Py2tex documentation

Jeroen van Maanen[*]

March 29, 2006

Abstract

The py2tex package allows you to typeset Python programs with LaTeX. It consists of some Python code to translate Python source to LaTeX and a LaTeX style file that contains the necessary definitions. The style file also adds some degree of customizability.

Contents

[*] E-mail: `jeroenvm@xs4all.nl`

1 Py2tex.py – Sat Apr 1 10:54:31 2006

py2tex.py – Translate Python source code to LaTeX code that can be typeset using the `py2tex` documentstyle option.

To typeset a Python module called `foo.py` with py2tex, create a LaTeX file along the following lines.

```
% frame.tex -- wrapper around foo
\documentstyle[...,py2tex,...]{...}
...
\begin{document}
...
\PythonSource{foo.pt}
...
\end{document}
```

Then give the command

```
    $ py2tex -o foo.pt foo.py
```

Finally run LaTeX on the previously constructed wrapper, like this

```
    $ latex frame
```

This will give you a `.dvi` file that you can print in the normal way.

Note that normally the comments are interpreted by LaTeX. This allows for formulae and other fancy stuff. However, if you don't need this, or if you want to typeset programs that were not specifically written to be typeset with py2tex, you can leave comments uninterpreted by calling the `py2tex` script with the `-v` option. The same effect can be obtained by ending a comment with '`%ASCII`'. It is also possible to switch back to interpreted mode by inserting a comment ending in '`%TeX`' or '`%LaTeX`'.

Here are some guidelines for writing Python code to be typeset using py2tex. Each line of Python code is typeset by LaTeX as a paragraph where, in case it is broken up into more than one line, all lines following the first are indented by one and a half standard indentation more than the indentation of the first line. Py2tex does not count parentheses to determine whether a line is a continuation of the previous or not. So if you want it to be indented appropriately, escape the end of the previous line with a backslash. Then py2tex will treat the joined lines as one line, and it will inform LaTeX that the escaped line breaks are good points to break it up again. Because LaTeX may decide to break the code at other positions (or not at all), these lines will not be numbered.

Consecutive lines that start with a single hash mark (#) right after the indentation are joined and typeset in a `\vbox` (more precise: a `\vtop`). This is called a block comment. Indentation changes have no effect within a block comment. It is possible to escape from the `\vbox` and set the remainder of the block comment in what Knuth calls 'outer vertical mode' by using the `\ESC` command. This can be used to incorporate long stretches of LaTeX code that can spread out over several pages. Unindented block comments are automatically escaped in their entirety.

If a line starts with at least two hash marks it is typeset as if it followed some Python code. The second hash mark also switches immediately back to Python mode (see below). This feature is also implemented for ASCII mode, while the general escape to Python mode is not. (This feature is intended to disable lines of Python code by placing two hash marks before them. This ensures that the formatting will be very similar to the uncommented version.)

Comments following Python code are typeset on the same line as the Python code, separated from it by a `\quad` space and the hash mark.

Both in block and in line comments the hash mark is used to switch between LaTeX and Python mode, just like the dollar sign ($) is used to switch between horizontal and math mode. This means that hash marks are not visible as such in the output. However, two consecutive hash marks are passed to LaTeX as one. This means that it is possible to typeset a hash mark by putting `\##` in a comment. (This can also be used to define LaTeX macros and to include `\halign` templates, albeit at the expense of doubling all hash marks.) Note that this works only in LaTeX mode, *not* in ASCII mode.

So if you type

```
# % LaTeX
# Hash mark in comment: \##,
# formula in comment: $i_0\to\infty$.
print chr (i) # where #040<=i<=0x7E
## print '#' # print one hash sign % ASCII
## print i_0 * '#' # where i_0 is #hash signs
```

you get

```
    || Hash mark in comment: #, formula in comment: $i_0 \to \infty$.
    print chr(i)   # where 040 ≤ i ≤ 0x7E
    # print '#'   # print one hash sign
    # print i_0 * '#'   # where i_0 is #hash signs
```

Triple quoted strings that occur as the first non-comment after a line that ends in a colon (:) are treated as documentation strings. There is are three different options for treating them. If docprocess = `'none'`, this results in the “Same ’ol behaviour”:

```
def trivial:
    || Comment before documentation string.
    '''
This␣function␣does␣nothing:

*␣efficiently

*␣noiselessly

*␣with␣style
'''
    pass
```

If docprocess = `'plain'`, docstrings are typeset as verbatim comments except with thick solid lines instead of thin double lines:

```
def trivial:
    || Comment before documentation string.
    |
    | This function does nothing:
    |
    | * efficiently
    |
    | * noiselessly
    |
    | * with style
    |
    pass
```

If docprocess = `'struct'`, docstrings are typeset as structured text as defined by the doc-sig. This is so people can potentially write programs that look good both under gendoc and py2tex.

```
def trivial:
```

Table 1. Some Python constructs get special typographic treatment

Python	LaTeX
`=`	$\leftarrow$
`==`	$=$
`<=, >=`	$\leq, \geq$
`!=, <>`	$\neq$
`<<, >>`	$\ll, \gg$
`and, or, not`	$\wedge, \vee, \neg$
`in, not in`	$\in, \notin$
`is, is not`	$\equiv, \not\equiv$

Comment before documentation string.

This function does nothing:

- efficiently
- noiselessly
- with style

pass

It is possible to include the formatted version of another Python source file using the `\PythonSource*` macro. This was done below to give an example of the use of class Interpret. The starred version of the macro is needed to drop the line numbers, otherwise they would be typeset through the lines that mark the block comment. The starred version of `\PythonSource` also drops the section heading. If you escape the block comment (using `\ESC`) you can use the unstarred version again.

Finally some remarks about the formatting of Python constructs. Identifiers (keywords, variables and functions) are typeset in sans serif. If an identifier consists of only one character, it is typeset in *math italic* instead of sans serif. Keywords are typeset in boldface, functions (actually: identifiers before opening parentheses) are typeset slanted. These typefaces can be changed by redefining some of the macros in `py2tex.sty`. See the documentation of the style file for customization instructions.

Some constructs that get special treatment are listed in Table 1. This special treatment is optional. If the class is initialized with an extra argument that evaluates to false, or if the *no_math* () method is used, then no special treatment is done for these constructs. (Special treatment can be turned back on half way through a file using the *math* () method.)

In strings, characters outside the range `'␣'`–`'~'` are typeset as standard escape sequences (*e.g.*, `TAB` is typeset as `'\t'`, `ESC` is typeset as `'\033'`). A floating point literal with an exponent has its exponent written out as a power of ten (*e.g.*, `3e-6` is typeset as $3{\cdot}10^{-6}$). Hexadecimal literals are typeset in a typewriter font with a lower case `x` and uppercase digits (*e.g.*, `0X007e` is typeset as `0x007E`). Octal literals are typeset in italics (*e.g.*, `0377` is typeset as *0377*).

```
import os, re, string, sys, time
```

Usage of class Interpret.

```
import py2tex
def translate (name, outfile):
    file ← Interpret (filename)
    outfile.write (file.translation ()[0])
    while file.translate () ≠ None:
        for scrap ∈ file.translation (): outfile.write (scrap)
    file.close ()
```

Note that sys.stdin is used if name ∈ (None, `'-'`).

