FancyVerbMathSpace}}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathSpace}
```

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.ltx` that works with active `'`. In verbatim contexts, `'` is made active by `\@noligs`.

```
\begingroup
\catcode`\'=\active
\catcode`\^=7
\gdef\FV@pr@m@s{%
  \ifx'\@let@token
    \expandafter\pr@@@s
  \else
    \ifx^\@let@token
      \expandafter\expandafter\expandafter\pr@@@t
    \else
      \egroup
    \fi
  \fi}
\endgroup
```

\FV@SetupMathFont Set the font back to default from the verbatim font.

```
\def\FV@SetupMathFont{%
  \everymath\expandafter{\the\everymath\fontfamily{\familydefault}\selectfont}}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathFont}
```

\FV@SetupMathLigs Make all characters in `\@noligs` behave normally, and switch to `\FV@pr@m@s`. The relevant definition from `latex.ltx`:

```
\def\verbatim@nolig@list{\do\`\do\<\do\>\do\,\do\'\do\-}
```

```
\def\FV@SetupMathLigs{%
  \everymath\expandafter{%
    \the\everymath
    \let\pr@m@s\FV@pr@m@s
    \begingroup\lccode`\~=`\'\lowercase{\endgroup\def~}{%
      \ifmmode\expandafter\active@math@prime\else'\fi}%
    \begingroup\lccode`\~=`\`\lowercase{\endgroup\def~}{`}%
    \begingroup\lccode`\~=`\<\lowercase{\endgroup\def~}{<}%
    \begingroup\lccode`\~=`\>\lowercase{\endgroup\def~}{>}%
    \begingroup\lccode`\~=`\,\lowercase{\endgroup\def~}{,}%
    \begingroup\lccode`\~=`\-\lowercase{\endgroup\def~}{-}%
  }%
}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@SetupMathLigs}
```

12.10.7 Ophaned label

`\FV@BeginListFrame@Lines`

When `frame=lines` is used with a label, the label can be orphaned. This overwrites the default definition to add `\penalty\@M`. The fix is attributed to http://tex.stackexchange.com/a/168021/10742.

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

`rulecolor`

```
\define@key{FV}{rulecolor}{%
  \ifstrempty{#1}%
   {\let\FancyVerbRuleColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FancyVerbRuleColor\relax}%
     {\def\@tempa{#1}%
      \FV@KVProcess@RuleColor#1\FV@Undefined}}}
\def\FV@KVProcess@RuleColor#1#2\FV@Undefined{%
  \ifx#1\color
  \else
    \expandafter\def\expandafter\@tempa\expandafter{%
      \expandafter\color\expandafter{\@tempa}}%
```

```
    \fi
    \let\FancyVerbRuleColor\@tempa}
\fvset{rulecolor=none}
```

fillcolor

```
\define@key{FV}{fillcolor}{%
  \ifstrempty{#1}%
   {\let\FancyVerbFillColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FancyVerbFillColor\relax}%
     {\def\@tempa{#1}%
      \FV@KVProcess@FillColor#1\FV@Undefined}}}
\def\FV@KVProcess@FillColor#1#2\FV@Undefined{%
  \ifx#1\color
  \else
    \expandafter\def\expandafter\@tempa\expandafter{%
      \expandafter\color\expandafter{\@tempa}}%
  \fi
  \let\FancyVerbFillColor\@tempa}
\fvset{fillcolor=none}
```

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}}
```

tab Redefine `\FancyVerbTab`.

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

tabcolor Set tab color, or allow it to adjust to surroundings (the default fancyvrb behavior). This involves re-creating the `showtabs` option to add `\FV@TabColor`.

```
\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}
```

space Redefine `\FancyVerbSpace`.

```
\define@key{FV}{space}{\def\FancyVerbSpace{#1}}
```

`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}}}}}
\define@booleankey{FV}{showspaces}%
 {\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}}}%
 {\def\FV@Space{\ }}
\fvset{spacecolor=none, showspaces=false}
```

`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`\^=7\catcode`\_=8\relax}
\FV@AddToHook\FV@CatCodesHook\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
    \catcode`\<=12\catcode`\>=12\relax}}
\FV@AddToHook\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~}{`}%
      \begingroup\lccode`\~=`\'\lowercase{\endgroup\def~}{'}}}%
   {}}
\g@addto@macro\FV@FormattingPrep@PreHook{\FancyVerbCurlyQuotes}
\fvset{curlyquotes=false}
```

fontencoding Add option for font encoding.

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

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

fancyvrb defines \FancyVerbFormatLine, which defines the formatting for each line. The introduction of line breaks introduces an issue for \FancyVerbFormatLine. Does it format the entire line, including any whitespace in the margins or behind line break symbols (that is, is it outside the \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 \parbox)? Since both might be desirable, \FancyVerbFormatLine is assigned to the entire line, and a new macro \FancyVerbFormatText is assigned to the text, within the \parbox.

An additional complication is that the fancyvrb documentation says that the default value is \def\FancyVerbFormatLine#1{#1}. But the actual default is \def\FancyVerbFormatLine#1{\FV@ObeyTabs{#1}}. That is, \FV@ObeyTabs needs to operate directly on the line to handle tabs. As a result, *all* fancyvrb commands that involve \FancyVerbFormatLine are patched, so that \def\FancyVerbFormatLine#1{#1}.

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

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

```
\def\FancyVerbFormatLine#1{#1}
```

\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.

```
\def\FancyVerbFormatText#1{#1}
```

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

```
\def\FV@ListProcessLine@NoBreak#1{%
  \hbox to \hsize{%
```

```
    \kern\leftmargin
    \hbox to \linewidth{%
      \FV@LeftListNumber
      \FV@LeftListFrame
      \FancyVerbFormatLine{%
        \FancyVerbHighlightLine{%
          \FV@ObeyTabs{\FancyVerbFormatText{#1}}}}\hss
      \FV@RightListFrame
      \FV@RightListNumber}%
    \hss}}
```

\FV@BProcessLine Redefined \FV@BProcessLine in which \FancyVerbFormatText is added and tab handling is explicit.

```
\def\FV@BProcessLine#1{%
  \hbox{\FancyVerbFormatLine{%
    \FancyVerbHighlightLine{%
      \FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}}
```

