

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

May 10, 2026

Abstract

This document is the documented code of the LuaLaTeX package `piton`. It is *not* its user's guide. The guide of utilisation is the document `piton.pdf` (with a French translation: `piton-french.pdf`).

The development of the extension `piton` is done on the following GitHub depot:
<https://github.com/fpantigny/piton>

1 Introduction

The main job of the package `piton` is to take in as input a computer listing and to send back to LaTeX as output that code *with interlaced LaTeX instructions of formatting*.

In fact, all that job is done by a LPEG called `LPEG1[<language>]` where `<language>` is a Lua string which is the name of the computer language. That LPEG, when matched against the string of a computer listing, returns as capture a Lua table containing data to send to LaTeX. The only thing to do after will be to apply `tex.tprint` to each element of that table.¹

In fact, there is a variant of the LPEG `LPEG1[<language>]`, called `LPEG2[<language>]`. The latter uses the first one and will be used to format the whole content of an environment `{Piton}` (with, in particular, small tuning for the beginning and the end).

Consider, for example, the following Python code:

```
def parity(x):  
    return x%2
```

The capture returned by the LPEG `LPEG1['python']` (in Lua, this may also be written `LPEG1.python`) against that code is the Lua table containing the following elements :

*This document corresponds to the version 4.12a of `piton`, at the date of 2026/05/10.

¹Recall that `tex.tprint` takes in as argument a Lua table whose first component is a “catcode table” and the second element a string. The string will be sent to LaTeX with the regime of catcodes specified by the catcode table. If no catcode table is provided, the standard catcodes of LaTeX will be used.

```

{ "\\_piton_begin_line:" }a
{ "{\\PitonStyle{Keyword}{ " }b
{ luatexbase.catcodetables.otherc, "def" }
{ "}} " }
{ luatexbase.catcodetables.other, " " }
{ "{\\PitonStyle{Name.Function}{ " }
{ luatexbase.catcodetables.other, "parity" }
{ "}} " }
{ luatexbase.catcodetables.other, "(" }
{ luatexbase.catcodetables.other, "x" }
{ luatexbase.catcodetables.other, ")" }
{ luatexbase.catcodetables.other, ":" }
{ "\\_piton_end_line: \\_piton_par: \\_piton_begin_line:" }
{ luatexbase.catcodetables.other, " " }
{ "{\\PitonStyle{Keyword}{ " }
{ luatexbase.catcodetables.other, "return" }
{ "}} " }
{ luatexbase.catcodetables.other, " " }
{ luatexbase.catcodetables.other, "x" }
{ "{\\PitonStyle{Operator}{ " }
{ luatexbase.catcodetables.other, "%" }
{ "}} " }
{ "{\\PitonStyle{Number}{ " }
{ luatexbase.catcodetables.other, "2" }
{ "}} " }
{ "\\_piton_end_line:" }

```

^aEach line of the computer listings will be encapsulated in a pair: `_@@_begin_line: – _@@_end_line:`. The token `_@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `_@@_begin_line:`. Both tokens `_@@_begin_line:` and `_@@_end_line:` will be nullified in the command `\piton` (since there can't be lines breaks in the argument of a command `\piton`).

^bThe lexical elements for which we have a piton style will be formatted via the use of the command `\PitonStyle`. Such an element is typeset in LaTeX via the syntax `{\\PitonStyle{style}{...}}` because the instructions inside an `\PitonStyle` may be both semi-global declarations like `\bfseries` and commands with one argument like `\fbox`.

^c`luatexbase.catcodetables.other` is a mere number which corresponds to the “catcode table” whose all characters have the catcode “other” (which means that they will be typeset by LaTeX verbatim).

We give now the LaTeX code which is sent back by Lua to TeX (we have written on several lines for legibility but no character `\r` will be sent to LaTeX). The characters which are greyed-out are sent to LaTeX with the catcode “other” (=12). All the others characters are sent with the regime of catcodes of L3 (as set by `\ExplSyntaxOn`).

```

\_piton_begin_line:{\\PitonStyle{Keyword}{def}}
{\\PitonStyle{Name.Function}{parity}}(x):\_piton_end_line:\_piton_par:
\_piton_begin_line:~~~~{\\PitonStyle{Keyword}{return}}
{x\\PitonStyle{Operator}{%}}{\\PitonStyle{Number}{2}}\_piton_end_line:

```

2 The L3 part of the implementation

2.1 Declaration of the package

```

1 < *STY >
2 \NeedsTeXFormat{LaTeX2e}
3 \ProvidesExplPackage
4   {piton}
5   {\PitonFileDate}
6   {\PitonFileVersion}
7   {Highlight computer listings with LPEG on LuaLaTeX}
8 \msg_new:nnn { piton } { latex-too-old }
9   {
10     Your~LaTeX~release-is-too-old. \\

```

```

11   You~need~at~least~the~version~of~2025-06-01.  \\\
12   If~you~use~Overleaf,~you~need~at~least~"TeX-Live-2025".\\
13   The~package~'piton'~won't~be~loaded.
14 }

15 \providecommand { \IfFormatAtLeastTF } { \@ifl@t@r \fmtversion }
16 \IfFormatAtLeastTF
17 { 2025-06-01 }
18 { }
19 { \msg_critical:nn { piton } { latex-too-old } }

```

The command `\text` provided by the package `amstext` will be used to allow the use of the command `\piton{...}` (with the standard syntax) in mathematical mode.

```

20 \RequirePackage { amstext }

```

The command `\marginalia` of the package `marginalia` will be used for the margin notes created by the keys `paperclip` and `annotation`.

```

21 \RequirePackage { marginalia }

```

The package `transparent` is compatible with `pdfmanagement` (which is not loaded by `piton` but which is used for the key `join` when it is loaded).

```

22 \RequirePackage { transparent }

```

```

23 \cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
24 \cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
25 \cs_new_protected:Npn \@@_warning:nn { \msg_warning:nnn { piton } }
26 \cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
27 \cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
28 \cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
29 \cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
30 \cs_new_protected:Npn \@@_msg_new:nn { \msg_new:nnn { piton } }

```

With Overleaf (and also TeXPage), by default, a document is compiled in non-stop mode. When there is an error, there is no way to the user to use the key `H` in order to have more information. That's why we decide to put that piece of information (for the messages with such information) in the main part of the message when the key `messages-for-Overleaf` is used (at load-time).

```

31 \cs_new_protected:Npn \@@_msg_new:nnn #1 #2 #3
32 {
33   \bool_if:NTF \g_@@_messages_for_Overleaf_bool
34     { \msg_new:nnn { piton } { #1 } { #2 } { #3 } }
35     { \msg_new:nnnn { piton } { #1 } { #2 } { #3 } }
36 }

```

We also create commands which will generate usually an error but only a warning on Overleaf. The argument is given by curryfication.

```

37 \cs_new_protected:Npn \@@_error_or_warning:n
38 { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:n \@@_error:n }
39 \cs_new_protected:Npn \@@_error_or_warning:nn
40 { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:nn \@@_error:nn }

```

We try to detect whether the compilation is done on Overleaf. We use `\c_sys_jobname_str` because, with Overleaf, the value of `\c_sys_jobname_str` is always “output”.

```

41 \bool_new:N \g_@@_messages_for_Overleaf_bool
42 \bool_gset:Nn \g_@@_messages_for_Overleaf_bool
43 {
44   \str_if_eq_p:on \c_sys_jobname_str { _region_ } % for Emacs
45   || \str_if_eq_p:on \c_sys_jobname_str { output } % for Overleaf
46 }

```

```

47 \@@_msg_new:nn { LuaLaTeX-mandatory }
48 {

```

```

49  LuaLaTeX-is-mandatory.\
50  The-package~'piton'~requires~the~engine~LuaLaTeX.\
51  \str_if_eq:onT \c_sys_jobname_str { output }
52  { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~
53    "Settings~>~Compiler"~and~if~you~use~TeXPage,
54    ~you~should~go~in~"Settings". \
55  \IfClassLoadedT { beamer }
56  {
57    Since~you~use~Beamer,~don't~forget~to~use~piton~in~frames~with~
58    the~key~'fragile'.\
59  }
60  \IfClassLoadedT { ltx-talk }
61  {
62    Since~you~use~'ltx-talk',~don't~forget~to~use~piton~in~
63    environments~'frame*'.\
64  }
65  }
66  \sys_if_engine luatex:F { \@@_fatal:n { LuaLaTeX-mandatory } }

67  \RequirePackage { luacode }

68  \@@_msg_new:nnn { piton.lua-not-found }
69  {
70    The~file~'piton.lua'~can't~be~found.~
71    If~you~want~to~know~how~to~retrieve~the~file~'piton.lua',~type~H~<return>.
72  }
73  {
74    On~the~site~CTAN,~go~to~the~page~of~'piton':~https://ctan.org/pkg/piton.~
75    The~file~'README.md'~explains~how~to~retrieve~the~files~'piton.sty'~and~
76    'piton.lua'.
77  }

78  \file_if_exist:nF { piton.lua } { \@@_fatal:n { piton.lua-not-found } }

```

We define a set of keys for the options at load-time.

```

79  \keys_define:nn { piton }
80  {
81    footnote .bool_gset:N = \g_@@_footnote_bool ,
82    footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
83    footnote .usage:n = load ,
84    footnotehyper .usage:n = load ,
85
86    beamer .bool_gset:N = \g_@@_beamer_bool ,
87    beamer .usage:n = load ,
88
89    unknown .code:n = \@@_error:n { Unknown-key-for-package }
90  }
91  \@@_msg_new:nn { Unknown-key-for-package }
92  {
93    Unknown-key.\
94    You~have~used~the~key~'\l_keys_key_str'~when~loading~piton~
95    but~the~only~keys~available~here~are~'beamer',~'footnote'~
96    and~'footnotehyper'.~Other~keys~are~available~in~
97    \token_to_str:N \PitonOptions.\
98    That~key~will~be~ignored.
99  }

```

We process the options provided by the user at load-time.

```

100  \ProcessKeyOptions

```

```

101 \IfClassLoadedT { beamer } { \bool_gset_true:N \g_@@_beamer_bool }
102 \IfClassLoadedT { ltx-talk } { \bool_gset_true:N \g_@@_beamer_bool }
103 \IfPackageLoadedT { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool }
104 \lua_now:e
105 {
106     piton = piton-or-{ }
107     piton.last_code = ''
108     piton.last_language = ''
109     piton.join = ''
110     piton.write = ''
111     piton.path_write = ''
112     \bool_if:NT \g_@@_beamer_bool { piton.beamer = true }
113 }

114 \RequirePackage { xcolor }

115 \@@_msg_new:nn { footnote-with-footnotehyper-package }
116 {
117     Footnote-forbidden.\
118     You-can't-use-the-option-'footnote'~because-the-package~
119     footnotehyper-has-already-been-loaded.~
120     If-you-want,~you-can-use-the-option-'footnotehyper'~and-the-footnotes~
121     within-the-environments-of-piton-will-be-extracted-with-the-tools~
122     of-the-package-footnotehyper.\
123     If-you-go-on,~the-package-footnote-won't-be-loaded.
124 }

125 \@@_msg_new:nn { footnotehyper-with-footnote-package }
126 {
127     You-can't-use-the-option-'footnotehyper'~because-the-package~
128     footnote-has-already-been-loaded.~
129     If-you-want,~you-can-use-the-option-'footnote'~and-the-footnotes~
130     within-the-environments-of-piton-will-be-extracted-with-the-tools~
131     of-the-package-footnote.\
132     If-you-go-on,~the-package-footnotehyper-won't-be-loaded.
133 }

134 \bool_if:NT \g_@@_footnote_bool
135 {

```

The class beamer has its own system to extract footnotes and that's why we have nothing to do if beamer is used.

```

136 \IfClassLoadedTF { beamer }
137 { \bool_gset_false:N \g_@@_footnote_bool }
138 {
139     \IfPackageLoadedTF { footnotehyper }
140     { \@@_error:n { footnote-with-footnotehyper-package } }
141     { \usepackage { footnote } }
142 }
143 }

144 \bool_if:NT \g_@@_footnotehyper_bool
145 {

```

The class beamer has its own system to extract footnotes and that's why we have nothing to do if beamer is used.

```

146 \IfClassLoadedTF { beamer }
147 { \bool_gset_false:N \g_@@_footnote_bool }
148 {
149     \IfPackageLoadedTF { footnote }
150     { \@@_error:n { footnotehyper-with-footnote-package } }
151     { \usepackage { footnotehyper } }
152     \bool_gset_true:N \g_@@_footnote_bool
153 }
154 }

```

The flag `\g_@@_footnote_bool` is raised and so, we will only have to test `\g_@@_footnote_bool` in order to know if we have to insert an environment `{savenotes}`.

2.2 Parameters and technical definitions

```
155 \dim_new:N \l_@@_rounded_corners_dim
156 \bool_new:N \l_@@_in_label_bool
157 \dim_new:N \l_@@_tmpc_dim
```

The listing that we have to format will be stored in `\l_@@_listing_tl`. That applies both for the command `\PitonInputFile` and the environment `{Piton}` (or another environment defined by `\NewPitonEnvironment`).

```
158 \tl_new:N \l_@@_listing_tl
```

The content of an environment such as `{Piton}` will be composed first in the following box, but that box will (sometimes) be *unboxed* at the end.

We need a global variable (see `\@@_add_bg_and_right_nb_to_output_box:`).

```
159 \box_new:N \g_@@_output_box
```

The following string will contain the name of the computer language considered (the initial value is `python`).

```
160 \str_new:N \l_piton_language_str
161 \str_set:Nn \l_piton_language_str { python }
```

Each time an environment of `piton` is used, the computer listing in the body of that environment will be stored in the following global string.

```
162 \tl_new:N \g_piton_last_code_tl
```

The following parameter corresponds to the key `path` (which is the path used to include files by `\PitonInputFile`). Each component of that sequence will be a string (type `str`).

```
163 \seq_new:N \l_@@_path_seq
```

The names of all the join files will be stored in the following sequence:

```
164 \seq_new:N \g_@@_join_seq
165 \str_new:N \l_@@_join_str
```

When the key `tcolorbox` is used, you will have to take into account the width of the graphical elements added by `tcolorbox` on both sides of the listing. We will put that quantity in the following variable.

```
166 \dim_new:N \l_@@_tcb_margins_dim
```

The following parameter corresponds to the key `box`.

```
167 \str_new:N \l_@@_box_str
```

In order to have a better control over the keys.

```
168 \bool_new:N \l_@@_in_PitonOptions_bool
169 \bool_new:N \l_@@_in_PitonInputFile_bool
```

We will compute (with Lua) the numbers of lines of the listings (or *chunks* of listings when `split-on-empty-lines` is in force) and store it in the following counter.

```
170 \int_new:N \g_@@_nb_lines_int
```

The same for the number of non-empty lines of the listings.

```
171 \int_new:N \l_@@_nb_non_empty_lines_int
```

The following counter will be used to count the lines during the composition. It will take into account all the lines, empty or not empty. It won't be used to print the numbers of the lines but will be used to allow or disallow line breaks (when `splittable` is in force) and for the color of the background (when `background-color` is used with a *list* of colors or when `\rowcolor` is used).

```
172 \int_new:N \g_@@_line_int
```

The following string corresponds to the key `background-color` of `\PitonOptions`.

```
173 \clist_new:N \l_@@_bg_color_clist
```

We will also keep in memory the length of the previous `clist` (for efficiency).

```
174 \int_new:N \l_@@_bg_colors_int

175 \tl_new:N \l_@@_space_in_string_tl
```

The following parameters correspond to the keys `begin-range` and `end-range` of the command `\PitonInputFile`.

```
176 \str_new:N \l_@@_begin_range_str
177 \str_new:N \l_@@_end_range_str
```

The following boolean corresponds to the key `math-comments` (available only in the preamble of the LaTeX document).

```
178 \bool_new:N \g_@@_math_comments_bool
```

The argument of `\PitonInputFile`.

```
179 \str_new:N \l_@@_file_name_str
```

The following line can't be deleted.

```
180 \bool_new:N \l_@@_tcolorbox_bool
```

The following boolean corresponds to the keys `paperclip` and `annotation`.

```
181 \bool_new:N \l_@@_paperclip_bool
182 \str_new:N \l_@@_paperclip_str
```

The listings embedded in the PDF by the key `paperclip` will be numbered by the following counter.

```
183 \int_new:N \g_@@_paperclip_int
```

The following boolean corresponds to the key `show-spaces`.

```
184 \bool_new:N \l_@@_show_spaces_bool
```

The following boolean corresponds to the key `break-lines-in-piton`.

```
185 \bool_new:N \l_@@_break_lines_in_piton_bool
```

The following flag will be raised when the key `max-width` is used (and when `width` is used with the key `min`, which is equivalent to `max-width=\linewidth`). Note also that the key `box` sets `width=min` (except if `min` is used with a numerical value).

```
186 \bool_new:N \l_@@_minimize_width_bool
```

The following dimension corresponds to the key `width`. It's meant to be the whole width of the environment (for instance, the width of the box of `tcolorbox` when the key `tcolorbox` is used). The initial value is 0 pt which means that the end user has not used the key. In that case, it will be set equal to the current value of `\linewidth` in `\@@_pre_composition:`.

However if `max-width` is used (or `width=min` which is equivalent to `max-width=\linewidth`), the actual width of the final environment in the PDF may (potentially) be smaller.

```
187 \dim_new:N \l_@@_width_dim
```

`\l_@@_listing_width_dim` will be the width of the listing taking into account the lines of code (of course) but also:

- `\l_@@_left_margin_dim` (for the numbers of lines);
- a small margin when `background-color` is in force²).

```
188 \dim_new:N \l_@@_listing_width_dim
```

However, if `max-width` is used (or `width=min` which is equivalent to `max-width=\linewidth`), that length will be computed once again in `\@@_create_output_box:`

`\l_@@_code_width_dim` will be the length of the lines of code, without the potential margins (for the backgrounds and for `length-margin` for the number of lines).

It will be computed in `\@@_compute_code_width:`.

```
189 \dim_new:N \l_@@_code_width_dim
```

²Remark that the mere use of `\rowcolor` does not add those small margins.

```
190 \box_new:N \l_@@_line_box
```

The following dimension corresponds to the keys `left-margin` and `right-margin`.

```
191 \dim_new:N \l_@@_left_margin_dim
192 \dim_new:N \l_@@_right_margin_dim
```

The following boolean will be set when the key `left-margin=auto` is used.

```
193 \bool_new:N \l_@@_left_margin_auto_bool
194 \bool_new:N \l_@@_right_margin_auto_bool
```

When the key `line-numbers/position` is set to `right`, we will have to keep in memory the numbers of the lines in the following sequence.

```
195 \seq_new:N \g_@@_visual_line_numbers_seq
```

Be careful. The following sequence `\g_@@_languages_seq` is not the list of the languages supported by `piton`. It's the list of the languages for which at least a user function has been defined. We need that sequence only for the command `\PitonClearUserFunctions` when it is used without its optional argument: it must clear the whole list of languages for which at least a user function has been defined.

```
196 \seq_new:N \g_@@_languages_seq
197 \cs_new_protected:Npn \@@_tab:
198 {
199   \bool_if:NTF \l_@@_show_spaces_bool
200   {
201     \hbox_set:Nn \l_tmpa_box
202     { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
203     \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
204     \(\mathcolor { gray }
205       { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \)
206   }
207   { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
208   \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
209 }
```

The following token list will be used only for the spaces in the strings.

```
210 \tl_set_eq:NN \l_@@_space_in_string_tl \nobreakspace
```

When the key `break-lines-in-piton` is set, that parameter will be replaced by `\space` (in `\piton` with the standard syntax) and when the key `show-spaces-in-strings` is set, it will be replaced by `□` (U+2423).

At each line, the following counter will count the spaces at the beginning.

```
211 \int_new:N \g_@@_indentation_int
```

In the environment `{Piton}`, the command `\label` will be linked to the following command.

```
212 \cs_new_protected:Npn \@@_label:n #1
213 {
214   \bool_if:NTF \l_@@_line_numbers_bool
215   {
216     \@bsphack
217     \protected@write \@auxout { }
218     {
219       \string \newlabel { #1 }
220       {
221         { \int_use:N \g_@@_visual_line_int }
222         { \thepage }
223         { }
224         { line.#1 }
225         { }
226       }
227     }
228     \@esphack
229     \IfPackageLoadedT { hyperref }
```



```

230         { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
231     }
232     { \@@_error:n { label-with-lines-numbers } }
233 }

```

The same goes for the command `\zlabel` if the `zref` package is loaded. Note that `\label` will also be linked to `\@@_zlabel:n` if the key `label-as-zlabel` is set to `true`.

```

234 \cs_new_protected:Npn \@@_zlabel:n #1
235 {
236     \bool_if:NTF \l_@@_line_numbers_bool
237     {
238         \@bsphack
239         \protected@write \@auxout { }
240         {
241             \string \zref@newlabel { #1 }
242             {
243                 \string \default { \int_use:N \g_@@_visual_line_int }
244                 \string \page { \thepage }
245                 \string \zc@type { line }
246                 \string \anchor { line.#1 }
247             }
248         }
249         \@esphack
250         \IfPackageLoadedT { hyperref }
251         { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
252     }
253     { \@@_error:n { label-with-lines-numbers } }
254 }

```

In the environments `{Piton}` the command `\rowcolor` will be linked to the following one.

```

255 \NewDocumentCommand { \@@_rowcolor:n } { o m }
256 {
257     \tl_gset:ce
258     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 }_ t1 }
259     { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
260     \bool_gset_true:N \g_@@_rowcolor_inside_bool
261 }

```

In the command `piton` (in fact in `\@@_piton_standard` and `\@@_piton_verbatim`, the command `\rowcolor` will be linked to the following one (in order to nullify its effect).

```

262 \NewDocumentCommand { \@@_noop_rowcolor } { o m } { }

```

The following commands correspond to the keys `marker/beginning` and `marker/end`. The values of that keys are functions that will be applied to the “*range*” specified by the end user in an individual `\PitonInputFile`. They will construct the markers used to find textually in the external file loaded by `piton` the part which must be included (and formatted).

These macros must *not* be protected.

```

263 \cs_new:Npn \@@_marker_beginning:n #1 { }
264 \cs_new:Npn \@@_marker_end:n #1 { }

```

The following token list will be evaluated at the end of `\@@_begin_line:...` `\@@_end_line:` and cleared at the end. It will be used by LPEG acting between the lines of the Python code in order to add instructions to be executed in vertical mode between the lines.

```

265 \tl_new:N \g_@@_after_line_tl

```

The spaces at the end of a line of code are deleted by `piton`. However, it’s not actually true: they are replace by `\@@_trailing_space:`.

```

266 \cs_new_protected:Npn \@@_trailing_space: { }

```

When we have to rescan some pieces of code, we will use `\@@_piton:n` and that command `\@@_piton:n` will set `\@@_trailing_space:` equal to `\space`.

```

267 \bool_new:N \g_@@_color_is_none_bool
268 \bool_new:N \g_@@_next_color_is_none_bool

269 \bool_new:N \g_@@_rowcolor_inside_bool

```

2.3 Detected commands

There are four keys for “detected commands and environments”: `detected-commands`, `raw-detected-commands`, `beamer-commands` and `beamer-environments`.

In fact, there is also `vertical-detected-commands` but has a special treatment.

For each of those keys, we keep a clist of the names of such detected commands and environments. For the commands, the corresponding clist will contain the name of the commands *wihtout* the backlash.

```

270 \clist_new:N \l_@@_detected_commands_clist
271 \clist_new:N \l_@@_raw_detected_commands_clist
272 \clist_new:N \l_@@_beamer_commands_clist
273 \clist_set:Nn \l_@@_beamer_commands_clist
274   { uncover , only , visible , invisible , alert , action }
275 \clist_new:N \l_@@_beamer_environments_clist
276 \clist_set:Nn \l_@@_beamer_environments_clist
277   { uncoverenv , onlyenv , visibleenv , invisibleenv , alertenv , actionenv }

```

Remark that, since we have used clists, these clists, as token lists are “purified”: there is no empty component and for each component, there is no space on both sides.

Of course, the value of those clists may be modified during the preamble of the document by using the corresponding key (`detected-commands`, etc.).

However, after the `\begin{document}`, it’s no longer possible to modify those clists because their contents will be used in the construction of the main LPEG for each computer language.

However, in a `\AtBeginDocument`, we will convert those clists into “toks registers” of TeX.

```

278 \hook_gput_code:nnn { begindocument } { . }
279   {
280     \newtoks \PitonDetectedCommands
281     \newtoks \PitonRawDetectedCommands
282     \newtoks \PitonBeamerCommands
283     \newtoks \PitonBeamerEnvironments

```

L3 does *not* support those “toks registers” but it’s still possible to affect to the “toks registers” the content of the clists with a L3-like syntax.

```

284 \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
285 \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
286 \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
287 \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
288 }

```

Then at the beginning of the document, when we will load the Lua file `piton.lua`, we will read those “toks registers” within Lua (with `tex.toks`) and convert them into Lua tables (and, then, use those tables to construct LPEG).

When the key `vertical-detected-commands` is used, we will have to redefine the corresponding commands in `\@@_pre_composition:`.

The instructions for these redefinitions will be put in the following token list.

```

289 \tl_new:N \g_@@_def_vertical_commands_tl

```

```

290 \cs_new_protected:Npn \@@_vertical_commands:n #1
291 {
292   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 }
293   \clist_map_inline:nn { #1 }
294   {
295     \cs_set_eq:cc { @@ _ old _ ##1 : } { ##1 }
296     \cs_new_protected:cn { @@ _ new _ ##1 : n }
297     {
298       \bool_if:nTF
299       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
300       {
301         \tl_gput_right:Nn \g_@@_after_line_tl
302         { \use:c { @@ _ old _ ##1 : } { #####1 } }
303       }
304       {
305         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int _ tl }
306         { \tl_gput_right:cn }
307         { \tl_gset:cn }
308         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 } _ tl }
309         { \use:c { @@ _ old _ ##1 : } { #####1 } }
310       }
311     }
312     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
313     { \cs_set_eq:cc { ##1 } { @@ _ new _ ##1 : n } }
314   }
315 }

```

2.4 Treatment of a line of code

```

316 \cs_new_protected:Npn \@@_replace_spaces:n #1
317 {
318   \tl_set:Nn \l_tmpa_tl { #1 }
319   \bool_if:NTF \l_@@_show_spaces_bool
320   {
321     \tl_set:Nn \l_@@_space_in_string_tl { } % U+2423
322     \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { } % U+2423
323   }
324   {

```

If the key `break-lines-in-Piton` is in force, we replace all the characters U+0020 (that is to say the spaces) by `\@@_breakable_space:`. Remark that, except the spaces inserted in the LaTeX comments (and maybe in the math comments), all these spaces are of catcode “other” (=12) and are unbreakable.

```

325     \bool_if:NT \l_@@_break_lines_in_Piton_bool
326     {
327       \tl_if_eq:NnF \l_@@_space_in_string_tl { }
328       { \tl_set_eq:NN \l_@@_space_in_string_tl \@@_breakable_space: }

```

In the following code, we have to replace all the spaces in the token list `\l_tmpa_tl`. That means that this replacement must be “recursive”: even the spaces which are within brace groups (`{...}`) must be replaced. For instance, the spaces in long strings of Python are within such groups since there are within a command `\PitonStyle{String.Long}{...}`. That’s why the use of `\tl_replace_all:Nnn` is not enough.

The first implementation was using `\tl_regex_replace_all:nnN`

```
\tl_regex_replace_all:nnN { \x20 } { \c { @@_breakable_space: } } \l_tmpa_tl
```

but that programming was certainly slow.

Now, we use `\tl_replace_all:Nvn` *but*, in the styles `String.Long.Internal` we replace the spaces with `\@@_breakable_space:` by another use of the same technic with `\tl_replace_all:Nvn`. We do the same jog for the *doc strings* of Python and for the comments.

```

329     \tl_replace_all:Nvn \l_tmpa_tl
330     \c_catcode_other_space_tl
331     \@@_breakable_space:

```

```

332     }
333   }
334   \l_tmpa_tl
335 }
336 \cs_generate_variant:Nn \@@_replace_spaces:n { o }

```

In the contents provided by Lua, each line of the Python code will be surrounded by `\@@_begin_line:` and `\@@_end_line:`.

`\@@_begin_line:` is a TeX command with a delimited argument (`\@@_end_line:` is the marker for the end of the argument).