The other methods can best be viewed as private to the class.

```
class Interpret:
    def __init__ (self, name, math ← 1, interpret ← 1, docprocess ← 'none'):
        if name = None:
            self._name ← '-'
        else:
            self._name ← name
        if self._name = '-':
            self._name ← '(stdin)'
            mtime ← time.asctime (time.localtime (time.time ()))
            self._file ← sys.stdin
        else:
            mtime ← time.asctime (time.localtime (os.stat (name)[8]))
            self._file ← open (self._name, 'r')
            self._name ← os.path.basename (self._name)
        preamble ← '\\File{%s}{%s}\n\n' % (self._name, mtime)
        if ¬math: preamble ← preamble + '\\PythonNoMath\n\n'
        self._translation ← [preamble, ]
        self._math ← math
        self._line_nr ← 0
        self._line ← None
        self._old_line ← None
        self._eof ← 0
        self._indent_stack ← [0]
        self._no_break ← 0
        self._interpret_comments ← interpret
        self._docprocess ← docprocess
        self._docstring ← 1
    def math (self):
        if ¬self._math:
            self._translation.append ('\\PythonMath\n')
            self._math ← 1
    def no_math (self):
        if ¬self._math:
            self._translation.append ('\\PythonNoMath\n')
            self._math ← 0
    def interpret (self):
        self._interpret_comments ← 1
    def verbatim (self):
        self._interpret_comments ← 0
    def close (self):
        self._file.close ()
        self._line_nr ← 0
        self._line ← None
        self._old_line ← None
        self._indent_stack ← [0]
        self._translation ← [ ]
        self._eof ← 1
        self._no_break ← 0
    def flush (self):
        self._file.flush ()
    def next_line (self):
        if self._old_line ≠ None:
            self._line ← self._old_line
            self._old_line ← None
            self._line_nr ← self._line_nr + 1
            return
        self._line ← self._file.readline ()
        if self._line = '':
            self._eof ← 1
            raise EOFError
        if self._line[−1] = '\n': self._line ← self._line[: −1]
        self._line_nr ← self._line_nr + 1
```

```
def undo_line (self):
    if self._line $\neq$ None:
        self._old_line $\leftarrow$ self._line
        self._line $\leftarrow$ None
        self._line_nr $\leftarrow$ self._line_nr $-$ 1
def close_tex (self, tex):
    while 1:
        if tex[$-2$: ] = '\\␣':
            tex $\leftarrow$ tex[: $-2$]
        elif tex[$-4$: ] = '\\BP␣':
            tex $\leftarrow$ tex[: $-4$]
        else:
            break
    if tex $\notin$ ('$', '${}'):
        self._translation.append (tex + '$')
def tr_indentation (self):
    length $\leftarrow$ white_re.match (self._line)
    if length $< 0$: raise error
    indent $\leftarrow$ 0
    for $c \in$ self._line[: length]:
        indent $\leftarrow$ indent + 1
        if $c$ = '\t':
            indent $\leftarrow$ indent + 8
            indent $\leftarrow$ indent & ~ 0x7
    self._line $\leftarrow$ self._line[length: ]
    while indent < self._indent_stack[$-1$]:
        del self._indent_stack[$-1$]
    if indent > self._indent_stack[$-1$]:
        self._indent_stack.append (indent)
    self._indentation $\leftarrow$ len (self._indent_stack) $-$ 1
def tr_comment_line (self):
    if self._interpret_comments:
        length $\leftarrow$ verbatim_re.search (self._line)
        if length $\geq 0$:
            self.verbatim ()
            self._line $\leftarrow$ self._line[: length]
        while 1:
            hash $\leftarrow$ string.find (self._line, '#')
            if hash $\geq 0$:
                if len (self._line) > hash + 1 $\wedge$ self._line[hash + 1] = '#':
                    self._translation.append (self._line[: hash] + '#')
                    self._line $\leftarrow$ self._line[hash + 2: ]
                    continue
                self._translation.append (self._line[: hash])
                self._line ← self._line[hash + 1: ]
                self.tr_code (0)    # No continued lines in comments.
                if len (self._line) ≤ 0: break
                if self._line[0] ≠ '#': raise error
                self._line ← self._line[1: ]
            else:
                break
        self._translation.append (self._line + '\n')
    else:
        length ← interpret_re.search (self._line)
        if length ≥ 0:
            self.interpret ()
            self._line ← self._line[: length]
        while len (self._line) > 0:
            length ← ordinary_re.match (self._line)
            if length > 0:
                self._translation.append (self._line[: length])
            if len (self._line) > length:
                char ← self._line[length]
                if char ∈ '<>\\{|}~':
                    self._translation.append ('{\\tt\\char`\\%s}' % char)
                else:
                    self._translation.append ('\\' + char)
            self._line ← self._line[length + 1: ]
        self._translation.append ('\n')
def tr_block_comment (self):
    if self._line[0] ≠ '#': raise error
    outer ← self._indentation = 0
    if outer:
        if self._line_nr > 1:

            self._translation.append ('\\PythonOuterBlock\n')
        else:
            self._translation.append ('\\PythonOuterBlock*\n')
    else:
        self._translation.append ('\\B{%d}{%d}{%%\n' %
              (self._line_nr, self._indentation))
    try:
        white ← white_re.match (self._line, 1)
        if white < 0: raise error
        self._line ← self._line[white: ]
        while 1:
            self.tr_comment_line ()
            self.next_line ()
            white ← white_re.match (self._line)
            if white < 0: raise error
            if len (self._line) > white ∧ self._line[white] = '#' ∧
                     self._line[white: white + 2] ≠ '##':
                self._line ← self._line[white + 1: ]
                white ← white_re.match (self._line)
                if white > 0: self._line ← self._line[white: ]
                continue
            self.undo_line ()
            return
    finally:
        if outer:
            self._translation.append ('\\PythonOuterBlockEnd\n')
        else:
            self._translation.append ('}\n')
def tr_comment (self):
    self._translation.append ('\\#\\␣')
    while self._line[: 2] = '##':
        self._line ← self._line[2: ]
        self.tr_code (0)   # No continued lines in comments.
        if self._line[: 1] = '#':
            self._translation.append ('\\quad\\#\\␣')
    if len (self._line) < 1:
        self._translation.append ('\n')
        return
    if self._line[0] ≠ '#': raise error
    white ← white_re.match (self._line, 1)
    if white < 0: raise error
    self._line ← self._line[white: ]
    self.tr_comment_line ()
    return
def tr_string (self, token):
    quote ← token[0]
    tl ← len (token)
    self._translation.append ('\S{' + token)
    while 1:
        pos ← string.find (self._line, quote)
        if pos > 0:
            self._translation.append ('\\verb*%s%s' %
                     (quote, ctrl_protect (self._line[: pos + 1])))
            if escape_re.match (self._line[: pos]) = pos:
                self._translation.append (quote)
                self._line ← self._line[pos + 1: ]
                continue
            self._line ← self._line[pos: ]
            pos ← 0
        if pos ≥ 0:
            if self._line[: tl] = token:
                self._translation.append (token + '}')
                self._line ← self._line[tl: ]
                return
            self._translation.append (quote)
            self._line ← self._line[1: ]
        else:
            self._translation.append ('\\verb*%s%s%s' %
                   (quote, ctrl_protect (self._line), quote))
            self._line ← ''
            if tl = 1:
                self._translation.append ('}')
                return
            self.next_line ()
            self._translation.append ('\n\\I{%d}{0}' % self._line_nr)
    return

def tr_docstring_plain (self):
    length ← quote_re.match (self._line)
    token ← self._line[: length]
    self._line ← self._line[length: ]
    quote ← token[0]
    tl ← len (token)

    if self._indentation = 0:
        self._translation.append ('\\PythonDocBlock\n')
    else:
        self._translation.append ('\DS{%s}{%s}{%%\n'%
            (self._line_nr, self._indentation))
    while 1:
        pos ← string.find (self._line, quote)
        if pos > 0:
            self._translation.append ('\\verb%s%s' %
                   (quote, ctrl_protect (self._line[: pos + 1])))
            if escape_re.match (self._line[: pos]) = pos:
                self._translation.append (quote)
                self._line ← self._line[pos + 1: ]
                continue
            self._line ← self._line[pos: ]
            pos ← 0
        if pos ≥ 0:
            if self._line[: tl] = token:
                self._line ← self._line[tl: ]
                break
            self._translation.append ('\\verb%s%s%s' %
                 (quote, ctrl_protect (self._line), quote))
        else:
            self._translation.append ('\\verb%s%s%s' %
                   (quote, ctrl_protect (self._line), quote))
            self._line ← ''
            if tl = 1:
                break
            self.next_line ()
            || XXX This assumes 8 spaces per tab.
            wchars ← white_re.match (self._line)
            spaces ← re.sub ('\t', '␣' ∗ 8, self._line[: wchars])
            indent ← white_re.match (spaces)
            # print ‘spaces‘, indent, self._indentation
            self._line ← spaces[self._indentation ∗ 4: ] + self._line[wchars: ]
            self._translation.append ('\\\\␣\n')
    if self._indentation = 0:
        self._translation.append ('\n\\PythonDocBlockEnd\n')
    else:
        self._translation.append ('}\n')

def tr_docstring_struct (self):
    length ← quote_re.match (self._line)
    token ← self._line[: length]
    self._line ← self._line[length: ]
    quote ← token[0]
    tl ← len (token)
    if self._indentation = 0:
        self._translation.append ('\\PythonDocBlock\n')
    else:
        self._translation.append ('\DS{%s}{%s}{%%\n'%
           (self._line_nr, self._indentation))
    docstring ← [ ]
    while 1:
        pos ← string.find (self._line, quote)
        if pos > 0:
            docstring.append (self._line[: pos])
            if escape_re.match (self._line[: pos]) = pos:
                docstring.append (quote)
                self._line ← self._line[pos + 1: ]
                continue
            self._line ← self._line[pos: ]
            pos ← 0
        if pos ≥ 0:
            if self._line[: tl] = token:
                self._line ← self._line[tl: ]
                break
            docstring.append (quote)
            self._line ← self._line[1: ]
        else:
            docstring.append (self._line)
            self._line ← ''
            if tl = 1:
                break
            self.next_line ()
            docstring.append ('\n')
    docstring ← string.joinfields (docstring, '')
    import struct2latex
    structstring ← str (struct2latex.LaTeX (docstring))
    if self._indentation = 0:
        self._translation.append ('%s\n\\PythonDocBlockEnd\n' %
              structstring)
    else:
        self._translation.append ('%s}' % structstring)

def tr_code (self, allow_continue ← 1):
    tex ← '$'
    try:
        careful ← 0
        while 1:
            white ← white_re.match (self._line)
            if white > 0:
                self._line ← self._line[white: ]
            if len (self._line) ≤ 0:  return
            if self._line = '\\':
                if allow_continue:
                    tex ← tex + '\\BP␣'
                    self.next_line ()
                    continue
                else:
                    self._line ← ''
                    return
            if self._line[0] = '#':  return
            length ← token_re.match (self._line)
            if length < 1:
```

```
        length $\leftarrow$ numeral_re.*match* (self._line)
        **if** length $< 1$:
            tex $\leftarrow$ tex + self._line$[0]$
            self._line $\leftarrow$ self._line$[1:]$
            careful $\leftarrow 0$
        **else**:
            token $\leftarrow$ self._line$[:$ length$]$
            self._line $\leftarrow$ self._line$[$length$:]$
            **if** careful: tex $\leftarrow$ tex + '\\␣'
            tex $\leftarrow$ tex + *tr_numeral* (token)
            careful $\leftarrow 1$
        **continue**
    token $\leftarrow$ self._line$[:$ length$]$
    self._line $\leftarrow$ self._line$[$length$:]$
    token $\leftarrow$ self.*double* (token)
    **if** token = ':':
        self._docstring $\leftarrow 1$
    **else**:
        self._docstring $\leftarrow 0$
    **if** token $\in$ ('{', '}'):
        tex $\leftarrow$ tex + '\\' + token
        careful $\leftarrow 0$
        **continue**
    **if** token $\in$ reserved_operators:
        tex $\leftarrow$ tex + '\\O{%s}' % token
        careful $\leftarrow 0$
        **continue**
    **if** token$[0]$ $\in$ string.letters + '_':
        **if** careful: tex $\leftarrow$ tex + '\\␣'
        new_careful $\leftarrow 1$
        **if** token $\in$ reserved:
            **if** tex$[-2:]$ $\notin$ ('$', '\\␣') $\wedge$ $\neg$careful:
                tex $\leftarrow$ tex + '\\␣'
            tex $\leftarrow$ tex + '\\K{%s}' % token
            **if** token = 'if': tex $\leftarrow$ tex + '\\,'
            **if** token $\notin$ single: tex $\leftarrow$ tex + '\\␣'
            new_careful $\leftarrow 0$
        **else**:
            token $\leftarrow$ *usc_protect* (token)
            length $\leftarrow$ function_re.*match* (self._line)
            **if** length $> 0$:
                self._line $\leftarrow$ self._line$[$length$:]$
                tex $\leftarrow$ tex + '\\F{%s}\\,(' % token
                new_careful $\leftarrow 0$
```

```
                    else:
                        if len (token) = 1:
                            tex ← tex + token
                        else:
                            tex ← tex + '\\V{%s}' % token
                careful ← new_careful
                continue
            if token[0] ∈ '\'"':
                self.close_tex (tex + '{}')
                self.tr_string (token)
                tex ← '${}'
                careful ← 0
                continue
            if '{' ∈ token ∨ '}' ∈ token:
                raise ValueError, "brace␣in␣token␣'%s'" % token
            tex ← tex + '\\Y{%s}' % token
            careful ← 0
    finally:
        self.close_tex (tex)
def double (self, token):
    if token ∉ ('not', 'is'): return token
    white ← white_re.match (self._line)
    if white > 0:
        self._line ← self._line[white: ]
    next_length ← token_re.match (self._line)
    if next_length > 0:
        next ← self._line[: next_length]
        if (token, next) ∈ (('not', 'in'), ('is', 'not')):
            self._line ← self._line[next_length: ]
            return token + '␣' + next
    return token
```