12.11.3 Line numbering

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

FV@NumberFirstLine

```
\newbool{FV@NumberFirstLine}
```

numberfirstline

```
\define@booleankey{FV}{numberfirstline}%
 {\booltrue{FV@NumberFirstLine}}%
 {\boolfalse{FV@NumberFirstLine}}
\fvset{numberfirstline=false}
```

FV@StepNumberFromFirst

```
\newbool{FV@StepNumberFromFirst}
```

stepnumberfromfirst

```
\define@booleankey{FV}{stepnumberfromfirst}%
 {\booltrue{FV@StepNumberFromFirst}}%
 {\boolfalse{FV@StepNumberFromFirst}}
\fvset{stepnumberfromfirst=false}
```

FV@StepNumberOffsetValues

```
\newbool{FV@StepNumberOffsetValues}
```

stepnumberoffsetvalues

```
\define@booleankey{FV}{stepnumberoffsetvalues}%
 {\booltrue{FV@StepNumberOffsetValues}}%
 {\boolfalse{FV@StepNumberOffsetValues}}
\fvset{stepnumberoffsetvalues=false}
```

\FV@Numbers@left 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@`).

```
\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}%
       {\@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
        \ifx\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}%
}
```

\FV@Numbers@right Redefine fancyvrb macro to account for `numberfirstline`, `stepnumberfromfirst`,

and stepnumberoffsetvalues.

```
\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
      \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
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \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\@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}%
       {\@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
        \ifx\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{%
    \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}%
       {\@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
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \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

```
\define@key{FV}{highlightlines}{\def\FV@HighlightLinesList{#1}}%
\fvset{highlightlines=}
```

highlightcolor
\FV@HighlightColor

Define color for highlighting. The default is LightCyan. A good alternative for a brighter color would be LemonChiffon.

```
\define@key{FV}{highlightcolor}{\def\FancyVerbHighlightColor{#1}}%
\let\FancyVerbHighlightColor\@empty
\ifcsname definecolor\endcsname
\ifx\definecolor\relax
\else
  \definecolor{FancyVerbHighlightColor}{rgb}{0.878, 1, 1}
  \fvset{highlightcolor=FancyVerbHighlightColor}
\fi\fi
\AtBeginDocument{%
  \ifx\FancyVerbHighlightColor\@empty
    \ifcsname definecolor\endcsname
    \ifx\definecolor\relax
    \else
    \definecolor{FancyVerbHighlightColor}{rgb}{0.878, 1, 1}
    \fvset{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
      \advance\@tempcntb\tw@
      \ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineMiddle
      \else
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineLast
      \fi
    \else
      \advance\@tempcntb\tw@
      \ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineFirst
      \else
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineSingle
      \fi
    \fi
  \else
    \let\FV@HighlightLine@Next\FancyVerbHighlightLineNormal
  \fi
  \FV@HighlightLine@Next{#1}%
}
```

`\FancyVerbHighlightLineNormal` 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.

```
\let\FancyVerbHighlightLineMiddle\FancyVerbHighlightLineFirst
```

`\FancyVerbHighlightLineLast` The last line in a multi-line range.

```
\let\FancyVerbHighlightLineLast\FancyVerbHighlightLineFirst
```

`\FancyVerbHighlightLineSingle` A single line not in a multi-line range.

```
\let\FancyVerbHighlightLineSingle\FancyVerbHighlightLineFirst
```

`\FV@HighlightLinesPrep` Process the list of lines to highlight (if any). A macro is created for each line to be highlighted. During highlighting, a line is highlighted if the corresponding macro exists. All of the macro creating is ultimately within the current environment group so it stays local. `\FancyVerbHighlightLine` is `\let` to a version that will invoke the necessary logic.

```
\def\FV@HighlightLinesPrep{%
  \ifx\FV@HighlightLinesList\@empty
  \else
    \let\FancyVerbHighlightLine\FV@HighlightLine
    \expandafter\FV@HighlightLinesPrep@i
  \fi}
\def\FV@HighlightLinesPrep@i{%
  \renewcommand{\do}[1]{%
    \ifstrempty{##1}{}{\FV@HighlightLinesParse##1-\FV@Undefined}}%
  \expandafter\docsvlist\expandafter{\FV@HighlightLinesList}}
\def\FV@HighlightLinesParse#1-#2\FV@Undefined{%
  \ifstrempty{#2}%
   {\FV@HighlightLinesParse@Single{#1}}%
   {\FV@HighlightLinesParse@Range{#1}#2\relax}}
\def\FV@HighlightLinesParse@Single#1{%
  \expandafter\let\csname FV@HighlightLine:\detokenize{#1}\endcsname\relax}
```

```
\newcounter{FV@HighlightLinesStart}
\newcounter{FV@HighlightLinesStop}
\def\FV@HighlightLinesParse@Range#1#2-{%
  \setcounter{FV@HighlightLinesStart}{#1}%
  \setcounter{FV@HighlightLinesStop}{#2}%
  \stepcounter{FV@HighlightLinesStop}%
  \FV@HighlightLinesParse@Range@Loop}
\def\FV@HighlightLinesParse@Range@Loop{%
  \ifnum\value{FV@HighlightLinesStart}<\value{FV@HighlightLinesStop}\relax
    \expandafter\let\csname FV@HighlightLine:\arabic{FV@HighlightLinesStart}\endcsname\relax
    \stepcounter{FV@HighlightLinesStart}%
    \expandafter\FV@HighlightLinesParse@Range@Loop
  \fi}
\g@addto@macro\FV@FormattingPrep@PreHook{\FV@HighlightLinesPrep}
```

12.12 Line breaking

The following code adds automatic line breaking functionality to fancyvrb's `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 `\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.

```
\newcount\FV@LoopCount
\newbox\FV@NCharsBox
\def\FV@SetToWidthNChars#1#2{%
  \FV@LoopCount=#2\relax
  \ifnum\FV@LoopCount>0
    \def\FV@NChars{}%
    \loop
    \ifnum\FV@LoopCount>0
      \expandafter\def\expandafter\FV@NChars\expandafter{\FV@NChars x}%
    \fi
    \advance\FV@LoopCount by -1
    \ifnum\FV@LoopCount>0
    \repeat
    \setbox\FV@NCharsBox\hbox{\FV@NChars}%
    #1=\wd\FV@NCharsBox
  \else
    #1=0pt\relax
  \fi
}
```

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.

