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 – Wed Feb 2 13:42:43 2005

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* $\leq i \leq$ `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*).

```
178 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.

```
186 class Interpret:
187     def __init__(self, name, math ← 1, interpret ← 1, docprocess ← 'none'):
188         if name = None:
189             self._name ← '-'
190         else:
191             self._name ← name
192         if self._name = '-':
193             self._name ← '(stdin)'
194             mtime ← time.asctime(time.localtime(time.time()))
196             self._file ← sys.stdin
197         else:
198             mtime ← time.asctime(time.localtime(os.stat(name)[8]))
200             self._file ← open(self._name, 'r')
201             self._name ← os.path.basename(self._name)
202         preamble ← '\\File{%s}{%s}\n\n' % (self._name, mtime)
203         if ¬math: preamble ← preamble + '\\PythonNoMath\n\n'
204         self._translation ← [preamble,]
205         self._math ← math
```

```
206         self._line_nr ← 0
207         self._line ← None
208         self._old_line ← None
209         self._eof ← 0
210         self._indent_stack ← [0]
211         self._no_break ← 0
212         self._interpret_comments ← interpret
213         self._docprocess ← docprocess
214         self._docstring ← 1
215     def math(self):
216         if ¬self._math:
217             self._translation.append('\\PythonMath\n')
218             self._math ← 1
219     def no_math(self):
220         if ¬self._math:
221             self._translation.append('\\PythonNoMath\n')
222             self._math ← 0
223     def interpret(self):
224         self._interpret_comments ← 1
225     def verbatim(self):
226         self._interpret_comments ← 0
227     def close(self):
228         self._file.close()
229         self._line_nr ← 0
230         self._line ← None
231         self._old_line ← None
232         self._indent_stack ← [0]
233         self._translation ← []
234         self._eof ← 1
235         self._no_break ← 0
236     def flush(self):
237         self._file.flush()
238     def next_line(self):
239         if self._old_line ≠ None:
240             self._line ← self._old_line
241             self._old_line ← None
242             self._line_nr ← self._line_nr + 1
243             return
244         self._line ← self._file.readline()
245         if self._line = '':
246             self._eof ← 1
247             raise EOFError
248         if self._line[−1] = '\n': self._line ← self._line[: −1]
249         self._line_nr ← self._line_nr + 1
```

```
    def undo_line(self):
        if self._line ≠ None:
            self._old_line ← self._line
            self._line ← None
            self._line_nr ← self._line_nr − 1
    def close_tex(self, tex):
        while 1:
            if tex[−2:] = '\\␣':
                tex ← tex[:−2]
            elif tex[−4:] = '\\BP␣':
                tex ← tex[:−4]
            else:
                break
        if tex ∉ ('$', '${}'):
            self._translation.append(tex + '$')
    def tr_indentation(self):
        length ← white_re.match(self._line)
        if length < 0: raise error
        indent ← 0
        for c ∈ self._line[:length]:
            indent ← indent + 1
            if c = '\t':
                indent ← indent + 8
                indent ← indent & ~ 0x7
        self._line ← 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 ← len(self._indent_stack) − 1
    def tr_comment_line(self):
        if self._interpret_comments:
            length ← verbatim_re.search(self._line)
            if length ≥ 0:
                self.verbatim()
                self._line ← self._line[:length]
            while 1:
                hash ← string.find(self._line, '#')
                if hash ≥ 0:
                    if len(self._line) > hash + 1 ∧ self._line[hash + 1] = '#':
                        self._translation.append(self._line[:hash] + '#')
                        self._line ← self._line[hash + 2:]
                        continue
                    self._translation.append(self._line[:hash])
```

```
296                         self._line ← self._line[hash + 1:]
297                         self.tr_code(0)   # No continued lines in comments.
298                         if len(self._line) ≤ 0: break
299                         if self._line[0] ≠ '#': raise error
300                         self._line ← self._line[1:]
301                     else:
302                         break
303                 self._translation.append(self._line + '\n')
304             else:
305                 length ← interpret_re.search(self._line)
306                 if length ≥ 0:
307                     self.interpret()
308                     self._line ← self._line[: length]
309                 while len(self._line) > 0:
310                     length ← ordinary_re.match(self._line)
311                     if length > 0:
312                         self._translation.append(self._line[: length])
313                     if len(self._line) > length:
314                         char ← self._line[length]
315                         if char ∈ '<>\\{|}~':
316                             self._translation.append(
                                    '{\\tt\\char`\\%s}' % char)
318                         else:
319                             self._translation.append('\\' + char)
320                     self._line ← self._line[length + 1:]
321                 self._translation.append('\n')
322     def tr_block_comment(self):
323         if self._line[0] ≠ '#': raise error
324         outer ← self._indentation = 0
325         if outer:
326             if self._line_nr > 1:

328                 self._translation.append('\\PythonOuterBlock\n')
330             else:
331                 self._translation.append('\\PythonOuterBlock*\n')
333         else:
334             self._translation.append('\\B{%d}{%d}{%%\n' %
                    (self._line_nr, self._indentation))
336         try:
337             white ← white_re.match(self._line, 1)
338             if white < 0: raise error
339             self._line ← self._line[white:]
340             while 1:
341                 self.tr_comment_line()
```

```
342                     self.next_line()
343                     white ← white_re.match(self._line)
344                     if white < 0: raise error
345                     if len(self._line) > white ∧ self._line[white] = '#' ∧
                            self._line[white: white + 2] ≠ '##':
347                         self._line ← self._line[white + 1:]
348                         white ← white_re.match(self._line)
349                         if white > 0: self._line ← self._line[white:]
350                         continue
351                     self.undo_line()
352                     return
353             finally:
354                 if outer:
355                     self._translation.append('\\PythonOuterBlockEnd\n')
357                 else:
358                     self._translation.append('}\n')
359     def tr_comment(self):
360         self._translation.append('\\#\\␣')
361         while self._line[:2] = '##':
362             self._line ← self._line[2:]
363             self.tr_code(0)   # No continued lines in comments.
364             if self._line[:1] = '#':
365                 self._translation.append('\\quad\\#\\␣')
366         if len(self._line) < 1:
367             self._translation.append('\n')
368             return
369         if self._line[0] ≠ '#': raise error
370         white ← white_re.match(self._line, 1)
371         if white < 0: raise error
372         self._line ← self._line[white:]
373         self.tr_comment_line()
374         return
375     def tr_string(self, token):
376         quote ← token[0]
377         tl ← len(token)
378         self._translation.append('\S{' + token)
379         while 1:
380             pos ← string.find(self._line, quote)
381             if pos > 0:
382                 self._translation.append('\\verb*%s%s' %
                        (quote, ctrl_protect(self._line[:pos + 1])))
384                 if escape_re.match(self._line[:pos]) = pos:
385                     self._translation.append(quote)
386                     self._line ← self._line[pos + 1:]
```

```
387                         continue
388                     self._line ← self._line[pos:]
389                     pos ← 0
390                 if pos ≥ 0:
391                     if self._line[:tl] = token:
392                         self._translation.append(token + '}')
393                         self._line ← self._line[tl:]
394                         return
395                     self._translation.append(quote)
396                     self._line ← self._line[1:]
397                 else:
398                     self._translation.append('\\verb*%s%s%s' %
                            (quote, ctrl_protect(self._line), quote))
400                     self._line ← ''
401                     if tl = 1:
402                         self._translation.append('}')
403                         return
404                     self.next_line()
405                     self._translation.append('\n\\I{%d}{0}' % self._line_nr)
407         return

409     def tr_docstring_plain(self):
410         length ← quote_re.match(self._line)
411         token ← self._line[:length]
412         self._line ← self._line[length:]
413         quote ← token[0]
414         tl ← len(token)

416         if self._indentation = 0:
417             self._translation.append('\\PythonDocBlock\n')
418         else:
419             self._translation.append('\DS{%s}{%s}{%%\n' %
420                 (self._line_nr, self._indentation))
421         while 1:
422             pos ← string.find(self._line, quote)
423             if pos > 0:
424                 self._translation.append('\\verb%s%s' %
                        (quote, ctrl_protect(self._line[:pos + 1])))
426                 if escape_re.match(self._line[:pos]) = pos:
427                     self._translation.append(quote)
428                     self._line ← self._line[pos + 1:]
429                     continue
430                 self._line ← self._line[pos:]
431                 pos ← 0
```

```
            if pos ≥ 0:
                if self._line[: tl] = token:
                    self._line ← self._line[tl: ]
                    break
                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:
                    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 ← numeral_re.match(self._line)
                if length < 1:
                    tex ← tex + self._line[0]
                    self._line ← self._line[1:]
                    careful ← 0
                else:
                    token ← self._line[:length]
                    self._line ← self._line[length:]
                    if careful: tex ← tex + '\\␣'
                    tex ← tex + tr_numeral(token)
                    careful ← 1
                continue
            token ← self._line[:length]
            self._line ← self._line[length:]
            token ← self.double(token)
            if token = ':':
                self._docstring ← 1
            else:
                self._docstring ← 0
            if token ∈ ('{', '}'):
                tex ← tex + '\\' + token
                careful ← 0
                continue
            if token ∈ reserved_operators:
                tex ← tex + '\\O{%s}' % token
                careful ← 0
                continue
            if token[0] ∈ string.letters + '_':
                if careful: tex ← tex + '\\␣'
                new_careful ← 1
                if token ∈ reserved:
                    if tex[−2:] ∉ ('$', '\\␣') ∧ ¬careful:
                        tex ← tex + '\\␣'
                    tex ← tex + '\\K{%s}' % token
                    if token = 'if': tex ← tex + '\\,'
                    if token ∉ single: tex ← tex + '\\␣'
                    new_careful ← 0
                else:
                    token ← usc_protect(token)
                    length ← function_re.match(self._line)
                    if length > 0:
                        self._line ← self._line[length:]
                        tex ← tex + '\\F{%s}\\,(' % token
```