Method *translate* () is the interface to the Interpret class. It calls the *tr_xxx*() methods to process indentation, code, comments and strings.

```
def translate (self):
    self._translation ← [ ]
    if self._eof: return None
    try:
        empty ← 0
        self.next_line ()
        while white_re.match (self._line) = len (self._line):
            empty ← empty + 1
            self.next_line ()
        if empty > 0:
            self._translation.append ('\\E{%d}' % empty)
        self.tr_indentation ()
        if len (self._line) $> 0 \land$ self._line$[0] =$ '#' $\land$ self._line$[:2] \neq$ '##':
            self.tr_block_comment ()
            self._no_break $\leftarrow 1$
        elif self._docprocess $\neq$ 'none' $\land$
              self._docstring $\land$ self._line$[:3] \in$ ('"""', "'''"):
            if self._docprocess $=$ 'plain':
                self.tr_docstring_plain ()
            elif self._docprocess $=$ 'struct':
                self.tr_docstring_struct ()
            else:
                raise ValueError, 'Illegal␣value␣for␣doc␣process.'
        else:
            self._translation.append ('\\I{%d}{%d}' %
                  (self._line_nr, self._indentation))
            self.tr_code ()
            if $\neg$self._no_break $\land$ self._translation$[-1][-8:] =$ '\\colon␣$' $\land$
                  self._translation$[0][:3] \neq$ '\\E{':
                self._translation.insert (0, '\\PB')
                self._no_break $\leftarrow 1$
            else:
                self._no_break $\leftarrow$ empty $\neq 0$
            if len (self._line) $> 0$:
                if self._line$[:1] \neq$ '#': raise error
                if self._translation$[-1][:1] =$ '$':
                    self._translation.append ('\\quad␣')
                self.tr_comment ()
            else:
                self._translation.append ('\n')
    except EOFError: pass
    return self._translation
def translation (self):
    return self._translation

error $\leftarrow$ 'py2tex␣error'
```

```
class Re:
    def __init__(self, regex):
        self._regex ← regex
    def match(self, string, pos ← 0):
        m ← self._regex.match(string, pos)
        result ← −1
        if m:
            result ← m.end(0)
        return result
    def search(self, string, pos ← 0):
        m ← self._regex.search(string, pos)
        result ← −1
        if m:
            result ← m.start(0)
        return result

class Regex:
    def compile(self, regex):
        return Re(re.compile(regex))

regex ← Regex()

interpret_re ← regex.compile('%[␣\t]*(La)?TeX[␣\t]*$')
verbatim_re ← regex.compile('%[␣\t]*ASCII[␣\t]*$')
ordinary_re ← regex.compile('[^#$%&<>\\\\^_{|}~]*')
white_re ← regex.compile('[␣\t]*')
function_re ← regex.compile('[␣\t]*\\(')
comment_re ← regex.compile('(##|[^#])*')
escape_re ← regex.compile('([^\\\\]|\\\\.)*\\\\')
numeral_re ← regex.compile(string.joinfields((
    '0[xX][0-9A-Fa-f]+',
    '[0-9]+\\.?[eE][+-]?[0-9]+[jJLl]?',
    '[0-9]*\\.[0-9]+[eE][+-]?[0-9]+[jJLl]?',
    '[1-9][0-9]*[jJLl]?',
    '0[0-7]*'), '|'))
```

```
token_re ← regex.compile (string.joinfields ((
    '[A-Za-z_][A-Za-z_0-9]*',
    "'('')?", '"("")?',
    '==?', '[<>!]=', '<>',
    '<<', '>>',
    '\\[]',
    '[*][*]',
    '[\\\\{}$&|^~%:*/+-]'), '|'))
quote_re ← regex.compile ('("("")?)|'"('('')?)")

TeX_code ← {
    '\\':'$\\backslash$', '|':'$\\vert$',
    '<':'$<$', '>':'$>$',
    '{':'$\\{$', '}':'$\\}$'}
reserved ← ('access', 'and', 'break', 'class', 'continue',
    'def', 'del', 'elif', 'else', 'except', 'exec',
    'finally', 'for', 'from', 'global', 'if',
    'import', 'in', 'is', 'is␣not', 'not', 'not␣in', 'or',
    'pass', 'print', 'raise', 'return', 'try', 'while')
single ← ('else', 'finally', 'try', '-', '+')
reserved_operators ←
        ('and', 'in', 'is', 'is␣not', 'not', 'not␣in', 'or', '**')
special_ctrl ← {'\a':'\\a', '\b':'\\b', '\f':'\\f',
    '\n':'\\n', '\r':'\\r', '\t':'\\t', '\v':'\\v'}

def usc_protect (ident):
    ident ← string.joinfields (string.splitfields (ident, '_'), '\\_')
    return ident

def ctrl_protect (str):
    result ← ''
    for $c \in$ str:
        $o$ ← ord ($c$)
        if $o < 32 \lor o \geq 127$:
            if special_ctrl.has_key ($c$):
                result ← result + special_ctrl[$c$]
            else:
                result ← '%s\\%03o' % (result, $o$)
        else:
            result ← result + $c$
    return result
```

```
def tr_numeral (token):
    end ← token[−1]    # Preserve the type signifier (jJlL) if any.
    numeral ← string.lower (token)
    if numeral[: 2] = '0x':
        || (0x1A, 0x2B)
        return '\\HEX{%s}' % string.upper (numeral[2: ])
    if ¬(end ∈ 'jJlL'):    # Check if end is a signifier.
        end ← ''
    else:
        numeral ← numeral[: −1]    # Strip the signifier.
    pos ← string.find (numeral, 'e')
    if pos ≥ 0:
        || ($12.4{\cdot}10^{-78}, .3333{\cdot}10^{+0}, .1{\cdot}10^{6}, 2.{\cdot}10^{1}, 0.{\cdot}10^{1}, 1{\cdot}10^{4}$)
        return '\\EXP{%s}{%s}{%s}' % (numeral[: pos], numeral[pos + 1: ], end)
    if numeral[: 1] = '0' ∧ numeral ≠ '0':
        || (0377, 037 8)
        return '\\OCT{%s}' % numeral[1: ]
    || (.333, 3.141592) (0, 1, 42)
    return '\\NUM{%s}{%s}' % (numeral, end)
```