```
\newboolean{FV@BreakLines}
\define@booleankey{FV}{breaklines}%
  {\FV@BreakLinestrue
    \let\FV@ListProcessLine\FV@ListProcessLine@Break}%
  {\FV@BreakLinesfalse
    \let\FV@ListProcessLine\FV@ListProcessLine@NoBreak}
\AtEndOfPackage{\fvset{breaklines=false}}
```

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

```
\def\FV@BreakLinesIndentationHook{}
\g@addto@macro\FV@FormattingPrep@PostHook{%
  \ifFV@BreakLines
    \FV@BreakLinesIndentationHook
  \fi}
```

\FV@BreakIndent Indentation of continuation lines.

\FV@BreakIndentNChars

```
\newdimen\FV@BreakIndent
\newcount\FV@BreakIndentNChars
\define@key{FV}{breakindent}{%
  \FV@BreakIndent=#1\relax
  \FV@BreakIndentNChars=0\relax}
\define@key{FV}{breakindentnchars}{\FV@BreakIndentNChars=#1\relax}
\g@addto@macro\FV@BreakLinesIndentationHook{%
  \ifnum\FV@BreakIndentNChars>0
    \FV@SetToWidthNChars{\FV@BreakIndent}{\FV@BreakIndentNChars}%
  \fi}
\fvset{breakindentnchars=0}
```

FV@BreakAutoIndent Auto indentation of continuation lines to indentation of original line. Adds to \FV@BreakIndent.

```
\newboolean{FV@BreakAutoIndent}
\define@booleankey{FV}{breakautoindent}%
  {\FV@BreakAutoIndenttrue}{\FV@BreakAutoIndentfalse}
\fvset{breakautoindent=true}
```

\FancyVerbBreakSymbolLeft The left-hand symbol indicating a break. Since breaking is done in such a way that a left-hand symbol will often be desired while a right-hand symbol may not be, a shorthand option breaksymbol is supplied. This shorthand convention is continued with other options applying to the left-hand symbol.

```
\define@key{FV}{breaksymbolleft}{\def\FancyVerbBreakSymbolLeft{#1}}
\define@key{FV}{breaksymbol}{\fvset{breaksymbolleft=#1}}
\fvset{breaksymbolleft=\tiny\ensuremath{\hookrightarrow}}
```

\FancyVerbBreakSymbolRight The right-hand symbol indicating a break.

```
\define@key{FV}{breaksymbolright}{\def\FancyVerbBreakSymbolRight{#1}}
\fvset{breaksymbolright={}}
```

\FV@BreakSymbolSepLeft Separation of left break symbol from the text.
\FV@BreakSymbolSepLeftNChars

```
\newdimen\FV@BreakSymbolSepLeft
\newcount\FV@BreakSymbolSepLeftNChars
\define@key{FV}{breaksymbolsepleft}{%
  \FV@BreakSymbolSepLeft=#1\relax
  \FV@BreakSymbolSepLeftNChars=0\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 Separation of right break symbol from the text.
\FV@BreakSymbolSepRightNChars

```
\newdimen\FV@BreakSymbolSepRight
\newcount\FV@BreakSymbolSepRightNChars
\define@key{FV}{breaksymbolsepright}{%
  \FV@BreakSymbolSepRight=#1\relax
  \FV@BreakSymbolSepRightNChars=0\relax}
\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}
```

\FV@BreakSymbolIndentLeft Additional left indentation to make room for the left break symbol.
\FV@BreakSymbolIndentLeftNChars

```
\newdimen\FV@BreakSymbolIndentLeft
\newcount\FV@BreakSymbolIndentLeftNChars
\define@key{FV}{breaksymbolindentleft}{%
  \FV@BreakSymbolIndentLeft=#1\relax
  \FV@BreakSymbolIndentLeftNChars=0\relax}
\define@key{FV}{breaksymbolindent}{\fvset{breaksymbolindentleft=#1}}
\define@key{FV}{breaksymbolindentleftnchars}{\FV@BreakSymbolIndentLeftNChars=#1\relax}
\define@key{FV}{breaksymbolindentnchars}{\fvset{breaksymbolindentleftnchars=#1}}
\g@addto@macro\FV@BreakLinesIndentationHook{%
  \ifnum\FV@BreakSymbolIndentLeftNChars>0
    \FV@SetToWidthNChars{\FV@BreakSymbolIndentLeft}{\FV@BreakSymbolIndentLeftNChars}%
  \fi}
\fvset{breaksymbolindentleftnchars=4}
```

\FV@BreakSymbolIndentRight Additional right indentation to make room for the right break symbol.
\FV@BreakSymbolIndentRightNChars

```
\newdimen\FV@BreakSymbolIndentRight
```

```
\newcount\FV@BreakSymbolIndentRightNChars
\define@key{FV}{breaksymbolindentright}{%
  \FV@BreakSymbolIndentRight=#1\relax
  \FV@BreakSymbolIndentRightNChars=0\relax}
\define@key{FV}{breaksymbolindentrightnchars}{\FV@BreakSymbolIndentRightNChars=#1\relax}
\g@addto@macro\FV@BreakLinesIndentationHook{%
  \ifnum\FV@BreakSymbolIndentRightNChars>0
    \FV@SetToWidthNChars{\FV@BreakSymbolIndentRight}{\FV@BreakSymbolIndentRightNChars}%
  \fi}
\fvset{breaksymbolindentrightnchars=4}
```

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, `\tiny`) don't escape.

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

```
\newcommand{\FancyVerbBreakSymbolLeftLogic}[1]{%
  \ifnum\value{linenumber}=1\relax\else{#1}\fi}
```

`FancyVerbLineBreakLast` 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.

```
\newcounter{FancyVerbLineBreakLast}
```

`\FV@SetLineBreakLast` Store the local line number for the last continuation line.

```
\newcommand{\FV@SetLineBreakLast}{%
  \setcounter{FancyVerbLineBreakLast}{\value{linenumber}}}
```

`\FancyVerbBreakSymbolRightLogic` Only insert a right break symbol if not on the last continuation line.

```
\newcommand{\FancyVerbBreakSymbolRightLogic}[1]{%
  \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.

