

The package `piton`*

F. Pantigny
fpantigny@wanadoo.fr

October 4, 2025

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 4.9 of `piton`, at the date of 2025/10/04.

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: \_piton_end_line:
{x{\PitonStyle{Operator}{%}}{\PitonStyle{Number}{2}}\_piton_end_line:

```

2 The L3 part of the implementation

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

```

21 \RequirePackage { transparent }

```

```

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

```

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

```

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

```

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

```

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

```

```

46 \@@_msg_new:nn { LuaLaTeX~mandatory }
47 {
48   LuaLaTeX~is~mandatory.\\
49   The~package~'piton'~requires~the~engine~LuaLaTeX.\\
50   \str_if_eq:onT \c_sys_jobname_str { output }
51   { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~the~"Menu"~and~
52     if~you~use~TeXPage,~you~should~go~in~"Settings".  \\\ }

```

```

53 \IfClassLoadedT { beamer }
54 {
55     Since-you-use-Beamer,~don't~forget~to~use~piton~in~frames~with~
56     the~key~'fragile'.\\
57 }
58 \IfClassLoadedT { ltx-talk }
59 {
60     Since-you-use~'ltx-talk',~don't~forget~to~use~piton~in~
61     environments~'frame*'.\\
62 }
63 That~error~is~fatal.
64 }
65 \sys_if_engine luatex:F { \@@_fatal:n { LuaLaTeX-mandatory } }

66 \RequirePackage { luacode }

67 \@@_msg_new:nnn { piton.lua-not-found }
68 {
69     The~file~'piton.lua'~can't~be~found.\\
70     This~error~is~fatal.\\
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 } }

```

The boolean `\g_@@_footnotehyper_bool` will indicate if the option `footnotehyper` is used.

```
79 \bool_new:N \g_@@_footnotehyper_bool
```

The boolean `\g_@@_footnote_bool` will indicate if the option `footnote` is used, but quickly, it will also be set to true if the option `footnotehyper` is used.

```
80 \bool_new:N \g_@@_footnote_bool
```

```
81 \bool_new:N \g_@@_beamer_bool
```

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

```

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

```

```
103 }
```

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

```
104 \ProcessKeyOptions

105 \IfClassLoadedT { beamer } { \bool_gset_true:N \g_@@_beamer_bool }
106 \IfClassLoadedT { ltx-talk } { \bool_gset_true:N \g_@@_beamer_bool }
107 \IfPackageLoadedT { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool }

108 \lua_now:e
109 {
110   piton = piton-or~{ }
111   piton.last_code = ''
112   piton.last_language = ''
113   piton.join = ''
114   piton.write = ''
115   piton.path_write = ''
116   \bool_if:NT \g_@@_beamer_bool { piton.beamer = true }
117 }

118 \RequirePackage { xcolor }

119 \@@_msg_new:nn { footnote~with~footnotehyper~package }
120 {
121   Footnote~forbidden.\\
122   You~can't~use~the~option~'footnote'~because~the~package~
123   footnotehyper~has~already~been~loaded.~
124   If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
125   within~the~environments~of~piton~will~be~extracted~with~the~tools~
126   of~the~package~footnotehyper.\\
127   If~you~go~on,~the~package~footnote~won't~be~loaded.
128 }

129 \@@_msg_new:nn { footnotehyper~with~footnote~package }
130 {
131   You~can't~use~the~option~'footnotehyper'~because~the~package~
132   footnote~has~already~been~loaded.~
133   If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
134   within~the~environments~of~piton~will~be~extracted~with~the~tools~
135   of~the~package~footnote.\\
136   If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
137 }

138 \bool_if:NT \g_@@_footnote_bool
139 {
```

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

```
140 \IfClassLoadedTF { beamer }
141 { \bool_gset_false:N \g_@@_footnote_bool }
142 {
143   \IfPackageLoadedTF { footnotehyper }
144   { \@@_error:n { footnote~with~footnotehyper~package } }
145   { \usepackage { footnote } }
146 }
147 }

148 \bool_if:NT \g_@@_footnotehyper_bool
149 {
```

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