However, we define also `\@@_end_line:` as no-op, because, when the last line of the listing is the end of an environment of Beamer (eg `\end{uncoverenv}`), we will have a token `\@@_end_line:` added at the end without any corresponding `\@@_begin_line:`.

```

337 \cs_set_protected:Npn \@@_end_line: { }

338 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
339 {
340   \group_begin:
341   \int_gzero:N \g_@@_indentation_int

```

We put the potential number of line, the potential left and right margins.

```

342   \hbox_set:Nn \l_@@_line_box
343   {
344     \skip_horizontal:N \l_@@_left_margin_dim
345     \bool_if:NT \l_@@_line_numbers_bool
346     {

```

`\l_tmpa_int` will be equal to 1 when the current line is not empty.

```

347     \int_set:Nn \l_tmpa_int
348     {
349       \lua_now:e
350       {
351         tex.sprint
352         (

```

The following expression gives an integer of Lua (*integer* is a sub-type of *number* introduced in Lua 5.3), the output will be of the form "3" (and not "3.0") which is what we want for `\int_set:Nn`.

```

353         piton.empty_lines
354         [ \int_eval:n { \g_@@_line_int + 1 } ]
355       )
356     }
357   }
358   \bool_lazy_or:nnT
359   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
360   { ! \l_@@_skip_empty_lines_bool }
361   { \int_gincr:N \g_@@_visual_line_int }
362
363   \bool_lazy_or:nnTF
364   { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
365   { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
366   {
367     \bool_lazy_or:nnTF
368     { \int_compare_p:nNn { \l_@@_numbers_step_int } = 1 }
369     {
370       \int_compare_p:nNn
371       {
372         \int_mod:nn
373         { \g_@@_visual_line_int }
374         { \l_@@_numbers_step_int }
375       }
376       = \c_one_int
377     }
378   {

```

```

379         \str_if_eq:eeTF \l_@@_line_numbers_position_str { left }
380         \@@_print_number_left:
381         {
382             \seq_gput_right:Ne \g_@@_visual_line_numbers_seq
383             { \int_use:N \g_@@_visual_line_int }
384         }
385     }
386     {
387         \str_if_eq:eeTF \l_@@_line_numbers_position_str { right }
388         { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
389     }
390 }
391 {
392     \str_if_eq:eeTF \l_@@_line_numbers_position_str { right }
393     { \seq_gput_right:Nn \g_@@_visual_line_numbers_seq { } }
394 }
395 }

```

If there is a background, we must remind that there is a left margin of 0.5 em for the background (which will be added later).

```

396     \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
397     {
... but if only if the key left-margin is not used !
398         \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
399         { \skip_horizontal:n { 0.5 em } }
400     }

```

```

401     \bool_if:NTF \l_@@_minimize_width_bool
402     {
403         \hbox_set:Nn \l_tmpa_box
404         {
405             \language = -1
406             \raggedright
407             \strut
408             \@@_replace_spaces:n { #1 }
409             \strut \hfil
410         }
411         \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
412         { \box_use:N \l_tmpa_box }
413         { \@@_vtop_of_code:n { #1 } }
414     }
415     { \@@_vtop_of_code:n { #1 } }
416 }

```

Now, the line of code is composed in the box `\l_@@_line_box`.

```

417     \box_set_dp:Nn \l_@@_line_box { \box_dp:N \l_@@_line_box + 1.25 pt }
418     \box_set_ht:Nn \l_@@_line_box { \box_ht:N \l_@@_line_box + 1.25 pt }
419     \box_use_drop:N \l_@@_line_box
420     \group_end:
421     \g_@@_after_line_tl
422     \tl_gclear:N \g_@@_after_line_tl
423 }

```

The following command will be used in `\@@_begin_line: ... \@@_end_line:.`

```

424     \cs_new_protected:Npn \@@_vtop_of_code:n #1
425     {
426         \vbox_top:n
427         {
428             \hsize = \l_@@_code_width_dim
429             \language = -1
430             \raggedright

```

```

431     \strut
432     \@@_replace_spaces:n { #1 }
433     \strut \hfil
434   }
435 }

```

The following command will be used when the key `background-color` is used or when the key `line-numbers` is used in conjunction with `line-numbers/position=right`.

The content of the line has been previously set in `\l_@@_line_box`.

That command is used only once, in `\@@_add_bg_and_right_nb_to_output_box::`.

```

436 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_line_and_use:
437 {
438   \vtop
439   {
440     \offinterlineskip
441     \hbox
442     {

```

The command `\@@_compute_and_set_color:` sets the current color but also sets the booleans `\g_@@_color_is_none_bool` and `\g_@@_next_color_is_none_bool`. It uses the current value of `\l_@@_bg_color_clist`, the value of `\g_@@_line_int` (the number of the current line) but also potential token lists of the form `\g_@@_color_12_tl` if the end user has used the command `\rowcolor`.

```

443       \group_begin:
444       \@@_compute_and_set_color:

```

The colored panels are overlapping. However, if the special color `none` is used we must not put such overlapping.

```

445       \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
446       \bool_if:NT \g_@@_next_color_is_none_bool
447       { \dim_sub:Nn \l_tmpa_dim { 2.5 pt } }

```

When `\g_@@_color_is_none_bool` is in force, we will compose a `\vrule` of width 0 pt. We need that `\vrule` because it will be a strut.

```

448       \bool_if:NTF \g_@@_color_is_none_bool
449       { \dim_zero:N \l_tmpb_dim }
450       { \dim_set_eq:NN \l_tmpb_dim \l_@@_listing_width_dim }
451       \dim_set:Nn \l_@@_tmpc_dim { \box_ht:N \l_@@_line_box }

```

Now, the colored panel.

```

452       \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
453       {
454         \int_compare:nNnTF \g_@@_line_int = \c_one_int
455         {
456           \begin{tikzpicture}[baseline = 0cm]
457             \fill (0,0)
458               [rounded-corners = \l_@@_rounded_corners_dim]
459               -- (0,\l_@@_tmpc_dim)
460               -- (\l_tmpb_dim,\l_@@_tmpc_dim)
461               [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
462               -- (0,-\l_tmpa_dim)
463               -- cycle ;
464           \end{tikzpicture}
465         }
466         {
467           \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
468           {
469             \begin{tikzpicture}[baseline = 0cm]
470               \fill (0,0) -- (0,\l_@@_tmpc_dim)
471                 -- (\l_tmpb_dim,\l_@@_tmpc_dim)
472                 [rounded-corners = \l_@@_rounded_corners_dim]
473                 -- (\l_tmpb_dim,-\l_tmpa_dim)
474                 -- (0,-\l_tmpa_dim)
475                 -- cycle ;
476             \end{tikzpicture}
477           }

```

```

478         {
479             \vrule height \l_@@_tmpc_dim
480             depth \l_tmpa_dim
481             width \l_tmpb_dim

```

For the case when line-numbers/position=right is in force with line-numbers.

```

482             % added 2026-01-02
483             \dim_compare:nNt \l_tmpb_dim = \c_zero_dim
484             { \skip_horizontal:N \l_@@_listing_width_dim }
485         }
486     }
487 }
488 {
489     \vrule height \l_@@_tmpc_dim
490     depth \l_tmpa_dim
491     width \l_tmpb_dim

```

For the case when line-numbers/position=right is in force with line-numbers.

```

492     % added 2026-01-02
493     \dim_compare:nNt \l_tmpb_dim = \c_zero_dim
494     { \skip_horizontal:N \l_@@_listing_width_dim }
495 }

```

The group is for the color of the background.

```

496     \group_end:
497     % added 2026-01-02
498     \bool_if:NT \l_@@_line_numbers_bool
499     {
500         \str_if_eq:eeT \l_@@_line_numbers_position_str { right }
501         {
502             \seq_gpop_right:NN \g_@@_visual_line_numbers_seq \l_tmpa_tl
503             \@@_print_number_right:
504         }
505     }
506 }
507 \bool_if:NT \g_@@_next_color_is_none_bool
508 { \skip_vertical:n { 2.5 pt } }
509 \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
510 \box_use_drop:N \l_@@_line_box
511 }
512 }

```

End of \@@_add_bg_and_right_nb_to_line_and_use:

The command \@@_compute_and_set_color: sets the current color but also sets the booleans \g_@@_color_is_none_bool and \g_@@_next_color_is_none_bool. It uses the current value of \l_@@_bg_color_clist, the value of \g_@@_line_int (the number of the current line) but also potential token lists of the form \g_@@_color_12_tl if the end user has used the command \rowcolor.

```

513 \cs_set_protected:Npn \@@_compute_and_set_color:
514 {
515     \int_compare:nNtF \l_@@_bg_colors_int = \c_zero_int
516     { \tl_set:Nn \l_tmpa_tl { none } }
517     {
518         \int_set:Nn \l_tmpb_int
519         { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
520         \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
521     }

```

The row may have a color specified by the command \rowcolor. We check that point now.

```

522     \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
523     {
524         \tl_set_eq:Nc \l_tmpa_tl { g_@@_color_ \int_use:N \g_@@_line_int _ tl }

```

We don't need any longer the variable and that's why we delete it (it must be free for the next environment of piton).

```

525     \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
526 }

```

```

527 \tl_if_eq:NnTF \l_tmpa_tl { none }
528 { \bool_gset_true:N \g_@@_color_is_none_bool }
529 {
530   \bool_gset_false:N \g_@@_color_is_none_bool
531   \@@_color:o \l_tmpa_tl
532 }

```

We are looking for the next color because we have to know whether that color is the special color `none` (for the vertical adjustment of the background color).

```

533 \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
534 { \bool_gset_false:N \g_@@_next_color_is_none_bool }
535 {
536   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
537   { \tl_set:Nn \l_tmpa_tl { none } }
538   {
539     \int_set:Nn \l_tmpb_int
540     { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
541     \tl_set:Nn \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
542   }
543   \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
544   {
545     \tl_set_eq:Nc \l_tmpa_tl
546     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
547   }
548   \tl_if_eq:NnTF \l_tmpa_tl { none }
549   { \bool_gset_true:N \g_@@_next_color_is_none_bool }
550   { \bool_gset_false:N \g_@@_next_color_is_none_bool }
551 }
552 }

```

The following command `\@@_color:n` will accept both the instruction `\@@_color:n { red!15 }` and the instruction `\@@_color:n { [rgb]{0.9,0.9,0} }`.

```

553 \cs_set_protected:Npn \@@_color:n #1
554 {
555   \tl_if_head_eq_meaning:nNTF { #1 } [
556   {
557     \tl_set:Nn \l_tmpa_tl { #1 }
558     \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
559     \exp_last_unbraced:No \color \l_tmpa_tl
560   }
561   { \color { #1 } }
562 }
563 \cs_generate_variant:Nn \@@_color:n { o }

```

The command `\@@_par:` will be inserted by Lua between two lines of the computer listing.

- In fact, it will be inserted between two commands `\@@_begin_line:...\@@_end_of_line:.`
- When the key `break-lines-in-Piton` is in force, a line of the computer listing (the *input*) may result in several lines in the PDF (the *output*).
- Remind that `\@@_par:` has a rather complex behaviour because it will finish and start paragraphs.

```

564 \cs_new_protected:Npn \@@_par:
565 {

```

We recall that `\g_@@_line_int` is *not* used for the number of line printed in the PDF (when `line-numbers` is in force)...

```

566   \int_gincr:N \g_@@_line_int

```

... it will be used to allow or disallow page breaks, and also by the command `\rowcolor`.

Each line in the listing is composed in a box of TeX (which may contain several lines when the key `break-lines-in-Piton` is in force) put in a paragraph.

```

567   \par

```


We now add a `\kern` because each line of code is overlapping vertically by a quantity of 2.5 pt in order to have a good background (when `background-color` is in force). We need to use a `\kern` (in fact `\par\kern...`) and not a `\vskip` because page breaks should *not* be allowed on that kern.

```
568 \kern -2.5 pt
```

Now, we control page breaks after the paragraph.

```
569 \@@_add_penalty_for_the_line:
570 }
```

After the command `\@@_par:`, we will usually have a command `\@@_begin_line:`.

The following command `\@@_breakable_space:` is for breakable spaces in the environments `{Piton}` and the listings of `\PitonInputFile` and *not* for the commands `\piton`.

```
571 \cs_set_protected:Npn \@@_breakable_space:
572 {
573   \discretionary
574     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
575     {
576       \hbox_overlap_left:n
577         {
578           {
579             \normalfont \footnotesize \color { gray }
580             \l_@@_continuation_symbol_tl
581           }
582           \skip_horizontal:n { 0.3 em }
583           \int_compare:nNnT \l_@@_bg_colors_int > \c_zero_int
584             { \skip_horizontal:n { 0.5 em } }
585         }
586       \bool_if:NT \l_@@_indent_broken_lines_bool
587       {
588         \hbox:n
589           {
590             \prg_replicate:nn { \g_@@_indentation_int } { ~ }
591             { \color { gray } \l_@@_csoi_tl }
592           }
593       }
594     }
595   { \hbox { ~ } }
596 }
```

2.5 PitonOptions

```
597 \bool_new:N \l_@@_line_numbers_bool
598 \bool_new:N \l_@@_skip_empty_lines_bool
599 \bool_set_true:N \l_@@_skip_empty_lines_bool
600 \bool_new:N \l_@@_line_numbers_absolute_bool
601 \tl_new:N \l_@@_line_numbers_format_tl
602 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
603 \bool_new:N \l_@@_label_empty_lines_bool
604 \bool_set_true:N \l_@@_label_empty_lines_bool
605 \int_new:N \l_@@_number_lines_start_int
606 \str_new:N \l_@@_line_numbers_position_str
607 \str_set:Nn \l_@@_line_numbers_position_str { left }
608 \bool_new:N \l_@@_resume_bool
609 \bool_new:N \g_@@_label_as_zlabel_bool

610 \keys_define:nn { PitonOptions / marker }
611 {
612   beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
613   beginning .value_required:n = true ,
614   end .cs_set:Np = \@@_marker_end:n #1 ,
```

```

615     end .value_required:n = true ,
616     include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
617     unknown .code:n = \@@_error:n { Unknown-key-for-marker }
618 }

619 \keys_define:nn { PitonOptions / line-numbers }
620 {
621     true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
622     false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
623
624     start .code:n =
625         \bool_set_true:N \l_@@_line_numbers_bool
626         \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
627     start .value_required:n = true ,
628
629     skip-empty-lines .code:n =
630         \bool_if:NF \l_@@_in_PitonOptions_bool
631         { \bool_set_true:N \l_@@_line_numbers_bool }
632         \str_if_eq:nnTF { #1 } { false }
633         { \bool_set_false:N \l_@@_skip_empty_lines_bool }
634         { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
635
636     label-empty-lines .code:n =
637         \bool_if:NF \l_@@_in_PitonOptions_bool
638         { \bool_set_true:N \l_@@_line_numbers_bool }
639         \str_if_eq:nnTF { #1 } { false }
640         { \bool_set_false:N \l_@@_label_empty_lines_bool }
641         { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
642
643     absolute .code:n =
644         \bool_if:NTF \l_@@_in_PitonOptions_bool
645         { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
646         { \bool_set_true:N \l_@@_line_numbers_bool }
647         \bool_if:NT \l_@@_in_PitonInputFile_bool
648         {
649             \bool_set_true:N \l_@@_line_numbers_absolute_bool
650             \bool_set_false:N \l_@@_skip_empty_lines_bool
651         } ,
652     absolute .value_forbidden:n = true ,
653
654     resume .code:n =
655         \bool_set_true:N \l_@@_resume_bool
656         \bool_if:NF \l_@@_in_PitonOptions_bool
657         { \bool_set_true:N \l_@@_line_numbers_bool } ,
658     resume .value_forbidden:n = true ,
659
660     sep .dim_set:N = \l_@@_numbers_sep_dim ,
661     sep .value_required:n = true ,
662     sep .initial:n = 0.7 em ,
663
664     step .int_set:N = \l_@@_numbers_step_int ,
665     step .initial:n = 1 ,
666     step .value_required:n = true ,
667
668     position .choices:nn = { left , right }
669     { \str_set_eq:NN \l_@@_line_numbers_position_str \l_keys_choice_tl } ,
670     position .value_required:n = true ,
671
672     format .tl_set:N = \l_@@_line_numbers_format_tl ,
673     format .value_required:n = true ,
674
675     unknown .code:n =
676         \@@_unknown_key:nn

```

```

677     { PitonOptions / line-numbers }
678     { Unknown~key~for~line-numbers }
679
680 }

```

Be careful! The name of the following set of keys must be considered as public! Hence, it should *not* be changed.

```

681 \keys_define:nn { PitonOptions }
682 {
683   indentations-for-Foxit .choices:nn = { true , false }
684   {
685     \tl_if_eq:VnTF \l_keys_value_tl { true }
686     { \@@_define_leading_space_Foxit: }
687     { \@@_define_leading_space_normal: }
688   } ,
689   box .choices:nn = { c , t , b , m }
690   { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
691   box .default:n = c ,
692   break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
693   break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,

```

First, we put keys that should be available only in the preamble.

```

694   detected-commands .code:n =
695     \clist_if_in:nnTF { #1 } { rowcolor }
696     {
697       \@@_error:n { rowcolor-in-detected-commands }
698       \clist_set:Nn \l_tmpa_clist { #1 }
699       \clist_remove_all:Nn \l_tmpa_clist { rowcolor }
700       \clist_put_right:No \l_@@_detected_commands_clist \l_tmpa_clist
701     }
702     { \clist_put_right:Nn \l_@@_detected_commands_clist { #1 } } ,
703   detected-commands .value_required:n = true ,
704   detected-commands .usage:n = preamble ,
705   vertical-detected-commands .code:n = \@@_vertical_commands:n { #1 } ,
706   vertical-detected-commands .value_required:n = true ,
707   vertical-detected-commands .usage:n = preamble ,
708   raw-detected-commands .code:n =
709     \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 } ,
710   raw-detected-commands .value_required:n = true ,
711   raw-detected-commands .usage:n = preamble ,
712   detected-beamer-commands .code:n =
713     \@@_error_if_not_in_beamer:
714     \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
715   detected-beamer-commands .value_required:n = true ,
716   detected-beamer-commands .usage:n = preamble ,
717   detected-beamer-environments .code:n =
718     \@@_error_if_not_in_beamer:
719     \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
720   detected-beamer-environments .value_required:n = true ,
721   detected-beamer-environments .usage:n = preamble ,

```

Remark that the command `\lua_escape:n` is fully expandable. That's why we use `\lua_now:e`.

```

722   begin-escape .code:n =
723     \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
724   begin-escape .value_required:n = true ,
725   begin-escape .usage:n = preamble ,
726
727   end-escape .code:n =
728     \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
729   end-escape .value_required:n = true ,
730   end-escape .usage:n = preamble ,
731
732   begin-escape-math .code:n =
733     \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,

```

```

734 begin-escape-math .value_required:n = true ,
735 begin-escape-math .usage:n = preamble ,
736
737 end-escape-math .code:n =
738   \lua_now:e { \piton_end_escape_math = "\lua_escape:n{#1}" } ,
739 end-escape-math .value_required:n = true ,
740 end-escape-math .usage:n = preamble ,
741
742 comment-latex .code:n = \lua_now:n { \comment_latex = "#1" } ,
743 comment-latex .value_required:n = true ,
744 comment-latex .usage:n = preamble ,
745
746 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
747 label-as-zlabel .usage:n = preamble ,
748
749 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
750 math-comments .usage:n = preamble ,

```

Now, general keys.

```

751 language .code:n =
752   \str_set:N \l_piton_language_str { \str_lowercase:n { #1 } } ,
753 language .value_required:n = true ,
754 path .code:n =
755   \seq_clear:N \l_@@_path_seq
756   \clist_map_inline:nn { #1 }
757   {
758     \str_set:Nn \l_tmpa_str { ##1 }
759     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
760   } ,
761 path .value_required:n = true ,

```

The initial value of the key `path` is not empty: it's `.`, that is to say a comma separated list with only one component which is `.`, the current directory.

```

762 path .initial:n = . ,
763 path-write .str_set:N = \l_@@_path_write_str ,
764 path-write .value_required:n = true ,
765 font-command .tl_set:N = \l_@@_font_command_tl ,
766 font-command .initial:n = \ttfamily ,
767 font-command .value_required:n = true ,
768 font-command+ .code:n
769   = { \tl_put_right:Nn \l_@@_font_command_tl { #1 } } ,
770 font-command+ .value_required:n = true ,
771 font-command~+ .meta:n = { font-command+ = #1 } ,
772 font-command~+ .value_required:n = true ,
773 gobble .int_set:N = \l_@@_gobble_int ,
774 gobble .default:n = -1 ,
775 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
776 auto-gobble .value_forbidden:n = true ,
777 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
778 env-gobble .value_forbidden:n = true ,
779 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
780 tabs-auto-gobble .value_forbidden:n = true ,
781
782 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
783
784 split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,

```

When the key `split-on-empty-lines` is in force, the correspondint token list will be inserted between the chunks of code (the computer listing provided by the end user is split in chunks on the empty lines in the code). That parameter must contain elements to be inserted in *vertical* mode by TeX.

```

785 split-separation .tl_set:N = \l_@@_split_separation_tl ,
786 split-separation .value_required:n = true ,
787 split-separation .initial:n = \vspace { \baselineskip } \vspace { -1.25pt } ,
788

```

```

789 split-separation+ .code:n =
790   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
791 split-separation+ .value_required:n = true ,
792 split-separation~+ .meta:n = { split-separation+ = #1 } ,
793 add-to-split-separation .meta:n = { split-separation+ = #1 } ,
794
795 marker .code:n =
796   \bool_lazy_or:nnTF
797     \l_@@_in_PitonInputFile_bool
798     \l_@@_in_PitonOptions_bool
799     { \keys_set:nn { PitonOptions / marker } { #1 } }
800     { \@@_error:n { Invalid-key } } ,
801 marker .value_required:n = true ,
802
803 line-numbers .code:n =
804   \keys_set:nn { PitonOptions / line-numbers } { #1 } ,

```

The following line is mandatory.

```

805 line-numbers .default:n = true ,

```

The following counter corresponds to the key `splittable` of `\PitonOptions`. If the value of `\l_@@_splittable_int` is equal to n , then no line break can occur within the first n lines or the last n lines of a listing (or a *chunk* of listings when the key `split-on-empty-lines` is in force).

```

806 splittable .int_set:N = \l_@@_splittable_int ,
807 splittable .default:n = 1 ,
808 splittable .initial:n = 100 ,
809 background-color .code:n =
810   \clist_set:Nn \l_@@_bg_color_clist { #1 }

```

We keep the length of the clist `\l_@@_bg_color_clist` in a counter for efficiency only.

```

811   \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
812 background-color .value_required:n = true ,

```

The package `piton` will also detect the lines of code which correspond to the user input in a Python console, that is to say the lines of code beginning with `>>>` and `....`. It's possible, with the key `prompt-background-color`, to require a background for these lines of code (and the other lines of code will have the standard background color specified by `background-color`).

```

813 prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl ,
814 prompt-background-color .value_required:n = true ,
815 prompt-background-color .initial:n = gray!15 ,

```

With the tuning `write=false`, the content of the environment won't be parsed and won't be printed on the PDF. However, the Lua variables `piton.last_code` and `piton.last_language` will be set (and, hence, `piton.get_last_code` will be operational). The keys `join` and `write` will be honoured.

```

816 print .bool_set:N = \l_@@_print_bool ,
817 print .value_required:n = true ,
818 print .initial:n = true ,
819
820 width .code:n =
821   \str_if_eq:nnTF { #1 } { min }
822   {
823     \bool_set_true:N \l_@@_minimize_width_bool
824     \dim_zero:N \l_@@_width_dim
825   }
826   {
827     \bool_set_false:N \l_@@_minimize_width_bool
828     \dim_set:Nn \l_@@_width_dim { #1 }
829   } ,
830 width .value_required:n = true ,
831
832 max-width .code:n =
833   \bool_set_true:N \l_@@_minimize_width_bool
834   \dim_set:Nn \l_@@_width_dim { #1 } ,
835 max-width .value_required:n = true ,
836
837 paperclip .code:n =

```

```

838     \bool_set_true:N \l_@@_paperclip_bool
839     \tl_if_novalue:nTF { #1 }
840     { \str_set:Nn \l_@@_paperclip_str { } }
841     { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
842
843     annotation .bool_set:N = \l_@@_annotation_bool ,
844
845     write .str_set:N = \l_@@_write_str ,
846     write .value_required:n = true ,
847     no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
848     no-write .value_forbidden:n = true ,
849     join .code:n =
850     \str_set:Nn \l_@@_join_str { #1 }
851     \seq_if_in:NnF \g_@@_join_seq { #1 }
852     { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
853     join .value_required:n = true ,
854     join-separation .str_set:N = \l_@@_join_separation_str ,
855     join-separation .value_required:n = true ,
856     no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
857     no-join .value_forbidden:n = true ,
858
859     left-margin .code:n =
860     \str_if_eq:nnTF { #1 } { auto }
861     {
862         \dim_zero:N \l_@@_left_margin_dim
863         \bool_set_true:N \l_@@_left_margin_auto_bool
864     }
865     {
866         \dim_set:Nn \l_@@_left_margin_dim { #1 }
867         \bool_set_false:N \l_@@_left_margin_auto_bool
868     } ,
869     left-margin .value_required:n = true ,
870
871     right-margin .code:n =
872     \str_if_eq:nnTF { #1 } { auto }
873     {
874         \dim_zero:N \l_@@_right_margin_dim
875         \bool_set_true:N \l_@@_right_margin_auto_bool
876     }
877     {
878         \dim_set:Nn \l_@@_right_margin_dim { #1 }
879         \bool_set_false:N \l_@@_right_margin_auto_bool
880     } ,
881     right-margin .value_required:n = true ,
882
883     tab-size .int_set:N = \l_@@_tab_size_int ,
884     tab-size .value_required:n = true ,
885     tab-size .initial:n = 4 ,
886
887     show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
888     show-spaces .value_forbidden:n = true ,
889
890     show-spaces-in-strings .code:n =
891     \tl_set:Nn \l_@@_space_in_string_tl { } , % U+2423
892     show-spaces-in-strings .value_forbidden:n = true ,
893
894     break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
895     break-lines-in-Piton .initial:n = true ,
896
897     break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
898
899     break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
900     break-lines .value_forbidden:n = true ,

```

```

901
902 indent-broken-lines .bool_set:N      = \l_@@_indent_broken_lines_bool ,
903
904 end-of-broken-line .tl_set:N          = \l_@@_end_of_broken_line_tl ,
905 end-of-broken-line .value_required:n = true ,
906 end-of-broken-line .initial:n        = \hspace* { 0.5em } \textbackslash ,
907
908 continuation-symbol .tl_set:N         = \l_@@_continuation_symbol_tl ,
909 continuation-symbol .value_required:n = true ,
910 continuation-symbol .initial:n        = + ,
911
912 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
913 continuation-symbol-on-indentation .value_required:n = true ,
914 continuation-symbol-on-indentation .initial:n = $\hookrightarrow$ ,
915
916 first-line .code:n = \@@_in_PitonInputFile:n
917   { \int_set:Nn \l_@@_first_line_int { #1 } } ,
918 first-line .value_required:n = true ,
919
920 last-line .code:n = \@@_in_PitonInputFile:n
921   { \int_set:Nn \l_@@_last_line_int { #1 } } ,
922 last-line .value_required:n = true ,
923
924 begin-range .code:n = \@@_in_PitonInputFile:n
925   { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
926 begin-range .value_required:n = true ,
927
928 end-range .code:n = \@@_in_PitonInputFile:n
929   { \str_set:Nn \l_@@_end_range_str { #1 } } ,
930 end-range .value_required:n = true ,
931
932 range .code:n = \@@_in_PitonInputFile:n
933   {
934     \str_set:Nn \l_@@_begin_range_str { #1 }
935     \str_set:Nn \l_@@_end_range_str { #1 }
936   } ,
937 range .value_required:n = true ,
938
939 env-used-by-split .code:n =
940   \lua_now:n { piton.env_used_by_split = '#1' } ,
941 env-used-by-split .initial:n = Piton ,
942
943 resume .meta:n = line-numbers/resume ,
944
945 unknown .code:n =
946   \@@_unknown_key:nn
947     { PitonOptions }
948     { Unknown~key~for~PitonOptions } ,
949
950 % deprecated
951 all-line-numbers .code:n =
952   \bool_set_true:N \l_@@_line_numbers_bool
953   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
954 rounded-corners .code:n =
955   \AtBeginDocument
956   {
957     \IfPackageLoadedTF { tikz }
958       { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
959       { \@@_err_rounded_corners_without_Tikz: }
960   } ,
961 rounded-corners .default:n = 4 pt
962 }
963 \hook_gput_code:nnn { begindocument } { . }