```
\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.

```
\let\FancyVerbBreakStop\relax
```

`\FV@Break@Token` Macro that controls token handling between `\FancyVerbBreakStart` and `\FancyVerbBreakStop`. Initially `\let` to `\relax`; later `\let` to `\FV@Break@AnyToken` or `\FV@Break@BeforeAfterToken`

as appropriate. There is no need to `\let\FV@Break@Token\relax` when `breakanywhere`, `breakbefore`, and `breakafter` are not in use. In that case, `\FancyVerbBreakStart` and `\FancyVerbBreakStop` are `\let` to `\relax`, and `\FV@Break@Token` is never invoked.

```
\let\FV@Break@Token\relax
```

`FV@BreakAnywhere` Allow line breaking (almost) anywhere. Set `\FV@Break` and `\FV@EndBreak` to be used, and `\let \FV@Break@Token` to the appropriate macro.

```
\newboolean{FV@BreakAnywhere}
\define@booleankey{FV}{breakanywhere}%
  {\FV@BreakAnywheretrue
    \let\FancyVerbBreakStart\FV@Break
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@AnyToken}%
  {\FV@BreakAnywherefalse
    \let\FancyVerbBreakStart\relax
    \let\FancyVerbBreakStop\relax}
\fvset{breakanywhere=false}
```

`\FV@BreakBefore` Allow line breaking (almost) anywhere, but only before specified characters.

```
\define@key{FV}{breakbefore}{%
  \ifstrempty{#1}%
   {\let\FV@BreakBefore\@empty
    \let\FancyVerbBreakStart\relax
    \let\FancyVerbBreakStop\relax}%
   {\def\FV@BreakBefore{#1}%
    \let\FancyVerbBreakStart\FV@Break
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@BeforeAfterToken}%
}
\fvset{breakbefore={}}
```

`FV@BreakBeforeGroup` Determine whether breaking before specified characters is always allowed before each individual character, or is only allowed before the first in a group of identical characters.

```
\newboolean{FV@BreakBeforeGroup}
\define@booleankey{FV}{breakbeforegroup}%
 {\FV@BreakBeforeGrouptrue}%
 {\FV@BreakBeforeGroupfalse}%
\fvset{breakbeforegroup=true}
```

`\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 exntensions 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.

```
\def\FV@BreakBeforePrep{%
  \ifx\FV@BreakBefore\@empty\relax
  \else
    \gdef\FV@BreakBefore@Def{}%
    \begingroup
    \def\FV@BreakBefore@Process##1##2\FV@Undefined{%
      \expandafter\FV@BreakBefore@Process@i\expandafter{##1}%
      \expandafter\ifx\expandafter\relax\detokenize{##2}\relax
      \else
        \FV@BreakBefore@Process##2\FV@Undefined
      \fi
    }%
    \def\FV@BreakBefore@Process@i##1{%
      \g@addto@macro\FV@BreakBefore@Def{%
        \@namedef{FV@BreakBefore@Token\detokenize{##1}}{}}%
    }%
    \FV@EscChars
    \expandafter\FV@BreakBefore@Process\FV@BreakBefore\FV@Undefined
    \endgroup
    \FV@BreakBefore@Def
    \FV@BreakBeforePrep@PygmentsHook
  \fi
}
\let\FV@BreakBeforePrep@PygmentsHook\relax
```

`\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}%
   {\def\FV@BreakAfter{#1}%
    \let\FancyVerbBreakStart\FV@Break
```

```
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@BeforeAfterToken}%
}
\fvset{breakafter={}}
```

`FV@BreakAfterGroup` Determine whether breaking after specified characters is always allowed after each individual character, or is only allowed after groups of identical characters.

```
\newboolean{FV@BreakAfterGroup}
\define@booleankey{FV}{breakaftergroup}%
 {\FV@BreakAfterGrouptrue}%
 {\FV@BreakAfterGroupfalse}%
\fvset{breakaftergroup=true}
```

`\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` is important; `\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
        \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {}%
           {\PackageError{fvextra}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {\PackageError{fvextra}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}%
           {}}%
      \fi
      \g@addto@macro\FV@BreakAfter@Def{%
        \@namedef{FV@BreakAfter@Token\detokenize{##1}}{}}%
    }%
    \FV@EscChars
```

```
    \expandafter\FV@BreakAfter@Process\FV@BreakAfter\FV@Undefined
    \endgroup
    \FV@BreakAfter@Def
    \FV@BreakAfterPrep@PygmentsHook
  \fi
}
\let\FV@BreakAfterPrep@PygmentsHook\relax
```

Now that `\FV@BreakBeforePrep` and `\FV@BreakAfterPrep` are defined, add them to `\FV@FormattingPrep@PreHook`, which is the fvextra extension to `\FV@FormattingPrep`. The ordering here is important, since `\FV@BreakAfterPrep` contains compatibility checks with `\FV@BreakBeforePrep`, and thus must be used after it. Also, we have to check for the pdfTeX engine with inputenc using UTF-8, and use the `UTF` macros instead when that is the case.

```
\g@addto@macro\FV@FormattingPrep@PreHook{%
  \ifFV@pdfTeXinputenc
    \ifdefstring{\inputencodingname}{utf8}%
     {\let\FV@BreakBeforePrep\FV@BreakBeforePrep@UTF
      \let\FV@BreakAfterPrep\FV@BreakAfterPrep@UTF}%
     {}%
  \fi
  \FV@BreakBeforePrep\FV@BreakAfterPrep}
```

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

```
\define@key{FV}{breakanywheresymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAnywhereSymbolPre{}}%
    {\def\FancyVerbBreakAnywhereSymbolPre{\hbox{#1}}}}
\fvset{breakanywheresymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

`\FancyVerbBreakAnywhereSymbolPost` The post-break symbol for breaks introduced by `breakanywhere`.

```
\define@key{FV}{breakanywheresymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAnywhereSymbolPost{}}%
    {\def\FancyVerbBreakAnywhereSymbolPost{\hbox{#1}}}}
\fvset{breakanywheresymbolpost={}}
```

`\FancyVerbBreakBeforeSymbolPre` The pre-break symbol for breaks introduced by `breakbefore`.

```
\define@key{FV}{breakbeforesymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakBeforeSymbolPre{}}%
    {\def\FancyVerbBreakBeforeSymbolPre{\hbox{#1}}}}
\fvset{breakbeforesymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

`\FancyVerbBreakBeforeSymbolPost` The post-break symbol for breaks introduced by `breakbefore`.

```
\define@key{FV}{breakbeforesymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakBeforeSymbolPost{}}%
```

```
    {\def\FancyVerbBreakBeforeSymbolPost{\hbox{#1}}}}
\fvset{breakbeforesymbolpost={}}
```

`\FancyVerbBreakAfterSymbolPre` The pre-break symbol for breaks introduced by `breakafter`.

```
\define@key{FV}{breakaftersymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAfterSymbolPre{}}%
    {\def\FancyVerbBreakAfterSymbolPre{\hbox{#1}}}}
\fvset{breakaftersymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

`\FancyVerbBreakAfterSymbolPost` The post-break symbol for breaks introduced by `breakafter`.