```
150 \IfClassLoadedTF { beamer }
151 { \bool_gset_false:N \g_@@_footnote_bool }
152 {
153   \IfPackageLoadedTF { footnote }
```

```

154     { \@@_error:n { footnotehyper~with~footnote~package } }
155     { \usepackage { footnotehyper } }
156     \bool_gset_true:N \g_@@_footnote_bool
157   }
158 }

```

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}`.

2.0.2 Parameters and technical definitions

```

159 \dim_new:N \l_@@_rounded_corners_dim
160 \bool_new:N \l_@@_in_label_bool
161 \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`).

```

162 \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_backgrounds_to_output_box:`).

```

163 \box_new:N \g_@@_output_box

```

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

```

164 \str_new:N \l_piton_language_str
165 \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.

```

166 \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`).

```

167 \seq_new:N \l_@@_path_seq

```

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

```

168 \seq_new:N \g_@@_join_seq

```

The following parameter corresponds to the key `path-write` (which is the path used when writing files from listings inserted in the environments of `piton` by use of the key `write`).

```

169 \str_new:N \l_@@_path_write_str

```

The following parameter corresponds to the key `tcolorbox`.

```

170 \bool_new:N \l_@@_tcolorbox_bool

```

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.

```

171 \dim_new:N \l_@@_tcb_margins_dim

```

The following parameter corresponds to the key `box`.

```

172 \str_new:N \l_@@_box_str

```

In order to have a better control over the keys.

```

173 \bool_new:N \l_@@_in_PitonOptions_bool
174 \bool_new:N \l_@@_in_PitonInputFile_bool

```

The following parameter corresponds to the key `font-command`.

```

175 \tl_new:N \l_@@_font_command_tl
176 \tl_set:Nn \l_@@_font_command_tl { \ttfamily }

```

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.

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

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

```
178 \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).

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

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).

```
180 \int_new:N \l_@@_splittable_int
```

An initial value of `splittable` equal to 100 is equivalent to say that the environments `{Piton}` are unbreakable.

```
181 \int_set:Nn \l_@@_splittable_int { 100 }
```

When the key `split-on-empty-lines` will be in force, then the following 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).

```
182 \tl_new:N \l_@@_split_separation_tl
183 \tl_set:Nn \l_@@_split_separation_tl
184 { \vspace { \baselineskip } \vspace { -1.25pt } }
```

That parameter must contain elements to be inserted in *vertical* mode by TeX.

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

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

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

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

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`).

```
187 \tl_new:N \l_@@_prompt_bg_color_tl
188 \tl_set:Nn \l_@@_prompt_bg_color_tl { gray!15 }
```

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

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

```
190 \str_new:N \l_@@_begin_range_str
191 \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).

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

The argument of `\PitonInputFile`.

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

The following flag corresponds to the key `print`. The initial value of that parameter will be `true` (and not `false`) since, of course, by default, we want to print the content of the environment `{Piton}`

```
194 \bool_new:N \l_@@_print_bool
195 \bool_set_true:N \l_@@_print_bool
```

The parameter `\l_@@_write_str` corresponds to the key `write`.

```
196 \str_new:N \l_@@_write_str
```

The parameter `\l_@@_join_str` corresponds to the key `join`.

```
197 \str_new:N \l_@@_join_str
```

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

```
198 \bool_new:N \l_@@_paperclip_bool
199 \str_new:N \l_@@_paperclip_str
200 \bool_new:N \l_@@_annotation_bool
```

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

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

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

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

The following booleans correspond to the keys `break-lines` and `indent-broken-lines`.

```
203 \bool_new:N \l_@@_break_lines_in_Piton_bool
204 \bool_set_true:N \l_@@_break_lines_in_Piton_bool
205 \bool_new:N \l_@@_indent_broken_lines_bool
```

The following token list corresponds to the key `continuation-symbol`.

```
206 \tl_new:N \l_@@_continuation_symbol_tl
207 \tl_set:Nn \l_@@_continuation_symbol_tl { + }
```

The following token list corresponds to the key `continuation-symbol-on-indentation`. The name has been shortened to `csoi`.