```

```

964 {
965   \IfPackageLoadedTF { tcolorbox }
966   {
967     \pgfkeysifdefined { / tcb / libload / breakable }
968     {
969       \keys_define:nn { PitonOptions }
970       {
971         tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
972       }
973     }
974     {
975       \keys_define:nn { PitonOptions }
976       { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
977     }
978   }
979   {
980     \keys_define:nn { PitonOptions }
981     { tcolorbox .code:n = \@@_error:n { tcolorbox-not-loaded } }
982   }
983 }

984 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
985 {
986   \@@_error:n { rounded-corners-without~Tikz }
987   \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
988 }

989 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
990 {
991   \bool_if:NTF \l_@@_in_PitonInputFile_bool
992   { #1 }
993   { \@@_error:n { Invalid~key } }
994 }

995 \NewDocumentCommand \PitonOptions { m }
996 {
997   \bool_set_true:N \l_@@_in_PitonOptions_bool
998   \keys_set:nn { PitonOptions } { #1 }
999   \bool_set_false:N \l_@@_in_PitonOptions_bool
1000 }

```

When using `\NewPitonEnvironment` a user may use `\PitonOptions` inside. However, the set of keys available should be different that in standard `\PitonOptions`. That's why we define a version of `\PitonOptions` with no restriction on the set of available keys and we will link that version to `\PitonOptions` in such environment.

```

1001 \NewDocumentCommand \@@_fake_PitonOptions { }
1002 { \keys_set:nn { PitonOptions } }

```

2.6 The numbers of the lines

The following counter will be used to count the lines in the code when the user requires the numbers of the lines to be printed (with `line-numbers`) whereas the counter `\g_@@_line_int` previously defined is *not* used for that functionality.

```

1003 \int_new:N \g_@@_visual_line_int

1004 \cs_new_protected:Npn \@@_incr_visual_line:
1005 { \bool_if:NF \l_@@_skip_empty_lines_bool { \int_gincr:N \g_@@_visual_line_int } }

```


The following command will be used when the numbers of lines are printed on the left (`line-numbers/position=left`). The number of line is in the counter `\g_@@_visual_line_int`.

```

1006 \cs_new_protected:Npn \@@_print_number_left:
1007 {
1008   \hbox_overlap_left:n
1009   {
1010     \@@_actually_print_number:n { \int_to_arabic:n { \g_@@_visual_line_int } }
1011     \skip_horizontal:N \l_@@_numbers_sep_dim
1012   }
1013 }

```

The following command will be used when the numbers of lines are printed on the right (`line-numbers/position=right`). The number of line is in `\l_tmpa_tl`.

```

1014 \cs_new_protected:Npn \@@_print_number_right:
1015 {
1016   \hbox_overlap_left:n
1017   {
1018     \@@_actually_print_number:n { \l_tmpa_tl }
1019     \int_compare:nNnT \l_@@_bg_colors_int > 0
1020     { \skip_horizontal:n { 0.1 em } }
1021   }
1022 }

```

`\@@_actually_print_number:` itself prints the number without the `\hbox_overlap_left:n`. It is used by both `\@@_print_number_left:` and `\@@_print_number_right:`

```

1023 \cs_new_protected:Npn \@@_actually_print_number:n #1
1024 {
1025   \group_begin:
1026   \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }

```

We put braces. Thus, the user may use the key `line-numbers/format` with a value such as `\fbox`.

```

1027   \l_@@_line_numbers_format_tl { #1 }
1028   \pdfextension literal { EMC }
1029   \group_end:
1030 }

```

2.7 The main commands and environments for the end user

```

1031 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
1032 {
1033   \tl_if_novalue:nTF { #3 }

```

The last argument is provided by curryfication.

```

1034   { \@@_NewPitonLanguage:nnn { #1 } { #2 } }

```

The two last arguments are provided by curryfication.

```

1035   { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
1036 }

```

The following property list will contain the definitions of the computer languages as provided by the end user. However, if a language is defined over another base language, the corresponding list will contain the *whole* definition of the language.

```

1037 \prop_new:N \g_@@_languages_prop

1038 \keys_define:nn { NewPitonLanguage }
1039 {
1040   morekeywords .code:n = ,
1041   otherkeywords .code:n = ,
1042   sensitive .code:n = ,
1043   keywordsprefix .code:n = ,
1044   moretexcs .code:n = ,

```

```

1045     morestring .code:n = ,
1046     morecomment .code:n = ,
1047     moredelim .code:n = ,
1048     moredirectives .code:n = ,
1049     tag .code:n = ,
1050     alsodigit .code:n = ,
1051     alsoletter .code:n = ,
1052     alsoother .code:n = ,
1053     unknown .code:n = \@@_error:n { Unknown~key~NewPitonLanguage }
1054 }

```

The function `\@@_NewPitonLanguage:nnn` will be used when the language is *not* defined above a base language (and a base dialect).

```

1055 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
1056 {

```

We store in `\l_tmpa_tl` the name of the language with the potential dialect, that is to say, for example : `[AspectJ]{Java}`. We use `\tl_if_blank:nF` because the end user may have written `\NewPitonLanguage[]{Java}{...}`.

```

1057     \tl_set:Nx \l_tmpa_tl
1058     {
1059         \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
1060         \str_lowercase:n { #2 }
1061     }

```

The following set of keys is only used to raise an error when a key is unknown!

```

1062     \keys_set:nn { NewPitonLanguage } { #3 }

```

We store in LaTeX the definition of the language because some languages may be defined with that language as base language.

```

1063     \prop_gput:Non \g_@@_languages_prop \l_tmpa_tl { #3 }

```

The Lua part of the package `piton` will be loaded in a `\AtBeginDocument`. Hence, we will put also in a `\AtBeginDocument` the use of the Lua function `piton.new_language` (which does the main job).

```

1064     \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1065 }
1066 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1067 {
1068     \hook_gput_code:nnn { begindocument } { . }
1069     { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1070 }
1071 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

Now the case when the language is defined upon a base language.

```

1072 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnnn #1 #2 #3 #4 #5
1073 {

```

We store in `\l_tmpa_tl` the name of the base language with the dialect, that is to say, for example : `[AspectJ]{Java}`. We use `\tl_if_blank:nF` because the end user may have used `\NewPitonLanguage[Handel]{C}[]{C}{...}`

```

1074     \tl_set:Nx \l_tmpa_tl
1075     {
1076         \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1077         \str_lowercase:n { #4 }
1078     }

```

We retrieve in `\l_tmpb_tl` the definition (as provided by the end user) of that base language. Caution: `\g_@@_languages_prop` does not contain all the languages provided by `piton` but only those defined by using `\NewPitonLanguage`.

```

1079     \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

We can now define the new language by using the previous function.

```

1080     { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1081     { \@@_error:n { Language~not~defined } }
1082 }

```

```
1083 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4
```

In the following line, we write #4,#3 and not #3,#4 because we want that the keys which correspond to base language appear before the keys which are added in the language we define.

```
1084 { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1085 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
```

```
1086 \NewDocumentCommand { \piton } { }
1087 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
1088 \NewDocumentCommand { \@@_piton_standard } { m }
1089 {
1090   \group_begin:
1091   \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1092   {
```

Remind that, when `break-strings-anywhere` is in force, multiple commands `\-` will be inserted between the characters of the string to allow the breaks. The `\exp_not:N` before `\space` is mandatory.

```
1093     \bool_lazy_or:nnT
1094     \l_@@_break_lines_in_piton_bool
1095     \l_@@_break_strings_anywhere_bool
1096     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1097 }
```

The following tuning of LuaTeX in order to avoid all breaks of lines on the hyphens.

```
1098 \automatichyphenmode = 1
```

Remark that the argument of `\piton` (with the normal syntax) is expanded in the TeX sens, (see the `\tl_set:Ne` below) and that's why we can provide the following escapes to the end user:

```
1099 \cs_set_eq:NN \ \ \c_backslash_str
1100 \cs_set_eq:NN \% \c_percent_str
1101 \cs_set_eq:NN \{ \c_left_brace_str
1102 \cs_set_eq:NN \} \c_right_brace_str
1103 \cs_set_eq:NN \$ \c_dollar_str
```

The standard command `_` is *not* expandable and we need here expandable commands. With the following code, we define an expandable command.

```
1104 \cs_set_eq:cN { ~ } \space
1105 \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
```

We redefine `\rowcolor` inside of `\piton` commands to do nothing.

```
1106 \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1107 \tl_set:Ne \l_tmpa_tl
1108 {
1109   \lua_now:e
1110   { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1111   { #1 }
1112 }
1113 \bool_if:NTF \l_@@_show_spaces_bool
1114 { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1115 {
1116   \bool_if:NT \l_@@_break_lines_in_piton_bool
```

With the following line, the spaces of catcode 12 (which were not breakable) are replaced by `\space`, and, thus, become breakable.

```
1117   { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl \space }
1118 }
```

The command `\text` is provided by the package `amstext` (loaded by `piton`).

```
1119 \if_mode_math:
1120   \text { \l_@@_font_command_tl \l_tmpa_tl }
1121 \else:
1122   \l_@@_font_command_tl \l_tmpa_tl
1123 \fi:
1124 \group_end:
```

```

1125 }

1126 \NewDocumentCommand { \@@_piton_verbatim } { v }
1127 {
1128   \group_begin:
1129   \automatichyphenmode = 1
1130   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

We redefine `\rowcolor` inside of `\piton` commands to do nothing.

```

1131   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor

1132   \tl_set:Ne \l_tmpa_tl
1133   {
1134     \lua_now:e
1135     { piton.Parse('\l_piton_language_str',token.scan_string()) }
1136     { #1 }
1137   }
1138   \bool_if:NT \l_@@_show_spaces_bool
1139   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1140   \if_mode_math:
1141     \text { \l_@@_font_command_tl \l_tmpa_tl }
1142   \else:
1143     \l_@@_font_command_tl \l_tmpa_tl
1144   \fi:
1145   \group_end:
1146 }

```

The following command does *not* correspond to a user command. It will be used when we will have to “rescan” some chunks of computer code. For example, it will be the initial value of the Piton style `InitialValues` (the default values of the arguments of a Python function).

```

1147 \cs_new_protected:Npn \@@_piton:n #1
1148 { \tl_if_blank:NF { #1 } { \@@_piton_i:n { #1 } } }

1149
1150 \cs_new_protected:Npn \@@_piton_i:n #1
1151 {
1152   \group_begin:
1153   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1154   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1155   \cs_set:cpn { pitonStyle _ Prompt } { }
1156   \cs_set_eq:NN \@@_leading_space: \space
1157   \cs_set_eq:NN \@@_trailing_space: \space
1158   \tl_set:Ne \l_tmpa_tl
1159   {
1160     \lua_now:e
1161     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1162     { #1 }
1163   }
1164   \bool_if:NT \l_@@_show_spaces_bool
1165   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1166   \@@_replace_spaces:o \l_tmpa_tl
1167   \group_end:
1168 }

```

`\@@_pre_composition:` will be used both in `\PitonInputFile` and in the environments such as `{Piton}`.

```

1169 \cs_new_protected:Npn \@@_pre_composition:
1170 {
1171   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1172   {
1173     \dim_set_eq:NN \l_@@_width_dim \linewidth

```

When the key `box` is used, `width=min` is activated (except when `width` has been used with a numerical value).

```

1174     \str_if_empty:NF \l_@@_box_str
1175     { \bool_set_true:N \l_@@_minimize_width_bool }
1176 }

```

We compute `\l_@@_listing_width_dim`. However, if `max-width` is used (or `width=min` which uses `max-width`), that length will be computed again in `\@@_create_output_box`: but **even in the case**, we have to compute that value now (because the maximal width set by `max-width` may be reached by some lines of the listing—and those lines would be wrapped).

```

1177     \dim_set:Nn \l_@@_listing_width_dim
1178     {
1179         \bool_if:NTF \l_@@_tcolorbox_bool
1180         {
1181             \l_@@_width_dim -
1182             ( \kvtcb@left@rule
1183               + \kvtcb@leftupper
1184               + \kvtcb@boxsep * 2
1185               + \kvtcb@rightupper
1186               + \kvtcb@right@rule )
1187         }
1188         { \l_@@_width_dim }
1189     }

1190     \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1191     \automatichyphenmode = 1
1192     \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1193     \g_@@_def_vertical_commands_tl
1194     \int_gzero:N \g_@@_line_int
1195     \int_gzero:N \g_@@_nb_lines_int
1196     \dim_zero:N \parindent
1197     \dim_zero:N \lineskip
1198     \dim_zero:N \parskip
1199
1200     % added 2026-01-02
1201     \seq_gclear:N \g_@@_visual_line_numbers_seq
1202
1203     \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

For efficiency, we keep in `\l_@@_bg_colors_int` the length of `\l_@@_bg_color_clist`.

```

1204     \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1205     { \bool_set_true:N \l_@@_bg_bool }
1206     \bool_gset_false:N \g_@@_rowcolor_inside_bool
1207     \IfPackageLoadedTF { zref-base }
1208     {
1209         \bool_if:NTF \g_@@_label_as_zlabel_bool
1210         { \cs_set_eq:NN \label \@@_zlabel:n }
1211         { \cs_set_eq:NN \label \@@_label:n }
1212         \cs_set_eq:NN \zlabel \@@_zlabel:n
1213     }
1214     { \cs_set_eq:NN \label \@@_label:n }
1215     \l_@@_font_command_tl
1216 }

```

When the parameters `line-numbers`, `line-numbers/position=left` and `left-margin` are in force (or if `line-numbers`, `line-numbers=right` and `right-margin` are in force), we have to compute the width of the maximal number of lines at the end of the environment to fix the correct value to `left-margin` (or `right-margin`).

The command `\@@_compute_margin:N` will do that job.

It's argument must be either `\l_@@_left_margin_dim` either `\l_@@_right_margin_dim`.

```

1217 \cs_new_protected:Npn \@@_compute_margin:N #1
1218 {
1219     \use:e
1220     {

```

```

1221     \bool_if:NTF \l_@@_skip_empty_lines_bool
1222     { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1223     { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1224     { \l_@@_listing_tl }
1225   }
1226   \hbox_set:Nn \l_tmpa_box
1227   {
1228     \l_@@_line_numbers_format_tl
1229     \int_to_arabic:n
1230     {
1231       \g_@@_visual_line_int
1232       +
1233       \bool_if:NTF \l_@@_skip_empty_lines_bool
1234       { \l_@@_nb_non_empty_lines_int }
1235       { \g_@@_nb_lines_int }
1236     }
1237   }
1238   \dim_set:Nn #1 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1239 }

```

The following command computes `\l_@@_code_width_dim`.

It will be used only once (in `\@@_create_output_box:`).

If there is a background (even a background with the color `none`), we subtract 0.5 em on both sides. However, if there is a left margin or a right margin, we use those margins. If the key `left-margin` has been used with the special value `auto` (this is meaningful only in conjunction with the key `line-numbers` and a value of `line-numbers/position` equal to `left`), the actual value for the left margin has yet computed (and stored in `left-margin`). Idem for the right margin.

```

1240 \cs_new_protected:Npn \@@_compute_code_width:
1241 {
1242   \dim_set:Nn \l_@@_code_width_dim
1243   {
1244     \l_@@_listing_width_dim
1245     -
1246     (
1247       \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1248       {
1249         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1250         { \l_@@_left_margin_dim }
1251         { 0.5 em }
1252       +
1253       \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1254       { \l_@@_right_margin_dim }
1255       { 0.5 em }
1256     )
1257     { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1258   )
1259 }
1260 }

```

The following command computes `\l_@@_listing_width_dim` and it will be used when `max-width` (or `width=min`) is used. Remind that the key `box` sets `width=min` (except when `width` is used with a numerical value).

It will be used only once (in `\@@_create_output_box:`).

The computation is the inverse of the computation done in `\@@_compute_code_width:`.

```

1261 \cs_new_protected:Npn \@@_recompute_listing_width:
1262 {
1263   \dim_set:Nn \l_@@_listing_width_dim
1264   {
1265     \box_wd:N \g_@@_output_box
1266     +
1267     \int_compare:nNnTF \l_@@_bg_colors_int > \c_zero_int
1268     {

```

```

1269         \dim_compare:nNnTF \l_@@_left_margin_dim > \c_zero_dim
1270         { \l_@@_left_margin_dim }
1271         { 0.5 em }
1272     +
1273     \dim_compare:nNnTF \l_@@_right_margin_dim > \c_zero_dim
1274     { \l_@@_right_margin_dim }
1275     { 0.5 em }
1276 }
1277 { \l_@@_left_margin_dim + \l_@@_right_margin_dim }
1278 }
1279 }

```

```

1280 \cs_new_protected:Npn \@@_store_body:n #1
1281 {

```

Now, we have to replace all the occurrences of `\obeyedline` by a character of end of line (`\r` in the strings of Lua).

```

1282     \tl_set:Nc \obeyedline { \char_generate:nn { 13 } { 11 } }
1283     \tl_set:Nc \l_@@_listing_tl { #1 }
1284     \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1285 }

```

The first argument of the following macro is one of the four strings: New, Renew, Provide and Declare.

```

1286 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1287 {
1288     \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1289     {
1290         \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1291         #4
1292         \@@_pre_composition:
1293         \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1294         {
1295             \int_gset:Nn \g_@@_visual_line_int
1296             { \l_@@_number_lines_start_int - 1 }
1297         }
1298         \bool_if:NT \g_@@_beamer_bool
1299         { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1300         \bool_if:NT \g_@@_footnote_bool \savenotes
1301         \@@_composition:
1302         \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1303         { \@@_create_paperclip_annotation: }
1304         \bool_if:NT \g_@@_footnote_bool \endsavenotes
1305         #5
1306     }
1307     { \ignorespacesafterend }
1308 }

```

`\marginalia` is a command of the package `marginalia` (loaded by `piton`).

```

1309 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1310 {
1311     \marginalia
1312     {
1313         \vspace* { - 0.8 em }
1314         \hbox:n
1315         {
1316             \vrule~height~0~pt~depth~12~pt~width~0~pt
1317             \bool_if:NT \l_@@_annotation_bool
1318             {
1319                 \lua_now:n
1320                 {

```

The function `piton.utf16` does a conversion from utf8 to utf16 big endian encoded in hexadecimal (with the BOM of big endian), which is suitable to be put in a string between angular brackets of the PDF. It's easier for a stream!

```

1321         pdf.immediateobj
1322         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1323     }
1324     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1325     {
1326         /Subtype /Text
1327         /Contents~\pdf_object_ref_last:
1328         /Name /Note
1329         /Subj (Computer~listing)

```

The following tries to specify that the note should not receive answers (since it is meant for an easy copy-past of the computer listing).

```

1330         /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1331         /F~512
1332         /C [0.8~0.8~0.8]
1333     }
1334     \hspace* { 7 mm }
1335 }
1336 \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1337 }
1338 }
1339 }

```

```

1340 \cs_new_protected:Npn \@@_create_paperclip:
1341 {
1342     \str_if_empty:NT \l_@@_paperclip_str
1343     {
1344         \int_gincr:N \g_@@_paperclip_int
1345         \str_set:Ne \l_@@_paperclip_str { listing~\int_use:N \g_@@_paperclip_int .txt }
1346     }

```

Here, we don't understand why the tostring is mandatory.

```

1347     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1348     \box_move_down:nn
1349     { 10 pt }
1350     {
1351         \hbox:n
1352         {
1353             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1354             {
1355                 /Subtype /FileAttachment
1356                 /Name /Paperclip
1357                 /F~8 % no zoom

```

/Contents will be used as info-bulle and description of the file in the panel of the embedded files.

```

1358             /Contents (The~computer~listing)
1359             /FS <<
1360                 /Type /Filespec
1361                 /F (\l_@@_paperclip_str)
1362                 /EF << /F~\pdf_object_ref_last: >>
1363                 /AFRelationship /Supplement
1364             >>
1365         }
1366     }
1367 }
1368 }

```

For the following commands, the arguments are provided by curryfication.

```

1369 \NewDocumentCommand { \NewPitonEnvironment } { }
1370 { \@@_DefinePitonEnvironment:nnnnn { New } }
1371 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1372 { \@@_DefinePitonEnvironment:nnnnn { Declare } }

```



```

1373 \NewDocumentCommand { \RenewPitonEnvironment } { }
1374 { \@@_DefinePitonEnvironment:nnnnn { Renew } }

1375 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1376 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1377 \cs_new_protected:Npn \@@_translate_beamer_env:n
1378 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1379 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1380 \cs_new_protected:Npn \@@_composition:
1381 {
1382   \str_if_empty:NT \l_@@_box_str
1383   {
1384     \mode_if_vertical:F
1385     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1386   }

1387   \bool_if:NT \l_@@_line_numbers_bool
1388   {
1389     \bool_lazy_and:nnT
1390     { \l_@@_left_margin_auto_bool }
1391     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { left } }
1392     { \@@_compute_margin:N \l_@@_left_margin_dim }
1393     \bool_lazy_and:nnT
1394     { \l_@@_right_margin_auto_bool }
1395     { \str_if_eq_p:ee \l_@@_line_numbers_position_str { right } }
1396     { \@@_compute_margin:N \l_@@_right_margin_dim }
1397   }

1398   \lua_now:e
1399   {
1400     piton.join_separation = "\l_@@_join_separation_str"
1401     piton.join = "\l_@@_join_str"
1402     piton.write = "\l_@@_write_str"
1403     piton.path_write = "\l_@@_path_write_str"
1404   }

1405   \noindent
1406   \bool_if:NTF \l_@@_print_bool
1407   {

```

When `split-on-empty-lines` is in force, each chunk will be formatted by an environment `{Piton}` (or the environment specified by `env-used-by-split`). Within each of these environments, we will come back here (but, of course, `split-on-empty-line` will have been set to `false`). The mechanism “retrieve” is mandatory.

```

1408   \bool_if:NTF \l_@@_split_on_empty_lines_bool
1409   { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1410   {
1411     \@@_create_output_box:

```

Now, the listing has been composed in `\g_@@_output_box` and `\l_@@_listing_width_dim` contains the width of the listing (with the potential margin for the numbers of lines).

```

1412   \bool_if:NTF \l_@@_tcolorbox_bool
1413   {
1414     \str_if_empty:NTF \l_@@_box_str
1415     \@@_composition_iii:
1416     \@@_composition_iv:
1417   }
1418   {
1419     \str_if_empty:NTF \l_@@_box_str
1420     \@@_composition_i:
1421     \@@_composition_ii:
1422   }
1423 }
1424 }

```

```

1425     { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1426 }

```

`\@@_composition_i:` is for the main case: the key `tcolorbox` is not used, nor the key `box`.

We can't do a mere `\vbox_unpack:N \g_@@_output_box` because that would not work inside a list of LaTeX (`{\itemize}` or `{\enumerate}`).

The composition in the box `\g_@@_output_box` was mandatory to be able to deal with the case of a conjunction of the keys `width=min` and `background-color=...`.

```

1427 \cs_new_protected:Npn \@@_composition_i:
1428 {

```

First, we “reverse” the box `\g_@@_output_box`: we put in the box `\g_tmpa_box` the boxes present in `\g_@@_output_box`, but in reversed order. The vertical spaces and the penalties are discarded.

```

1429   \box_clear:N \g_tmpa_box

```

The box `\g_@@_line_box` will be used as an auxiliary box.

```

1430   \box_clear_new:N \g_@@_line_box

```

We unpack `\g_@@_output_box` in `\l_tmpa_box` used as a scratched box.

```

1431   \vbox_set:Nn \l_tmpa_box
1432   {
1433     \vbox_unpack_drop:N \g_@@_output_box
1434     \bool_gset_false:N \g_tmpa_bool
1435     \unskip \unskip
1436     \bool_gset_false:N \g_tmpa_bool
1437     \bool_do_until:nn \g_tmpa_bool
1438     {
1439       \unskip \unskip \unskip
1440       \unpenalty \unkern
1441       \box_set_to_last:N \l_@@_line_box
1442       \box_if_empty:NTF \l_@@_line_box
1443       { \bool_gset_true:N \g_tmpa_bool }
1444       {
1445         \vbox_gset:Nn \g_tmpa_box
1446         {
1447           \vbox_unpack:N \g_tmpa_box
1448           \box_use:N \l_@@_line_box
1449         }
1450       }
1451     }
1452   }

```

Now, we will loop over the boxes in `\g_tmpa_box` and compose the boxes in the TeX flow.

```

1453   \bool_gset_false:N \g_tmpa_bool
1454   \int_zero:N \g_@@_line_int
1455   \bool_do_until:nn \g_tmpa_bool
1456   {

```

We retrieve the last box of `\g_tmpa_box` (and store it in `\g_@@_line_box`) and keep the other boxes in `\g_tmpa_box`.

```

1457     \vbox_gset:Nn \g_tmpa_box
1458     {
1459       \vbox_unpack_drop:N \g_tmpa_box
1460       \box_gset_to_last:N \g_@@_line_box
1461     }

```

If the box that we have retrieved is void, that means that, in fact, there is no longer boxes in `\g_tmpa_box` and we will exit the loop.

```

1462     \box_if_empty:NTF \g_@@_line_box
1463     { \bool_gset_true:N \g_tmpa_bool }
1464     {
1465       \box_use:N \g_@@_line_box
1466       \int_gincr:N \g_@@_line_int
1467       \par
1468       \kern -2.5 pt

```

We will determine the penalty by reading the Lua table `piton.lines_status`. That will use the current value of `\g_@@_line_int`.

```
1469 \@@_add_penalty_for_the_line:
```

We now add the instructions corresponding to the *vertical detected commands* that are potentially used in the corresponding line of the listing.

```
1470 \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 }
1471 { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 } }
1472 \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1473 { \mode_leave_vertical: }
1474 }
1475 }
1476 \skip_vertical:n { 2.5 pt }
1477 }
```

`\@@_composition_ii`: will be used when the key `box` is in force but *not* the key `tcolorbox`.

```
1478 \cs_new_protected:Npn \@@_composition_ii:
1479 {
1480 \use:e { \begin { minipage } [ \l_@@_box_str ] }
1481 { \l_@@_listing_width_dim }
```

Here, `\vbox_unpack:N`, instead of `\box_use:N` is mandatory for the vertical position of the box.

```
1482 \vbox_unpack:N \g_@@_output_box
```

`\kern` is mandatory here (`\skip_vertical:n` won't work).

```
1483 \kern 2.5 pt
1484 \end { minipage }
1485 }
```

`\@@_composition_iii`: will be used when the key `tcolorbox` is in force but *not* the key `box`.

```
1486 \cs_new_protected:Npn \@@_composition_iii:
1487 {
1488 \use:e
1489 {
1490 \begin { tcolorbox }
```

Even though we use the key `breakable` of `{tcolorbox}`, our environment will be breakable only when the key `splittable` of `piton` is used.

```
1491 [ breakable , text~width = \l_@@_listing_width_dim ]
1492 }
1493 \par
1494 \vbox_unpack:N \g_@@_output_box
1495 \end { tcolorbox }
1496 }
```

`\@@_composition_iv`: will be used when both keys `tcolorbox` and `box` are in force.

```
1497 \cs_new_protected:Npn \@@_composition_iv:
1498 {
1499 \use:e
1500 {
1501 \begin { tcolorbox }
1502 [
1503 hbox ,
1504 text~width = \l_@@_listing_width_dim ,
1505 nobeforeafter ,
1506 box~align =
1507 \str_case:Nn \l_@@_box_str
1508 {
1509 t { top }
1510 b { bottom }
1511 c { center }
1512 m { center }
1513 }
1514 ]
1515 }
```

```

1516   \box_use:N \g_@@_output_box
1517   \end { tcolorbox }
1518 }

```

The following function will add the correct vertical penalty after a line of code in order to control the breaks of the pages. We use the Lua table `piton.lines_status` which has been written by `piton.ComputeLinesStatus` for this aim. Each line has a “status” (equal to 0, 1 or 2) and that status directly says whether a break is allowed.

```

1519 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1520 {
1521   \int_case:nn
1522   {
1523     \lua_now:e
1524     {
1525       tex.sprint
1526       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1527     }
1528   }
1529   { 1 { \penalty 100 } 2 \nobreak }
1530 }

```

`\@@_create_output_box:` is used only once, in `\@@_composition:`.