```
\define@key{FV}{breakaftersymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAfterSymbolPost{}}%
    {\def\FancyVerbBreakAfterSymbolPost{\hbox{#1}}}}
\fvset{breakaftersymbolpost={}}
```

`\FancyVerbBreakAnywhereBreak` The macro governing breaking for `breakanywhere=true`.

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

`\FancyVerbBreakBeforeBreak` The macro governing breaking for `breakbefore=true`.

```
\newcommand{\FancyVerbBreakBeforeBreak}{%
  \discretionary{\FancyVerbBreakBeforeSymbolPre}%
   {\FancyVerbBreakBeforeSymbolPost}{}}
```

`\FancyVerbBreakAfterBreak` The macro governing breaking for `breakafter=true`.

```
\newcommand{\FancyVerbBreakAfterBreak}{%
  \discretionary{\FancyVerbBreakAfterSymbolPre}%
   {\FancyVerbBreakAfterSymbolPost}{}}
```

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.

```
\newsavebox{\FV@LineBox}
```

`\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@CleanRemainingChars#1\FV@Undefined{}
\def\FV@GetLineIndent{\afterassignment\FV@CheckIndentChar\let\FV@NextChar=}
\def\FV@CheckIndentChar{%
  \ifx\FV@NextChar\FV@Undefined\relax
    \let\FV@Next=\relax
  \else
    \ifx\FV@NextChar\FV@FVSpaceToken\relax
      \g@addto@macro{\FV@LineIndentChars}{\FV@FVSpaceToken}%
      \let\FV@Next=\FV@GetLineIndent
    \else
      \ifx\FV@NextChar\FV@FVTabToken\relax
        \g@addto@macro{\FV@LineIndentChars}{\FV@FVTabToken}%
        \let\FV@Next=\FV@GetLineIndent
      \else
        \let\FV@Next=\FV@CleanRemainingChars
      \fi
    \fi
  \fi
  \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.

\FV@TrueTab@UseWidth Version of \FV@TrueTab that uses pre-computed tab widths.

```
\def\FV@TrueTab@UseWidth{%
  \@tempdima=\csname FV@TrueTab:Width\arabic{FV@TrueTabCounter}\endcsname sp\relax
  \stepcounter{FV@TrueTabCounter}%
  \hbox to\@tempdima{\hss\FV@TabChar}}
```

Line scanning and break insertion macros

The strategy here is to scan a line token-by-token, and insert breaks at appropriate points. An alternative would be to make characters active, and have them expand to literal versions of themselves plus appropriate breaks. Both approaches have advantages and drawbacks. A catcode-based approach could work, but in general would require redefining some existing active characters to insert both appropriate breaks and their original definitions. The current approach works regardless of catcodes. It is also convenient for working with macros that expand to single characters, such as those created in highlighting code with Pygments (which is used by minted and pythontex). In that case, working with active characters would not be enough, and scanning for macros (or redefining them) is necessary. With the current approach, working with more complex macros is also straightforward. Adding support for line breaks within a macro simply requires wrapping macro contents with `\FancyVerbBreakStart...\FancyVerbBreakStop`. A catcode-based approach could require `\scantokens` or a similar retokenization in some cases, but would have the advantage that in other cases no macro redefinition would be needed.

`\FV@Break` The entry macro for breaking lines, either anywhere or before/after specified characters. The current line (or argument) will be scanned token by token/group by group, and accumulated (with added potential breaks) in `\FV@TmpLine`. After scanning is complete, `\FV@TmpLine` will be inserted. 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.[8] The last token is tracked with `\FV@LastToken`, to allow lookbehind when breaking by groups of identical characters. `\FV@LastToken` is `\let` to `\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 `\FV@LastToken`, it is vital always to use `\let\FV@LastToken=...` so that `\let\FV@LastToken==` will work (so that the equals sign `=` won't break things).

The current definition of `\FV@Break@Token` is swapped for a UTF-8 compatible one under pdfTeX when necessary. The standard macros are defined next, since they make the algorithms simpler to understand. The more complex `UTF` variants are defined later.

```
\def\FV@Break{%
  \def\FV@TmpLine{}%
  \let\FV@LastToken=\FV@Undefined
  \ifFV@pdfTeXinputenc
    \ifdefstring{\inputencodingname}{utf8}%
     {\ifx\FV@Break@Token\FV@Break@AnyToken
```

[8] 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.

```
        \let\FV@Break@Token\FV@Break@AnyToken@UTF
      \else
        \ifx\FV@Break@Token\FV@Break@BeforeAfterToken
          \let\FV@Break@Token\FV@Break@BeforeAfterToken@UTF
        \fi
      \fi}%
    {}%
  \fi
  \FV@Break@Scan
}
```

`\FV@EndBreak`

```
\def\FV@EndBreak{\FV@TmpLine}
```

`\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.

```
\begingroup
\catcode`\$=3%
\gdef\FV@Break@Scan{%
  \@ifnextchar\FV@EndBreak%
   {}%
   {\ifx\@let@token$\relax
      \let\FV@Break@Next\FV@Break@Math
    \else
      \ifx\@let@token\bgroup\relax
        \let\FV@Break@Next\FV@Break@Group
      \else
        \let\FV@Break@Next\FV@Break@Token
      \fi
    \fi
    \FV@Break@Next}%
}
\endgroup
```

`\FV@Break@Math` Grab an entire math span, and insert it into `\FV@TmpLine`. Due to grouping, this works even when math contains things like `\text{$x$}`. After dealing with the math span, continue scanning.

```
\begingroup
\catcode`\$=3%
\gdef\FV@Break@Math$#1${%
  \g@addto@macro{\FV@TmpLine}{$#1$}%
  \let\FV@LastToken=\FV@Undefined
  \FV@Break@Scan}
\endgroup
```

`\FV@Break@Group` Grab the group, and insert it into `\FV@TmpLine` (as a group) before continuing scanning.