```
208 \tl_new:N \l_@@_csoi_tl
209 \tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow ; $ }
```

The following token list corresponds to the key `end-of-broken-line`.

```
210 \tl_new:N \l_@@_end_of_broken_line_tl
211 \tl_set:Nn \l_@@_end_of_broken_line_tl { \hspace* { 0.5em } \textbackslash }
```

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

```
212 \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).

```
213 \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.

```
214 \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).

```
215 \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:`.

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

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


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

The following dimension corresponds to the key `left-margin`.

```
218 \dim_new:N \l_@@_left_margin_dim
```

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

```
219 \bool_new:N \l_@@_left_margin_auto_bool
```

The following dimension corresponds to the key `numbers-sep` of `\PitonOptions`.

```
220 \dim_new:N \l_@@_numbers_sep_dim
221 \dim_set:Nn \l_@@_numbers_sep_dim { 0.7 em }
```

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.

```
222 \seq_new:N \g_@@_languages_seq

223 \int_new:N \l_@@_tab_size_int
224 \int_set:Nn \l_@@_tab_size_int { 4 }

225 \cs_new_protected:Npn \@@_tab:
226 {
227   \bool_if:NTF \l_@@_show_spaces_bool
228   {
229     \hbox_set:Nn \l_tmpa_box
230     { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
231     \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
232     \< \mathcolor { gray }
233     { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \>
234   }
235   { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
236   \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
237 }
```

The following integer corresponds to the key `gobble`.

```
238 \int_new:N \l_@@_gobble_int
```

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

```
239 \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.

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

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

```
241 \cs_new_protected:Npn \@@_label:n #1
242 {
243   \bool_if:NTF \l_@@_line_numbers_bool
244   {
245     \@bsphack
246     \protected@write \@auxout { }
247     {
248       \string \newlabel { #1 }
249       {
250         { \int_use:N \g_@@_visual_line_int }
251         { \thepage }
252         { }
253         { line.#1 }
254         { }

```

```

255     }
256   }
257   \@esphack
258   \IfPackageLoadedT { hyperref }
259     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
260   }
261   { \@@_error:n { label-with-lines-numbers } }
262 }

```

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.

```

263 \cs_new_protected:Npn \@@_zlabel:n #1
264 {
265   \bool_if:NTF \l_@@_line_numbers_bool
266   {
267     \@esphack
268     \protected@write \@auxout { }
269     {
270       \string \zref@newlabel { #1 }
271       {
272         \string \default { \int_use:N \g_@@_visual_line_int }
273         \string \page { \thepage }
274         \string \zc@type { line }
275         \string \anchor { line.#1 }
276       }
277     }
278     \@esphack
279     \IfPackageLoadedT { hyperref }
280       { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
281   }
282   { \@@_error:n { label-with-lines-numbers } }
283 }

```

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

```

284 \NewDocumentCommand { \@@_rowcolor:n } { o m }
285 {
286   \tl_gset:ce
287     { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 }_ t1 }
288     { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
289   \bool_gset_true:N \g_@@_rowcolor_inside_bool
290 }

```

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).

```

291 \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.

```

292 \cs_new:Npn \@@_marker_beginning:n #1 { }
293 \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.

```

294 \tl_new:N \g_@@_after_line_t1

```

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

```
295 \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`.

```
296 \bool_new:N \g_@@_color_is_none_bool
297 \bool_new:N \g_@@_next_color_is_none_bool
```

```
298 \bool_new:N \g_@@_rowcolor_inside_bool
```

2.0.3 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 *without* the backlash.

```
299 \clist_new:N \l_@@_detected_commands_clist
300 \clist_new:N \l_@@_raw_detected_commands_clist
301 \clist_new:N \l_@@_beamer_commands_clist
302 \clist_set:Nn \l_@@_beamer_commands_clist
303   { uncover , only , visible , invisible , alert , action }
304 \clist_new:N \l_@@_beamer_environments_clist
305 \clist_set:Nn \l_@@_beamer_environments_clist
306   { 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.

```
307 \hook_gput_code:nnn { begindocument } { . }
308   {
309     \newtoks \PitonDetectedCommands
310     \newtoks \PitonRawDetectedCommands
311     \newtoks \PitonBeamerCommands
312     \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.