2 Py2tex – Sun Apr 2 20:58:58 2006

!/usr/local/bin/python

Py2tex, script to translate Python source to LaTeX code.

```
import getopt, os, sys
ospath = os.path
if not ospath.isabs(sys.argv[0]):
    sys.path.insert(0, ospath.dirname(sys.argv[0]))
from py2tex import Interpret
```

The `-m` and `-n` options affect the typographic treatment of the tokens =, ==, <=, >=, !=, <>, <<, >>, **in**, **not in**, **is**, and **is not**. When `-n` is in effect these tokens are printed as they appear in the Python source. When `-m` (the default) is in effect they are translated to mathematical symbols that are designed for use in typeset documents. (Please read Chapter *Book Printing versus Ordinary Typing* from the TEXbook before you use the `-n` option.) The `-o` option causes the script to write the LaTeX output to the specified file, rather than standard output.

The `-d` option affects the way the script handles documentation strings. The option `-d␣none` treats documentation strings as ordinary strings. The option `-d␣plain` typesets the docstrings like verbatim comments except with thick solid lines instead of thin double lines. (OK, so that's not clear, try it and see.) Finally, `-d␣struct` typesets the docstrings as structured text as defined by the doc-sig.

The `-i` and `-v` options determine whether the comments will be interpreted by (La)TEX (`-i`) or typeset verbatim (`-v`).

```
    || Default values.
interpret = 1
math = 1
output = None
docprocess = 'none'
```

```
    || Parse options.
optlist, args = getopt.getopt (sys.argv[1: ], 'imno:vd:')
for pair in optlist:
    key = pair[0]
    if pair[0] == '-m':
        math = 1
    if pair[0] == '-n':
        math = 0
    if pair[0] == '-o':
        output = pair[1]
    if pair[0] == '-d':
        docprocess = pair[1]
    if pair[0] == '-i':
        interpret = 1
    if pair[0] == '-v':
        interpret = 0

if args == [ ]:
    args = ['-']

    || Open output file.
if output == None:
    outfile = sys.stdout
else:
    outfile = open (output, 'w')

    || Translate source files.
for name in args:
    file = Interpret (name, math, interpret, docprocess)
    outfile.write (file.translation ()[0])
    while file.translate () != None:
        for scrap in file.translation ():
            outfile.write (scrap)

    || Close output file.
outfile.close ()
```

3 Py2tex.sty

The py2tex documentstyle option can be used to typeset files generated by the py2tex script. Directions on the usage of the script and the documentstyle option can be found in py2tex.py.

The implementation and customization of the documentstyle are documented in py2tex.doc.

This file can be used both as a style file for LaTeX documents, and as a package for LaTeX2ε documents.

```
\@ifundefined{ProvidesPackage}{}%
  {\ProvidesPackage{py2tex}}
```

3.1 Customization

If you would like to change the definition of one or more macros in this section, you are advised to make a new style file along the following lines, rather than change this file.

```
% mypy.sty
\input py2tex.sty
⟨new definitions⟩
% EOF
```

Such a derived style file can be used as a document style option instead of py2tex.

In the rest of this section the customizable macros and their default definitions are documented.

The \PythonFile macro is meant to typeset a heading. It is called with the name of the source file as the first parameter and a time stamp as the second parameter. It uses the \PythonSection command to generate the header. By default it uses the \section command, but the \PythonSection macro can be \let equal to an arbitrary sectioning command (or any other command that takes two parameters with syntax [#1]{#2}).

```
\let\PythonSection=\section
\def\PythonFile#1#2{\PythonSection[\upcasechar#1]%
  {\upcasechar#1\thinspace--\thinspace#2}\bigskip}
\def\upcasechar#1{\uppercase{#1}}
```

The \PythonEmptyLines macro is called to typeset empty lines in the source. The number of empty lines is given as a parameter, but is ignored by default. The default behavior is to typeset just one blank line.

```
\def\PythonEmptyLines#1{\PythonPageBreak
  \vskip\baselineskip }
\def\PythonNumber#1{\llap{\rm\small #1\ }}
```

The \PythonCalcIndent macro is called once, just before the \input macro, to calculate the indentation level. By default it measures the width of a box with the keyword **def** and some whitespace in it.

```
\def\PythonCalcIndent{%
  \setbox0=\hbox{$\K{def}\ $}\PythonDent=\wd0
  \advance\PythonDent by .8 pt }
```

The following macros are used to typeset various Python constructs. Note that they are all designed to be used in math mode. By default, variables are typeset upright and functions slanted. Use the macro `PythonSlantedVariables` to have it just the other way round. (I personally prefer the default setting, except when I use many one-letter variables which are typeset in math italics.)

```
\ifx\selectfont\undefined
  \let\PythonFont=\relax
  \let\PythonSlFont=\sl
  \let\PythonBfFont=\bf
\else
  \message{NFSS font settings}
  \let\PythonFont=\sffamily
  \def\PythonSlFont{\PythonFont\slshape}
  \def\PythonBfFont{\PythonFont\bfseries}
\fi
\def\PythonSlantedFunctions{%
  \def\PythonFunction##1{\mbox{\PythonSlFont ##1\/}}%
  \def\PythonVariable##1{\mbox{\PythonFont ##1}}}
\def\PythonSlantedVariables{%
  \def\PythonFunction##1{\mbox{\PythonFont ##1}}%
  \def\PythonVariable##1{\mbox{\PythonSlFont ##1\/}}}
\PythonSlantedFunctions
\def\PythonKeyword#1{\mbox{\PythonBfFont #1}}%
\def\PythonOperator#1{\mathrel{\PythonKeyword{#1}}}
\def\PythonSymbol#1{#1}
\def\PythonHexadecimal#1{\mbox{\tt 0x#1}}
\def\PythonOctal#1{\mbox{\it 0#1\/}}
\def\PythonExponentFloat#1#2#3{#1{\cdot}10^{#2}{\mathrm\relax #3}}
\def\PythonPlainNumber#1#2{#1{\mathrm\relax#2}}
\def\PythonBreakPoint{\penalty 100\relax }
```

At the end of this file there is a section that specifies how the operators and relations should be typeset. These definitions are at the end because they use the macro `\PythonDefIntern`. This macro can also be used to override these definitions. Likewise the macro `\PythonDef` can be used to determine how certain variables and/or functions should be typeset. For examples of the use of these macros, take a look at the source code of the following fragment.

definitions.py

if $\vec{a} = [a_1, a_2]$:
    print *print*$_i$ $(\vec{a})$

Somewhat more intricate customization.

print *repr* (REPR), *str* (STR), *foo* (bar)

3.2 Implementation

In this section the implementation of the style is documented.

First a dimension register is allocated to hold the standard indentation. Furthermore an `\if` construct is initialized that is used to distinguish between the normal and the starred form of `\PythonSource`.

```
\newdimen\PythonDent \PythonDent=2em
\newif\ifOuterPython
```

The `\PythonSource` macro checks for the star, then it sets the `OuterPython` flag accordingly, and calls `\@PythonSource`.

```
\def\PythonSource{%
  \@ifstar
    {\OuterPythonfalse\@PythonSource}%
    {\OuterPythontrue\@PythonSource}}
```

The `\@PythonSource` macro does the real work.

```
\def\@PythonSource#1{\begingroup
  \PythonMode
```

Then a lot of short versions of Python specific macros are `\let` equal to their long forms.

```
  \let\B=  \PythonBlockComment
  \let\BP= \PythonBreakPoint
  \let\DS= \PythonDocString
  \let\E=  \PythonEmptyLines
  \let\ESC=\par
  \let\EXP=\PythonExponentFloat
  \let\F=  \Python@function
  \let\HEX=\PythonHexadecimal
  \let\I=  \PythonIndent
  \let\K=  \Python@keyword
  \let\M=  \PythonMetaVariable
  \let\NUM=\PythonPlainNumber
  \let\O=  \Python@operator
  \let\OCT=\PythonOctal
  \let\PB= \PythonPageBreak
  \let\S=  \PythonString
  \let\V=  \Python@variable
  \let\Y=  \Python@symbol
```