It creates (and modifies when there are backgrounds or numbers of the lines on the right) `\g_@@_output_box`.

```

1531 \cs_new_protected:Npn \@@_create_output_box:
1532 {
1533   \@@_compute_code_width:
1534   \vbox_gset:Nn \g_@@_output_box
1535   { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1536   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1537   \bool_lazy_any:nT
1538   {
1539     { \int_compare_p:nNn \l_@@_bg_colors_int > \c_zero_int }
1540     { \g_@@_rowcolor_inside_bool }
1541     {
1542       \l_@@_line_numbers_bool
1543       &&
1544       \str_if_eq_p:ee \l_@@_line_numbers_position_str { right }
1545     }
1546   }
1547   \@@_add_bg_and_right_nb_to_output_box:
1548 }

```

We add the backgrounds after the composition of the box `\g_@@_output_box` by a loop over the lines in that box. Idem when the key `line-numbers` is used in conjunction with `line-numbers/position=right`.

The backgrounds will have a width equal to `\l_@@_listing_width_dim`.

That command will be used only once, in `\@@_create_output_box:`.

```

1549 \cs_new_protected:Npn \@@_add_bg_and_right_nb_to_output_box:
1550 {
1551   \int_gset_eq:NN \g_@@_line_int \g_@@_nb_lines_int

```

`\l_tmpa_box` is only used to *unpack* the vertical box `\g_@@_output_box`.

```

1552   \vbox_set:Nn \l_tmpa_box
1553   {
1554     \vbox_unpack_drop:N \g_@@_output_box

```

We will raise `\g_tmpa_bool` to exit the loop `\bool_do_until:nn` below.

```

1555     \bool_gset_false:N \g_tmpa_bool
1556     \unskip \unskip

```

We begin the loop.

```

1557     \bool_do_until:nn \g_tmpa_bool

```

```

1558     {
1559         \unskip \unskip \unskip
1560         \int_set_eq:NN \l_tmpa_int \lastpenalty
1561         \unpenalty \unkern

```

In standard TeX (not LuaTeX), the only way to loop over the sub-boxes of a given box is to use the TeX primitive `\lastbox` (via `\box_set_to_last:N` of L3). Of course, it would be interesting to replace that programming by a programming in Lua of LuaTeX...

```

1562         \box_set_to_last:N \l_@@_line_box
1563         \box_if_empty:NTF \l_@@_line_box
1564         { \bool_gset_true:N \g_tmpa_bool }
1565         {

```

`\g_@@_line_int` will be used in `\@@_add_bg_and_right_nb_to_line_and_use:`.

```

1566         \vbox_gset:Nn \g_@@_output_box
1567         {

```

The command `\@@_add_bg_and_right_nb_to_line_and_use:` will add a background to the line (in `\l_@@_line_box`) but will also put the line in the current box. The background will have a width equal to `\l_@@_listing_width_dim`.

```

1568             \@@_add_bg_and_right_nb_to_line_and_use:
1569             \kern -2.5 pt
1570             \penalty \l_tmpa_int
1571             \vbox_unpack:N \g_@@_output_box
1572         }
1573     }
1574     \int_gdecr:N \g_@@_line_int
1575 }
1576 }
1577 }

```

The following will be used when the end user has used `print=false`.

```

1578 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1579 {
1580     \lua_now:e
1581     {
1582         piton.GobbleParseNoPrint
1583         (
1584             '\l_piton_language_str' ,
1585             \int_use:N \l_@@_gobble_int ,
1586             token.scan_argument ( )
1587         )
1588     }
1589 }
1590 \cs_generate_variant:Nn \@@_gobble_parse_no_print:n { o }

```

The following function will be used when the key `split-on-empty-lines` is not in force. It will retrieve the first empty line, gobble the spaces at the beginning of the lines and parse the code. The argument is provided by curryfication.

```

1591 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1592 {
1593     \lua_now:e
1594     {
1595         piton.RetrieveGobbleParse
1596         (
1597             '\l_piton_language_str' ,
1598             \int_use:N \l_@@_gobble_int ,
1599             \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1600             { \int_eval:n { - \l_@@_splittable_int } }
1601             { \int_use:N \l_@@_splittable_int } ,
1602             token.scan_argument ( )
1603         )
1604     }
1605 }

```

```
1606 \cs_generate_variant:Nn \@@_retrieve_gobble_parse:n { o }
```

The following function will be used when the key `split-on-empty-lines` is in force. It will gobble the spaces at the beginning of the lines (if the key `gobble` is in force), then split the code at the empty lines and, eventually, parse the code. The argument is provided by curryfication.

```
1607 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1608 {
1609   \lua_now:e
1610   {
1611     piton.RetrieveGobbleSplitParse
1612     (
1613       '\l_piton_language_str' ,
1614       \int_use:N \l_@@_gobble_int ,
1615       \int_use:N \l_@@_splittable_int ,
1616       token.scan_argument ( )
1617     )
1618   }
1619 }
1620 \cs_generate_variant:Nn \@@_retrieve_gobble_split_parse:n { o }
```

Now, we define the environment `{Piton}`, which is the main environment provided by the package `piton`. Of course, you use `\NewPitonEnvironment`.

```
1621 \bool_if:NTF \g_@@_beamer_bool
1622 {
1623   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1624   {
1625     \keys_set:nn { PitonOptions } { #2 }
1626     \begin { actionenv } < #1 >
1627   }
1628   { \end { actionenv } }
1629 }
1630 {
1631   \NewPitonEnvironment { Piton } { 0 { } }
1632   { \keys_set:nn { PitonOptions } { #1 } }
1633   { }
1634 }

1635 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1636 {
1637   \mode_if_vertical:F { \par }
1638   \group_begin:
1639   \seq_concat:NNN
1640     \l_file_search_path_seq
1641     \l_@@_path_seq
1642     \l_file_search_path_seq
1643   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1644   {
1645     \@@_input_file:nn { #1 } { #2 }
1646     #4
1647   }
1648   { #5 }
1649   \group_end:
1650 }

1651 \cs_new_protected:Npn \@@_unknown_file:n #1
1652 { \msg_error:nnn { piton } { Unknown~file } { #1 } }
1653 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1654 {
1655   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1656   {
```

The following line is for `latexmk` (suggestion of Y. Salmon).

```

1657     \iow_log:n { No~file~#3 }
1658     \@@_unknown_file:n { #3 }
1659   }
1660 }
1661 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1662 {
1663   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1664   {

```

The following line is for latexmk (suggestion of Y. Salmon).

```

1665     \iow_log:n { No~file~#3 }
1666     \@@_unknown_file:n { #3 }
1667   }
1668 }
1669 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1670 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

The following command uses as implicit argument the name of the file in `\l_@@_file_name_str`.

```

1671 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1672 {

```

We recall that, if we are in Beamer, the command `\PitonInputFile` is “overlay-aware” and that’s why there is an optional argument between angular brackets (`<` and `>`).

```

1673   \tl_if_novalue:nF { #1 }
1674   {
1675     \bool_if:NTF \g_@@_beamer_bool
1676     { \begin { uncoverenv } < #1 > }
1677     { \@@_error_or_warning:n { overlay~without~beamer } }
1678   }
1679   \group_begin:

```

The following line is to allow tools such as latexmk to be aware that the file read by `\PitonInputFile` is loaded during the compilation of the LaTeX document.

```

1680   \iow_log:e { (\l_@@_file_name_str) }
1681   \int_zero_new:N \l_@@_first_line_int
1682   \int_zero_new:N \l_@@_last_line_int
1683   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1684   \bool_set_true:N \l_@@_in_PitonInputFile_bool
1685   \keys_set:nn { PitonOptions } { #2 }
1686   \bool_if:NT \l_@@_line_numbers_absolute_bool
1687   { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1688   \bool_if:NTF
1689   {
1690     (
1691       \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1692       || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1693     )
1694     && ! \str_if_empty_p:N \l_@@_begin_range_str
1695   }
1696   {
1697     \@@_error_or_warning:n { bad-range-specification }
1698     \int_zero:N \l_@@_first_line_int
1699     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1700   }
1701   {
1702     \str_if_empty:NF \l_@@_begin_range_str
1703     {
1704       \@@_compute_range:
1705       \bool_lazy_or:nnT
1706       \l_@@_marker_include_lines_bool
1707       { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1708       {
1709         \int_decr:N \l_@@_first_line_int
1710         \int_incr:N \l_@@_last_line_int
1711       }
1712     }

```

```

1713     }
1714     \@@_pre_composition:
1715     \bool_if:NT \l_@@_line_numbers_absolute_bool
1716     { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1717     \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1718     {
1719         \int_gset:Nn \g_@@_visual_line_int
1720         { \l_@@_number_lines_start_int - 1 }
1721     }

```

The following case arises when the code `line-numbers/absolute` is in force without the use of a marked range.

```

1722     \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1723     { \int_gzero:N \g_@@_visual_line_int }
1724     \lua_now:e
1725     {

```

The following command will store the content of the file (or only a part of that file) in `\l_@@_listing_tl`.

```

1726         piton.ReadFile(
1727             '\l_@@_file_name_str' ,
1728             \int_use:N \l_@@_first_line_int ,
1729             \int_use:N \l_@@_last_line_int )
1730     }
1731     \@@_composition:
1732     \group_end:

```

We recall that, if we are in Beamer, the command `\PitonInputFile` is “overlay-aware” and that’s why we close now an environment `{uncoverenv}` that we have opened at the beginning of the command.

```

1733     \tl_if_novalue:nF { #1 }
1734     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1735 }

```

The following command computes the values of `\l_@@_first_line_int` and `\l_@@_last_line_int` when `\PitonInputFile` is used with textual markers.

```

1736 \cs_new_protected:Npn \@@_compute_range:
1737 {

```

We store the markers in L3 strings (`str`) in order to do safely the following replacement of `\#`.

```

1738     \str_set:Nc \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1739     \str_set:Nc \l_tmpb_str { \@@_marker_end:n { \l_@@_end_range_str } }

```

We replace the sequences `\#` which may be present in the prefixes and suffixes added to the markers by the functions `\@@_marker_beginning:n` and `\@@_marker_end:n`.

```

1740     \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1741     \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1742     \lua_now:e
1743     {
1744         piton.ComputeRange
1745         ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1746     }
1747 }

```

2.8 The styles

The following command is fundamental: it will be used by the Lua code.

```

1748 \NewDocumentCommand { \PitonStyle } { m }
1749 {
1750     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1751     { \use:c { pitonStyle _ #1 } }
1752 }

```


The following variant will be rarely used. It applies only a local style and only when that style exists (no error will be raised when the style does not exist). That command will be used in particular for the language “expl”.

```

1753 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1754 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1755 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1756 {
1757   \str_clear_new:N \l_@@_SetPitonStyle_option_str
1758   \str_set:Nx \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1759   \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1760   { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1761   \keys_set:nn { piton / Styles } { #2 }
1762 }

1763 \cs_new_protected:Npn \@@_math_scantokens:n #1
1764 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

1765 \clist_new:N \g_@@_styles_clist
1766 \clist_gset:Nn \g_@@_styles_clist
1767 {
1768   Comment ,
1769   Comment.Internal ,
1770   Comment.LaTeX ,
1771   Discard ,
1772   Exception ,
1773   FormattingType ,
1774   Identifier.Internal ,
1775   Identifier ,
1776   InitialValues ,
1777   Interpol.Inside ,
1778   Keyword ,
1779   Keyword.Governing ,
1780   Keyword.Constant ,
1781   Keyword2 ,
1782   Keyword3 ,
1783   Keyword4 ,
1784   Keyword5 ,
1785   Keyword6 ,
1786   Keyword7 ,
1787   Keyword8 ,
1788   Keyword9 ,
1789   Name.Builtin ,
1790   Name.Class ,
1791   Name.Constructor ,
1792   Name.Decorator ,
1793   Name.Field ,
1794   Name.Function ,
1795   Name.Module ,
1796   Name.Namespace ,
1797   Name.Table ,
1798   Name.Type ,
1799   Number ,
1800   Number.Internal ,
1801   Operator ,
1802   Operator.Word ,
1803   Preproc ,
1804   Prompt ,
1805   String.Doc ,
1806   String.Doc.Internal ,
1807   String.Interpol ,
1808   String.Long ,

```

```

1809   String.Long.Internal ,
1810   String.Short ,
1811   String.Short.Internal ,
1812   Tag ,
1813   TypeParameter ,
1814   UserFunction ,

```

`TypeExpression` is an internal style for expressions which defines types in OCaml.

```

1815   TypeExpression ,

```

Now, specific styles for the languages created with `\NewPitonLanguage` with the syntax of listings.

```

1816   Directive
1817 }

1818 \clist_map_inline:Nn \g_@@_styles_clist
1819 {
1820   \keys_define:nn { piton / Styles }
1821   {
1822     #1 .value_required:n = true ,
1823     #1 .code:n =
1824       \tl_set:cn
1825       {
1826         pitonStyle _
1827         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1828         { \l_@@_SetPitonStyle_option_str _ }
1829         #1
1830       }
1831     { ##1 }
1832   }
1833 }

1834
1835 \keys_define:nn { piton / Styles }
1836 {
1837   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1838   String      .value_required:n = true ,
1839   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1840   Comment.Math .value_required:n = true ,
1841   unknown     .code:n = \@@_unknown_style:
1842 }

```

For the language `expl`, it's possible to create “on the fly” some styles of the form `Module.name` or `Type.name`. For the other languages, it's not possible.

```

1843 \cs_new_protected:Npn \@@_unknown_style:
1844 {
1845   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1846   {
1847     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1848     \seq_get_left:NN \l_tmpa_seq \l_tmpa_str

```

Now, the first part of the key (before the first period) is stored in `\l_tmpa_str`.

```

1849     \bool_lazy_and:nnTF
1850     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1851     {
1852       \str_if_eq_p:Vn \l_tmpa_str { Module }
1853       ||
1854       \str_if_eq_p:Vn \l_tmpa_str { Type }
1855     }

```

Now, we will create a new style.

```

1856     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1857     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1858   }
1859   { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1860 }

```

```

1861 \SetPitonStyle[OCaml]
1862 {
1863   TypeExpression =
1864   {
1865     \SetPitonStyle [ OCaml ]
1866     {
1867       Identifier = \PitonStyle { Name.Type } ,
1868       Name.Builtin = \PitonStyle { Name.Type}
1869     }
1870     \@@_piton:n
1871   }
1872 }

```

We add the word `String` to the list of the styles because we will use that list in the error message for an unknown key in `\SetPitonStyle`.

```

1873 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1874 \clist_gsort:Nn \g_@@_styles_clist
1875 {
1876   \str_compare:nNnTF { #1 } < { #2 }
1877   \sort_return_same:
1878   \sort_return_swapped:
1879 }

```

```

1880 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1881
1882 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1883
1884 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1885 {
1886   \tl_set:Nn \l_tmpa_tl { #1 }

```

We have to begin by a substitution for the spaces. Otherwise, they would be gobbled in the `\tl_map_inline:Nn`.

```

1887   \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1888   \seq_clear:N \l_tmpa_seq
1889   \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1890   \seq_use:Nn \l_tmpa_seq { \- }
1891 }

```

```

1892 \cs_new_protected:Npn \@@_comment:n #1
1893 {
1894   \PitonStyle { Comment }
1895   {
1896     \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1897     {
1898       \tl_set:Nn \l_tmpa_tl { #1 }
1899       \tl_replace_all:NVn \l_tmpa_tl
1900       \c_catcode_other_space_tl
1901       \@@_breakable_space:
1902       \l_tmpa_tl
1903     }
1904     { #1 }
1905   }
1906 }

```

```

1907 \cs_new_protected:Npn \@@_string_long:n #1
1908 {

```

```

1909 \PitonStyle { String.Long }
1910 {
1911     \bool_if:NTF \l_@@_break_strings_anywhere_bool
1912     { \@@_actually_break_anywhere:n { #1 } }
1913     {

```

We have, when `break-lines-in-Piton` is in force, to replace the spaces by `\@@_breakable_space:` because, when we have done a similar job in `\@@_replace_spaces:n` used in `\@@_begin_line:`, that job was not able to do the replacement in the brace group `{...}` of `\PitonStyle{String.Long}{...}` because we used a `\tl_replace_all:Nvn`. At that time, it would have been possible to use a `\tl_regex_replace_all:Nnn` but it is notoriously slow.

```

1914         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1915         {
1916             \tl_set:Nn \l_tmpa_tl { #1 }
1917             \tl_replace_all:Nvn \l_tmpa_tl
1918             \c_catcode_other_space_tl
1919             \@@_breakable_space:
1920             \l_tmpa_tl
1921         }
1922         { #1 }
1923     }
1924 }
1925 }
1926 \cs_new_protected:Npn \@@_string_short:n #1
1927 {
1928     \PitonStyle { String.Short }
1929     {
1930         \bool_if:NT \l_@@_break_strings_anywhere_bool
1931         { \@@_actually_break_anywhere:n }
1932         { #1 }
1933     }
1934 }
1935 \cs_new_protected:Npn \@@_string_doc:n #1
1936 {
1937     \PitonStyle { String.Doc }
1938     {
1939         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1940         {
1941             \tl_set:Nn \l_tmpa_tl { #1 }
1942             \tl_replace_all:Nvn \l_tmpa_tl
1943             \c_catcode_other_space_tl
1944             \@@_breakable_space:
1945             \l_tmpa_tl
1946         }
1947         { #1 }
1948     }
1949 }
1950 \cs_new_protected:Npn \@@_number:n #1
1951 {
1952     \PitonStyle { Number }
1953     {
1954         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1955         { \@@_actually_break_anywhere:n }
1956         { #1 }
1957     }
1958 }

```

2.9 The initial styles

The initial styles are inspired by the style “manni” of Pygments.

```

1959 \SetPitonStyle

```

```

1960 {
1961     Comment                = \color [ HTML ] { 0099FF } \itshape ,
1962     Comment.Internal       = \@@_comment:n ,
1963     Exception              = \color [ HTML ] { CC0000 } ,
1964     Keyword                = \color [ HTML ] { 006699 } \bfseries ,
1965     Keyword.Governing      = \color [ HTML ] { 006699 } \bfseries ,
1966     Keyword.Constant       = \color [ HTML ] { 006699 } \bfseries ,
1967     Name.Builtin           = \color [ HTML ] { 336666 } ,
1968     Name.Decorator         = \color [ HTML ] { 9999FF } ,
1969     Name.Class             = \color [ HTML ] { 00AA88 } \bfseries ,
1970     Name.Function          = \color [ HTML ] { CC00FF } ,
1971     Name.Namespace        = \color [ HTML ] { 00CCFF } ,
1972     Name.Constructor       = \color [ HTML ] { 006000 } \bfseries ,
1973     Name.Field             = \color [ HTML ] { AA6600 } ,
1974     Name.Module            = \color [ HTML ] { 0060A0 } \bfseries ,
1975     Name.Table             = \color [ HTML ] { 309030 } ,
1976     Number                 = \color [ HTML ] { FF6600 } ,
1977     Number.Internal        = \@@_number:n ,
1978     Operator               = \color [ HTML ] { 555555 } ,
1979     Operator.Word          = \bfseries ,
1980     String                 = \color [ HTML ] { CC3300 } ,
1981     String.Long.Internal   = \@@_string_long:n ,
1982     String.Short.Internal  = \@@_string_short:n ,
1983     String.Doc.Internal    = \@@_string_doc:n ,
1984     String.Doc             = \color [ HTML ] { CC3300 } \itshape ,
1985     String.Interpol        = \color [ HTML ] { AA0000 } ,
1986     Comment.LaTeX          = \normalfont \color [ rgb ] { .468, .532, .6 } ,
1987     Name.Type              = \color [ HTML ] { 336666 } ,
1988     InitialValues          = \@@_piton:n ,
1989     Interpol.Inside        = { \l_@@_font_command_tl \@@_piton:n } ,
1990     TypeParameter          = \color [ HTML ] { 336666 } \itshape ,
1991     Preproc                = \color [ HTML ] { AA6600 } \slshape ,

```

We need the command `\@@_identifier:n` because of the command `\SetPitonIdentifier`. The command `\@@_identifier:n` will potentially call the style `Identifier` (which is a user-style, not an internal style).

```

1992     Identifier.Internal    = \@@_identifier:n ,
1993     Identifier             = ,
1994     Directive              = \color [ HTML ] { AA6600 } ,
1995     Tag                    = \colorbox { gray!10 } ,
1996     UserFunction           = \PitonStyle { Identifier } ,
1997     Prompt                 = ,
1998     Discard                = \use_none:n
1999 }

```

2.10 Styles specific to the language expl

```

2000 \clist_new:N \g_@@_expl_styles_clist
2001 \clist_gset:Nn \g_@@_expl_styles_clist
2002 {
2003     Scope.l ,
2004     Scope.g ,
2005     Scope.c
2006 }
2007 \clist_map_inline:Nn \g_@@_expl_styles_clist
2008 {
2009     \keys_define:nn { piton / Styles }
2010     {
2011         #1 .value_required:n = true ,
2012         #1 .code:n =
2013             \tl_set:cn
2014             {

```

```

2015         pitonStyle _
2016         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
2017         { \l_@@_SetPitonStyle_option_str _ }
2018         #1
2019     }
2020     { ##1 }
2021 }
2022 }
2023 \SetPitonStyle [ expl ]
2024 {
2025     Scope.l          = ,
2026     Scope.g          = \bfseries ,
2027     Scope.c          = \slshape ,
2028     Type.bool        = \color [ HTML ] { AA6600 } ,
2029     Type.box         = \color [ HTML ] { 267910 } ,
2030     Type.clist       = \color [ HTML ] { 309030 } ,
2031     Type.fp          = \color [ HTML ] { FF3300 } ,
2032     Type.int         = \color [ HTML ] { FF6600 } ,
2033     Type.seq         = \color [ HTML ] { 309030 } ,
2034     Type.skip        = \color [ HTML ] { OCC060 } ,
2035     Type.str         = \color [ HTML ] { CC3300 } ,
2036     Type.tl          = \color [ HTML ] { AA2200 } ,
2037     Module.bool      = \color [ HTML ] { AA6600 } ,
2038     Module.box       = \color [ HTML ] { 267910 } ,
2039     Module.cs        = \bfseries \color [ HTML ] { 006699 } ,
2040     Module.exp       = \bfseries \color [ HTML ] { 404040 } ,
2041     Module.hbox      = \color [ HTML ] { 267910 } ,
2042     Module.prg       = \bfseries ,
2043     Module.clist     = \color [ HTML ] { 309030 } ,
2044     Module.fp        = \color [ HTML ] { FF3300 } ,
2045     Module.int       = \color [ HTML ] { FF6600 } ,
2046     Module.seq       = \color [ HTML ] { 309030 } ,
2047     Module.skip      = \color [ HTML ] { OCC060 } ,
2048     Module.str       = \color [ HTML ] { CC3300 } ,
2049     Module.tl        = \color [ HTML ] { AA2200 } ,
2050     Module.vbox      = \color [ HTML ] { 267910 }
2051 }

```

If the key `math-comments` has been used in the preamble of the LaTeX document, we change the style `Comment.Math` which should be considered only at an “internal style”. However, maybe we will document in a future version the possibility to write change the style *locally* in a document)).

```

2052 \hook_gput_code:nnn { begindocument } { . }
2053 {
2054     \bool_if:NT \g_@@_math_comments_bool
2055     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
2056 }

```

2.11 Highlighting some identifiers

```

2057 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
2058 {
2059     \clist_set:Nn \l_tmpa_clist { #2 }
2060     \tl_if_novalue:nTF { #1 }
2061     {
2062         \clist_map_inline:Nn \l_tmpa_clist
2063         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
2064     }
2065     {
2066         \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
2067         \str_if_eq:onT \l_tmpa_str { current-language }
2068         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
2069         \clist_map_inline:Nn \l_tmpa_clist

```

```

2070         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
2071     }
2072 }
2073 \cs_new_protected:Npn @@_identifier:n #1
2074 {
2075     \str_set:Nn \l_tmpa_str { #1 }
2076     \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2077     {
2078         \cs_if_exist_use:cF { PitonIdentifier _ \l_tmpa_str }
2079         { \PitonStyle { Identifier } }
2080     }
2081     { #1 }
2082 }

```

In particular, we have an highlighting of the identifiers which are the names of Python functions previously defined by the user. Indeed, when a Python function is defined, the style `Name.Function.Internal` is applied to that name. We define now that style (we define it directly and we short-cut the function `\SetPitonStyle`).

```

2083 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
2084 {

```

First, the element is composed in the TeX flow with the style `Name.Function` which is provided to the end user.

```

2085     { \PitonStyle { Name.Function } { #1 } }
2086     \str_set:Nn \l_tmpa_str { #1 }

```

Now, we specify that the name of the new Python function is a known identifier that will be formatted with the Piton style `UserFunction`. Of course, here the affectation is global because we have to exit many groups and even the environments `{Piton}`.

```

2087     \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ \l_tmpa_str }
2088     { \PitonStyle { UserFunction } }

```

Now, we put the name of that new user function in the dedicated sequence (specific of the current language). **That sequence will be used only by `\PitonClearUserFunctions`.**

```

2089     \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
2090     { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
2091     \seq_gput_right:co { g_@@_functions _ \l_piton_language_str _ seq } \l_tmpa_str

```

We update `g_@@_languages_seq` which is used only by the command `\PitonClearUserFunctions` when it's used without its optional argument.

```

2092     \seq_if_in:Nof \g_@@_languages_seq { \l_piton_language_str }
2093     { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
2094 }

```

```

2095 \NewDocumentCommand \PitonClearUserFunctions { ! o }
2096 {
2097     \tl_if_novalue:nTF { #1 }

```

If the command is used without its optional argument, we will deleted the user language for all the computer languages.

```

2098         { \@@_clear_all_functions: }
2099         { \@@_clear_list_functions:n { #1 } }
2100     }

```

```

2101 \cs_new_protected:Npn @@_clear_list_functions:n #1
2102 {
2103     \clist_set:Nn \l_tmpa_clist { #1 }
2104     \clist_map_function:NN \l_tmpa_clist @@_clear_functions_i:n
2105     \clist_map_inline:nn { #1 }
2106     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2107 }

```

```

2108 \cs_new_protected:Npn @@_clear_functions_i:n #1
2109 { \@@_clear_functions_ii:n { \str_lowercase:n { #1 } } }

```

The following command clears the list of the user-defined functions for the language provided in argument (mandatory in lower case).

```

2110 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2111 {
2112   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2113   {
2114     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2115     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} }
2116     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2117   }
2118 }
2119 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

2120 \cs_new_protected:Npn \@@_clear_functions:n #1
2121 {
2122   \@@_clear_functions_i:n { #1 }
2123   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2124 }

```

The following command clears all the user-defined functions for all the computer languages.

```

2125 \cs_new_protected:Npn \@@_clear_all_functions:
2126 {
2127   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2128   \seq_gclear:N \g_@@_languages_seq
2129 }

```

```

2130 \AtEndDocument
2131 {

```

For the files written on the disk (with the key `write`), all the job is done by Lua.

```

2132   \lua_now:n { piton.write_files_now ( ) }

```

For the files joined in the PDF, we have a modern version which uses the package `pdfmanagement` of LaTeX and a legacy mechanism.

```

2133   \IfPDFManagementActiveTF
2134   { \@@_join_files: }
2135   { \@@_join_files_legacy: }
2136 }

```

If the new package `pdfmanagement` is used, we insert the file directly in the catalog of the PDF file.

```

2137 \cs_new_protected:Npn \@@_join_files:
2138 {
2139   \seq_map_inline:Nn \g_@@_join_seq
2140   {
2141     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2142     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2143     \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2144     {
2145       <<
2146         /Type /Filespec
2147         /UF <\l_tmpa_str>
2148         /EF << /F~\pdf_object_ref_last: >>
2149         /Desc (Computer~listing)
2150         /AFRelationship /Supplement
2151       >>
2152     }
2153   }
2154 }

```

The legacy version of `\@@_join_files:` will be used when the new package `pdfmanagement` is *not* used. In that case, we can't insert the file directly in the catalog of the pdf file. Therefore, we insert the file linked to an annotation in a page of the PDF file. We try to make the annotation itself invisible with several technics.