```
313   \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
314   \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
315   \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
316   \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
317 }
```

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.

```
318 \tl_new:N \g_@@_def_vertical_commands_tl
```

```

319 \cs_new_protected:Npn \l_@@_vertical_commands:n #1
320 {
321   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 }
322   \clist_map_inline:nn { #1 }
323   {
324     \cs_set_eq:cc { @@ _ old _ ##1 : } { ##1 }
325     \cs_new_protected:cn { @@ _ new _ ##1 : n }
326     {
327       \bool_if:nTF
328       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
329       {
330         \tl_gput_right:Nn \g_@@_after_line_tl
331         { \use:c { @@ _ old _ ##1 : } { #####1 } }
332       }
333       {
334         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int _ tl }
335         { \tl_gput_right:cn }
336         { \tl_gset:cn }
337         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 } _ tl }
338         { \use:c { @@ _ old _ ##1 : } { #####1 } }
339       }
340     }
341     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
342     { \cs_set_eq:cc { ##1 } { @@ _ new _ ##1 : n } }
343   }
344 }

```

2.0.4 Treatment of a line of code

```

345 \cs_new_protected:Npn \l_@@_replace_spaces:n #1
346 {
347   \tl_set:Nn \l_tmpa_tl { #1 }
348   \bool_if:NTF \l_@@_show_spaces_bool
349   {
350     \tl_set:Nn \l_@@_space_in_string_tl { \_ } % U+2423
351     \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { \_ } % U+2423
352   }
353   {

```

If the key `break-lines-in-Piton` is in force, we replace all the characters U+0020 (that is to say the spaces) by `\l_@@_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.

```

354     \bool_if:NT \l_@@_break_lines_in_Piton_bool
355     {
356       \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
357       { \tl_set_eq:NN \l_@@_space_in_string_tl \l_@@_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 `\l_@@_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.

```

358     \tl_replace_all:Nvn \l_tmpa_tl
359     \c_catcode_other_space_tl
360     \l_@@_breakable_space:

```

```

361     }
362   }
363   \l_tmpa_tl
364 }
365 \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:`.

```

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

367 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
368 {
369   \group_begin:
370   \int_gzero:N \g_@@_indentation_int

```

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

```

371   \hbox_set:Nn \l_@@_line_box
372   {
373     \skip_horizontal:N \l_@@_left_margin_dim
374     \bool_if:NT \l_@@_line_numbers_bool
375     {

```

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

```

376       \int_set:Nn \l_tmpa_int
377       {
378         \lua_now:e
379         {
380           tex.sprint
381           (

```

The following expression gives a 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`.

```

382           piton.empty_lines
383           [ \int_eval:n { \g_@@_line_int + 1 } ]
384         )
385       }
386     }
387     \bool_lazy_or:nnT
388     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
389     { ! \l_@@_skip_empty_lines_bool }
390     { \int_gincr:N \g_@@_visual_line_int }
391     \bool_lazy_or:nnT
392     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
393     { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
394     { \@@_print_number: }
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 }

```

Of course, the following command will be used when the key `background-color` is used.

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

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

```

436 \cs_new_protected:Npn \@@_add_background_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     \@@_compute_and_set_color:

```

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

```

444     \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
445     \bool_if:NT \g_@@_next_color_is_none_bool
446     { \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.

```

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