Normally the file name and time are put into a heading and lines are numbered, but this is turned off in the starred version of the `\PythonSource` macro.

```
  \ifOuterPython
    \let\File=\PythonFile
    \let\PythonNr=\PythonNumber
  \else
    \let\File\@gobbletwo
    \let\PythonNr\@gobble
  \fi
```

Finally calculate the indentation level.

```
  \PythonCalcIndent
```

Now `\input` the file. The `\par` ensures that hanging indentation is not lost for the last line of code.

```
  \input #1
  \par\endgroup}
```

The `\PythonMode` macro sets some TEX parameters in order to typeset Python code, rather than running text. This macro is complementary to the `\TextMode` macro defined below.

```
\def\PythonMode{
  \par
  \parskip=0mm plus 1 pt
  \parindent=0mm
  \rightskip=0mm plus .5\hsize
  \interlinepenalty=300 }
```

The `\PythonIndent` macro is used to start a new line of Python code. It starts a new paragraph with the proper indentation and one and a half standard indentation more hanging indentation. Furthermore it calls `\PythonNr` to typeset the line number.

```
\def\PythonIndent#1#2{\endgraf\penalty 500
  \hangindent=#2\PythonDent
  \advance\hangindent by 1.5\PythonDent
  \hangafter=1
  \leavevmode\strut\PythonNr{#1}%
  \hskip #2\PythonDent\relax }
```

The `\PythonOuterBlock` and `\PythonOuterBlockEnd` macros delimit an unindented block comment. An outer block does not imply grouping and is delimited by `\OuterMarkers`. The starred form of `\PythonOuterBlock` leaves out the opening marker.

```
\def\PythonOuterBlock{\TextMode
  \@ifstar{}{\@start@outer@block}}
\def\@start@outer@block{%
  \par\OuterMarker\nobreak\vskip -\parskip}
\def\PythonOuterBlockEnd{%
  \par\nobreak\OuterMarker\PythonMode}
```

The `\PythonBlockComment` macro starts a block comment. It defines `\subtract` to yield the amount of indentation to subtract from the width of the box containing the comment and calls \PythonInnerBlock to do the real work.

```
\def\PythonBlockComment#1#2{\PythonPageBreak
  \PythonIndent{#1}{#2}%
  \def\subtract{-#2\PythonDent}\PythonInnerBlock}
```

The \PythonInnerBlock macro starts a \hbox containing the lines that mark a block comment and a \vtop that contains the actual comment (So the line number will be aligned with the first line of the comment). It uses \subtract defined by \PythonBlocComment to reduce the width of the \vtop. It also subtracts the width of the marker from the width of the \vtop.

```
\def\PythonInnerBlock#{\hbox\bgroup\strut \Marker
  \vtop\bgroup
    \TextMode
    \let\ESC=\PythonEscapeBlockComment
    \advance\hsize by \subtract
    \setbox0=\hbox{\Marker}\advance\hsize by -\wd0
    \textwidth=\hsize
    \linewidth=\hsize
```

The next command causes the \hbox to be wrapped up immediately when the \vtop is completed.

```
    \aftergroup\egroup
```

Gobble the opening brace before reading the comment.

```
    \let\next=}
```

The \PythonDocBlock macro starts a block that contains a doc string.

```
\def\PythonDocBlock{\TextMode
  \@ifstar{}{\@start@doc@block}}
\def\@start@doc@block{%
  \par\DocOuterMarker\nobreak\vskip -\parskip}
```

The \PythonDocBlockEnd macro ends a block that contains a doc string.

```
\def\PythonDocBlockEnd{%
  \par\nobreak\DocOuterMarker\PythonMode}

```

The \PythonDocString macro formats a doc string in a way similar to the \PythonInnerBlock macro, except that it uses a different marker.

```
\def\PythonDocString#1#2{\PythonPageBreak
  \PythonIndent{#1}{#2}%
  \def\subtract{-#2\PythonDent}\PythonDocStringHelper}

\def\PythonDocStringHelper#{\hbox\bgroup\strut \DocStringMarker
  \vtop\bgroup
    \TextMode
    \advance
    \hsize by \subtract
    \setbox0=\hbox{\DocStringMarker}\advance\hsize by -\wd0
    \textwidth=\hsize
    \linewidth=\hsize
    \aftergroup\egroup
    \let\next=}
```

The `\TextMode` macro sets some TeX parameters to typeset running text rather than Python code.

```
\def\TextMode{\par
  \rightskip=0mm%
  \parskip=\baselineskip
  \advance\parskip by 0mm plus 1pt
  \interlinepenalty=0}
```

The `\PythonEscapeBlockComment` macro can be used in block comments by the name `\ESC` to escape the `\vtop` containing the comment and typeset material in outer vertical mode. First the `\vtop` started by `\PythonBlockComment` is closed. This also closes the `\hbox` around it, leaving us in outer vertical mode. Then two levels of grouping are opened. One to contain parameter settings local to the escaped comment and one in order to end the last paragraph in the comment–with an `\aftergroup` construction–before closing the outer level of grouping.

```
\def\PythonEscapeBlockComment{\par
    \vskip.5\baselineskip\vskip.5\MarkerSep
  \egroup\par\nobreak
  \bgroup
    \vskip-.5\baselineskip\vskip-.5\MarkerSep
    \EscapeMarker\nobreak
    \TextMode
    \bgroup
      \vskip -\parskip
      \aftergroup\EndEscape}
\def\EndEscape{\par\nobreak\EscapeMarker\egroup}
```

The `\MarkerSep` dimension variable determines the amount of whitespace separating the lines typeset with the `\Marker` and `\OuterMarker` macros.

```
\newdimen\MarkerSep \MarkerSep=2pt
```

The `\Marker` macro is used to typeset the lines that mark a block comment.

```
\def\Marker{\vrule\hskip\MarkerSep\vrule\ }
```

The `\DocStringMarker` macro is used to typeset the lines that mark a doc string.

```
\def\DocStringMarker{\vrule width\MarkerSep\ }
```

The `\OuterMarker` macro is used to typeset the lines that mark unindented comment blocks and escaped sections of block comments.

```
\def\OuterMarker{\par\nointerlineskip
  \vbox to \baselineskip{\vss
```

```
    \hrule width\textwidth \vskip\MarkerSep
    \hrule width\textwidth \vss}%
  \nointerlineskip}
\let\EscapeMarker=\OuterMarker
```

The \DocOuterMarker macro is used to typeset the lines that mark unindented doc string blocks.

```
\def\DocOuterMarker{\par\nointerlineskip
  \vbox to \baselineskip{\vss
    \hrule height\MarkerSep width\textwidth \vss}%
  \nointerlineskip}
```

The \PythonPageBreak macro is called at several points to allow a page to be short rather than break the code at an ugly point. (Breaking before block comments and empty lines is considered good and so is breaking before a line that has less indentation than the next, except when it is preceded by a block comment.)

```
\def\PythonPageBreak{\par
  \vskip 0mm plus  4\baselineskip \penalty -200
  \vskip 0mm plus -4\baselineskip \relax }
```

The \PythonString macro starts a group in which the left quote character is active and prints as an undirected quote.

```
\input{ts1enc.def}
\input{t1enc.def}
%\DeclareTextSymbolDefault{\textquotesingle}{TS1}
\DeclareTextSymbolDefault{\textquoteright}{T1}
\DeclareTextSymbolDefault{\textquotedbl}{T1}
{\catcode`\'=\active
  \catcode`\"=\active
  \gdef\PythonString#{\bgroup\tt
    \catcode`\'=\active\def'{\textquoteright}%
    \catcode`\"=\active\def"{\textquotedbl}%
    \let\next= }}
```

The \PythonDef defines how a function or variable sould be typeset. Usage: \PythonDef{name}{definition}. In the definition #1 refers to the type of identifier (either V or F), #2 is the default macro for this type (either \PythonFunction or \PythonVariable) and #3 refers to the name of the identifier.

E.g., \PythonDef{row_alpha}{\langle\alpha\rangle} has the effect that #row_alpha# will be typeset as $\langle\alpha\rangle$.

```
\def\prefix@user{ExcUser@}
\def\prefix@intern{ExcIntern@}
\def\Python@def#1{\endgroup\expandafter\def
    \csname \@prefix #1\endcsname ##1##2##3}
\def\PythonDef{\let\@prefix=\prefix@user
  \@prepare\Python@def}
```

```
\def\PythonDefIntern{\let\@prefix=\prefix@intern
  \@prepare\Python@def}
\def\Python@let#1{\endgroup
  \expandafter\let\csname \@prefix #1\endcsname }
\def\PythonLet{\let\@prefix=\prefix@user
  \@prepare\Python@let}
\def\PythonLetIntern{\let\@prefix=\prefix@intern
  \@prepare\Python@let}
\def\PythonDefault#1{\PythonLet{#1}\relax}
\def\PythonDefaultIntern#1{\PythonLetIntern{#1}\relax}
```

The \Python@function macro calls \ExcUser@#1 or, if that doesn't exist, PythonFunction. The \Python@variable macro does the same, but calls the macro \PythonVariable by default.