```

2155 \cs_new_protected:Npn \@@_join_files_legacy:
2156 {
2157   \seq_map_inline:Nn \g_@@_join_seq
2158   {
2159     \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2160     \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2161     \pdfextension annot~width~0pt~height~0pt~depth~0pt

```

The entry /F in the PDF dictionary of the annotation is an unsigned 32-bit integer containing flags specifying various characteristics of the annotation. The bit in position 2 means *Hidden*. However, despite that bit which means *Hidden*, some PDF readers show the annotation. That's why we have used width 0pt height 0pt depth 0pt.

```

2162   {
2163     /Subtype /FileAttachment
2164     /F~2
2165     /Name /Paperclip
2166     /Contents (Computer~listing)
2167     /FS <<
2168       /Type /Filespec

```

We have previously converted the name of the embedded file in utf16/hex with the BOM of big endian and now we can write a PDF string between < and > (with that encoding).

```

2169     /UF <\l_tmpa_str>

```

It would have been possible to write \pdffeedback lastobj~0~R instead \pdf_object_ref_last: since LuaTeX is the only engine allowed by piton. Remark that \pdf_object_ref_last: is in the LaTeX kernel (not in the package pdfmanagement).

```

2170     /EF << /F~\pdf_object_ref_last: >>
2171     /AFRelationship /Supplement
2172   >>
2173 }
2174 }
2175 }

```

2.12 Spaces of indentation

```

2176 \cs_new_protected:Npn \@@_define_leading_space_normal:
2177 {
2178   \cs_set_protected:Npn \@@_leading_space:
2179   {
2180     \int_gincr:N \g_@@_indentation_int

```

Be careful: the \hbox:n is mandatory.

```

2181     \hbox:n { ~ }
2182   }
2183 }

```

```

2184 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2185 {
2186   \cs_set_protected:Npn \@@_leading_space:
2187   {
2188     \int_gincr:N \g_@@_indentation_int
2189     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2190     {
2191       \color { white }
2192       \transparent { 0 }
2193       . % previously : □ U+2423
2194     }
2195     \pdfextension literal { EMC }
2196   }
2197 }
2198 \@@_define_leading_space_Foxit:

```

2.13 Security

```

2199 \AddToHook { env / piton / before }
2200 { \@@_fatal:n { No-environment-piton } }

```

2.14 The error messages of the package

When there is a unknown key, we try a “normal form” of the key and, when that normal form exists, we add that information in the error message.

The normal form is the lower case form of the key, with all the spaces replaced by hyphens (there is never spaces in the keys of piton).

#1 is a clist of names of sets of keys and #2 is the error message to send.

```

2201 \cs_new_protected:Npn \@@_unknown_key:nn #1 #2
2202 {
2203   \str_set_eq:NN \l_tmpa_str \l_keys_key_str
2204   \str_replace_all:Nnn \l_tmpa_str { ~ } { - }
2205   \str_set:Nx \l_tmpa_str { \str_lowercase:f { \l_tmpa_str } }
2206   \bool_set_false:N \l_tmpa_bool
2207   \clist_map_inline:nn { #1 }
2208   {
2209     \keys_if_exist:neT { ##1 } { \l_tmpa_str }
2210     {
2211       \@@_error:n { key~with~normal~form~exists }
2212       \bool_set_true:N \l_tmpa_bool
2213       \clist_map_break:
2214     }
2215   }
2216   \bool_if:NF \l_tmpa_bool { \@@_error:n { #2 } }
2217 }
2218 \@@_msg_new:nn { key~with~normal~form~exists }
2219 {
2220   The~key~'\l_keys_key_str'~does~not~exists.~It~will~be~ignored.\\
2221   Maybe~you~want~to~use~the~key~'\l_tmpa_str'.
2222 }
2223 \@@_msg_new:nn { No-environment-piton }
2224 {
2225   There~is~no~environment-piton!\\
2226   There~is~an~environment~{Piton}~and~a~command~
2227   \token_to_str:N \piton\ but~there~is~no~environment~
2228   {piton}.
2229 }
2230 \@@_msg_new:nn { rounded-corners-without~Tikz }
2231 {
2232   TikZ~not~used \\
2233   You~can't~use~the~key~'rounded-corners'~because~
2234   you~have~not~loaded~the~package~TikZ.~
2235   If~you~go~on,~your~key~will~be~ignored.~
2236   You~won't~have~similar~error~till~the~end~of~the~document.
2237 }
2238 \@@_msg_new:nn { tcolorbox~not~loaded }
2239 {
2240   tcolorbox~not~loaded \\
2241   You~can't~use~the~key~'tcolorbox'~because~
2242   you~have~not~loaded~the~package~tcolorbox.~
2243   Use~\token_to_str:N \usepackage[breakable]{tcolorbox}.~
2244   If~you~go~on,~that~key~will~be~ignored.
2245 }
2246 \@@_msg_new:nn { library~breakable~not~loaded }
2247 {
2248   breakable~not~loaded \\
2249   You~can't~use~the~key~'tcolorbox'~because~
2250   you~have~not~loaded~the~library~'breakable'~of~tcolorbox'.~

```

```

2251 Use~\token_to_str:N \tcbuselibrary{breakable}~in-the~preamble~
2252 of~your~document.~
2253 If~you~go~on,~that~key~will~be~ignored.
2254 }

2255 \@@_msg_new:nn { Language~not~defined }
2256 {
2257   Language~not~defined \\
2258   The~language~'\l_tmpa_tl'~has~not~been~defined~previously.~
2259   If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2260   will~be~ignored.
2261 }

2262 \@@_msg_new:nn { bad~version~of~piton.lua }
2263 {
2264   Bad~number~version~of~'piton.lua'\\
2265   The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2266   version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2267   address~that~issue.
2268 }

2269 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2270 {
2271   Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
2272   The~key~'\l_keys_key_str'~is~unknown.~
2273   This~key~will~be~ignored.
2274 }

2275 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
2276 {
2277   The~style~'\l_keys_key_str'~is~unknown.\\
2278   This~setting~will~be~ignored.~
2279   The~available~styles~are~(in~alphabetic~order):~
2280   \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2281 }

2282 \@@_msg_new:nn { Invalid~key }
2283 {
2284   Wrong~use~of~key.\\
2285   You~can't~use~the~key~'\l_keys_key_str'~here.~
2286   Your~key~will~be~ignored.
2287 }

2288 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2289 {
2290   Unknown~key. \\
2291   The~key~'line~numbers / \l_keys_key_str'~is~unknown.~
2292   The~available~keys~of~the~family~'line~numbers'~are~(in~
2293   alphabetic~order):~
2294   absolute,~false,~label~empty~lines,~position,~resume,~skip~empty~lines,~
2295   sep,~start~and~true.~
2296   Your~key~will~be~ignored.
2297 }

2298 \@@_msg_new:nn { Unknown~key~for~marker }
2299 {
2300   Unknown~key. \\
2301   The~key~'marker / \l_keys_key_str'~is~unknown.~
2302   The~available~keys~of~the~family~'marker'~are~(in~
2303   alphabetic~order):~ beginning,~end~and~include~lines.~
2304   Your~key~will~be~ignored.
2305 }

2306 \@@_msg_new:nn { bad~range~specification }
2307 {
2308   Incompatible~keys.\\
2309   You~can't~specify~the~range~of~lines~to~include~by~using~both~
2310   markers~and~explicit~number~of~lines.~

```

```

2311     Your-whole-file~'\l_@@_file_name_str'~will~be~included.
2312 }
2313 \cs_new_nopar:Nn \@@_thepage:
2314 {
2315     \thepage
2316     \cs_if_exist:NT \insertframenumber
2317     {
2318         ~(frame~\insertframenumber
2319         \cs_if_exist:NT \beamer@slidenumber { ,~slide~\insertslidenumber }
2320         )
2321     }
2322 }

```

We don't give the name `syntax error` for the following error because you should not give a name with a space because such space could be replaced by U+2423 when the key `show-spaces` is in force in the command `\piton`.

```

2323 \@@_msg_new:nn { SyntaxError }
2324 {
2325     Syntax~Error~on~page~\@@_thepage:~\\
2326     Your~code~of~the~language~'\l_piton_language_str'~is~not~
2327     syntactically~correct.~
2328     It~won't~be~printed~in~the~PDF~file.
2329 }
2330 \@@_msg_new:nn { FileError }
2331 {
2332     File~Error.~\\
2333     It's~not~possible~to~write~on~the~file~'#1'~\\
2334     \sys_if_shell_unrestricted:F
2335     { (try~to~compile~with~'lua\latex--shell-escape').\\ }
2336     If~you~go~on,~nothing~will~be~written~on~that~file.
2337 }
2338 \@@_msg_new:nn { InexistentDirectory }
2339 {
2340     Inexistent~directory.~\\
2341     The~directory~'\l_@@_path_write_str'~
2342     given~in~the~key~'path-write'~does~not~exist.~
2343     Nothing~will~be~written~on~'\l_@@_write_str'.
2344 }
2345 \@@_msg_new:nn { begin-marker-not-found }
2346 {
2347     Marker~not~found.~\\
2348     The~range~'\l_@@_begin_range_str'~provided~to~the~
2349     command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2350     The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2351 }
2352 \@@_msg_new:nn { end-marker-not-found }
2353 {
2354     Marker~not~found.~\\
2355     The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2356     provided~to~the~command~\token_to_str:N \PitonInputFile\
2357     has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2358     be~inserted~till~the~end.
2359 }
2360 \@@_msg_new:nn { Unknown~file }
2361 {
2362     Unknown~file.~\\
2363     The~file~'#1'~is~unknown.~
2364     Your~command~\token_to_str:N \PitonInputFile\ will~be~ignored.
2365 }
2366 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2367 {

```

```

2368 \bool_if:NF \g_@@_beamer_bool
2369 { \@@_error_or_warning:n { Without~beamer } }
2370 }
2371 \@@_msg_new:nn { Without~beamer }
2372 {
2373   Key~'\l_keys_key_str'~without~Beamer.\\
2374   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2375   are~not~in~Beamer.~However,~you~can~go~on.
2376 }
2377 \@@_msg_new:nn { rowcolor~in~detected-commands }
2378 {
2379   'rowcolor'~forbidden~in~'detected-commands'.\\
2380   You~should~put~'rowcolor'~in~'raw-detected-commands'.~
2381   That~key~will~be~ignored.
2382 }
2383 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2384 {
2385   Unknown~key. \\
2386   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2387   It~will~be~ignored.~
2388   For~a~list~of~the~available~keys,~type~H~<return>.
2389 }
2390 {
2391   The~available~keys~are~(in~alphabetic~order):~
2392   annotation,~
2393   add-to-split-separation,~
2394   auto-gobble,~
2395   background-color,~
2396   begin-range,~
2397   box,~
2398   break-lines,~
2399   break-lines-in-piton,~
2400   break-lines-in-Piton,~
2401   break-numbers-anywhere,~
2402   break-strings-anywhere,~
2403   continuation-symbol,~
2404   continuation-symbol-on-indentation,~
2405   detected-beamer-commands,~
2406   detected-beamer-environments,~
2407   detected-commands,~
2408   end-of-broken-line,~
2409   end-range,~
2410   env-gobble,~
2411   env-used-by-split,~
2412   font-command(+),~
2413   gobble,~
2414   indent-broken-lines,~
2415   join,~
2416   label-as-zlabel,~
2417   language,~
2418   left-margin,~
2419   line-numbers/,~
2420   marker/,~
2421   math-comments,~
2422   no-join,~
2423   no-write,~
2424   path,~
2425   path-write,~
2426   print,~
2427   prompt-background-color,~
2428   raw-detected-commands,~
2429   resume,~
2430   right-margin,~

```

```

2431 rounded-corners,~
2432 show-spaces,~
2433 show-spaces-in-strings,~
2434 splittable,~
2435 splittable-on-empty-lines,~
2436 split-on-empty-lines,~
2437 split-separation,~
2438 tabs-auto-gobble,~
2439 tab-size,~
2440 tcolorbox,~
2441 varwidth,~
2442 vertical-detected-commands,~
2443 width-and-write.
2444 }

2445 \@@_msg_new:nn { label-with-lines-numbers }
2446 {
2447   You-can't-use-the-command~\token_to_str:N \label\
2448   or~\token_to_str:N \zlabel\ because-the-key~'line-numbers'
2449   ~is-not-active.~If-you-go-on,~that-command-will-ignored.
2450 }

2451 \@@_msg_new:nn { overlay-without-beamer }
2452 {
2453   You-can't-use-an-argument~<...>~for-your-command~
2454   \token_to_str:N \PitonInputFile\ because-you-are-not~
2455   in-Beamer.~If-you-go-on,~that-argument-will-be-ignored.
2456 }

2457 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2458 {
2459   The-key~'label-as-zlabel'~requires-the-package~'zref'.~
2460   Please-load-the-package~'zref'~before-setting-the-key.
2461 }
2462 \hook_gput_code:nnn { begindocument } { . }
2463 {
2464   \bool_if:NT \g_@@_label_as_zlabel_bool
2465   {
2466     \IfPackageLoadedF { zref-base }
2467     { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2468   }
2469 }

```

2.15 We load piton.lua

```

2470 \cs_new_protected:Npn \@@_test_version:n #1
2471 {
2472   \str_if_eq:onF \PitonFileVersion { #1 }
2473   { \@@_error:n { bad-version-of-piton.lua } }
2474 }

2475 \hook_gput_code:nnn { begindocument } { . }
2476 {
2477   \lua_load_module:n { piton }
2478   \lua_now:n
2479   {
2480     tex.sprint ( luatexbase.catcodetables.expl ,
2481                 [[\@@_test_version:n {}] .. piton_version .. "]" )

```

```

2482     }
2483 }
</STY>

```

3 The Lua part of the implementation

The Lua code will be loaded via a `{luacode*}` environment. The environment is by itself a Lua block and the local declarations will be local to that block. All the global functions (used by the L3 parts of the implementation) will be put in a Lua table called `piton`.

```

2484 ⟨*LUA⟩
2485 piton.comment_latex = piton.comment_latex or ">"
2486 piton.comment_latex = "#" .. piton.comment_latex

```

The table `piton.write_files` will contain the contents of all the files that we will write on the disk in the `\AtEndDocument` (if the user has used the key `write-file`). The table `piton.join_files` is similar for the key `join`.

```

2487 piton.write_files = { }
2488 piton.join_files = { }

2489 local sprintL3
2490 function sprintL3 ( s )
2491     tex.sprint ( luatexbase.catcodetables.expl , s )
2492 end

```

3.1 Special functions dealing with LPEG

We will use the Lua library `lpeg` which is built in LuaTeX. That’s why we define first aliases for several functions of that library.

```

2493 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2494 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2495 local B, R = lpeg.B, lpeg.R

```

The following line is mandatory.

```

2496 lpeg.locale(lpeg)

```

3.2 The functions Q, K, WithStyle, etc.

The function `Q` takes in as argument a pattern and returns a LPEG *which does a capture* of the pattern. That capture will be sent to LaTeX with the catcode “other” for all the characters: it’s suitable for elements of the computer listings that `piton` will typeset verbatim (thanks to the catcode “other”).

```

2497 local Q
2498 function Q ( pattern )
2499     return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2500 end

```

The function `L` takes in as argument a pattern and returns a LPEG *which does a capture* of the pattern. That capture will be sent to LaTeX with standard LaTeX catcodes for all the characters: the elements captured will be formatted as normal LaTeX codes. It’s suitable for the “LaTeX comments” in the environments `{Piton}` and the elements between `begin-escape` and `end-escape`. That function won’t be much used.

```

2501 local L
2502 function L ( pattern ) return
2503     Ct ( C ( pattern ) )
2504 end

```

The function `Lc` (the `c` is for *constant*) takes in as argument a string and returns a LPEG *with does a constant capture* which returns that string. The elements captured will be formatted as L3 code. It will be used to send to LaTeX all the formatting LaTeX instructions we have to insert in order to do the syntactic highlighting (that's the main job of `piton`). That function, unlike the previous one, will be widely used.

```

2505 local Lc
2506 function Lc ( string ) return
2507   Cc ( { luatexbase.catcodetables.expl , string } )
2508 end

```

The function `K` creates a LPEG which will return as capture the whole LaTeX code corresponding to a Python chunk (that is to say with the LaTeX formatting instructions corresponding to the syntactic nature of that Python chunk). The first argument is a Lua string corresponding to the name of a `piton` style and the second element is a pattern (that is to say a LPEG without capture)

```

2509 local K
2510 function K ( style , pattern ) return
2511   Lc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] )
2512   * Q ( pattern )
2513   * Lc "}" ]
2514 end

```

The formatting commands in a given `piton` style (eg. the style `Keyword`) may be semi-global declarations (such as `\bfseries` or `\slshape`) or LaTeX macros with an argument (such as `\fbox` or `\colorbox{yellow}`). In order to deal with both syntaxes, we have used two pairs of braces: `{\PitonStyle{Keyword}}{text to format}`.

The following function `WithStyle` is similar to the function `K` but should be used for multi-lines elements.

```

2515 local WithStyle
2516 function WithStyle ( style , pattern ) return
2517   Ct ( Cc "Open" * Cc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] * Cc "}" ] )
2518   * pattern
2519   * Ct ( Cc "Close" )
2520 end

```

The following LPEG catches the Python chunks which are in LaTeX escapes (and that chunks will be considered as normal LaTeX constructions).

```

2521 Escape = P ( false )
2522 EscapeClean = P ( false )
2523 if piton.begin_escape then
2524   Escape =
2525     P ( piton.begin_escape )
2526     * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2527     * P ( piton.end_escape )

```

The LPEG `EscapeClean` will be used in the LPEG `Clean` (and that LPEG is used to “clean” the code by removing the formatting elements).

```

2528 EscapeClean =
2529   P ( piton.begin_escape )
2530   * ( 1 - P ( piton.end_escape ) ) ^ 1
2531   * P ( piton.end_escape )
2532 end

2533 EscapeMath = P ( false )
2534 if piton.begin_escape_math then
2535   EscapeMath =
2536     P ( piton.begin_escape_math )
2537     * Lc "$"
2538     * L ( ( 1 - P ( piton.end_escape_math ) ) ^ 1 )
2539     * Lc "$"
2540     * P ( piton.end_escape_math )
2541 end

```


The basic syntactic LPEG

```
2542 local alpha , digit = lpeg.alpha , lpeg.digit
2543 local space = P " "
```

Remember that, for LPEG, the Unicode characters such as à, â, ç, etc. are in fact strings of length 2 (2 bytes) because lpeg is not Unicode-aware.

```
2544 local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "í"
2545               + "ô" + "û" + "ü" + "Å" + "Ä" + "Ç" + "É" + "È" + "Ê" + "Ë"
2546               + "Ī" + "Ĭ" + "Ō" + "Ū" + "Ū"
2547
2548 local alphanum = letter + digit
```

The following LPEG `identifier` is a mere pattern (that is to say more or less a regular expression) which matches the Python identifiers (hence the name).

```
2549 local identifier = letter * alphanum ^ 0
```

On the other hand, the LPEG `Identifier` (with a capital) also returns a *capture*.

```
2550 local Identifier = K ( 'Identifier.Internal' , identifier )
```

By convention, we will use names with an initial capital for LPEG which return captures.

The following functions allow to recognize numbers that contains `_` among their digits, for example `1_000_000`, but also floating point numbers, numbers with exponents and numbers with different bases.³

```
2551 local allow_underscores_except_first
2552 function allow_underscores_except_first ( p )
2553     return p * (P "_" + p)^0
2554 end
2555 local allow_underscores
2556 function allow_underscores ( p )
2557     return (P "_" + p)^0
2558 end
2559 local digits_to_number
2560 function digits_to_number(prefix, digits)
2561     -- The edge cases of what is allowed in number literals is modelled after
2562     -- OCaml numbers, which seems to be the most permissive language
2563     -- in this regard (among C, OCaml, Python & SQL).
2564     return prefix
2565         * allow_underscores_except_first(digits^1)
2566         * (P "." * #(1 - P ".") * allow_underscores(digits))^~1
2567         * (S "eE" * S "+-~"^-1 * allow_underscores_except_first(digits^1))^~1
2568 end
```

Here is the first use of our function `K`. That function will be used to construct LPEG which capture Python chunks for which we have a dedicated `piton` style. For example, for the numbers, `piton` provides a style which is called `Number`. The name of the style is provided as a Lua string in the second argument of the function `K`. By convention, we use single quotes for delimiting the Lua strings which are names of `piton` styles (but this is only a convention).

```
2569 local Number =
2570     K ( 'Number.Internal' ,
2571         digits_to_number (P "0x" + P "0X", R "af" + R "AF" + digit)
2572         + digits_to_number (P "0o" + P "0O", R "07")
2573         + digits_to_number (P "0b" + P "0B", R "01")
2574         + digits_to_number ( "" , digit )
2575     )
```

³The edge cases such as

We will now define the LPEG Word.

We have a problem in the following LPEG because, obviously, we should adjust the list of symbols with the delimiters of the current language (no?).

```
2576 local lpeg_central = 1 - S " '\r[{}]" - digit
```

We recall that `piton.begin_escape` and `piton.end_escape` are Lua strings corresponding to the keys `begin-escape` and `end-escape`.

```
2577 if piton.begin_escape then
2578   lpeg_central = lpeg_central - piton.begin_escape
2579 end
2580 if piton.begin_escape_math then
2581   lpeg_central = lpeg_central - piton.begin_escape_math
2582 end
2583 local Word = Q ( lpeg_central ^ 1 )
```

```
2584 local Space = Q " " ^ 1
2585 local SkipSpace = Q " " ^ 0
2586
2587 local Punct = Q ( S ".,:;!" )
2588
2589 local Tab = "\t" * Lc [[ \@_tab: ]]
```

```
2590 local LeadingSpace = Lc [[ \@_leading_space: ]] * P " "
```

```
2591 local Delim = Q ( S "[{}]" )
```

The following LPEG catches a space (U+0020) and replaces it by `\l_@@_space_in_string_t1`. It will be used in the strings. Usually, `\l_@@_space_in_string_t1` will contain a space and therefore there won't be any difference. However, when the key `show-spaces-in-strings` is in force, `\l_@@_space_in_string_t1` will contain `␣` (U+2423) in order to visualize the spaces.

```
2592 local SpaceInString = space * Lc [[ \l_@@_space_in_string_t1 ]]
```

3.3 The option 'detected-commands' and al.

We create four Lua tables called `detected_commands`, `raw_detected_commands`, `beamer_commands` and `beamer_environments`.

On the TeX side, the corresponding data have first been stored as clists.

Then, in a `\AtBeginDocument`, they have been converted in “toks registers” of TeX.

Now, on the Lua side, we are able to access to those “toks registers” with the special pseudo-table `tex.toks` of LuaTeX.

Remark that we can safely use `explode('')` to convert such “toks registers” in Lua tables since, in a clist of L3, there is no empty component and, for each component, there is no space on both sides (the `explode` of the Lua of LuaTeX is unable to do itself such purification of the components).

```
2593 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
2594 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
2595 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
2596 local beamer_environments = tex.toks.PitonBeamerEnvironments : explode ( ',' )
```

We will also create some LPEG.

According to our conventions, a LPEG with a name in camelCase is a LPEG which doesn't do any capture.

```
2597 local detectedCommands = P ( false )
2598 for _ , x in ipairs ( detected_commands ) do
2599   detectedCommands = detectedCommands + P ( "\\\" .. x )
2600 end
```

Further, we will have a LPEG called `DetectedCommands` (in `PascalCase`) which will be a LPEG *with* captures.

```

2601 local rawDetectedCommands = P ( false )
2602 for _ , x in ipairs ( raw_detected_commands ) do
2603   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
2604 end
2605
2606 local beamerCommands = P ( false )
2607 for _ , x in ipairs ( beamer_commands ) do
2608   beamerCommands = beamerCommands + P ( "\\\" .. x )
2609 end
2610
2611 local beamerEnvironments = P ( false )
2612 for _ , x in ipairs ( beamer_environments ) do
2613   beamerEnvironments = beamerEnvironments + P ( x )
2614 end

```

Several tools for the construction of the main LPEG

```

2613 local LPEG0 = { }
2614 local LPEG1 = { }
2615 local LPEG2 = { }
2616 local LPEG_cleaner = { }

```

For each language, we will need a pattern to match expressions with balanced braces. Those balanced braces must *not* take into account the braces present in strings of the language. However, the syntax for the strings is language-dependent. That's why we write a Lua function `Compute_braces` which will compute the pattern by taking in as argument a pattern for the strings of the language (at least the shorts strings). The argument of `Compute_braces` must be a pattern *which does no captures*.

```

2617 local Compute_braces
2618 function Compute_braces ( lpeg_string ) return
2619   P { "E" ,
2620     E =
2621     (
2622       "{ " * V "E" * "}"
2623       +
2624       lpeg_string
2625       +
2626       ( 1 - S "{" )
2627     ) ^ 0
2628   }
2629 end

```

The following Lua function will compute the lpeg `DetectedCommands` which is a LPEG with captures.

```

2630 local Compute_DetectedCommands
2631 function Compute_DetectedCommands ( lang , braces ) return
2632   Ct (
2633     Cc "Open"
2634     * C ( detectedCommands * space ^ 0 * P "{" )
2635     * Cc "}"
2636   )
2637   * ( braces
2638     / ( function ( s )
2639         if s ~= '' then return
2640           LPEG1[lang] : match ( s )
2641         end
2642       end )
2643   )
2644   * P "}"
2645   * Ct ( Cc "Close" )
2646 end

```

```

2647 local Compute_RawDetectedCommands
2648 function Compute_RawDetectedCommands ( lang , braces ) return
2649   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2650 end

2651 local Compute_LPEG_cleaner
2652 function Compute_LPEG_cleaner ( lang , braces ) return
2653   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
2654         * ( braces
2655           / ( function ( s )
2656               if s ~= '' then return
2657                 LPEG_cleaner[lang] : match ( s )
2658             end
2659           end )
2660         )
2661         * "}"
2662         + EscapeClean
2663         + C ( P ( 1 ) )
2664         ) ^ 0 ) / table.concat
2665 end

```

The following function `ParseAgain` will be used in the definitions of the LPEG of the different computer languages when we will need to *parse again* a small chunk of code. It's a way to avoid the use of a actual *grammar* of LPEG (in a sens, a recursive regular expression).

Remark that there is no `piton` style associated to a chunk of code which is analyzed by `ParseAgain`. If we wish a `piton` style available to the end user (if he wish to format that element with a uniform font instead of an analyze by `ParseAgain`), we have to use `\@@_piton:n`.

```

2666 local ParseAgain
2667 function ParseAgain ( code )
2668   if code ~= '' then return

```

The variable `piton.language` is set in the function `piton.Parse`.

```

2669   LPEG1[piton.language] : match ( code )
2670 end
2671 end

```

Constructions for Beamer If the class `Beamer` is used, some environments and commands of `Beamer` are automatically detected in the listings of `piton`.

```

2672 local Beamer = P ( false )

```

The following Lua function will be used to compute the LPEG `Beamer` for each computer language. According to our conventions, the LPEG `Beamer`, with its name in PascalCase does captures.

```

2673 local Compute_Beamer
2674 function Compute_Beamer ( lang , braces )

```

We will compute in `lpeg` the LPEG that we will return.

```

2675   local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
2676   lpeg = lpeg +
2677     Ct ( Cc "Open"
2678         * C ( beamerCommands
2679             * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2680             * P "{"
2681           )
2682         * Cc "}"
2683       )
2684     * ( braces /
2685         ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2686     * "}"
2687     * Ct ( Cc "Close" )

```

For the command `\alt`, the specification of the overlays (between angular brackets) is mandatory.

```

2688 lpeg = lpeg +
2689   L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2690   * ( braces /
2691     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2692   * L ( P "}{" )
2693   * ( braces /
2694     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2695   * L ( P "}" )

```

For `\temporal`, the specification of the overlays (between angular brackets) is mandatory.

```

2696 lpeg = lpeg +
2697   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2698   * ( braces
2699     / ( function ( s )
2700       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2701   * L ( P "}{" )
2702   * ( braces
2703     / ( function ( s )
2704       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2705   * L ( P "}{" )
2706   * ( braces
2707     / ( function ( s )
2708       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2709   * L ( P "}" )

```

Now, the environments of Beamer.