```

Now, the colored panel.

```

451 \dim_compare:nNnTF \l_@@_rounded_corners_dim > \c_zero_dim
452 {
453   \int_compare:nNnTF \g_@@_line_int = \c_one_int
454   {
455     \begin{tikzpicture}[baseline = 0cm]
456       \fill (0,0)
457         [rounded-corners = \l_@@_rounded_corners_dim]
458         -- (0,\l_@@_tmpc_dim)
459         -- (\l_tmpb_dim,\l_@@_tmpc_dim)
460         [sharp-corners] -- (\l_tmpb_dim,-\l_tmpa_dim)
461         -- (0,-\l_tmpa_dim)
462         -- cycle ;
463     \end{tikzpicture}
464   }
465   {
466     \int_compare:nNnTF \g_@@_line_int = \g_@@_nb_lines_int
467     {
468       \begin{tikzpicture}[baseline = 0cm]
469         \fill (0,0) -- (0,\l_@@_tmpc_dim)
470           -- (\l_tmpb_dim,\l_@@_tmpc_dim)
471           [rounded-corners = \l_@@_rounded_corners_dim]
472           -- (\l_tmpb_dim,-\l_tmpa_dim)
473           -- (0,-\l_tmpa_dim)
474           -- cycle ;
475       \end{tikzpicture}
476     }
477     {
478       \vrule height \l_@@_tmpc_dim
479       depth \l_tmpa_dim
480       width \l_tmpb_dim
481     }
482   }
483 }
484 {
485   \vrule height \l_@@_tmpc_dim
486   depth \l_tmpa_dim
487   width \l_tmpb_dim
488 }
489 }
490 \bool_if:NT \g_@@_next_color_is_none_bool
491 { \skip_vertical:n { 2.5 pt } }
492 \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
493 \box_use_drop:N \l_@@_line_box
494 }
495 }

```

End of \@@_add_background_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.

```

496 \cs_set_protected:Npn \@@_compute_and_set_color:
497 {
498   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
499   { \tl_set:Nn \l_tmpa_tl { none } }
500   {
501     \int_set:Nn \l_tmpb_int
502     { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }

```

```

503     \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
504 }

```

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

```

505     \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
506     {
507         \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`).

```

508         \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
509     }
510     \tl_if_eq:NnTF \l_tmpa_tl { none }
511     { \bool_gset_true:N \g_@@_color_is_none_bool }
512     {
513         \bool_gset_false:N \g_@@_color_is_none_bool
514         \@@_color:o \l_tmpa_tl
515     }

```

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).

```

516     \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
517     { \bool_gset_false:N \g_@@_next_color_is_none_bool }
518     {
519         \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
520         { \tl_set:Nn \l_tmpa_tl { none } }
521         {
522             \int_set:Nn \l_tmpb_int
523             { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
524             \tl_set:Nc \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
525         }
526         \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
527         {
528             \tl_set_eq:Nc \l_tmpa_tl
529             { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
530         }
531         \tl_if_eq:NnTF \l_tmpa_tl { none }
532         { \bool_gset_true:N \g_@@_next_color_is_none_bool }
533         { \bool_gset_false:N \g_@@_next_color_is_none_bool }
534     }
535 }

```

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} }`.

```

536 \cs_set_protected:Npn \@@_color:n #1
537 {
538     \tl_if_head_eq_meaning:nNTF { #1 } [
539     {
540         \tl_set:Nn \l_tmpa_tl { #1 }
541         \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
542         \exp_last_unbraced:No \color \l_tmpa_tl
543     }
544     { \color { #1 } }
545 }
546 \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.


```

547 \cs_new_protected:Npn \@@_par:
548   {

```

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

```

549   \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.

```

550   \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.

```

551   \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

552   \@@_add_penalty_for_the_line:
553   }

```

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`.

```

554 \cs_set_protected:Npn \@@_breakable_space:
555   {
556     \discretionary
557     { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
558     {
559       \hbox_overlap_left:n
560       {
561         {
562           \normalfont \footnotesize \color { gray }
563           \l_@@_continuation_symbol_tl
564         }
565         \skip_horizontal:n { 0.3 em }
566         \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
567         { \skip_horizontal:n { 0.5 em } }
568       }
569       \bool_if:NT \l_@@_indent_broken_lines_bool
570       {
571         \hbox:n
572         {
573           \prg_replicate:nn { \g_@@_indentation_int } { ~ }
574           { \color { gray } \l_@@_csoi_tl }
575         }
576       }
577     }
578     { \hbox { ~ } }
579   }