The \Python@keyword, \Python@operator and \Python@symbol call either \ExcIntern@#1 or \PythonKeyword, \PythonOperator or \PythonSymbol respectively.

```
\def\Python@function{\Python@identifier
  UF\PythonFunction}
\def\Python@variable{\Python@identifier
  UV\PythonVariable}
\def\Python@symbol{\@prepare\Python@identifier
  IY\PythonSymbol}
\def\Python@keyword{\Python@identifier
  IK\PythonKeyword}
\def\Python@operator{\Python@identifier
  IO\PythonOperator}
\chardef\other=12
\def\@prepare{\begingroup
  \def\do##1{\catcode`##1=\other}\dospecials
  \catcode`\{=1 \catcode`\}=2 }
{\catcode`\_=\other \gdef\@underscore{_}}
\def\global@let@tempa#1{\global\let\@tempa#1}
\def\Python@identifier#1#2#3#4{%
  \if #2Y\relax \endgroup \fi
  \begingroup\let\_=\@underscore \relax
    \if #1U\relax \let\@prefix=\prefix@user
    \else \let\@prefix=\prefix@intern \fi
    \@ifundefined{\@prefix #4}{%
        \global\let\@tempa=\@gobble
      }{\expandafter\global@let@tempa
          \csname \@prefix #4\endcsname
      }\endgroup\let\@tempb=\@tempa
      \@tempb{#2}#3{#4}}
```

3.3 More customization

Here are at last the promised definitions that state how the various Python constructs should be typeset.

```
\PythonDefIntern{[]}{[\,]}
\PythonDefIntern{&}{\mathbin\&}
\PythonDefIntern{|}{\mathbin\vert}
\PythonDefIntern{^}{\mathbin{{}^\wedge}}
\PythonDefIntern{~}{\mathop{{}^\sim}}
\PythonDefIntern{%}{\mathbin{\%}}
\PythonDefIntern{:}{\colon}
```

There are two predefined ways to handle assignment and equality. The default one is to type set the assignment operator as a left arrow (←) and the equality relation as an equals sign (=). The alternative is to typeset these tokens as themselves, *i.e.*, = and == respectively.

```
\def\PythonToAssign{%
  \PythonDefIntern{=}{\leftarrow}%
  \PythonDefIntern{==}{=}}
\def\PythonIsAssign{%
  \PythonDefaultIntern{=}%
  \PythonDefIntern{==}{\mathrel{==}}}
```

By default, the relations and operators are typeset in their corresponding mathematical notation. The alternative is to have them typeset as they occur in the source. Note that `\PythonMath` implies `\PythonToAssign` and that `\PythonNoMath` implies `\PythonIsAssign`.

```
\def\PythonMath{%
  \PythonToAssign
  \PythonDefIntern{and}{\land}%
  \PythonDefIntern{in}{\in}%
  \PythonDefIntern{is}{\equiv}%
  \PythonDefIntern{is not}{\not\equiv}%
  \PythonDefIntern{not}{\neg}%
  \PythonDefIntern{not in}{\not\in}%
  \PythonDefIntern{or}{\lor}%
  \PythonDefIntern{<=}{\le}%
  \PythonDefIntern{>=}{\ge}%
  \PythonDefIntern{!=}{\ne}%
  \PythonDefIntern{<>}{\ne}%
  \PythonDefIntern{<<}{\ll}%
  \PythonDefIntern{>>}{\gg}}
\def\PythonNoMath{%
  \PythonIsAssign
  \PythonDefaultIntern{and}%
  \PythonDefaultIntern{in}%
  \PythonDefaultIntern{is}%
  \PythonDefaultIntern{is not}%
  \PythonDefIntern{not}{{##2{##3}}\mathbin{}}%
  \PythonDefaultIntern{not in}%
  \PythonDefaultIntern{or}%
  \PythonDefIntern{<=}{\mathrel{<=}}%
  \PythonDefIntern{>=}{\mathrel{>=}}%
  \PythonDefIntern{!=}{\mathrel{!\!=}}%
  \PythonDefIntern{<>}{\mathrel{<>}}%
  \PythonDefIntern{<<}{\mathrel{<\!<}}%
  \PythonDefIntern{>>}{\mathrel{>\!>}}}
\PythonMath
```

The \PythonSubscript and \PythonSubscriptV macros can be used to typeset the suffix of an identifier with an underscore, as a subscript. For example \PythonLet{part_i}\PythonSubscript will cause part_i to be typeset as part$_i$. The V-version of the macro is intented to be used with identifiers where the base consists of only one letter. For example, the command \PythonLet{a_1}\PythonSubscriptV will cause a_1 to be typeset as a_1.

```
\def\Ident@Base#1\_#2.{#1}
\def\Ident@Sub#1\_#2.{#2}
\def\PythonSubscript#1#2#3{%
  #2{\Ident@Base#3.}_{\Ident@Sub#3.}}
\def\PythonSubscriptV#1#2#3{%
  \Ident@Base#3._{\Ident@Sub#3.}}
```

4 Struct2latex.py – Sun Apr 2 21:53:53 2006

Convert structured text to LaTeX.

LaTeX - A class that converts structured text (cf. the *doc-sig*[1]) into a format readable by LaTeX. Based on the class *HTML* authored by Jim Fulton which appears in *StructuredText.py*.

Usage (this is long and rambling so I can test it with itself...):

1. Put *struct2latex.py* someplace that python and can find it.
2. Create your LaTeX file by:
   1. (a) Creating a **LaTeX** object (e.g., `st = LaTeX(string)`) .
   2. (b) Getting the LaTeXified string by converting the **LaTeX** object to a string (.e.g, `lt = str(st)` or `print st`) .
   3. (c) Save your LaTeXified string somewhere.
3. You should be able to include the LaTeX text in any LaTeX file. Two ways I use it are:
   - Use the text by itself by putting it in a stub file. For example:

     ```
     \\documentstyle[11pt]{article}

     \\begin{document}

     \\include{docstring}

     \\end{document}
     ```

   - I'm using use it to support structured text in *py2tex*.
4. Run LaTeX.
5. Once you have a dvi file you're on your own....

There are some caveats (of course):

Characters I believe all the LaTeX special characters (&%#_{}~^\) should be properly escaped (with the exception of $ - see below, but no guarantees.

- And now it should allow bullet lists that are adjacent to work.

[1] http://www.python.org/sigs/doc-sig/

- This is provided by the magic of regsub.gsub.
- But who knows it may have some horrible side effects...

Equations I thought, “as long as we’re using LaTeX, we should have access to equations.” So, `$` is used to invoke math mode, just as in LaTeX. For example, `$x = \oint y\,dy$` produces $x = \oint y\,dy$. $ obeys the same rules as ’, so you usually shouldn’t have to quote it - although that would probably be safer...

Quotes The normal LaTeX style quotes work fine as long as there is no white space inside the quote (’).

```
import re, string
import StructuredText
ST ← StructuredText
regex ← ST.regex
regsub ← ST.regsub

href_re ← regex.compile ('[.][.]␣(".+")[␣\t]*(.*)\n')
line2_re ← regex.compile ('.*\n([␣\t]*\n)*([␣\t]*)')
slashable_re ← regex.compile ('[$&%#_{}]')
quotable_re ← regex.compile ('[~^\\\\]')
eqn_re ← regex.compile (
      "[␣\t\n(]\\"+
      "$([^␣\t\n’$]([^\n’]*[^␣\t\n’])?)\\$"+
      "([)␣\t\n,.:;!?])"
)
carrot_re ← regex.compile ("\\^")

expand_bullet ← regex.compile ('\n[␣\t\n]*[o*-][␣\t\n]')
expand_deflist ← regex.compile ('\n[␣\t\n]*[^\n]+[␣\t]+--[␣\t\n]')

def _split (s):
      Split a string into normal and quoted pieces.

      Splits a string into normal and quoted (or math mode) sections. Returns
      a list where the even elements are normal text, and the odd elements are
      quoted.  The appropiate quote tags ($ and \\verb) are applied to the
      quoted text.
      r ← [ ]
      while 1:
          epos ← eqn_re.search (s)
          qpos ← ST.code.search (s)
          if epos = qpos:   # = −1
              break
          elif (qpos = −1) ∨ (epos ≠ −1 ∧ epos < qpos):
              r.append (s[: epos])
              end ← epos + eqn_re.match (s[epos: ])
              arg ← [eqn_re.group (1), eqn_re.group (3)]
              if ¬arg[1]: arg[1] ← ''
              r.append ('␣$%s$%s␣' % tuple (arg))
          else:   # (epos = −1) ∨ (qpos ≠ −1 ∧ epos > qpos):
              r.append (s[: qpos])
              end ← qpos + ST.code.match (s[qpos: ])
              arg ← [
                  regsub.gsub (carrot_re, '^\\\\verb@\\g<0>@\\\\verb^',
                  ST.code.group (1)),
                  ST.code.group (3)
              ]
              if ¬arg[1]: arg[1] ← ''
              r.append ('␣\\verb^%s^%s␣' % tuple (arg))
          s ← s[end: ]
      r.append (s)
      return r

def _ctag (str, hrefs ← ()):
```

Quote, tag, and escape the text.