```
\def\FV@Break@Group#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}{}{\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Scan}
```

`\FV@Break@AnyToken` 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\FV@Break...
  \ifcat\noexpand#1a%
    \g@addto@macro{\FV@TmpLine}...
  \else
...
\endgroup
```

```
\def\FV@Break@AnyToken#1{%
  \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAnywhereBreak#1}%
  \FV@Break@Scan}
```

`\FV@Break@BeforeAfterToken` 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.

```
\def\FV@Break@BeforeAfterToken#1{%
  \ifcsname FV@BreakBefore@Token\detokenize{#1}\endcsname
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak
  \else
    \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \let\FV@Break@Next\FV@Break@AfterTokenBreak
    \else
```

```
      \let\FV@Break@Next\FV@Break@BeforeAfterTokenNoBreak
    \fi
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterTokenNoBreak#1{%
  \g@addto@macro{\FV@TmpLine}{#1}%
  \let\FV@LastToken=#1%
  \FV@Break@Scan}
\def\FV@Break@BeforeTokenBreak#1{%
  \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
   {\ifx#1\FV@LastToken\relax
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \let\FV@LastToken=#1%
      \fi
    \else
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \let\FV@LastToken=#1%
      \fi
    \fi}%
   {\ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
      \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
      \def\FV@RescanToken{#1}%
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
      \let\FV@Break@Next\FV@Break@Scan
      \let\FV@LastToken=#1%
    \fi}%
  \FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan{%
  \expandafter\FV@Break@AfterTokenBreak\FV@RescanToken}
\def\FV@Break@AfterTokenBreak#1{%
  \let\FV@LastToken=#1%
  \@ifnextchar\FV@FVSpaceToken%
   {\g@addto@macro{\FV@TmpLine}{#1}\FV@Break@Scan}%
   {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
     {\ifx\@let@token#1\relax
        \g@addto@macro{\FV@TmpLine}{#1}%
```

```
        \let\FV@Break@Next\FV@Break@Scan
      \else
        \ifx\@let@token\bgroup\relax
          \g@addto@macro{\FV@TmpLine}{#1}%
          \let\FV@Break@Next\FV@Break@AfterTokenBreak@Group
        \else
          \g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
          \let\FV@Break@Next\FV@Break@Scan
        \fi
      \fi}%
     {\g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
      \let\FV@Break@Next\FV@Break@Scan}%
    \FV@Break@Next}%
}
\def\FV@Break@AfterTokenBreak@Group#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}%
   {\let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@i}%
   {\let\FV@Break@Next\FV@Break@Scan\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Next}
\def\FV@Break@AfterTokenBreak@Group@i{%
  \@ifnextchar\FV@LastToken%
   {\FV@Break@Scan}%
   {\g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \FV@Break@Scan}}
```

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:

```
\ifFV@pdfTeXinputenc
```

`\FV@BreakBeforePrep@UTF` We need `UTF` variants of the `breakbefore` and `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).

```
\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
        \expandafter\let\expandafter\FV@Break@Next\csname 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
}
```

`\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
        \expandafter\let\expandafter\FV@Break@Next\csname 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{%
      \ifcsname FV@BreakBefore@Token\detokenize{##1}\endcsname
```

```
        \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {}%
           {\PackageError{fvextra}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {\PackageError{fvextra}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}%
           {}}%
       \fi
       \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
}
```

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

```
\def\FV@Break@AnyToken@UTF#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\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}%
}
\def\FV@Break@AnyToken@UTF@i#1{%
  \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAnywhereBreak#1}%
  \FV@Break@Scan}
```

`\FV@Break@BeforeAfterToken@UTF` 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.

```
\def\FV@Break@BeforeAfterToken@UTF#1{%
```

```
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@UTF@octets@after\FV@Break@BeforeAfterToken@UTF@i
  \else
    \let\FV@Break@Next\FV@Break@BeforeAfterToken@UTF@i
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterToken@UTF@i#1{%
  \ifcsname FV@BreakBefore@Token\detokenize{#1}\endcsname
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak@UTF
  \else
    \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \let\FV@Break@Next\FV@Break@AfterTokenBreak@UTF
    \else
      \let\FV@Break@Next\FV@Break@BeforeAfterTokenNoBreak@UTF
    \fi
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterTokenNoBreak@UTF#1{%
  \g@addto@macro{\FV@TmpLine}{#1}%
  \def\FV@LastToken{#1}%
  \FV@Break@Scan}
\def\FV@Break@BeforeTokenBreak@UTF#1{%
  \def\FV@CurrentToken{#1}%
  \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
   {\ifx\FV@CurrentToken\FV@LastToken\relax
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \def\FV@LastToken{#1}%
      \fi
    \else
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \def\FV@LastToken{#1}%
      \fi
    \fi}%
   {\ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
      \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
```

```
      \def\FV@RescanToken{#1}%
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
      \let\FV@Break@Next\FV@Break@Scan
      \def\FV@LastToken{#1}%
    \fi}%
  \FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF{%
  \expandafter\FV@Break@AfterTokenBreak@UTF\expandafter{\FV@RescanToken}}
\def\FV@Break@AfterTokenBreak@UTF#1{%
  \def\FV@LastToken{#1}%
  \@ifnextchar\FV@FVSpaceToken%
   {\g@addto@macro{\FV@TmpLine}{#1}\FV@Break@Scan}%
   {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
     {\g@addto@macro{\FV@TmpLine}{#1}%
      \ifx\@let@token\bgroup\relax
        \let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@UTF
      \else
        \let\FV@Break@Next\FV@Break@AfterTokenBreak@UTF@i
      \fi}%
     {\g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
      \let\FV@Break@Next\FV@Break@Scan}%
    \FV@Break@Next}%
}
\def\FV@Break@AfterTokenBreak@UTF@i#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@UTF@octets@after\FV@Break@AfterTokenBreak@UTF@i
  \else
    \def\FV@NextToken{#1}%
    \ifx\FV@LastToken\FV@NextToken
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \fi
    \let\FV@Break@Next\FV@Break@Scan
  \fi
  \FV@Break@Next#1}
\def\FV@Break@AfterTokenBreak@Group@UTF#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}%
   {\let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@UTF@i}%
   {\let\FV@Break@Next\FV@Break@Scan\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Next}
\def\FV@Break@AfterTokenBreak@Group@UTF@i{%
  \@ifnextchar\bgroup%
   {\FV@Break@Scan}%
   {\FV@Break@AfterTokenBreak@Group@UTF@ii}}
\def\FV@Break@AfterTokenBreak@Group@UTF@ii#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
```