```

2.0.5 PitonOptions

```

580 \bool_new:N \l_@@_line_numbers_bool
581 \bool_new:N \l_@@_skip_empty_lines_bool
582 \bool_set_true:N \l_@@_skip_empty_lines_bool
583 \bool_new:N \l_@@_line_numbers_absolute_bool
584 \tl_new:N \l_@@_line_numbers_format_tl
585 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
586 \bool_new:N \l_@@_label_empty_lines_bool
587 \bool_set_true:N \l_@@_label_empty_lines_bool
588 \int_new:N \l_@@_number_lines_start_int

```

```

589 \bool_new:N \l_@@_resume_bool
590 \bool_new:N \l_@@_split_on_empty_lines_bool
591 \bool_new:N \l_@@_splittable_on_empty_lines_bool
592 \bool_new:N \g_@@_label_as_zlabel_bool

593 \keys_define:nn { PitonOptions / marker }
594 {
595     beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
596     beginning .value_required:n = true ,
597     end .cs_set:Np = \@@_marker_end:n #1 ,
598     end .value_required:n = true ,
599     include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
600     include-lines .default:n = true ,
601     unknown .code:n = \@@_error:n { Unknown~key~for~marker }
602 }

603 \keys_define:nn { PitonOptions / line-numbers }
604 {
605     true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
606     false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
607
608     start .code:n =
609         \bool_set_true:N \l_@@_line_numbers_bool
610         \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
611     start .value_required:n = true ,
612
613     skip-empty-lines .code:n =
614         \bool_if:NF \l_@@_in_PitonOptions_bool
615         { \bool_set_true:N \l_@@_line_numbers_bool }
616         \str_if_eq:nnTF { #1 } { false }
617         { \bool_set_false:N \l_@@_skip_empty_lines_bool }
618         { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
619     skip-empty-lines .default:n = true ,
620
621     label-empty-lines .code:n =
622         \bool_if:NF \l_@@_in_PitonOptions_bool
623         { \bool_set_true:N \l_@@_line_numbers_bool }
624         \str_if_eq:nnTF { #1 } { false }
625         { \bool_set_false:N \l_@@_label_empty_lines_bool }
626         { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
627     label-empty-lines .default:n = true ,
628
629     absolute .code:n =
630         \bool_if:NTF \l_@@_in_PitonOptions_bool
631         { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
632         { \bool_set_true:N \l_@@_line_numbers_bool }
633         \bool_if:NT \l_@@_in_PitonInputFile_bool
634         {
635             \bool_set_true:N \l_@@_line_numbers_absolute_bool
636             \bool_set_false:N \l_@@_skip_empty_lines_bool
637         } ,
638     absolute .value_forbidden:n = true ,
639
640     resume .code:n =
641         \bool_set_true:N \l_@@_resume_bool
642         \bool_if:NF \l_@@_in_PitonOptions_bool
643         { \bool_set_true:N \l_@@_line_numbers_bool } ,
644     resume .value_forbidden:n = true ,
645
646     sep .dim_set:N = \l_@@_numbers_sep_dim ,
647     sep .value_required:n = true ,
648
649     format .tl_set:N = \l_@@_line_numbers_format_tl ,

```

```

650 format .value_required:n = true ,
651
652 unknown .code:n = \@_error:n { Unknown-key-for-line-numbers }
653 }

```

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

```

654 \keys_define:nn { PitonOptions }
655 {
656   indentations-for-Foxit .choices:nn = { true , false }
657   {
658     \tl_if_eq:VnTF \l_keys_value_tl { true }
659     { \@_define_leading_space_Foxit: }
660     { \@_define_leading_space_normal: }
661   } ,
662   box .choices:nn = { c , t , b , m }
663   { \str_set_eq:NN \l_@@_box_str \l_keys_choice_tl } ,
664   box .default:n = c ,
665   break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
666   break-strings-anywhere .default:n = true ,
667   break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,
668   break-numbers-anywhere .default:n = true ,