This is a modified version of the `ctag` function appearing in StructuredText.py. The differences include,

- it uses _split, so that it avoids escaping text in quotes or in mathmode.
- it processes hrefs.
- it escapes LaTeX special characters.
- it doesn't try to find duplicate list items - that got moved into LaTeX.

```
    if str ≡ None: str ← ''
    str ← '␣%s' % str    # prepend a space
    str ← _split (str)
    for $i \in$ xrange (len (str)):
        if $\neg i$ % 2:
            str[$i$] ← regsub.gsub (quotable_re, '\\\\verb@\\g<0>@', str[$i$])
            str[$i$] ← regsub.gsub (slashable_re, '\\\\\\g<0>', str[$i$])
            str[$i$] ← regsub.gsub (ST.strong, '␣{\\\\bfseries␣\\1}\\2', str[$i$])
            str[$i$] ← regsub.gsub (ST.em, '␣{\\\\itshape␣\\1}\\2', str[$i$])
            for ref, link ∈ hrefs:
                tag ← '{\slshape␣%s}\\footnote{%s}' % (ref[1: $-1$], link)
                str[$i$] ← string.joinfields (string.split (str[$i$], ref), tag)
    return string.joinfields (str)

def _strip_hrefs (string):
```

Strip hrefs out of a string.

Strip the hrefs of the form ' *string.* Return string, as well as a dictionary containing the stripped references.

```
    hrefs ← [ ]
    $s$ ← string
    $l$ ← href_re.search ($s$)
    while $l \neq -1$:
        hrefs.append (href_re.group (1, 2))
        $s$ ← $s[l + 1:]$
        $l$ ← href_re.search ($s$)
    string ← regsub.gsub (href_re, '', string)
    return string, hrefs

def _separate_bullets (string):
    Separate list items by a newline.
    string ← regsub.gsub (expand_bullet, '\n\\g<0>', string)
    string ← regsub.gsub (expand_deflist, '\n\\g<0>', string)
    return string

class LaTeX (ST.StructuredText):

    Translate StructuredText to LaTeX.

    This is loosely based on Jim Fulton's class HTML.

    def __init__ (self, aStructuredString, level ← 1, isdoc ← 1):
        Create a LaTeX object.

        self.level ← level
        aStructuredString ← ST.untabify (aStructuredString)
        if isdoc:
            if line2_re.match (aStructuredString) $\neq -1$:
                aStructuredString ← line2_re.group (2) + aStructuredString
            aStructuredString, self.hrefs ← _strip_hrefs (aStructuredString)
            aStructuredString ← _separate_bullets (aStructuredString)
        paragraphs ← regsub.split (aStructuredString, ST.paragraph_divider)
        paragraphs ← map (ST.indent_level, paragraphs)
        self.structure ← ST.structure (paragraphs)

    def _str (self, structure, level):
        Translate structure to LaTeX.

        Driver for the translation. Based on HTML._str. Differences include:

          1. changed the handling of examples so that bullets could have
             examples too.
        if type (structure) = type (''):
            return structure
        $r$ ← ''
        for $s \in$ structure:
            # print $s[0]$, '\n', len ($s[1]$), '\n\n'
            if ST.example.search ($s[0]$) $\geq 0 \wedge s[1]$:
                s0, s1 ← $s[0]$, self.pre ($s[1]$)
            elif $s[0][-2:]$ = '::' $\wedge s[1]$:
                s0, s1 ← $s[0][:-1]$, self.pre ($s[1]$)
            else:
                s0, s1 ← $s[0], s[1]$
            ||
            if ST.bullet.match (s0) $\geq 0$:
                $p$ ← ST.bullet.group $(1)$
                $r$ ← self.ul ($r, p$, self._str (s1, level))
            elif ST.ol.match (s0) $\geq 0$:
                $p$ ← ST.ol.group $(3)$
                $r$ ← self.ol ($r, p$, self._str (s1, level))
            elif ST.olp.match (s0) $\geq 0$:
                $p$ ← ST.olp.group $(1)$
                $r$ ← self.ol ($r, p$, self._str (s1, level))
            elif ST.dl.match (s0) $\geq 0$:
                $t, d$ ← ST.dl.group $(1, 2)$
                $r$ ← self.dl ($r, t, d$, self._str (s1, level))
            elif ST.nl.search (s0) $< 0 \wedge$ s1:
                || Treat as a heading
                $t$ ← s0
                $r$ ← self.head ($r, t$, level, self._str (s1, level $+ 1$))
            else:
                $r$ ← self.normal ($r$, s0, self._str (s1, level))
        return $r$

    def ul (self, before, $p$, after):
        | Process an unordered list.
        if before[$-14$:] = '\\end{itemize}\n':
            return '''\
%s
\\item␣%s%s

\\end{itemize}
''' % (before[: $-15$], _ctag ($p$, self.hrefs), after)
        else:
            return '''\
%s\\begin{itemize}

\\item␣%s%s

\\end{itemize}
''' % (before, _ctag ($p$, self.hrefs), after)

    def ol (self, before, $p$, after):
        ┃Process an ordered list.
        if before[−16:] = '\\end{enumerate}\n':
            return '''\
%s
\\item␣%s%s
\\end{enumerate}
''' % (before[: −16], _ctag (p, self.hrefs), after)
        else:
            return '''\
%s\\begin{enumerate}

\\item␣%s%s

\\end{enumerate}
''' % (before, _ctag (p, self.hrefs), after)

    def dl (self, before, t, d, after):
        ┃Process a description list.
        if before[−18:] = '\\end{description}\n':
            return '''\
%s
\\item[%s]%s%s

\\end{description}
''' % (before[: −18], _ctag (t, self.hrefs), _ctag (d, self.hrefs), after)
        else:
            return '''\
%s\\begin{description}

\\item[%s]%s%s

\\end{description}
''' % (before, _ctag (t, self.hrefs), _ctag (d, self.hrefs), after)

    def head (self, before, t, level, d):
        ┃Process a heading.
        t ← "{\\bfseries␣%s␣}" % _ctag (t, self.hrefs)
        return '''\
%s\\begin{description}
\\item[%s]\\␣

%s
\\end{description}
''' % (before, t, d)
```

```
    def normal(self, before, p, after):
        | Process a normal paragraph.
        return '%s\n%s\n%s\n' % (before, _ctag(p, self.hrefs), after)

    def pre(self, structure, tagged ← 0):
        | Process some pre-formatted (example) text.
        if ¬structure: return ''
        if tagged:
            r ← ''
        else:
            r ← '\\begin{verbatim}\n'
        for s ∈ structure:
            r ← "%s%s\n\n%s" % (r, s[0], self.pre(s[1], 1))
        if ¬tagged: r ← r + '\\end{verbatim}\n'
        return r

    def __str__(self):
        | Return the translated text.
        return self._str(self.structure, self.level)

if __name__ = '__main__':
    print LaTeX(__doc__)
```

5 StructuredText.py – Sun Apr 2 22:54:15 2006

!/usr/local/bin/python – # -*- python -*- $What$

Structured Text Manipulation

Parse a structured text string into a form that can be used with structured formats, like html.

Structured text is text that uses indentation and simple symbology to indicate the structure of a document.

A structured string consists of a sequence of paragraphs separated by one or more blank lines. Each paragraph has a level which is defined as the minimum indentation of the paragraph. A paragraph is a sub-paragraph of another paragraph if the other paragraph is the last preceedeing paragraph that has a lower level.

Special symbology is used to indicate special constructs:

- A paragraph that begins with a `-`, `*`, or `o` is treated as an unordered list (bullet) element.
- A paragraph that begins with a sequence of digits followed by a white-space character is treated as an ordered list element.
- A paragraph that begins with a sequence of sequences, where each sequence is a sequence of digits or a sequence of letters followed by a period, is treated as an ordered list element.
- A paragraph with a first line that contains some text, followed by some white-space and `--` is treated as a descriptive list element. The leading text is treated as the element title.
- Sub-paragraphs of a paragraph that ends in the word `example` or the word `examples` is treated as example code and is output as is.
- Text enclosed single quotes (with white-space to the left of the first quote and whitespace or puctuation to the right of the second quote) is treated as example code.
- Text surrounded by `*` characters (with white-space to the left of the first `*` and whitespace or puctuation to the right of the second `*`) is emphasized.

- Text surrounded by ** characters (with white-space to the left of the first ** and whitespace or puctuation to the right of the second **) is emphasized.