```
    \let\FV@UTF@octets@after\FV@Break@AfterTokenBreak@Group@UTF@ii
  \else
    \def\FV@NextToken{#1}%
    \ifx\FV@LastToken\FV@NextToken
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \fi
    \let\FV@Break@Next\FV@Break@Scan
  \fi
  \FV@Break@Next#1}
```

End the conditional creation of the pdfTeX `UTF` macros:

```
\fi
```

Line processing before scanning

`\FV@makeLineNumber` The lineno package is used for formatting wrapped lines and inserting break symbols. We need a version of lineno's `\makeLineNumber` that is adapted for our purposes. This is adapted directly from the example `\makeLineNumber` that is given in the lineno documentation under the discussion of internal line numbers. The `\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, `\FV@SetLineBreakLast` is disabled by `\let`ting it to `\relax`.

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

`\FV@RaggedRight` We need a copy of the default `\raggedright` to ensure that everything works with classes or packages that use a special definition.

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

`\FV@LineWidth` This is the effective line width within a broken line.

```
\newdimen\FV@LineWidth
```

\FV@SaveLineBox This is the macro that does most of the work. It was inspired by Marco Daniel's code at http://tex.stackexchange.com/a/112573/10742.

This macro is invoked when a line is too long. We modify `\FV@LineWidth` to take into account `breakindent` and `breakautoindent`, and insert `\hbox`es to fill the empty space. We also account for `breaksymbolindentleft` and `breaksymbolindentright`, but *only* when there are actually break symbols. The code is placed in a `\parbox`. Break symbols are inserted via lineno's `internallinenumbers*`, which does internal line numbers without continuity between environments (the `linenumber` counter is automatically reset). The beginning of the line has negative `\hspace` inserted to pull it out to the correct starting position. `\strut`s are 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.

```
\def\FV@SaveLineBox#1{%
  \savebox{\FV@LineBox}{%
    \advance\FV@LineWidth by -\FV@BreakIndent
    \hbox to \FV@BreakIndent{\hfill}%
    \ifthenelse{\boolean{FV@BreakAutoIndent}}%
     {\let\FV@LineIndentChars\@empty
      \FV@GetLineIndent#1\FV@Undefined
      \savebox{\FV@LineIndentBox}{\FV@LineIndentChars}%
      \hbox to \wd\FV@LineIndentBox{\hfill}%
      \advance\FV@LineWidth by -\wd\FV@LineIndentBox
      \setcounter{FV@TrueTabCounter}{0}}%
     {}%
    \ifdefempty{\FancyVerbBreakSymbolLeft}{}%
     {\hbox to \FV@BreakSymbolIndentLeft{\hfill}%
      \advance\FV@LineWidth by -\FV@BreakSymbolIndentLeft}%
    \ifdefempty{\FancyVerbBreakSymbolRight}{}%
     {\advance\FV@LineWidth by -\FV@BreakSymbolIndentRight}%
    \parbox[t]{\FV@LineWidth}{%
      \FV@RaggedRight
      \leftlinenumbers*
      \begin{internallinenumbers*}%
      \let\makeLineNumber\FV@makeLineNumber
      \noindent\hspace*{-\FV@BreakIndent}%
      \ifdefempty{\FancyVerbBreakSymbolLeft}{}{%
        \hspace*{-\FV@BreakSymbolIndentLeft}}%
      \ifthenelse{\boolean{FV@BreakAutoIndent}}%
       {\hspace*{-\wd\FV@LineIndentBox}}%
       {}%
      \FV@BreakByTokenAnywhereHook
      \strut\FancyVerbFormatText{%
        \FancyVerbBreakStart #1\FancyVerbBreakStop}\nobreak\strut
      \end{internallinenumbers*}
    }%
```

```
    \ifdefempty{\FancyVerbBreakSymbolRight}{}%
     {\hbox to \FV@BreakSymbolIndentRight{\hfill}}%
  }%
}
\let\FV@BreakByTokenAnywhereHook\relax
```

\FV@ListProcessLine@Break

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 \FV@SaveLineBox, the counter FancyVerbLineBreakLast is set to the internal line number of the last segment of the broken line. During the second use of \FV@SaveLineBox, we disable this (\let\FV@SetLineBreakLast\relax) so that the value of FancyVerbLineBreakLast remains fixed and thus may be used to determine when a right-hand break symbol should be inserted.

```
\def\FV@ListProcessLine@Break#1{%
  \hbox to \hsize{%
  \kern\leftmargin
  \hbox to \linewidth{%
  \FV@LineWidth\linewidth
  \ifx\FV@RightListFrame\relax\else
    \advance\FV@LineWidth by -\FV@FrameSep
    \advance\FV@LineWidth by -\FV@FrameRule
  \fi
  \ifx\FV@LeftListFrame\relax\else
    \advance\FV@LineWidth by -\FV@FrameSep
    \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}{%
    \FancyVerbFormatLine{%
      %\FancyVerbHighlightLine  %<-- Default definition using \rlap breaks breaking
       {\FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}%
  \ifx\FV@Tab\FV@TrueTab
    \let\FV@TrueTabSaveWidth\relax
  \fi
  \ifdim\wd\FV@LineBox>\FV@LineWidth
    \setcounter{FancyVerbLineBreakLast}{0}%
```

```
    \ifx\FV@Tab\FV@TrueTab
      \let\FV@Tab\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
  \else
    \FV@LeftListNumber
    \FV@LeftListFrame
    \FancyVerbFormatLine{%
      \FancyVerbHighlightLine{%
        \parbox[t]{\FV@LineWidth}{%
          \noindent\strut\FV@ObeyTabs{\FancyVerbFormatText{#1}}\strut}}}%
    \FV@RightListFrame
    \FV@RightListNumber
  \fi}%
  \hss}\baselineskip\z@\lineskip\z@}
```

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.

`\FV@BreakBeforePrep@Pygments` Pygments converts some characters into macros to ensure that they appear literally. As a result, `breakbefore` and `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.