```

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

```

669 detected-commands .code:n =
670   \clist_if_in:nnTF { #1 } { rowcolor }
671   {
672     \@_error:n { rowcolor-in-detected-commands }
673     \clist_set:Nn \l_tmpa_clist { #1 }
674     \clist_remove_all:Nn \l_tmpa_clist { rowcolor }
675     \clist_put_right:No \l_@@_detected_commands_clist \l_tmpa_clist
676   }
677   { \clist_put_right:Nn \l_@@_detected_commands_clist { #1 } } ,
678 detected-commands .value_required:n = true ,
679 detected-commands .usage:n = preamble ,
680 vertical-detected-commands .code:n = \@_vertical_commands:n { #1 } ,
681 vertical-detected-commands .value_required:n = true ,
682 vertical-detected-commands .usage:n = preamble ,
683 raw-detected-commands .code:n =
684   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 } ,
685 raw-detected-commands .value_required:n = true ,
686 raw-detected-commands .usage:n = preamble ,
687 detected-beamer-commands .code:n =
688   \@_error_if_not_in_beamer:
689   \clist_put_right:Nn \l_@@_beamer_commands_clist { #1 } ,
690 detected-beamer-commands .value_required:n = true ,
691 detected-beamer-commands .usage:n = preamble ,
692 detected-beamer-environments .code:n =
693   \@_error_if_not_in_beamer:
694   \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
695 detected-beamer-environments .value_required:n = true ,
696 detected-beamer-environments .usage:n = preamble ,

```

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

```

697 begin-escape .code:n =
698   \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
699 begin-escape .value_required:n = true ,
700 begin-escape .usage:n = preamble ,
701
702 end-escape .code:n =
703   \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
704 end-escape .value_required:n = true ,
705 end-escape .usage:n = preamble ,
706

```

```

707 begin-escape-math .code:n =
708   \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
709 begin-escape-math .value_required:n = true ,
710 begin-escape-math .usage:n = preamble ,
711
712 end-escape-math .code:n =
713   \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
714 end-escape-math .value_required:n = true ,
715 end-escape-math .usage:n = preamble ,
716
717 comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
718 comment-latex .value_required:n = true ,
719 comment-latex .usage:n = preamble ,
720
721 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
722 label-as-zlabel .default:n = true ,
723 label-as-zlabel .usage:n = preamble ,
724
725 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
726 math-comments .default:n = true ,
727 math-comments .usage:n = preamble ,

```

Now, general keys.

```

728 language .code:n =
729   \str_set:Ne \l_piton_language_str { \str_lowercase:n { #1 } } ,
730 language .value_required:n = true ,
731 path .code:n =
732   \seq_clear:N \l_@@_path_seq
733   \clist_map_inline:nn { #1 }
734   {
735     \str_set:Nn \l_tmpa_str { ##1 }
736     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
737   } ,
738 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.

```

739 path .initial:n = . ,
740 path-write .str_set:N = \l_@@_path_write_str ,
741 path-write .value_required:n = true ,
742 font-command .tl_set:N = \l_@@_font_command_tl ,
743 font-command .value_required:n = true ,
744 gobble .int_set:N = \l_@@_gobble_int ,
745 gobble .default:n = -1 ,
746 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
747 auto-gobble .value_forbidden:n = true ,
748 env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
749 env-gobble .value_forbidden:n = true ,
750 tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
751 tabs-auto-gobble .value_forbidden:n = true ,
752
753 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
754 splittable-on-empty-lines .default:n = true ,
755
756 split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,
757 split-on-empty-lines .default:n = true ,
758
759 split-separation .tl_set:N = \l_@@_split_separation_tl ,
760 split-separation .value_required:n = true ,
761
762 add-to-split-separation .code:n =
763   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
764 add-to-split-separation .value_required:n = true ,
765

```

```

766 marker .code:n =
767     \bool_lazy_or:nnTF
768     \l_@@_in_PitonInputFile_bool
769     \l_@@_in_PitonOptions_bool
770     { \keys_set:nn { PitonOptions / marker } { #1 } }
771     { \@@_error:n { Invalid-key } } ,
772 marker .value_required:n = true ,
773
774 line-numbers .code:n =
775     \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
776 line-numbers .default:n = true ,
777
778 splittable .int_set:N      = \l_@@_splittable_int ,
779 splittable .default:n     = 1 ,
780 background-color .code:n =
781     \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.

```

782     \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
783 background-color .value_required:n = true ,
784 prompt-background-color .tl_set:N      = \l_@@_prompt_bg_color_tl ,
785 prompt-background-color .value_required:n = true ,