```

2710 for _ , x in ipairs ( beamer_environments ) do
2711   lpeg = lpeg +
2712     Ct ( Cc "Open"
2713       * C (
2714         P ( [[\begin{]] .. x .. "]" )
2715         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2716       )
2717       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
2718       * Cc ( [[\end{]] .. x .. "]" )
2719     )
2720   * (

```

We catch all the content of the Beamer environment which a LPEG which is a grammar because there may be nested environments of the same type (added 2025/11/14).

```

2721     (
2722       P { "E" ,
2723         E = (
2724           P ( [[\begin{]] .. x .. "]" )
2725           * V "E"
2726           * P ( [[\end{]] .. x .. "]" )
2727         +
2728         (
2729           1
2730           - P ( [[\begin{]] .. x .. "]" )
2731           - P ( [[\end{]] .. x .. "]" )
2732         )
2733       ) ^ 0
2734     }
2735   )
2736   / ( function ( s )
2737     if s ~= '' then return
2738       LPEG1[lang] : match ( s )
2739     end
2740   end )
2741 )

```

```

2742         * P ( [[\end{}} .. x .. "}" )
2743         * Ct ( Cc "Close" )
2744     end

```

Now, you can return the value we have computed.

```

2745     return lpeg
2746 end

```

The following LPEG is in relation with the key `math-comments`. It will be used in all the languages.

```

2747 local CommentMath =
2748     P "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * P "$" -- $

```

EOL There may be empty lines in the transcription of the prompt, *id est* lines of the form ... without space after and that's why we need `P " " ^ -1` with the `^ -1`.

```

2749 local Prompt =
2750     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
2751     * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

2752 local EOL =
2753     P "\r"
2754     *
2755     (
2756         space ^ 0 * -1
2757         +
2758         Cc "EOL"
2759     )
2760     * ( LeadingSpace ^ 0 * # ( 1 - S " \r" ) ) ^ -1

```

The following LPEG `CommentLaTeX` is for what is called in that document the “LaTeX comments”.

```

2761 local CommentLaTeX =
2762     P ( piton.comment_latex )
2763     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
2764     * L ( ( 1 - P "\r" ) ^ 0 )
2765     * Lc "}}"
2766     * ( EOL + -1 )

```

3.4 The language Python

We open a Lua local scope for the language Python (of course, there will be also global definitions).

```

2767 --python Python
2768 do

```

Some strings of length 2 are explicit because we want the corresponding ligatures available in some fonts such as *Fira Code* to be active.

```

2769 local Operator =
2770     K ( 'Operator' ,
2771         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "*"
2772         + S "--+/*%=<>.&| " )
2773
2774 local OperatorWord =
2775     K ( 'Operator.Word' , P "in" + "is" + "and" + "or" + "not" )

```

The keyword `in` in a construction such as “`for i in range(n)`” must be formatted as a keyword and not as an `Operator.Word` and that’s why we write the following LPEG `For`.

```

2776 local For = K ( 'Keyword' , P "for" )
2777         * Space
2778         * Identifier
2779         * Space
2780         * K ( 'Keyword' , P "in" )
2781
2782 local Keyword =
2783   K ( 'Keyword' ,
2784     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
2785     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
2786     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
2787     "try" + "while" + "with" + "yield" + "yield from" )
2788   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
2789
2790 local Builtin =
2791   K ( 'Name.Builtin' ,
2792     P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
2793     "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
2794     "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
2795     "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
2796     "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
2797     "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
2798     + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
2799     "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
2800     "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
2801     "vars" + "zip" )
2802
2803 local Exception =
2804   K ( 'Exception' ,
2805     P "ArithmeticError" + "AssertionError" + "AttributeError" +
2806     "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
2807     "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
2808     "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
2809     "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
2810     "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
2811     "NotImplementedError" + "OSError" + "OverflowError" +
2812     "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
2813     "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
2814     "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
2815     + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
2816     "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
2817     "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
2818     "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
2819     "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
2820     "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
2821     "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
2822     "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
2823     "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
2824     "RecursionError" )
2825
2826 local RaiseException = K ( 'Keyword' , P "raise" ) * SkipSpace * Exception * Q "("

```

In Python, a “decorator” is a statement whose begins by `@` which patches the function defined in the following statement.

```

2827 local Decorator = K ( 'Name.Decorator' , P "@" * letter ^ 1 )

```

The following LPEG `DefClass` will be used to detect the definition of a new class (the name of that new class will be formatted with the `piton` style `Name.Class`).

Example: `class myclass:`

```

2828 local DefClass =
2829     K ( 'Keyword' , "class" ) * Space * K ( 'Name.Class' , identifier )

```

If the word `class` is not followed by a identifier, it will be caught as keyword by the LPEG `Keyword` (useful if we want to type a list of keywords).

The following LPEG `ImportAs` is used for the lines beginning by `import`. We have to detect the potential keyword `as` because both the name of the module and its alias must be formatted with the `piton` style `Name.Namespace`.

Example: `import numpy as np`

Moreover, after the keyword `import`, it's possible to have a comma-separated list of modules (if the keyword `as` is not used).

Example: `import math, numpy`

```

2830 local ImportAs =
2831     K ( 'Keyword' , "import" )
2832     * Space
2833     * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
2834     * (
2835         ( Space * K ( 'Keyword' , "as" ) * Space
2836           * K ( 'Name.Namespace' , identifier ) )
2837         +
2838         ( SkipSpace * Q "," * SkipSpace
2839           * K ( 'Name.Namespace' , identifier ) ) ^ 0
2840     )

```

Be careful: there is no commutativity of `+` in the previous expression.

The LPEG `FromImport` is used for the lines beginning by `from`. We need a special treatment because the identifier following the keyword `from` must be formatted with the `piton` style `Name.Namespace` and the following keyword `import` must be formatted with the `piton` style `Keyword` and must *not* be caught by the LPEG `ImportAs`.

Example: `from math import pi`

```

2841 local FromImport =
2842     K ( 'Keyword' , "from" )
2843     * Space * K ( 'Name.Namespace' , identifier )
2844     * Space * K ( 'Keyword' , "import" )

```

The strings of Python For the strings in Python, there are four categories of delimiters (without counting the prefixes for f-strings and raw strings). We will use, in the names of our LPEG, prefixes to distinguish the LPEG dealing with that categories of strings, as presented in the following tabular.

	Single	Double
Short	'text'	"text"
Long	'''test'''	"""test"""

We have also to deal with the interpolations in the f-strings. Here is an example of a f-string with an interpolation and a format instruction⁴ in that interpolation:

```
\python{f'Total price: {total+1:.2f} €'}
```

The interpolations beginning by `%` (even though there is more modern techniques now in Python).

```

2845 local PercentInterpol =
2846     K ( 'String.Interpol' ,
2847         P "%"

```

⁴There is no special `piton` style for the formatting instruction (after the colon): the style which will be applied will be the style of the encompassing string, that is to say `String.Short` or `String.Long`.


```

2848 * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2849 * ( S "-#0 +" ) ^ 0
2850 * ( digit ^ 1 + "*" ) ^ -1
2851 * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2852 * ( S "HLL" ) ^ -1
2853 * S "sdfFeExXorgiGauc%"
2854 )

```

We can now define the LPEG for the four kinds of strings. It's not possible to use our function `K` because of the interpolations which must be formatted with another `piton` style that the rest of the string.⁵

```

2855 local SingleShortString =
2856   WithStyle ( 'String.Short.Internal' ,

```

First, we deal with the f-strings of Python, which are prefixed by `f` or `F`.

```

2857   Q ( P "f'" + "F'" )
2858   * (
2859     K ( 'String.Interpol' , "{" )
2860     * K ( 'Interpol.Inside' , ( 1 - S "}'':" ) ^ 0 )
2861     * Q ( P ":" * ( 1 - S "}'':" ) ^ 0 ) ^ -1
2862     * K ( 'String.Interpol' , "}" )
2863     +
2864     SpaceInString
2865     +
2866     Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
2867   ) ^ 0
2868   * Q ""
2869 +

```

Now, we deal with the standard strings of Python, but also the “raw strings”.

```

2870   Q ( P "'" + "r'" + "R'" )
2871   * ( Q ( ( P "\\'" + "\\\\" + 1 - S " 'r%" ) ^ 1 )
2872     + SpaceInString
2873     + PercentInterpol
2874     + Q "%"
2875   ) ^ 0
2876   * Q "" )
2877 local DoubleShortString =
2878   WithStyle ( 'String.Short.Internal' ,
2879     Q ( P "f\\"" + "F\\"" )
2880     * (
2881       K ( 'String.Interpol' , "{" )
2882       * K ( 'Interpol.Inside' , ( 1 - S "}'\':" ) ^ 0 )
2883       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}'\':" ) ^ 0 ) ) ^ -1
2884       * K ( 'String.Interpol' , "}" )
2885       +
2886       SpaceInString
2887       +
2888       Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\"" ) ^ 1 )
2889     ) ^ 0
2890     * Q "\"
2891   +
2892   Q ( P "\\"" + "r\\"" + "R\\"" )
2893   * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"r%" ) ^ 1 )
2894     + SpaceInString
2895     + PercentInterpol
2896     + Q "%"
2897   ) ^ 0
2898   * Q "\" )

```

⁵The interpolations are formatted with the `piton` style `Interpol.Inside`. The initial value of that style is `\\@@_piton:n` which means that the interpolations are parsed once again by `piton`.

```

2899
2900 local ShortString = SingleShortString + DoubleShortString

```

Beamer The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

2901 local braces =
2902   Compute_braces
2903   (
2904     ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
2905     * ( P '\\"' + 1 - S "\"" ) ^ 0 * "\""
2906   +
2907     ( P "'" + "r'" + "R'" + "f'" + "F'" )
2908     * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
2909   )
2910
2911 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

2912 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
2913 + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

2914 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

2915 local SingleLongString =
2916   WithStyle ( 'String.Long.Internal' ,
2917     ( Q ( S "fF" * P "'''" )
2918       * (
2919         K ( 'String.Interpol' , "{" )
2920         * K ( 'Interpol.Inside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
2921         * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
2922         * K ( 'String.Interpol' , "}" )
2923       +
2924         Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
2925       +
2926         EOL
2927     ) ^ 0
2928   +
2929     Q ( ( S "rR" ) ^ -1 * "'''" )
2930     * (
2931       Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
2932       +
2933       PercentInterpol
2934       +
2935       P "%"
2936       +
2937       EOL
2938     ) ^ 0
2939   )
2940   * Q "'''" )

```

```

2941 local DoubleLongString =
2942   WithStyle ( 'String.Long.Internal' ,
2943     (
2944       Q ( S "fF" * "\"\\\"" )
2945       * (
2946         K ( 'String.Interpol', "{ " )
2947         * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\"\\\"" ) ^ 0 )
2948         * Q ( ":" * (1 - S "}:\\r" - "\"\\\"" ) ^ 0 ) ^ -1
2949         * K ( 'String.Interpol' , "}" )
2950         +
2951         Q ( ( 1 - S "{}\\r" - "\"\\\"" ) ^ 1 )
2952         +
2953         EOL
2954       ) ^ 0
2955       +
2956       Q ( S "rR" ^ -1 * "\"\\\"" )
2957       * (
2958         Q ( ( 1 - P "\"\\\"" - S "%\\r" ) ^ 1 )
2959         +
2960         PercentInterpol
2961         +
2962         P "%"
2963         +
2964         EOL
2965       ) ^ 0
2966     )
2967   * Q "\"\\\""
2968 )
2969 local LongString = SingleLongString + DoubleLongString

```

We have a LPEG for the Python docstrings. That LPEG will be used in the LPEG `DefFunction` which deals with the whole preamble of a function definition (which begins with `def`).

```

2970 local StringDoc =
2971   K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
2972   * ( K ( 'String.Doc.Internal' , (1 - P "\"\\\"" - "\\r" ) ^ 0 ) * EOL
2973     * Tab ^ 0
2974   ) ^ 0
2975   * K ( 'String.Doc.Internal' , ( 1 - P "\"\\\"" - "\\r" ) ^ 0 * "\"\\\"" )

```

The comments in the Python listings We define different LPEG dealing with comments in the Python listings.

```

2976 local Comment =
2977   WithStyle
2978   ( 'Comment.Internal' ,
2979     Q "#" * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
2980   )
2981   * ( EOL + -1 )

```

DefFunction The following LPEG **expression** will be used for the parameters in the *argspec* of a Python function. It's necessary to use a *grammar* because that pattern mainly checks the correct nesting of the delimiters (and it's known in the theory of formal languages that this can't be done with regular expressions *stricto sensu* only).

```

2982 local expression =
2983   P { "E" ,
2984     E = ( "'" * ( P "\\'" + 1 - S "'\\r" ) ^ 0 * "'"
2985       + "\\\"" * ( P "\\\"" + 1 - S "\\\"\\r" ) ^ 0 * "\\\""
2986       + "{" * V "F" * "}"
2987       + "(" * V "F" * ")" )

```

```

2988         + "[" * V "F" * "]"
2989         + ( 1 - S "{ } ( [ ] \r , " ) ) ^ 0 ,
2990     F = (   "{" * V "F" * "}"
2991           + "(" * V "F" * ")"
2992           + "[" * V "F" * "]"
2993           + ( 1 - S "{ } ( [ ] \r \'"' " ) ) ^ 0
2994     }

```

We will now define a LPEG Params that will catch the list of parameters (that is to say the *argspec*) in the definition of a Python function. For example, in the line of code

```
def MyFunction(a,b,x=10,n:int): return n
```

the LPEG Params will be used to catch the chunk `a,b,x=10,n:int`.

```

2995     local Params =
2996     P { "E" ,
2997         E = ( V "F" * ( Q " , " * V "F" ) ^ 0 ) ^ -1 ,
2998         F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
2999           * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3000           * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
3001     }

```

The following LPEG DefFunction catches a keyword `def` and the following name of function *but also everything else until a potential docstring*. That's why this definition of LPEG must occur (in the file `piton.sty`) after the definition of several other LPEG such as `Comment`, `CommentLaTeX`, `Params`, `StringDoc`...

```

3002     local DefFunction =
3003     K ( 'Keyword' , "def" )
3004     * Space
3005     * K ( 'Name.Function.Internal' , identifier )
3006     * SkipSpace
3007     * Q "(" * Params * Q ")"
3008     * SkipSpace
3009     * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
3010     * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
3011     * Q ":"
3012     * ( SkipSpace
3013       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
3014       * Tab ^ 0
3015       * SkipSpace
3016       * StringDoc ^ 0 -- there may be additional docstrings
3017     ) ^ -1

```

Remark that, in the previous code, `CommentLaTeX` *must* appear before `Comment`: there is no commutativity of the addition for the *parsing expression grammars* (PEG).

If the word `def` is not followed by an identifier and parenthesis, it will be caught as keyword by the LPEG `Keyword` (useful if, for example, the end user wants to speak of the keyword `def`).

Miscellaneous

```

3018     local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
3019 local EndKeyword
3020     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3021     EscapeMath + -1
```

First, the main loop :

```
3022 local Main =
3023     space ^ 0 * EOL -- faut-il le mettre en commentaire ?
3024     + Space
3025     + Tab
3026     + Escape + EscapeMath
3027     + Beamer
3028     + CommentLaTeX
3029     + DetectedCommands
3030     + Prompt
3031     + LongString
3032     + Comment
3033     + ExceptionInConsole
3034     + Delim
3035     + Operator
3036     + OperatorWord * EndKeyword
3037     + ShortString
3038     + Punct
3039     + FromImport
3040     + RaiseException
3041     + DefFunction
3042     + DefClass
3043     + For
3044     + Keyword * EndKeyword
3045     + Decorator
3046     + Builtin * EndKeyword
3047     + Identifier
3048     + Number
3049     + Word
```

Here, we must not put local, of course.

```
3050 LPEG1.python = Main ^ 0
```

We recall that each line in the Python code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`⁶.

```
3051 LPEG2.python =
3052     Ct (
3053         ( space ^ 0 * "\r" ) ^ -1
3054         * Lc [[ \@@_begin_line: ]]
3055         * LeadingSpace ^ 0
3056         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3057         * -1
3058         * Lc [[ \@@_end_line: ]]
3059     )
```

End of the Lua scope for the language Python.

```
3060 end
```

⁶Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

3.5 The language OCaml

We open a Lua local scope for the language OCaml (of course, there will be also global definitions).

```

3061 --ocaml Ocaml OCaml
3062 do

3063     local SkipSpace = ( Q " " + EOL ) ^ 0
3064     local Space = ( Q " " + EOL ) ^ 1

3065     local braces = Compute_braces ( '\'' * ( 1 - S "\"" ) ^ 0 * '\'' )

3066     if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
3067     DetectedCommands =
3068         Compute_DetectedCommands ( 'ocaml' , braces )
3069         + Compute_RawDetectedCommands ( 'ocaml' , braces )
3070     local Q

```

Usually, the following version of the function Q will be used without the second argument (**strict**), that is to say in a looser way. However, in some circumstances, we will need the “strict” version, for instance in DefFunction.

```

3071     function Q ( pattern, strict )
3072         if strict ~= nil then
3073             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3074         else
3075             return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
3076                 + Beamer + DetectedCommands + EscapeMath + Escape
3077         end
3078     end

3079     local K
3080     function K ( style , pattern, strict ) return
3081         Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
3082         * Q ( pattern, strict )
3083         * Lc "}"
3084     end

3085     local WithStyle
3086     function WithStyle ( style , pattern ) return
3087         Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{ ]] .. style .. "}{" ) * Cc "}" )
3088         * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
3089         * Ct ( Cc "Close" )
3090     end

```

The following LPEG corresponds to the balanced expressions (balanced according to the parenthesis). Of course, we must write (1 - S "(") with outer parenthesis.

```

3091     local balanced_parens =
3092         P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

The strings of OCaml

```

3093     local ocaml_string =
3094         P "\""
3095         * (
3096             P " "
3097             +
3098             P ( ( P '\\\'' + 1 - S " \r" ) ^ 1 )
3099             +
3100             EOL -- ?
3101         ) ^ 0
3102     * P "\""

```

```

3103 local String =
3104   WithStyle
3105     ( 'String.Long.Internal' ,
3106       Q "\""
3107       * (
3108         SpaceInString
3109         +
3110         Q ( ( P '\\\\' + 1 - S "\r" ) ^ 1 )
3111         +
3112         EOL
3113       ) ^ 0
3114       * Q "\""
3115     )

```

Now, the “quoted strings” of OCaml (for example {`ext`|`Essai`|`ext`}).

For those strings, we will do two consecutive analysis. First an analysis to determine the whole string and, then, an analysis for the potential visual spaces and the EOL in the string.

The first analysis require a match-time capture. For explanations about that programmation, see the paragraphe *Lua’s long strings* in www.inf.puc-rio.br/~roberto/lpeg.

```

3116 local ext = ( R "az" + "_" ) ^ 0
3117 local open = "{" * Cg ( ext , 'init' ) * "/"
3118 local close = "/" * C ( ext ) * "}"
3119 local closeeq =
3120   Cmt ( close * Cb ( 'init' ) ,
3121     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3122 local QuotedStringBis =
3123   WithStyle ( 'String.Long.Internal' ,
3124     (
3125       Space
3126       +
3127       Q ( ( 1 - S "\r" ) ^ 1 )
3128       +
3129       EOL
3130     ) ^ 0 )

```

We use a “function capture” (as called in the official documentation of the LPEG) in order to do the second analysis on the result of the first one.

```

3131 local QuotedString =
3132   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3133   ( function ( s ) return QuotedStringBis : match ( s ) end )

```

In OCaml, the delimiters for the comments are (`*` and `*`). There are unsymmetrical and OCaml allows those comments to be nested. That’s why we need a grammar.

In these comments, we embed the math comments (between `$` and `$`) and we embed also a treatment for the end of lines (since the comments may be multi-lines).

```

3134 local comment =
3135   P {
3136     "A" ,
3137     A = Q "(*"
3138       * ( V "A"
3139         + Q ( ( 1 - S "\r$" - "(" - "*" ) ^ 1 ) -- $
3140         + Q ( ocaml_string )
3141         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3142         + EOL
3143       ) ^ 0
3144       * Q "*)"
3145   }
3146 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```
3147 local Delim = Q ( P "[" + "]" + S "[]" )
3148 local Punct = Q ( S ",:;! " )
```

The identifiers caught by `cap_identifier` begin with a capital. In OCaml, it's used for the constructors of types and for the names of the modules.

```
3149 local cap_identifier = R "AZ" * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
```

We consider `::` and `[]` as constructors (of the lists) as does the Tuareg mode of Emacs.

```
3150 local Constructor =
3151   P "::"
```

Don't use `\hspace` instead of `\kern`

```
3152 * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3153 +
3154 P "[]"
3155 * Lc ([[\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3156 K ( 'Name.Constructor' ,
3157     Q "`" ^ -1 * cap_identifier
3158     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
```

```
3159 local ModuleType = K ( 'Name.Type' , cap_identifier )
```

```
3160 local OperatorWord =
3161   K ( 'Operator.Word' ,
3162       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

In OCaml, some keywords are considered as *governing keywords* with some special syntactic characteristics.

```
3163 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3164   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3165   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3166   "struct" + "type" + "val"
```

```
3167 local Keyword =
3168   K ( 'Keyword' ,
3169       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3170       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3171       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3172       + "virtual" + "when" + "while" + "with" )
3173   + K ( 'Keyword.Constant' , P "true" + "false" )
3174   + K ( 'Keyword.Governing' , governing_keyword )
```

```
3175 local EndKeyword
3176   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3177   + EscapeMath + -1
```

Now, the identifier. Recall that we have also a LPEG `cap_identifier` for the identifiers beginning with a capital letter.

```
3178 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3179   - ( OperatorWord + Keyword ) * EndKeyword
```

We have the internal style `Identifier.Internal` in order to be able to implement the mechanism `\SetPitonIdentifier`. The final user has access to a style called `Identifier`.

```
3180 local Identifier = K ( 'Identifier.Internal' , identifier )
```


In OCaml, *character* is a type different of the type `string`.

```

3181 local ocaml_char =
3182   P "" *
3183   (
3184     ( 1 - S "\\\" )
3185     + "\\\"
3186     * ( S "\\ntbr \"
3187         + digit * digit * digit
3188         + P "x" * ( digit + R "af" + R "AF" )
3189                   * ( digit + R "af" + R "AF" )
3190                   * ( digit + R "af" + R "AF" )
3191         + P "o" * R "03" * R "07" * R "07" )
3192   )
3193   * ""
3194 local Char =
3195   K ( 'String.Short.Internal', ocaml_char )

```

For the parameter of the types (for example : ``\a` as in ``a` list).

```

3196 local TypeParameter =
3197   K ( 'TypeParameter' ,
3198       "" * Q "_" ^ -1 * alpha ^ 1 * digit ^ 0 * ( # ( 1 - P "" ) + -1 ) )

```

DotNotation Now, we deal with the notations with points (eg: `List.length`). In OCaml, such notation is used for the fields of the records and for the modules.

```

3199 local DotNotation =
3200   (
3201     K ( 'Name.Module' , cap_identifier )
3202     * Q "."
3203     * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3204     +
3205     Identifier
3206     * Q "."
3207     * K ( 'Name.Field' , identifier )
3208   )
3209   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

The records

```

3210 local expression_for_fields_type =
3211   P { "E" ,
3212       E = ( "{" * V "F" * "}"
3213           + "(" * V "F" * ")"
3214           + TypeParameter
3215           + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
3216       F = ( "{" * V "F" * "}"
3217           + "(" * V "F" * ")"
3218           + ( 1 - S "{}()[]\r\"" ) + TypeParameter ) ^ 0
3219   }

```

```

3220 local expression_for_fields_value =
3221   P { "E" ,
3222       E = ( "{" * V "F" * "}"
3223           + "(" * V "F" * ")"
3224           + "[" * V "F" * "]"
3225           + ocaml_string + ocaml_char
3226           + ( 1 - S "{}()[];" ) ) ^ 0 ,
3227       F = ( "{" * V "F" * "}"
3228           + "(" * V "F" * ")"
3229           + "[" * V "F" * "]"
3230           + ocaml_string + ocaml_char
3231           + ( 1 - S "{}()[]\"" ) ) ^ 0
3232   }

```

```

3233 local OneFieldDefinition =
3234   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3235   * K ( 'Name.Field' , identifier ) * SkipSpace
3236   * Q ":" * SkipSpace
3237   * K ( 'TypeExpression' , expression_for_fields_type )
3238   * SkipSpace

```

```

3239 local OneField =
3240   K ( 'Name.Field' , identifier ) * SkipSpace
3241   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3242   * ( C ( expression_for_fields_value ) / ParseAgain )
3243   * SkipSpace

```

The records.

```

3244 local RecordVal =
3245   Q "{" * SkipSpace
3246   *
3247   (
3248     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3249   ) ^ -1
3250   *
3251   (
3252     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3253   )
3254   * SkipSpace
3255   * Q ";" ^ -1
3256   * SkipSpace
3257   * Comment ^ -1
3258   * SkipSpace
3259   * Q "}"
3260 local RecordType =
3261   Q "{" * SkipSpace
3262   *
3263   (
3264     OneFieldDefinition
3265     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3266   )
3267   * SkipSpace
3268   * Q ";" ^ -1
3269   * SkipSpace
3270   * Comment ^ -1
3271   * SkipSpace
3272   * Q "}"
3273 local Record = RecordType + RecordVal

```

```

3274 local Operator =
3275   P "||" *

```

Don't use \hspace instead of \kern!

```

3276 Lc([{\PitonStyle{Operator}{\kern0.1em/\kern-0.2em/\kern0.1em}}])
3277 +
3278 K ( 'Operator' ,
3279   P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3280   "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3281   + S "--+/*%=<>&@|" )

3282 local Builtin =
3283   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```

```

3284 local Exception =
3285   K ( 'Exception' ,
3286     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3287     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3288     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3289   LPEG_cleaner.ocaml = Compute_LPEG_cleaner ( 'ocaml' , braces )

```

An argument in the definition of a OCaml function may be of the form (pattern:type). pattern may be a single identifier but it's not mandatory. First instance, it's possible to write in OCaml:

```
let head (a::q) = a
```

First, we write a pattern (in the LPEG sens!) to match what will be the pattern (in the OCaml sens).

```

3290 local pattern_part =
3291   ( P "(" * balanced_parens * ")" + ( 1 - S ":" ) + P ":" ) ^ 0

```

For the “type” part, the LPEG-pattern will merely be `balanced_parens`.