  $Id: StructuredText.py,v 1.2 1999/05/01 00:56:45 daniel Exp $

Jim Fulton, jim@digicool.com

(540) 371-6909

```

import re

class Re:
    def __init__(self, regex):
        self._regex ← regex
        self._match ← None
    def match(self, string, pos ← $0$):
        $m$ ← self._regex.match(string, pos)
        self._match ← $m$
        result ← $-1$
        if $m$:
            result ← $m$.end($0$)
        return result
    def search(self, string, pos ← $0$):
        $m$ ← self._regex.search(string, pos)
        self._match ← $m$
        result ← $-1$
        if $m$:
            result ← $m$.start($0$)
        return result
    def sub(self, replacement, string):
        return self._regex.sub(replacement, string)
    def split(self, string):
        $m$ ← self._regex.search(string)
        if $\neg m$: return [string, ]
        $g$ ← len($m$.groups())
        result ← self._regex.split(string)
        $i \leftarrow 1$
        if $g > 0$:
            while $i <$ len(result):
                for $j \in$ range($g$):
                    if $i <$ len(result):
                        del result[$i$]
                $i \leftarrow i + 1$
        return result
    def group(self, *indices):
        if ¬indices: indices ← ($0$, )
        result ← ()
        if len(indices) $= 1$:
                result ← self._match.group (indices[0])
        else:
                result ← [ ]
                for index ∈ indices:
                        result.append (self._match.group (index))
                result ← tuple (result)
        return result

class Regex:
        def compile (self, regex):
                return Re (re.compile (regex))

regex ← Regex ()

class Regsub:
        def gsub (self, pattern, replacement, string):
                return pattern.sub (replacement, string)
        def split (self, string, separator):
                return separator.split (string)

regsub ← Regsub ()

indent_tab ← regex.compile ('(\n|^)(␣*)\t')
indent_space ← regex.compile ('\n(␣*)')
paragraph_divider ← regex.compile ('(\n␣*)+\n')

def untabify (aString):
        ▌\ Convert indentation tabs to spaces.
        result ← ''
        rest ← aString
        while 1:
                start ← indent_tab.search (rest)
                if start ≥ 0:
                        lnl ← len (indent_tab.group (1))
                        indent ← len (indent_tab.group (2))
                        result ← result + rest[: start]
                        rest ← "\n%s%s" % ('␣' ∗ ((indent/8 + 1) ∗ 8),
                                rest[start + indent + 1 + lnl: ])
                else:
                        return result + rest

def indent_level (aString):
        \ Find the minimum indentation for a string, not counting blank lines.
        start ← 0
        text ← '\n' + aString
        indent ← l ← len (text)
        while 1:
            start ← indent_space.search (text, start)
            if start ≥ 0:
                i ← len (indent_space.group (1))
                start ← start + i + 1
                if start < l ∧ text[start] ≠ '\n':   # Skip blank lines
                    if ¬i: return (0, aString)
                    if i < indent: indent ← i
            else:
                return (indent, aString)

def paragraphs (list, start):
        l ← len (list)
        level ← list[start][0]
        i ← start + 1
        while i < l ∧ list[i][0] > level: i ← i + 1
        return i − 1 − start

def structure (list):
        if ¬list: return [ ]
        i ← 0
        l ← len (list)
        r ← [ ]
        while i < l:
            sublen ← paragraphs (list, i)
            i ← i + 1
            r.append ((list[i − 1][1], structure (list[i: i + sublen])))
            i ← i + sublen
        return r

bullet ← regex.compile ('[␣\t\n]*[o*-][␣\t\n]+([^\\0]*)')
example ← regex.compile ('[␣\t\n]examples?:[␣\t\n]*$')
dl ← regex.compile ('([^\n]+)[␣\t]+--[␣\t\n]+([^\\0]*)')
nl ← regex.compile ('\n')
ol ← regex.compile ('[␣\t]*(([0-9]+|[a-zA-Z]+)\.)+'+
        '[␣\t\n]+([^\\0]*|$)')
olp ← regex.compile ('[␣\t]*\([0-9]+\)[␣\t\n]+([^\\0]*|$)')
em ← regex.compile ("[␣\t\n]\*([^␣\t][^\n*]*[^␣\t])\*"+
        "([␣\t\n,.:;!?])")
code ← regex.compile ("[␣\t\n(]'([^␣\t']([^\n']*[^␣\t'])?)'"+
    "([)␣\t\n,.:;!?])")
strong ←
        regex.compile("[␣\t\n]\*\*([^␣\t][^\n*]*[^␣\t])\*\*([␣\t\n,.:;!?])")
extra_dl ← regex.compile("</dl>\n<dl>")
extra_ul ← regex.compile("</ul>\n<ul>")
extra_ol ← regex.compile("</ol>\n<ol>")

class StructuredText:

    \ Model text as structured collection of paragraphs.

    Structure is implied by the indentation level.

    This class is intended as a base classes that do actual text output formatting.

    def __init__(self, aStructuredString, level ← 1):
        \ Convert a string containing structured text into a structured text
        object.

        Arguments:

         aStructuredString  The string to be parsed.

         level  The level of top level headings to be created.
        self.level ← level
        paragraphs ←
                regsub.split(untabify(aStructuredString), paragraph_divider)
        paragraphs ← map(indent_level, paragraphs)

    self.structure ← structure(paragraphs)

    def __str__(self):
        return str(self.structure)


class HTML(StructuredText):

    \ An HTML structured text formatter.

    def __str__(self):
        ┃\ Return an HTML string representation of the structured text data.
        $s \leftarrow$ self._str (self.structure, self.level)
        if $s \equiv$ None: $s \leftarrow$ ''
        $s \leftarrow$ regsub.gsub (extra_dl, '\n', $s$)
        $s \leftarrow$ regsub.gsub (extra_ul, '\n', $s$)
        $s \leftarrow$ regsub.gsub (extra_ol, '\n', $s$)
        $s \leftarrow$ regsub.gsub (strong, '␣<strong>\\1</strong>\\2', $s$)
        $s \leftarrow$ regsub.gsub (code, '␣<code>\\1</code>\\3', $s$)
        $s \leftarrow$ regsub.gsub (em, '␣<em>\\1</em>\\2', $s$)
        return $s$

    def ul (self, before, $p$, after):
        if $p$: $p \leftarrow$ "<p>%s</p>" % $p$
        return ('%s<ul><li>%s\n%s\n</ul>\n'
            %(before, $p$, after))

    def ol (self, before, $p$, after):
        if $p$: $p \leftarrow$ "<p>%s</p>" % $p$
        return ('%s<ol><li>%s\n%s\n</ol>\n'
            %(before, $p$, after))

    def dl (self, before, $t$, $d$, after):
        return ('%s<dl><dt>%s<dd><p>%s</p>\n%s\n</dl>\n'
            %(before, $t$, $d$, after))

    def head (self, before, $t$, level, $d$):
        || if level <= 6: t="<h%d>%s</h%d>" % (level,t,level)
        $t \leftarrow$ "<p><strong>%s</strong><p>" % $t$
        return ('%s<dl><dt>%s\n<dd>%s\n</dl>\n'
            %(before, $t$, $d$))

    def normal (self, before, $p$, after):
        return '%s<p>%s</p>\n%s\n' % (before, $p$, after)

    def _str (self, structure, level):
        $r \leftarrow$ ''
        for $s \in$ structure:
            || print s[0],'\n', len(s[1]), '\n\n'
            if bullet.match $(s[0]) \geq 0$:
                $p \leftarrow$ bullet.group $(1)$
                $r \leftarrow$ self.ul $(r, p,$ self._str $(s[1],$ level$))$
            elif ol.match $(s[0]) \geq 0$:
                $p \leftarrow$ ol.group $(3)$
                $r \leftarrow$ self.ul $(r, p,$ self._str $(s[1],$ level$))$
            elif olp.match $(s[0]) \geq 0$:
                $p \leftarrow$ olp.group $(1)$
                $r \leftarrow$ self.ol $(r, p,$ self._str $(s[1],$ level$))$
            elif dl.match $(s[0]) \geq 0$:
                $t, d \leftarrow$ dl.group $(1, 2)$
                $r \leftarrow$ self.dl $(r, t, d,$ self._str $(s[1],$ level$))$
            elif example.search $(s[0]) \geq 0 \wedge s[1]$:
                || Introduce an example, using pre tags:
                $r \leftarrow$ self.normal $(r, s[0],$ self.pre $(s[1]))$
            elif nl.search $(s[0]) < 0 \wedge s[1]$:
                || Treat as a heading
                $t \leftarrow s[0]$
                $r \leftarrow$ self.head $(r, t,$ level, self._str $(s[1],$ level $+ 1))$
            else:
                $r \leftarrow$ self.normal $(r, s[0],$ self._str $(s[1],$ level$))$
        return $r$

    def pre (self, structure, tagged $\leftarrow 0$):
        if $\neg$structure: return ''
        if tagged:
            $r \leftarrow$ ''
        else:
            $r \leftarrow$ '<pre>\n'
        for $s \in$ structure:
            $r \leftarrow$ "%s%s\n\n%s" % $(r, s[0],$ self.pre $(s[1], 1))$
        if $\neg$tagged: $r \leftarrow r +$ '</pre>\n'
        return $r$


def main ():
    import sys

    print HTML (sys.stdin.read ())

if __name__ = "__main__": main ()

```