```
                                    new_careful ← 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
```

```
596     ‖ Method translate() is the interface to the Interpret class. It calls the
        ‖ tr_xxx() methods to process indentation, code, comments and strings.
599   def translate(self):
600       self._translation ← []
601       if self._eof: return None
602       try:
603           empty ← 0
604           self.next_line()
605           while white_re.match(self._line) = len(self._line):
606               empty ← empty + 1
607               self.next_line()
608           if empty > 0:
609               self._translation.append('\\E{%d}' % empty)
610           self.tr_indentation()
611           if len(self._line) > 0 ∧ self._line[0] = '#' ∧ self._line[:2] ≠ '##':
613               self.tr_block_comment()
614               self._no_break ← 1
615           elif self._docprocess ≠ 'none' ∧
                    self._docstring ∧ self._line[:3] ∈ ('"""', "'''"):
617               if self._docprocess = 'plain':
618                   self.tr_docstring_plain()
619               elif self._docprocess = 'struct':
620                   self.tr_docstring_struct()
621               else:
622                   raise ValueError, 'Illegal␣value␣for␣doc␣process.'
623           else:
624               self._translation.append('\\I{%d}{%d}' %
                        (self._line_nr, self._indentation))
626               self.tr_code()
627               if ¬self._no_break ∧ self._translation[−1][−8:] = '\\colon␣$' ∧
                        self._translation[0][:3] ≠ '\\E{':
630                   self._translation.insert(0, '\\PB')
631                   self._no_break ← 1
632               else:
633                   self._no_break ← empty ≠ 0
634               if len(self._line) > 0:
635                   if self._line[:1] ≠ '#': raise error
636                   if self._translation[−1][:1] = '$':
637                       self._translation.append('\\quad␣')
638                   self.tr_comment()
639               else:
640                   self._translation.append('\n')
641       except EOFError: pass
642       return self._translation
```

```
643     def translation(self):
644         return self._translation

646 error ← 'py2tex␣error'

648 class Re:
649     def __init__(self, regex):
650         self._regex ← regex
651     def match(self, string, pos ← 0):
652         m ← self._regex.match(string, pos)
653         result ← −1
654         if m:
655             result ← m.end(0)
656         return result
657     def search(self, string, pos ← 0):
658         m ← self._regex.search(string, pos)
659         result ← −1
660         if m:
661             result ← m.start(0)
662         return result

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

668 regex ← Regex()

670 interpret_re ← regex.compile('%[␣\t]*(La)?TeX[␣\t]*$')
671 verbatim_re ← regex.compile('%[␣\t]*ASCII[␣\t]*$')
672 ordinary_re ← regex.compile('[^#$%&<>\\^_{|}~]*')
673 white_re ← regex.compile('[␣\t]*')
674 function_re ← regex.compile('[␣\t]*\\(')
675 comment_re ← regex.compile('(##|[^#])*')
676 escape_re ← regex.compile('([^\\\\]|\\\\.)*\\\\')
677 numeral_re ← regex.compile(string.joinfields((
678     '0[xX][0-9A-Fa-f]+',
679     '[0-9]+\\.?[eE][+-]?[0-9]+[jJLl]?',
680     '[0-9]*\\.[0-9]+[eE][+-]?[0-9]+[jJLl]?',
681     '[1-9][0-9]*[jJLl]?',
682     '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 ∈ str:
        o ← ord (c)
        if o < 32 ∨ o ≥ 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:
        || (12.4·10^−78, .3333·10^+0, .1·10^6, 2.·10^1, 0.·10^1, 1·10^4)
        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 – Sat Nov 22 01:06:17 2003

!/usr/local/bin/python

Py2tex, script to translate Python source to LaTeX code.

```
import getopt, sys
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'
```

```
35      || Parse options.
36 optlist, args = getopt.getopt (sys.argv[1:], 'imno:vd:')
37 for pair in optlist:
38      key = pair[0]
39      if pair[0] == '-m':
40          math = 1
41      if pair[0] == '-n':
42          math = 0
43      if pair[0] == '-o':
44          output = pair[1]
45      if pair[0] == '-d':
46          docprocess = pair[1]
47      if pair[0] == '-i':
48          interpret = 1
49      if pair[0] == '-v':
50          interpret = 0

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

55      || Open output file.
56 if output == None:
57      outfile = sys.stdout
58 else:
59      outfile = open (output, 'w')

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

69      || Close output file.
70 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 `\OuterMarker`s. 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}
\DeclareTextSymbolDefault{\textquotesingle}{TS1}
{\catcode`\'=\active
  \gdef\PythonString#{\bgroup\tt
    \catcode`\'=\active\def'{\textquotesingle}%
    \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.}}
```