```
\def\FV@BreakBeforePrep@Pygments#1{%
  \ifcsname FV@BreakBefore@Token\@backslashchar\endcsname
    \@namedef{FV@BreakBefore@Token#1Zbs }{}%
  \fi
```

```
\ifcsname FV@BreakBefore@Token\FV@underscorechar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zus }{}%
\fi
\ifcsname FV@BreakBefore@Token\@charlb\endcsname
  \@namedef{FV@BreakBefore@Token#1Zob }{}%
\fi
\ifcsname FV@BreakBefore@Token\@charrb\endcsname
  \@namedef{FV@BreakBefore@Token#1Zcb }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{^}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zca }{}%
\fi
\ifcsname FV@BreakBefore@Token\FV@ampchar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zam }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{<}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zlt }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{>}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zgt }{}%
\fi
\ifcsname FV@BreakBefore@Token\FV@hashchar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zsh }{}%
\fi
\ifcsname FV@BreakBefore@Token\@percentchar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zpc }{}%
\fi
\ifcsname FV@BreakBefore@Token\FV@dollarchar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zdl }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{-}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zhy }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{'}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zsq }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{"}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zdq }{}%
\fi
\ifcsname FV@BreakBefore@Token\FV@tildechar\endcsname
  \@namedef{FV@BreakBefore@Token#1Zti }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{@}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zat }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{[}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zlb }{}%
\fi
\ifcsname FV@BreakBefore@Token\detokenize{]}\endcsname
  \@namedef{FV@BreakBefore@Token#1Zrb }{}%
```

```
  \fi
}
```

`\FV@BreakAfterPrep@Pygments`

```
\def\FV@BreakAfterPrep@Pygments#1{%
  \ifcsname FV@BreakAfter@Token\@backslashchar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zbs }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\FV@underscorechar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zus }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\@charlb\endcsname
    \@namedef{FV@BreakAfter@Token#1Zob }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\@charrb\endcsname
    \@namedef{FV@BreakAfter@Token#1Zcb }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{^}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zca }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\FV@ampchar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zam }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{<}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zlt }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{>}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zgt }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\FV@hashchar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zsh }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\@percentchar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zpc }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\FV@dollarchar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zdl }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{-}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zhy }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{'}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zsq }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\detokenize{"}\endcsname
    \@namedef{FV@BreakAfter@Token#1Zdq }{}%
  \fi
  \ifcsname FV@BreakAfter@Token\FV@tildechar\endcsname
    \@namedef{FV@BreakAfter@Token#1Zti }{}%
  \fi
```

```
    \ifcsname FV@BreakAfter@Token\detokenize{@}\endcsname
      \@namedef{FV@BreakAfter@Token#1Zat }{}%
    \fi
    \ifcsname FV@BreakAfter@Token\detokenize{[}\endcsname
      \@namedef{FV@BreakAfter@Token#1Zlb }{}%
    \fi
    \ifcsname FV@BreakAfter@Token\detokenize{]}\endcsname
      \@namedef{FV@BreakAfter@Token#1Zrb }{}%
    \fi
}
```

breakbytoken

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.

```
\newbool{FV@breakbytoken}
\define@booleankey{FV}{breakbytoken}%
 {\booltrue{FV@breakbytoken}}%
 {\boolfalse{FV@breakbytoken}\boolfalse{FV@breakbytokenanywhere}}
```

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`.

```
\newbool{FV@breakbytokenanywhere}
\define@booleankey{FV}{breakbytokenanywhere}%
 {\booltrue{FV@breakbytokenanywhere}\booltrue{FV@breakbytoken}}%
 {\boolfalse{FV@breakbytokenanywhere}\boolfalse{FV@breakbytoken}}
```

\FancyVerbBreakByTokenAnywhereBreak

This is the break introduced when `breakbytokenanywhere=true`. Alternatives would be `\discretionary{}{}{}` or `\linebreak[0]`.

```
\def\FancyVerbBreakByTokenAnywhereBreak{\allowbreak{}}
```

\VerbatimPygments

This is the command that activates Pygments features. It must be invoked before `\begin{Verbatim}`, etc., but inside a `\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 pythontex), and the Pygments macro that should actually be used (`\PYG`⟨*style_name*⟩ for minted and pythontex). The two are distinguished because it can be convenient to highlight everything using the same literal macro name, and then `\let` it to appropriate values to change styles, rather than redoing all highlighting to change styles. It modifies `\FV@PygmentsHook`, which is at the beginning of `\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, `#1` is the Pygments macro that literally appears, and `#2` is the macro that is actually to be used.

The `breakbefore` and `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-`breakbytoken` case, `#1` is redefined to use `#2` internally, bringing in `\FancyVerbBreakStart` and `\FancyVerbBreakStop` to allow line breaks.

In the `breakbytoken` cases, an `\hbox` is used to prevent breaks within the macro (breaks could occur at spaces even without `\FancyVerbBreakStart`). The `breakbytokenanywhere` case is similar but a little tricky. `\FV@BreakByTokenAnywhereHook`, which is inside `\FV@SaveLineBox` where line breaking occurs, is used to define `\FV@BreakByTokenAnywhereBreak` so that it will "do nothing" the first time it is used and on subsequent invocations become `\FancyVerbBreakByTokenAnywhereBreak`. Because the hook is within the `internallinenumbers*` environment, the redefinition doesn't escape, and the default global definition of `\FV@BreakByTokenAnywhereBreak` as `\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 `\FV@PYG` is defined to avoid problems in case `#1=#2`. There is also a check for a non-existant `#2` (`\PYG`⟨*style_name*⟩ may not be created until a later compile in the pythontex case); if `#2` does not exist, fall back to `#1`. For the existance check, `\ifx...\relax` must be used instead of `\ifcsname`, because `#2` will be a macro, and will typically be created with `\csname...\endcsname` which will `\let` the macro to `\relax` if it doesn't already exist.

```
\def\FV@VerbatimPygments#1#2{%
  \edef\FV@PYG@Literal{\expandafter\FV@DetokMacro@StripSpace\detokenize{#1}}%
  \def\FV@BreakBeforePrep@PygmentsHook{%
    \expandafter\FV@BreakBeforePrep@Pygments\expandafter{\FV@PYG@Literal}}%
  \def\FV@BreakAfterPrep@PygmentsHook{%
    \expandafter\FV@BreakAfterPrep@Pygments\expandafter{\FV@PYG@Literal}}%
  \ifx#2\relax
    \let\FV@PYG#1%
  \else
    \let\FV@PYG#2%
  \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}{##2}}}}%
   {\def#1##1##2{%
```

```
      \FV@PYG{##1}{\FancyVerbBreakStart##2\FancyVerbBreakStop}}}%
}
\let\FV@BreakByTokenAnywhereBreak\relax
\def\FV@DetokMacro@StripSpace#1 {#1}
```