We can now write a LPEG `Argument` which catches a argument of function (in the definition of the function).

```
3292 local Argument =
```

The following line is for the labels of the labeled arguments. Maybe we will, in the future, create a style for those elements.

```

3293   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3294   *

```

Now, the argument itself, either a single identifier, or a construction between parentheses

```

3295   (
3296     K ( 'Identifier.Internal' , identifier )
3297     +
3298     Q "(" * SkipSpace
3299     * ( C ( pattern_part ) / ParseAgain )
3300     * SkipSpace

```

Of course, the specification of type is optional.

```

3301     * ( Q ":" * #(1- P"=")
3302         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3303     ) ^ -1
3304     * Q ")"
3305   )

```

Despite its name, the LPEG `DefFunction` deals also with `let open` which opens locally a module.

```

3306 local DefFunction =
3307   K ( 'Keyword.Governing' , "let open" )
3308   * Space
3309   * K ( 'Name.Module' , cap_identifier )
3310   +
3311   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3312   * Space
3313   * K ( 'Name.Function.Internal' , identifier )
3314   * Space
3315   * (

```

We use here the argument `strict` in order to allow a correct analyse of `let x = \uncover<2->{y}` (elsewhere, it's interpreted as a definition of a OCaml function).

```

3316     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3317     +
3318     Argument * ( SkipSpace * Argument ) ^ 0
3319     * (
3320       SkipSpace
3321       * Q ":" * #( 1 - P "=" )
3322       * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3323     ) ^ -1
3324   )

```

DefModule

```

3325 local DefModule =
3326   K ( 'Keyword.Governing' , "module" ) * Space
3327   *
3328   (
3329     K ( 'Keyword.Governing' , "type" ) * Space
3330     * K ( 'Name.Type' , cap_identifier )
3331     +
3332     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3333     *
3334     (
3335       Q "(" * SkipSpace
3336       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3337       * Q ":" * # ( 1 - P "=" ) * SkipSpace
3338       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3339       *
3340       (
3341         Q "," * SkipSpace
3342         * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3343         * Q ":" * # ( 1 - P "=" ) * SkipSpace
3344         * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3345       ) ^ 0
3346       * Q ")"
3347     ) ^ -1
3348     *
3349     (
3350       Q "=" * SkipSpace
3351       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3352       * Q "("
3353       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3354       *
3355       (
3356         Q ","
3357         *
3358         K ( 'Name.Module' , cap_identifier ) * SkipSpace
3359       ) ^ 0
3360       * Q ")"
3361     ) ^ -1
3362   )
3363   +
3364   K ( 'Keyword.Governing' , P "include" + "open" )
3365   * Space
3366   * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3367 local DefType =
3368   K ( 'Keyword.Governing' , "type" )
3369   * Space
3370   * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3371   * SkipSpace
3372   * ( Q "+=" + Q "=" )
3373   * SkipSpace
3374   * (
3375     RecordType
3376     +

```

The following lines are a suggestion of Y. Salmon.

```

3377   WithStyle
3378   (
3379     'TypeExpression' ,
3380     (
3381       (
3382         EOL

```

```

3383         + comment
3384         + Q ( 1
3385             - P ";;"
3386             - P "type"
3387             - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3388         )
3389     ) ^ 0
3390     *
3391     (
3392         # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3393         + Q ";;"
3394         + -1
3395     )
3396 )
3397 )
3398 )

3399 local prompt =
3400     Q "utop[" * digit^1 * Q "> "
3401 local start_of_line = P(function(subject, position)
3402     if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3403         return position
3404     end
3405     return nil
3406 end)
3407 local Prompt = #start_of_line * K( 'Prompt', prompt )
3408 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3409                 * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3410                 * (K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3411 local Main =
3412     space ^ 0 * EOL
3413     + Space
3414     + Tab
3415     + Escape + EscapeMath
3416     + Beamer
3417     + DetectedCommands
3418     + TypeParameter
3419     + String + QuotedString + Char
3420     + Comment
3421     + Prompt + Answer

```

For the labels (maybe we will write in the future a dedicated LPEG pour those tokens).

```

3422     + Q "~" * Identifier * ( Q ":" ) ^ -1
3423     + Q ":" * # (1 - P ":" ) * SkipSpace
3424         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3425     + Exception
3426     + DefType
3427     + DefFunction
3428     + DefModule
3429     + Record
3430     + Keyword * EndKeyword
3431     + OperatorWord * EndKeyword
3432     + Builtin * EndKeyword
3433     + DotNotation * EndKeyword
3434     + Constructor
3435     + Identifier
3436     + Punct
3437     + Delim -- Delim is before Operator for a correct analysis of [| et |]
3438     + Operator

```

```

3439     + Number
3440     + Word

```

Here, we must not put `local`, of course.

```

3441   LPEG1.ocaml = Main ^ 0

```

```

3442   LPEG2.ocaml =
3443   Ct (

```

The following lines are in order to allow, in `\piton` (and not in `{Piton}`), judgments of type (such as `f : my_type -> 'a list`) or single expressions of type such as `my_type -> 'a list` (in that case, the argument of `\piton` *must* begin by a colon).

```

3444       (
3445       (
3446         P ":"
3447         +
3448         (
3449           ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3450           * Identifier
3451           * SkipSpace
3452           * Q ":"
3453         )
3454       )
3455       * # ( 1 - S ":@" )
3456       * SkipSpace
3457       * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 ) * -1
3458     )
3459   +
3460   (
3461     ( space ^ 0 * "\r" ) ^ -1
3462     * Lc [[ \@@_begin_line: ]]
3463     * LeadingSpace ^ 0
3464     * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3465       + space ^ 0 * EOL
3466       + Main
3467     ) ^ 0
3468     * -1
3469     * Lc [[ \@@_end_line: ]]
3470   )
3471 )

```

End of the Lua scope for the language OCaml.

```

3472 end

```

3.6 The language C

We open a Lua local scope for the language C (of course, there will be also global definitions).

```

3473 --c C c++ C++
3474 do

```

```

3475   local Delim = Q ( S "{[()]}")
3476   local Punct = Q ( S ",:;!")

```

Some strings of length 2 are explicit because we want the corresponding ligatures available in some fonts such as *Fira Code* to be active.

```

3477   local identifier = letter * alphanum ^ 0
3478
3479   local Operator =

```

```

3480 K ( 'Operator' ,
3481     P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3482     + S "~+/*%=<>&.@|!" )
3483
3484 local Keyword =
3485     K ( 'Keyword' ,
3486         P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3487         "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3488         "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3489         "register" + "restricted" + "return" + "static" + "static_assert" +
3490         "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3491         "union" + "using" + "virtual" + "volatile" + "while"
3492     )
3493     + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3494
3495 local Builtin =
3496     K ( 'Name.Builtin' ,
3497         P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3498
3499 local Type =
3500     K ( 'Name.Type' ,
3501         P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3502         "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3503         "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3504         "void" + "wchar_t" ) * Q "*" ^ 0
3505
3506 local DefFunction =
3507     Type
3508     * Space
3509     * Q "*" ^ -1
3510     * K ( 'Name.Function.Internal' , identifier )
3511     * SkipSpace
3512     * # P "("

```

We remind that the marker # of LPEG specifies that the pattern will be detected but won't consume any character.

The following LPEG DefClass will be used to detect the definition of a new class (the name of that new class will be formatted with the piton style Name.Class).

Example: `class myclass:`

```

3513 local DefClass =
3514     K ( 'Keyword' , "class" ) * Space * K ( 'Name.Class' , identifier )

```

If the word `class` is not followed by a identifier, it will be caught as keyword by the LPEG Keyword (useful if we want to type a list of keywords).

```

3515 local Character =
3516     K ( 'String.Short' ,
3517         P "[\''"] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

The strings of C

```

3518 String =
3519     WithStyle ( 'String.Long.Internal' ,
3520         Q "\"\"
3521         * ( SpaceInString
3522             + K ( 'String.Interpol' ,
3523                 "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3524             )
3525             + Q ( ( P "\\\"\" + 1 - S " \"\" ) ^ 1 )
3526             ) ^ 0
3527         * Q "\"\"
3528     )

```

Beamer The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3529 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3530 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end

3531 DetectedCommands =
3532   Compute_DetectedCommands ( 'c' , braces )
3533   + Compute_RawDetectedCommands ( 'c' , braces )

3534 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

3535 local Preproc = K ( 'Preproc' , "#" * ( 1 - P "r" ) ^ 0 ) * ( EOL + -1 )

```

The comments in the C listings We define different LPEG dealing with comments in the C listings.

```

3536 local Comment =
3537   WithStyle ( 'Comment.Internal' ,
3538     Q "/" * ( CommentMath + Q ( ( 1 - S "$r" ) ^ 1 ) ) ^ 0 ) -- $
3539     * ( EOL + -1 )
3540
3541 local LongComment =
3542   WithStyle ( 'Comment.Internal' ,
3543     Q "/*"
3544     * ( CommentMath + Q ( ( 1 - P "*/" - S "$r" ) ^ 1 ) + EOL ) ^ 0
3545     * Q "*/"
3546     ) -- $

```

The main LPEG for the language C

```

3547 local EndKeyword
3548   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3549     EscapeMath + -1

```

First, the main loop :

```

3550 local Main =
3551   space ^ 0 * EOL
3552   + Space
3553   + Tab
3554   + Escape + EscapeMath
3555   + CommentLaTeX
3556   + Beamer
3557   + DetectedCommands
3558   + Preproc
3559   + Comment + LongComment
3560   + Delim
3561   + Operator
3562   + Character
3563   + String
3564   + Punct
3565   + DefFunction
3566   + DefClass
3567   + Type * ( Q "*" ^ -1 + EndKeyword )
3568   + Keyword * EndKeyword
3569   + Builtin * EndKeyword
3570   + Identifier
3571   + Number
3572   + Word

```


Here, we must not put `local`, of course.

```
3573   LPEG1.c = Main ^ 0
```

We recall that each line in the C code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`⁷.

```
3574   LPEG2.c =
3575   Ct (
3576       ( space ^ 0 * P "\r" ) ^ -1
3577       * Lc [[ \@@_begin_line: ]]
3578       * LeadingSpace ^ 0
3579       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3580       * -1
3581       * Lc [[ \@@_end_line: ]]
3582   )
```

End of the Lua scope for the language C.

```
3583 end
```

3.7 The language SQL

We open a Lua local scope for the language SQL (of course, there will be also global definitions).

```
3584 --sql SQL
3585 do

3586   local LuaKeyword
3587   function LuaKeyword ( name ) return
3588       Lc [[ {\PitonStyle{Keyword}{ }}
3589       * Q ( Cmt (
3590           C ( letter * alphanum ^ 0 ) ,
3591           function ( _ , _ , a ) return a : upper ( ) == name end
3592       )
3593   )
3594   * Lc "}"
3595 end
```

In the identifiers, we will be able to catch those contening spaces, that is to say like "last name".

```
3596   local identifier =
3597       letter * ( alphanum + "-" ) ^ 0
3598       + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"
3599   local Operator =
3600       K ( 'Operator' , P "=" + "!=" + "<>" + ">=" + ">" + "<=" + "<" + S "*/" )
```

In SQL, the keywords are case-insensitive. That's why we have a little complication. We will catch the keywords with the identifiers and, then, distinguish the keywords with a Lua function. However, some keywords will be caught in special LPEG because we want to detect the names of the SQL tables.

The following function converts a comma-separated list in a “set”, that is to say a Lua table with a fast way to test whether a string belongs to that set (eventually, the indexation of the components of the table is no longer done by integers but by the strings themselves).

```
3601   local Set
3602   function Set ( list )
3603       local set = { }
3604       for _ , l in ipairs ( list ) do set[l] = true end
3605       return set
3606   end
```

⁷Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

We now use the previous function `Set` to create the “sets” `set_keywords` and `set_builtin`. That list of keywords comes from https://sqlite.org/lang_keywords.html.

```

3607 local set_keywords = Set
3608 {
3609     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3610     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3611     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3612     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3613     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3614     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3615     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3616     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3617     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3618     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3619     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3620     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3621     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3622     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3623     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3624     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3625     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3626     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3627     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3628     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3629 }
3630
3630 local set_builtins = Set
3631 {
3632     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3633     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3634     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3635 }

```

The LPEG Identifier will catch the identifiers of the fields but also the keywords and the built-in functions of SQL. It will *not* catch the names of the SQL tables.

```

3636 local Identifier =
3637   C ( identifier ) /
3638   (
3639     function ( s )
3640       if set_keywords [ s : upper ( ) ] then return

```

Remind that, in Lua, it's possible to return *several* values.

```

3641     { [[{\PitonStyle{Keyword}{}}] } ,
3642     { luatexbase.catcodetables.other , s } ,
3643     { "}" } }
3644   else
3645     if set_builtins [ s : upper ( ) ] then return
3646     { [[{\PitonStyle{Name.Builtin}{}}] } ,
3647     { luatexbase.catcodetables.other , s } ,
3648     { "}" } }
3649   else return
3650     { [[{\PitonStyle{Name.Field}{}}] } ,
3651     { luatexbase.catcodetables.other , s } ,
3652     { "}" } }
3653   end
3654 end
3655 end
3656 )

```

The strings of SQL

```

3657 local String = K ( 'String.Long.Internal' , "'" * ( 1 - P "'" ) ^ 1 * "'" )

```

Beamer The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3658 local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
3659 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end

3660 DetectedCommands =
3661   Compute_DetectedCommands ( 'sql' , braces )
3662   + Compute_RawDetectedCommands ( 'sql' , braces )
3663   LPEG_cleaner.sql = Compute_LPEG_cleaner ( 'sql' , braces )

```

The comments in the SQL listings We define different LPEG dealing with comments in the SQL listings.

```

3664 local Comment =
3665   WithStyle ( 'Comment.Internal' ,
3666     Q "--" -- syntax of SQL92
3667     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3668     * ( EOL + -1 )
3669
3670 local LongComment =
3671   WithStyle ( 'Comment.Internal' ,
3672     Q "/*"
3673     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3674     * Q "*/"
3675     ) -- $

```

The main LPEG for the language SQL

```

3676 local EndKeyword
3677   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3678     EscapeMath + -1
3679
3680 local TableField =
3681   K ( 'Name.Table' , identifier )
3682   * Q "."
3683   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
3684
3685 local OneField =
3686   (
3687     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
3688     +
3689     K ( 'Name.Table' , identifier )
3690     * Q "."
3691     * K ( 'Name.Field' , identifier )
3692     +
3693     K ( 'Name.Field' , identifier )
3694   )
3695   * (
3696     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
3697     ) ^ -1
3698   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
3699
3700 local OneTable =
3701   K ( 'Name.Table' , identifier )
3702   * (
3703     Space
3704     * LuaKeyword "AS"
3705     * Space
3706     * K ( 'Name.Table' , identifier )
3707   ) ^ -1
3708
3709 local WeCatchTableNames =

```

```

3709     LuaKeyword "FROM"
3710     * ( Space + EOL )
3711     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
3712     + (
3713         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
3714         + LuaKeyword "TABLE"
3715     )
3716     * ( Space + EOL ) * OneTable
3717     local EndKeyword
3718     = Space + Punct + Delim + EOL + Beamer
3719     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

3720     local Main =
3721         space ^ 0 * EOL
3722         + Space
3723         + Tab
3724         + Escape + EscapeMath
3725         + CommentLaTeX
3726         + Beamer
3727         + DetectedCommands
3728         + Comment + LongComment
3729         + Delim
3730         + Operator
3731         + String
3732         + Punct
3733         + WeCatchTableNames
3734         + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
3735         + Number
3736         + Word

```

Here, we must not put local, of course.

```

3737     LPEG1.sql = Main ^ 0

```

We recall that each line in the code to parse will be sent back to LaTeX between a pair `\@@_begin_line: - \@@_end_line:`⁸.

```

3738     LPEG2.sql =
3739         Ct (
3740             ( space ^ 0 * "\r" ) ^ -1
3741             * Lc [[ \@@_begin_line: ]]
3742             * LeadingSpace ^ 0
3743             * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3744             * -1
3745             * Lc [[ \@@_end_line: ]]
3746         )

```

End of the Lua scope for the language SQL.

```

3747     end

```

3.8 The language “Minimal”

We open a Lua local scope for the language “Minimal” (of course, there will be also global definitions).

```

3748     --minimal Minimal
3749     do

```

⁸Remember that the `\@@_end_line:` must be explicit because it will be used as marker in order to delimit the argument of the command `\@@_begin_line:`

```

3750 local Punct = Q ( S ",:;!\" )
3751
3752 local Comment =
3753   WithStyle ( 'Comment.Internal' ,
3754     Q "\""
3755     * ( CommentMath + Q ( ( 1 - S "$\" ) ^ 1 ) ) ^ 0 -- $
3756     )
3757     * ( EOL + -1 )
3758
3759 local String =
3760   WithStyle ( 'String.Short.Internal' ,
3761     Q "\"
3762     * ( SpaceInString
3763       + Q ( ( P "[\" ] + 1 - S \" \" ) ^ 1 )
3764       ) ^ 0
3765     * Q "\"
3766     )

```

The argument of `Compute_braces` must be a pattern *which does no catching* corresponding to the strings of the language.

```

3767 local braces = Compute_braces ( P "\" * ( P "\\\" + 1 - P \" \" ) ^ 1 * \" \" )
3768
3769 if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
3770
3771 DetectedCommands =
3772   Compute_DetectedCommands ( 'minimal' , braces )
3773   + Compute_RawDetectedCommands ( 'minimal' , braces )
3774
3775 LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
3776
3777 local identifier = letter * alphanum ^ 0
3778
3779 local Identifier = K ( 'Identifier.Internal' , identifier )
3780
3781 local Delim = Q ( S "{[()]}" )
3782
3783 local Main =
3784   space ^ 0 * EOL
3785   + Space
3786   + Tab
3787   + Escape + EscapeMath
3788   + CommentLaTeX
3789   + Beamer
3790   + DetectedCommands
3791   + Comment
3792   + Delim
3793   + String
3794   + Punct
3795   + Identifier
3796   + Number
3797   + Word

```

Here, we must not put `local`, of course.

```

3798 LPEG1.minimal = Main ^ 0
3799
3800 LPEG2.minimal =
3801   Ct (
3802     ( space ^ 0 * \"\" ) ^ -1
3803     * Lc [ [ @@_begin_line: ] ]
3804     * LeadingSpace ^ 0
3805     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3806     * -1

```

```

3807         * Lc [[ \@@_end_line: ]]
3808     )

```

End of the Lua scope for the language “Minimal”.

```

3809 end

```

3.9 The language “Verbatim”

We open a Lua local scope for the language “Verbatim” (of course, there will be also global definitions).

```

3810 --verbatim Verbatim
3811 do

```

Here, we don’t use `braces` as done with the other languages because we don’t have to take into account the strings (there is no string in the language “Verbatim”).

```

3812     local braces =
3813         P { "E" ,
3814             E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3815         }
3816
3817     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
3818
3819     DetectedCommands =
3820         Compute_DetectedCommands ( 'verbatim' , braces )
3821         + Compute_RawDetectedCommands ( 'verbatim' , braces )
3822
3823     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

3824     local lpeg_central = 1 - S " \\r"
3825     if piton.begin_escape then
3826         lpeg_central = lpeg_central - piton.begin_escape
3827     end
3828     if piton.begin_escape_math then
3829         lpeg_central = lpeg_central - piton.begin_escape_math
3830     end
3831     local Word = Q ( lpeg_central ^ 1 )
3832
3833     local Main =
3834         space ^ 0 * EOL
3835         + Space
3836         + Tab
3837         + Escape + EscapeMath
3838         + Beamer
3839         + DetectedCommands
3840         + Q [[\]]
3841         + Word

```

Here, we must not put `local`, of course.

```

3842     LPEG1.verbatim = Main ^ 0
3843
3844     LPEG2.verbatim =
3845         Ct (
3846             ( space ^ 0 * "\r" ) ^ -1
3847             * Lc [[ \@@_begin_line: ]]
3848             * LeadingSpace ^ 0
3849             * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3850             * -1
3851             * Lc [[ \@@_end_line: ]]
3852         )

```

End of the Lua scope for the language “verbatim”.

```

3853 end

```

3.10 The language expl

We open a Lua local scope for the language `expl` of LaTeX3 (of course, there will be also global definitions).

```

3854 --EXPL expl
3855 do
3856   local Comment =
3857     WithStyle
3858     ( 'Comment.Internal' ,
3859       Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3860     )
3861     * ( EOL + -1 )

```

First, we begin with a special function to analyse the “keywords”, that is to say the control sequences beginning by “\”.

```

3862   local analyze_cs
3863   function analyze_cs ( s )
3864     local i = s : find ( ":" )
3865     if i then

```

First, the case of what might be called a “function” in `expl`, for instance, `\tl_set:Nn` or `\int_compare:nNnTF`.

```

3866       local name = s : sub ( 2 , i - 1 )
3867       local parts = name : explode ( "_" )
3868       local module = parts[1]
3869       if module == "" then module = parts[3] end

```

Remind that, in Lua, we can return *several* values.

```

3870       return
3871       { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3872         { luatexbase.catcodetables.other , s } ,
3873         { "}" } }
3874     else
3875       local p = s : sub ( 1 , 3 )
3876       if p == [[\l_]] or p == [[\g_]] or p == [[\c_]] then

```

The case of what might be called a “variable”, for instance, `\l_tmpa_int` or `\g__module_text_tl`.

```

3877       local scope = s : sub(2,2)
3878       local parts = s : explode ( "_" )
3879       local module = parts[2]
3880       if module == "" then module = parts[3] end
3881       local type = parts[#parts]
3882       return
3883       { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
3884         { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3885         { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
3886         { luatexbase.catcodetables.other , s } ,
3887         { "}}}}"} }
3888     else

```

We have a control sequence which is neither a “function” neither a “variable” of `expl`. It’s a control sequence of standard LaTeX and we don’t format it.

```

3889       return { luatexbase.catcodetables.other , s }
3890     end
3891   end
3892 end

```

Here, we don’t use `braces` as done with the other languages because we don’t have to take into account the strings (there is no string in the language `expl`).

```

3893   local braces =
3894     P { "E" ,
3895       E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3896     }

```

```

3897
3898 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
3899
3900 DetectedCommands =
3901   Compute_DetectedCommands ( 'expl' , braces )
3902   + Compute_RawDetectedCommands ( 'expl' , braces )
3903
3904 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
3905 local control_sequence = P "\\\" * ( R "Az" + "_" + ":" + "@" ) ^ 1
3906 local ControlSequence = C ( control_sequence ) / analyze_cs
3907
3908 local def_function
3909   = P [[\cs_]]
3910   * ( P "set" + "new" )
3911   * ( P "_protected" ) ^ -1
3912   * P ":N" * ( P "p" ) ^ -1 * "n"
3913
3914 local DefFunction =
3915   C ( def_function ) / analyze_cs
3916   * Space
3917   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
3918   * ControlSequence -- Q ( ControlSequence ) ?
3919   * Lc "}"
3920
3921 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
3922
3923 local Main =
3924   space ^ 0 * EOL
3925   + Space
3926   + Tab
3927   + Escape + EscapeMath
3928   + Beamer
3929   + Comment
3930   + DetectedCommands
3931   + DefFunction
3932   + ControlSequence
3933   + Word

```

Here, we must not put local, of course.

```

3931 LPEG1.expl = Main ^ 0
3932
3933 LPEG2.expl =
3934   Ct (
3935     ( space ^ 0 * "\r" ) ^ -1
3936     * Lc [[ \@@_begin_line: ]]
3937     * LeadingSpace ^ 0
3938     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3939     * -1
3940     * Lc [[ \@@_end_line: ]]
3941   )

```

End of the Lua scope for the language expl of LaTeX3.

```

3942 end

```

3.11 The function Parse

The function `Parse` is the main function of the package `piton`. It parses its argument and sends back to LaTeX the code with interlaced formatting LaTeX instructions. In fact, everything is done by the LPEG corresponding to the considered language (`LPEG2[language]`) which returns as capture a Lua table containing data to send to LaTeX.

```

3943 function piton.Parse ( language , code )

```


The variable `piton.language` will be used by the function `ParseAgain`.

```

3944   piton.language = language
3945   local t = LPEG2[language] : match ( code )
3946   if not t then
3947       sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
3948       return -- to exit in force the function
3949   end
3950   local left_stack = {}
3951   local right_stack = {}
3952   for _ , one_item in ipairs ( t ) do
3953       if one_item == "EOL" then
3954           for i = #right_stack, 1, -1 do
3955               tex.sprint ( right_stack[i] )
3956           end

```

We remind that the `\@@_end_line:` must be explicit since it's the marker of end of the command `\@@_begin_line:`.

```

3957       sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
3958       tex.sprint ( table.concat ( left_stack ) )
3959   else

```

Here is an example of an item beginning with "Open".

```
{ "Open" , "\begin{uncoverenv}<2>" , "\end{uncoverenv}" }
```

In order to deal with the ends of lines, we have to close the environment (`{uncoverenv}` in this example) at the end of each line and reopen it at the beginning of the new line. That's why we use two Lua stacks, called `left_stack` and `right_stack`. `left_stack` will be for the elements like `\begin{uncoverenv}<2>` and `right_stack` will be for the elements like `\end{uncoverenv}`.

```

3960       if one_item[1] == "Open" then
3961           tex.sprint ( one_item[2] )
3962           table.insert ( left_stack , one_item[2] )
3963           table.insert ( right_stack , one_item[3] )
3964       else
3965           if one_item[1] == "Close" then
3966               tex.sprint ( right_stack[#right_stack] )
3967               left_stack[#left_stack] = nil
3968               right_stack[#right_stack] = nil
3969           else
3970               tex.tprint ( one_item )
3971           end
3972       end
3973   end
3974 end
3975 end

```

There is the problem of the conventions of end of lines (`\n` in Unix and Linux but `\r\n` in Windows). The function `my_file_lines` will read a file line by line after replacement of the potential `\r\n` by `\n` (that means that we go the convention UNIX).

```

3976 local my_file_lines
3977 function my_file_lines ( filename )
3978     local f = io.open ( filename , 'rb' )
3979     local s = f : read ( '*a' )
3980     f : close ( )

```

À la fin, on doit bien mettre `(.-)` et pas `(.*)`.

```

3981     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
3982 end

```

Recall that, in Lua, `gmatch` returns an *iterator*.

```

3983 function piton.ReadFile ( name , first_line , last_line )
3984     local s = ''
3985     local i = 0
3986     for line in my_file_lines ( name ) do

```

```

3987     i = i + 1
3988     if i >= first_line then
3989         s = s .. '\r' .. line
3990     end
3991     if i >= last_line then break end
3992 end

```

We extract the BOM of utf-8, if present.

```

3993     if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
3994         s = s : sub ( 5 , -1 )
3995     end
3996     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
3997     tex.sprint ( luatexbase.catcodetables.other , s )
3998     sprintL3 ( "}" )
3999 end

4000 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
4001     local s
4002     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4003     piton.GobbleParse ( lang , n , splittable , s )
4004 end

```

3.12 Two variants of the function Parse with integrated preprocessors

The following command will be used by the user command `\piton`. For that command, we have to undo the duplication of the symbols `#`.

```

4005 function piton.ParseBis ( lang , code )
4006     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
4007 end

```

Of course, `gsub` spans the string only once for the substitutions, which means that `####` will be replaced by `##` as expected and not by `#`.

The following command will be used when we have to parse some small chunks of code that have yet been parsed. They are re-scanned by LaTeX because it has been required by `\@@_piton:n` in the `piton` style of the syntactic element. In that case, you have to remove the potential `\@@_breakable_space:` that have been inserted when the key `break-lines` is in force.

```

4008 function piton.ParseTer ( lang , code )

```

Be careful: we have to write `[[\@@_breakable_space:]]` with a space after the name of the LaTeX command `\@@_breakable_space:`.

```

4009     return piton.Parse
4010     (
4011         lang ,
4012         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
4013     )
4014 end

```

3.13 Preprocessors of the function Parse for gobble

We deal now with preprocessors of the function `Parse` which are needed when the “gobble mechanism” is used.

The following LPEG returns as capture the minimal number of spaces at the beginning of the lines of code.

```

4015 local AutoGobbleLPEG =
4016     ( (
4017         P " " ^ 0 * "\r"

```

```

4018      +
4019      Ct ( C " " ^ 0 ) / table.getn
4020      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
4021    ) ^ 0
4022    * ( Ct ( C " " ^ 0 ) / table.getn
4023      * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4024  ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

4025 local TabsAutoGobbleLPEG =
4026   (
4027     (
4028       P "\t" ^ 0 * "\r"
4029       +
4030       Ct ( C "\t" ^ 0 ) / table.getn
4031       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
4032     ) ^ 0
4033     * ( Ct ( C "\t" ^ 0 ) / table.getn
4034       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
4035   ) / math.min

```

The following LPEG returns as capture the number of spaces at the last line, that is to say before the `\end{Piton}` (and usually it's also the number of spaces before the corresponding `\begin{Piton}` because that's the traditional way to indent in LaTeX).

```

4036 local EnvGobbleLPEG =
4037   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
4038   * Ct ( C " " ^ 0 * -1 ) / table.getn

```

The function `gobble` gobbles n characters on the left of the code. The negative values of n have special significations.

```

4039 function piton.Gobble ( n , code )
4040   if n == 0 then return
4041     code
4042   else
4043     if n == -1 then
4044       n = AutoGobbleLPEG : match ( code )

```

for the case of an empty environment (only blank lines)

```

4045     if tonumber(n) then else n = 0 end
4046   else
4047     if n == -2 then
4048       n = EnvGobbleLPEG : match ( code )
4049     else
4050       if n == -3 then
4051         n = TabsAutoGobbleLPEG : match ( code )
4052         if tonumber(n) then else n = 0 end
4053       end
4054     end
4055   end

```

We have a second test `if n == 0` because, even if the key like `auto-gobble` is in force, it's possible that, in fact, there is no space to gobble...

```

4056     if n == 0 then return
4057       code
4058     else return

```

We will now use a LPEG that we have to compute dynamically because it depends on the value of n .

```

4059   ( Ct (
4060     ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4061     * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
4062   ) ^ 0 )
4063   / table.concat

```

```

4064         ) : match ( code )
4065     end
4066 end
4067 end

```

In the following code, `n` is the value of `\l_@@gobble_int`.
`splittable` is the value of `\l_@@splittable_int`.

```

4068 function piton.GobbleParse ( lang , n , splittable , code )
4069     piton.ComputeLinesStatus ( code , splittable )
4070     piton.last_code = piton.Gobble ( n , code )
4071     piton.last_language = lang

```

We count the number of lines of the computer listing. The result will be stored by Lua in `\g_@@nb_lines_int`.

```

4072     piton.CountLines ( piton.last_code )
4073     piton.Parse ( lang , piton.last_code )
4074     piton.join_and_write ( )
4075 end

```

The following function will be used when the end user has used the key `join` or the key `write`. The value of the key `join` has been written in the Lua variable `piton.join`.

```

4076 function piton.join_and_write ( )
4077     if piton.join ~= '' then
4078         if not piton.join_files [ piton.join ] then
4079             piton.join_files [ piton.join ] = piton.get_last_code ( )
4080         else
4081             if piton.join_separation == '' then
4082                 piton.join_files [ piton.join ] =
4083                     piton.join_files [ piton.join ]
4084                     .. "\r\n"
4085                 .. piton.get_last_code ( )
4086             else
4087                 piton.join_files [ piton.join ] =
4088                     piton.join_files [ piton.join ]
4089                     .. "\r\n"
4090                     .. ( piton.join_separation : gsub ( '##' , '#' ) )
4091                     .. "\r\n"
4092                     .. piton.get_last_code ( )
4093             end
4094         end
4095     end

```

Now, if the end user has used the key `write` to write the listing of the environment on an external file (on the disk).

We have written the values of the keys `write` and `path-write` in the Lua variables `piton.write` and `piton.path_write`.

If `piton.write` is not empty, that means that the key `write` has been used for the current environment and, hence, we have to write the content of the listing on the corresponding external file.

```

4096     if piton.write ~= '' then

```

We will write on `file_name` the full name (with the path) of the file in which we will write.

```

4097         local file_name = ''
4098         if piton.path_write == '' then
4099             file_name = piton.write
4100         else

```

If `piton.path-write` is not empty, that means that we will not write on a file in the current directory but in another directory. First, we verify that that directory actually exists.

```

4101             local attr = lfs.attributes ( piton.path_write )
4102             if attr and attr.mode == "directory" then
4103                 file_name = piton.path_write .. "/" .. piton.write
4104             else

```

If the directory does *not* exist, you raise an (non-fatal) error since TeX is not able to create a new directory.

```

4105     sprintL3 [[ \@@_error_or_warning:n { InexistentDirectory } ]]
4106     end
4107     end
4108     if file_name ~= '' then

```

Now, `file_name` contains the complete name of the file on which we will have to write. Maybe the file does not exist but we are sure that the directory exist.

The Lua table `piton.write_files` is a table of Lua strings corresponding to all the files that we will write on the disk in the `\AtEndDocument`. They correspond to the use of the key `write` (and `path-write`).

```

4109     if not piton.write_files [ file_name ] then
4110         piton.write_files [ file_name ] = piton.get_last_code ( )
4111     else
4112         piton.write_files [ file_name ] =
4113         piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4114     end
4115     end
4116     end
4117 end

```

The following command will be used when the end user has set `print=false`.

```

4118 function piton.GobbleParseNoPrint ( lang , n , code )
4119     piton.last_code = piton.Gobble ( n , code )
4120     piton.last_language = lang
4121     piton.join_and_write ( )
4122 end

```

The following function will be used when the key `split-on-empty-lines` is in force. With that key, the computer listing is split in chunks at the empty lines (usually between the abstract functions defined in the computer code). LaTeX will be able to change the page between the chunks. The second argument `n` corresponds to the value of the key `gobble` (number of spaces to gobble).

```

4123 function piton.GobbleSplitParse ( lang , n , splittable , code )
4124     local chunks
4125     chunks =
4126     (
4127         Ct (
4128             (
4129                 P " " ^ 0 * "\r"
4130                 +
4131                 C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4132                     - ( P " " ^ 0 * ( P "\r" + -1 ) )
4133                 ) ^ 1
4134             )
4135         ) ^ 0
4136     )
4137     ) : match ( piton.Gobble ( n , code ) )
4138     sprintL3 [[ \begingroup ]]
4139     sprintL3
4140     (
4141         [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4142         .. "language = " .. lang .. ","
4143         .. "splittable = " .. splittable .. "}"
4144     )
4145     for k , v in pairs ( chunks ) do
4146         if k > 1 then
4147             sprintL3 ( [[ \l_@@_split_separation_tl ]] )
4148         end
4149         tex.print
4150         (
4151             [[\begin{}} .. piton.env_used_by_split .. "}" .. "\r"

```

```

4152         .. v
4153         .. [[\end{}} .. piton.env_used_by_split .. "}\r"
4154     )
4155 end
4156 sprintL3 [[ \endgroup ]]
4157 end

4158 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4159     local s
4160     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4161     piton.GobbleSplitParse ( lang , n , splittable , s )
4162 end

```

The following Lua string will be inserted between the chunks of code created when the key `split-on-empty-lines` is in force. It's used only once: you have given a name to that Lua string only for legibility. The token list `\l_@@_split_separation_tl` corresponds to the key `split-separation`. That token list must contain elements inserted in *vertical mode* of TeX.

```

4163 piton.string_between_chunks =
4164 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4165 .. [[ \global \g_@@_line_int = 0 ]]

```

The counter `\g_@@_line_int` will be used to control the points where the code may be broken by a change of page (see the key `splittable`).

The following public Lua function is provided to the developer.

```

4166 function piton.get_last_code ( )
4167     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4168         : gsub ( '\r\n?' , '\n' )
4169 end

```

3.14 To count the number of lines

```

4170 local CountBeamerEnvironments
4171 function CountBeamerEnvironments ( code ) return
4172     (
4173         Ct (
4174             (
4175                 P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4176                 +
4177                 ( 1 - P "\r" ) ^ 0 * "\r"
4178             ) ^ 0
4179             * ( 1 - P "\r" ) ^ 0
4180             * -1
4181         ) / table.getn
4182     ) : match ( code )
4183 end

```

The following function counts the lines of code except the lines which contains only instructions for the environments of Beamer.

It is used in `GobbleParse` and at the beginning of `\@@_composition:` (in some rare circumstances). Be careful. We have tried a version with `string.gsub` without success.

```

4184 function piton.CountLines ( code )
4185     local count
4186     count =
4187         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4188             *
4189             (
4190                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4191                 + space ^ 0
4192             ) ^ -1
4193             * -1

```

```

4194         ) / table.getn
4195     ) : match ( code )
4196     if piton.beamer then
4197         count = count - 2 * CountBeamerEnvironments ( code )
4198     end
4199     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4200 end

```

The following function is only used once (in `piton.GobbleParse`). We have written an autonomous function only for legibility. The number of lines of the code will be stored in `\l_@@_nb_non_empty_lines_int`. It will be used to compute the largest number of lines to write (when `line-numbers` is in force).

```

4201 function piton.CountNonEmptyLines ( code )
4202     local count = 0

```

The following code is not clear. We should try to replace it by use of the `string` library of Lua.

```

4203     count =
4204         ( Ct ( ( P " " ^ 0 * "\r"
4205             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4206             * ( 1 - P "\r" ) ^ 0
4207             * -1
4208             ) / table.getn
4209         ) : match ( code )
4210     count = count + 1
4211     if piton.beamer then
4212         count = count - 2 * CountBeamerEnvironments ( code )
4213     end
4214     sprintL3
4215     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4216 end

```

The following function stores in `\l_@@_first_line_int` and `\l_@@_last_line_int` the numbers of lines of the file `file_name` corresponding to the strings `marker_beginning` and `marker_end`. `s` is the marker of the beginning and `t` is the marker of the end.

```

4217 function piton.ComputeRange ( s , t , file_name )
4218     local first_line = -1
4219     local count = 0
4220     local last_found = false
4221     for line in io.lines ( file_name ) do
4222         if first_line == -1 then
4223             if line : sub ( 1 , #s ) == s then
4224                 first_line = count
4225             end
4226         else
4227             if line : sub ( 1 , #t ) == t then
4228                 last_found = true
4229                 break
4230             end
4231         end
4232         count = count + 1
4233     end
4234     if first_line == -1 then
4235         sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4236     else
4237         if not last_found then
4238             sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4239         end
4240     end
4241     sprintL3 (
4242         [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 '
4243         .. [[ \global \l_@@_last_line_int = ]] .. count )
4244 end

```

3.15 To determine the empty lines of the listings

Despite its name, the Lua function `ComputeLinesStatus` computes `piton.lines_status` but also `piton.empty_lines`.

In `piton.empty_lines`, a line will have the number 0 if it's a empty line (in fact a blank line, with only spaces) and 1 elsewhere.

In `piton.lines_status`, each line will have a status with regard the breaking points allowed (for the changes of pages).

- 0 if the line is empty and a page break is allowed;
- 1 if the line is not empty but a page break is allowed after that line;
- 2 if a page break is *not* allowed after that line (empty or not empty).

`splittable` is the value of `\l_@@splittable_int`. However, if `splittable-on-empty-lines` is in force, `splittable` is the opposite of `\l_@@splittable_int`.

```
4245 function piton.ComputeLinesStatus ( code , splittable )
```

The lines in the listings which correspond to the beginning or the end of an environment of Beamer (eg. `\begin{uncoverenv}`) must be retrieved (those lines have *no* number and therefore, *no* status).

```
4246   local lpeg_line_beamer
4247   if piton.beamer then
4248     lpeg_line_beamer =
4249       space ^ 0
4250       * P [[\begin{}} * beamerEnvironments * "]"
4251       * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4252       +
4253       space ^ 0
4254       * P [[\end{}} * beamerEnvironments * "]"
4255   else
4256     lpeg_line_beamer = P ( false )
4257   end
4258   local lpeg_empty_lines =
4259     Ct (
4260       ( lpeg_line_beamer * "\r"
4261         +
4262         P " " ^ 0 * "\r" * Cc ( 0 )
4263         +
4264         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4265       ) ^ 0
4266       *
4267       ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4268     )
4269     * -1
4270   local lpeg_all_lines =
4271     Ct (
4272       ( lpeg_line_beamer * "\r"
4273         +
4274         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4275       ) ^ 0
4276       *
4277       ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4278     )
4279     * -1
```

We begin with the computation of `piton.empty_lines`. It will be used in conjunction with `line-numbers`.

```
4280   piton.empty_lines = lpeg_empty_lines : match ( code )
```


Now, we compute `piton.lines_status`. It will be used in conjunction with `splittable` and `splittable-on-empty-lines`.

Now, we will take into account the current value of `\l_@@_splittable_int` (provided by the *absolute value* of the argument `splittable`).

```

4281 local lines_status
4282 local s = splittable
4283 if splittable < 0 then s = - splittable end
4284
4284 if splittable > 0 then
4285   lines_status = lpeg_all_lines : match ( code )
4286 else

```

Here, we should try to copy `piton.empty_lines` but it's not easy.

```

4287   lines_status = lpeg_empty_lines : match ( code )
4288   for i , x in ipairs ( lines_status ) do
4289     if x == 0 then
4290       for j = 1 , s - 1 do
4291         if i + j > #lines_status then break end
4292         if lines_status[i+j] == 0 then break end
4293         lines_status[i+j] = 2
4294       end
4295       for j = 1 , s - 1 do
4296         if i - j == 1 then break end
4297         if lines_status[i-j-1] == 0 then break end
4298         lines_status[i-j-1] = 2
4299       end
4300     end
4301   end
4302 end

```

In all cases (whatever is the value of `splittable-on-empty-lines`) we have to deal with both extremities of the listing to format.

First from the beginning of the code.

```

4303   for j = 1 , s - 1 do
4304     if j > #lines_status then break end
4305     if lines_status[j] == 0 then break end
4306     lines_status[j] = 2
4307   end

```

Now, from the end of the code.

```

4308   for j = 1 , s - 1 do
4309     if #lines_status - j == 0 then break end
4310     if lines_status[#lines_status - j] == 0 then break end
4311     lines_status[#lines_status - j] = 2
4312   end

```

```

4313   piton.lines_status = lines_status
4314 end

```

```

4315 function piton.TranslateBeamerEnv ( code )
4316   local s
4317   s =
4318   (
4319     Ct (
4320       (
4321         space ^ 0
4322         * C (
4323           ( P "\\begin{" + "\\end{" )
4324           * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * "\r"
4325         )
4326         + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4327       ) ^ 0
4328     *
4329     (

```

```

4330      (
4331          space ^ 0
4332          * C (
4333              ( P "\\begin{" + "\\end{" )
4334              * beamerEnvironments * "}" * ( 1 - P "\r" ) ^ 0 * -1
4335          )
4336          + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4337      ) ^ -1
4338  )
4339  ) ^ -1 / table.concat
4340  ) : match ( code )
4341  sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4342  tex.sprint ( luatexbase.catcodetables.other , s )
4343  sprintL3 ( "]" )
4344  end

```

3.16 To create new languages with the syntax of listings

```

4345 function piton.new_language ( lang , definition )
4346     lang = lang : lower ( )

4347     local alpha , digit = lpeg.alpha , lpeg.digit
4348     local extra_letters = { "@" , "_" , "$" } --

```

The command `add_to_letter` (triggered by the key `)` don't write right away in the LPEG pattern of the letters in an intermediate `extra_letters` because we may have to retrieve letters from that "list" if there appear in a key `alsoother`.

```

4349     function add_to_letter ( c )
4350         if c ~= " " then table.insert ( extra_letters , c ) end
4351     end

```

For the digits, it's straitforward.

```

4352     function add_to_digit ( c )
4353         if c ~= " " then digit = digit + c end
4354     end

```

The main use of the key `alsoother` is, for the language LaTeX, when you have to retrieve some characters from the list of letters, in particular `@` and `_` (which, by default, are not allowed in the name of a control sequence in TeX).

(In the following LPEG we have a problem when we try to add `{` and `}`).

```

4355     local other = S " :_@+~*/<>!?.() [] ~^=#&\"'\\$" --
4356     local extra_others = { }
4357     function add_to_other ( c )
4358         if c ~= " " then

```

We will use `extra_others` to retrieve further these characters from the list of the letters.

```

4359         extra_others[c] = true

```

The LPEG pattern `other` will be used in conjunction with the key `tag` (mainly for languages such as HTML and XML) for the character `/` in the closing tags `</...>`.

```

4360         other = other + P ( c )
4361     end
4362 end

```

Now, the first transformation of the definition of the language, as provided by the end user in the argument definition of `piton.new_language`.

```

4363     local def_table
4364     if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4365         def_table = {}
4366     else
4367         local strict_braces =

```

```

4368     P { "E" ,
4369         E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4370         F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4371     }
4372     local cut_definition =
4373     P { "E" ,
4374         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
4375         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4376             * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4377     }
4378     def_table = cut_definition : match ( definition )
4379 end

```

The definition of the language, provided by the end user of piton is now in the Lua table `def_table`. We will use it *several times*.

The following LPEG will be used to extract arguments in the values of the keys (`morekeywords`, `morecomment`, `morestring`, etc.).

```

4380     local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4381     local tex_arg = tex_braced_arg + C ( 1 )
4382     local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4383     local args_for_tag
4384     = tex_option_arg
4385       * space ^ 0
4386       * tex_arg
4387       * space ^ 0
4388       * tex_arg
4389     local args_for_morekeywords
4390     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4391       * space ^ 0
4392       * tex_option_arg
4393       * space ^ 0
4394       * tex_arg
4395       * space ^ 0
4396       * ( tex_braced_arg + Cc ( nil ) )
4397     local args_for_moredelims
4398     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4399       * args_for_morekeywords
4400     local args_for_morecomment
4401     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4402       * space ^ 0
4403       * tex_option_arg
4404       * space ^ 0
4405       * C ( P ( 1 ) ^ 0 * -1 )

```

We scan the definition of the language (i.e. the table `def_table`) in order to detect the potential key **sensitive**. Indeed, we have to catch that key before the treatment of the keywords of the language. We will also look for the potential keys `alsodigit`, `alsoletter` and `tag`.

```

4406     local sensitive = true
4407     local style_tag , left_tag , right_tag
4408     for _ , x in ipairs ( def_table ) do
4409         if x[1] == "sensitive" then
4410             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4411                 sensitive = true
4412             else
4413                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4414             end
4415         end
4416         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4417         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4418         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4419         if x[1] == "tag" then

```

```

4420     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4421     style_tag = style_tag or [[\PitonStyle{Tag}]]
4422 end
4423 end

```

Now, the LPEG for the numbers. Of course, it uses `digit` previously computed.

```

4424 local Number =
4425   K ( 'Number.Internal' ,
4426     ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4427       + digit ^ 0 * "." * digit ^ 1
4428       + digit ^ 1 )
4429     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4430     + digit ^ 1
4431   )
4432 local string_extra_letters = ""
4433 for _ , x in ipairs ( extra_letters ) do
4434   if not ( extra_others[x] ) then
4435     string_extra_letters = string_extra_letters .. x
4436   end
4437 end
4438 local letter = alpha + S ( string_extra_letters )
4439   + P "â" + "à" + "ç" + "ê" + "ë" + "ê" + "ï" + "î"
4440   + "ô" + "û" + "ü" + "Ë" + "Ê" + "Ç" + "É" + "È" + "Ê" + "Ë"
4441   + "ï" + "î" + "Û" + "Ü" + "Û"
4442 local alphanum = letter + digit
4443 local identifier = letter * alphanum ^ 0
4444 local Identifier = K ( 'Identifier.Internal' , identifier )

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the keywords.

The following LPEG does *not* catch the optional argument between square brackets in first position.

```

4445 local split_clist =
4446   P { "E" ,
4447     E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4448       * ( P "{" ) ^ 1
4449       * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4450       * ( P "}" ) ^ 1 * space ^ 0 ,
4451     F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4452   }

```

The following function will be used if the keywords are not case-sensitive.

```

4453 local keyword_to_lpeg
4454 function keyword_to_lpeg ( name ) return
4455   Q ( Cmt (
4456     C ( identifier ) ,
4457     function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4458   end
4459 )
4460 )
4461 end
4462 local Keyword = P ( false )
4463 local PrefixedKeyword = P ( false )

```

Now, we actually treat all the keywords and also the key `moredirectives`.

```

4464 for _ , x in ipairs ( def_table )
4465 do if x[1] == "morekeywords"
4466   or x[1] == "otherkeywords"
4467   or x[1] == "moredirectives"
4468   or x[1] == "moretexcs"
4469 then
4470   local keywords = P ( false )
4471   local style = [[\PitonStyle{Keyword}]]
4472   if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4473   style = tex_option_arg : match ( x[2] ) or style
4474   local n = tonumber ( style )

```

```

4475     if n then
4476       if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4477     end
4478     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4479       if x[1] == "moretexcs" then
4480         keywords = Q ( [[\]] .. word ) + keywords
4481       else
4482         if sensitive

```

The documentation of `lstlistings` specifies that, for the key `morekeywords`, if a keyword is a prefix of another keyword, then the prefix must appear first. However, for the `lpeg`, it's rather the contrary. That's why, here, we add the new element *on the left*.

```

4483         then keywords = Q ( word ) + keywords
4484         else keywords = keyword_to_lpeg ( word ) + keywords
4485       end
4486     end
4487   end
4488   Keyword = Keyword +
4489     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4490 end

```

Of course, the feature with the key `keywordsprefix` is designed for the languages TeX, LaTeX, et al. In that case, there is two kinds of keywords (= control sequences).

- those beginning with `\` and a sequence of characters of catcode “letter”;
- those beginning by `\` followed by one character of catcode “other”.

The following code addresses both cases. Of course, the LPEG pattern `letter` must catch only characters of catcode “letter”. That's why we have a key `alsoletter` to add new characters in that category (e.g. : when we want to format L3 code). However, the LPEG pattern is allowed to catch *more* than only the characters of catcode “other” in TeX.

```

4491   if x[1] == "keywordsprefix" then
4492     local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4493     PrefixedKeyword = PrefixedKeyword
4494       + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4495   end
4496 end

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the strings.

```

4497   local long_string = P ( false )
4498   local Long_string = P ( false )
4499   local LongString = P ( false )
4500   local central_pattern = P ( false )
4501   for _ , x in ipairs ( def_table ) do
4502     if x[1] == "morestring" then
4503       arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4504       arg2 = arg2 or [[\PitonStyle{String.Long}]]
4505       if arg1 ~= "s" then
4506         arg4 = arg3
4507       end
4508       central_pattern = 1 - S ( " \r" .. arg4 )
4509       if arg1 : match "b" then
4510         central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4511       end

```

In fact, the specifier `d` is point-less: when it is not in force, it's still possible to double the delimiter with a correct behaviour of `piton` since, in that case, `piton` will compose *two* contiguous strings...

```

4512     if arg1 : match "d" or arg1 == "m" then
4513       central_pattern = P ( arg3 .. arg3 ) + central_pattern
4514     end
4515     if arg1 == "m"
4516     then prefix = B ( 1 - letter - ")" - "]" )
4517     else prefix = P ( true )
4518     end

```

First, a pattern *without captures* (needed to compute braces).

```

4519     long_string = long_string +
4520         prefix
4521         * arg3
4522         * ( space + central_pattern ) ^ 0
4523         * arg4

```

Now a pattern *with captures*.

```

4524     local pattern =
4525         prefix
4526         * Q ( arg3 )
4527         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4528         * Q ( arg4 )

```

We will need Long_string in the nested comments.

```

4529     Long_string = Long_string + pattern
4530     LongString = LongString +
4531         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4532         * pattern
4533         * Ct ( Cc "Close" )
4534     end
4535 end

```

The argument of Compute_braces must be a pattern *which does no catching* corresponding to the strings of the language.

```

4536     local braces = Compute_braces ( long_string )
4537     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4538
4539     DetectedCommands =
4540         Compute_DetectedCommands ( lang , braces )
4541         + Compute_RawDetectedCommands ( lang , braces )
4542
4543     LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

4544     local CommentDelim = P ( false )
4545
4546     for _ , x in ipairs ( def_table ) do
4547         if x[1] == "morecomment" then
4548             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4549             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

If the letter i is present in the first argument (eg: morecomment = [si]{(*){*}), then the corresponding comments are discarded.

```

4550             if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4551             if arg1 : match "l" then
4552                 local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4553                     : match ( other_args )
4554                 if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4555                 if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4556                 CommentDelim = CommentDelim +
4557                     Ct ( Cc "Open"
4558                         * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4559                         * Q ( arg3 )
4560                         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4561                         * Ct ( Cc "Close" )
4562                         * ( EOL + -1 )
4563             else
4564                 local arg3 , arg4 =
4565                     ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4566                 if arg1 : match "s" then
4567                     CommentDelim = CommentDelim +
4568                         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4569                         * Q ( arg3 )

```

```

4570         * (
4571             CommentMath
4572             + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4573             + EOL
4574         ) ^ 0
4575         * Q ( arg4 )
4576         * Ct ( Cc "Close" )
4577     end
4578     if arg1 : match "n" then
4579         CommentDelim = CommentDelim +
4580         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
4581         * P { "A" ,
4582             A = Q ( arg3 )
4583             * ( V "A"
4584                 + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4585                     - S "\r$" ) ^ 1 ) -- $
4586                 + long_string
4587                 + "$" -- $
4588                 * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4589                 * "$" -- $
4590                 + EOL
4591             ) ^ 0
4592             * Q ( arg4 )
4593         }
4594         * Ct ( Cc "Close" )
4595     end
4596 end
4597 end

```

For the keys `moredelim`, we have to add another argument in first position, equal to `*` or `**`.

```

4598 if x[1] == "moredelim" then
4599     local arg1 , arg2 , arg3 , arg4 , arg5
4600     = args_for_moredelims : match ( x[2] )
4601     local MyFun = Q
4602     if arg1 == "*" or arg1 == "**" then
4603         function MyFun ( x )
4604             if x ~= '' then return
4605                 LPEG1[lang] : match ( x )
4606             end
4607         end
4608     end
4609     local left_delim
4610     if arg2 : match "i" then
4611         left_delim = P ( arg4 )
4612     else
4613         left_delim = Q ( arg4 )
4614     end
4615     if arg2 : match "l" then
4616         CommentDelim = CommentDelim +
4617         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )
4618         * left_delim
4619         * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4620         * Ct ( Cc "Close" )
4621         * ( EOL + -1 )
4622     end
4623     if arg2 : match "s" then
4624         local right_delim
4625         if arg2 : match "i" then
4626             right_delim = P ( arg5 )
4627         else
4628             right_delim = Q ( arg5 )
4629         end
4630         CommentDelim = CommentDelim +
4631         Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )

```

```

4632         * left_delim
4633         * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4634         * right_delim
4635         * Ct ( Cc "Close" )
4636     end
4637 end
4638 end
4639
4640 local Delim = Q ( S "{[()]}" )
4641 local Punct = Q ( S "=:;!\\"'" )
4642
4643 local Main =
4644     space ^ 0 * EOL
4645     + Space
4646     + Tab
4647     + Escape + EscapeMath
4648     + CommentLaTeX
4649     + Beamer
4650     + DetectedCommands
4651     + CommentDelim

```

We must put `LongString` before `Delim` because, in PostScript, the strings are delimited by parenthesis and those parenthesis would be caught by `Delim`.

```

4651     + LongString
4652     + Delim
4653     + PrefixedKeyword
4654     + Keyword * ( -1 + # ( 1 - alphanum ) )
4655     + Punct
4656     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4657     + Number
4658     + Word

```

The LPEG `LPEG1[lang]` is used to reformat small elements, for example the arguments of the “detected commands”.

Of course, here, we must not put `local`, of course.

```

4659 LPEG1[lang] = Main ^ 0

```

The LPEG `LPEG2[lang]` is used to format general chunks of code.

```

4660 LPEG2[lang] =
4661     Ct (
4662         ( space ^ 0 * P "\r" ) ^ -1
4663         * Lc [[ \@@_begin_line: ]]
4664         * LeadingSpace ^ 0
4665         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
4666         * -1
4667         * Lc [[ \@@_end_line: ]]
4668     )

```

If the key `tag` has been used. Of course, this feature is designed for the languages such as HTML and XML.

```

4669 if left_tag then
4670     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
4671     * Q ( left_tag * other ^ 0 ) -- $
4672     * ( ( 1 - P ( right_tag ) ) ^ 0 )
4673     / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
4674     * Q ( right_tag )
4675     * Ct ( Cc "Close" )
4676
4677     MainWithoutTag
4678     = space ^ 1 * -1
4679     + space ^ 0 * EOL
4680     + Space
4681     + Tab
4682     + Escape + EscapeMath
4683     + CommentLaTeX
4684     + Beamer

```



```

4684         + DetectedCommands
4685         + CommentDelim
4686         + Delim
4687         + LongString
4688         + PrefixedKeyword
4689         + Keyword * ( -1 + # ( 1 - alphanum ) )
4690         + Punct
4691         + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4692         + Number
4693         + Word
4694 LPEG0[lang] = MainWithoutTag ^ 0
4695 local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
4696               + Beamer + DetectedCommands + CommentDelim + Tag
4697 MainWithTag
4698     = space ^ 1 * -1
4699     + space ^ 0 * EOL
4700     + Space
4701     + LPEGaux
4702     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
4703 LPEG1[lang] = MainWithTag ^ 0
4704 LPEG2[lang] =
4705     Ct (
4706         ( space ^ 0 * P "\r" ) ^ -1
4707         * Lc [[ \@_begin_line: ]]
4708         * Beamer
4709         * LeadingSpace ^ 0
4710         * LPEG1[lang]
4711         * -1
4712         * Lc [[ \@_end_line: ]]
4713     )
4714 end
4715 end

```

3.17 We write the files (key 'write') and join the files in the PDF (key 'join')

```

4716 function piton.write_files_now ( )
4717     for file_name , file_content in pairs ( piton.write_files ) do
4718         local file = io.open ( file_name , "w" )
4719         if file then
4720             file : write ( file_content )
4721             file : close ( )
4722         else
4723             sprintL3
4724             ( [[ \@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
4725         end
4726     end
4727 end

```

3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

4728 function piton.utf16 ( str )
4729     local hex = { "FEFF" } -- BOM UTF-16BE
4730     for _, codepoint in utf8.codes(str) do
4731         table.insert(hex, string.format("%04X", codepoint))
4732     end
4733     return table.concat(hex)
4734 end
4735 </LUA>

```