```

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.

```

786     print .bool_set:N = \l_@@_print_bool ,
787     print .value_required:n = true ,
788
789 width .code:n =
790     \str_if_eq:nnTF { #1 } { min }
791     {
792         \bool_set_true:N \l_@@_minimize_width_bool
793         \dim_zero:N \l_@@_width_dim
794     }
795     {
796         \bool_set_false:N \l_@@_minimize_width_bool
797         \dim_set:Nn \l_@@_width_dim { #1 }
798     } ,
799 width .value_required:n = true ,
800
801 max-width .code:n =
802     \bool_set_true:N \l_@@_minimize_width_bool
803     \dim_set:Nn \l_@@_width_dim { #1 } ,
804 max-width .value_required:n = true ,
805
806 paperclip .code:n =
807     \bool_set_true:N \l_@@_paperclip_bool
808     \tl_if_novalue:nTF { #1 }
809     { \str_set:Nn \l_@@_paperclip_str { } }
810     { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
811
812 annotation .bool_set:N = \l_@@_annotation_bool ,
813 annotation .default:n = true ,
814
815 write .str_set:N = \l_@@_write_str ,
816 write .value_required:n = true ,
817 no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
818 no-write .value_forbidden:n = true ,
819 join .code:n =
820     \str_set:Nn \l_@@_join_str { #1 }
821     \seq_if_in:NnF \g_@@_join_seq { #1 }
822     { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
823 join .value_required:n = true ,
824 no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,

```

```

825 no-join .value_forbidden:n = true ,
826 left-margin .code:n =
827   \str_if_eq:nmTF { #1 } { auto }
828   {
829     \dim_zero:N \l_@@_left_margin_dim
830     \bool_set_true:N \l_@@_left_margin_auto_bool
831   }
832   {
833     \dim_set:Nn \l_@@_left_margin_dim { #1 }
834     \bool_set_false:N \l_@@_left_margin_auto_bool
835   } ,
836 left-margin .value_required:n = true ,
837
838 tab-size .int_set:N = \l_@@_tab_size_int ,
839 tab-size .value_required:n = true ,
840 show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
841 show-spaces .value_forbidden:n = true ,
842 show-spaces-in-strings .code:n =
843   \tl_set:Nn \l_@@_space_in_string_tl { } , % U+2423
844 show-spaces-in-strings .value_forbidden:n = true ,
845 break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
846 break-lines-in-Piton .default:n = true ,
847 break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
848 break-lines-in-piton .default:n = true ,
849 break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
850 break-lines .value_forbidden:n = true ,
851 indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
852 indent-broken-lines .default:n = true ,
853 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
854 end-of-broken-line .value_required:n = true ,
855 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
856 continuation-symbol .value_required:n = true ,
857 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
858 continuation-symbol-on-indentation .value_required:n = true ,
859
860 first-line .code:n = \@@_in_PitonInputFile:n
861   { \int_set:Nn \l_@@_first_line_int { #1 } } ,
862 first-line .value_required:n = true ,
863
864 last-line .code:n = \@@_in_PitonInputFile:n
865   { \int_set:Nn \l_@@_last_line_int { #1 } } ,
866 last-line .value_required:n = true ,
867
868 begin-range .code:n = \@@_in_PitonInputFile:n
869   { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
870 begin-range .value_required:n = true ,
871
872 end-range .code:n = \@@_in_PitonInputFile:n
873   { \str_set:Nn \l_@@_end_range_str { #1 } } ,
874 end-range .value_required:n = true ,
875
876 range .code:n = \@@_in_PitonInputFile:n
877   {
878     \str_set:Nn \l_@@_begin_range_str { #1 }
879     \str_set:Nn \l_@@_end_range_str { #1 }
880   } ,
881 range .value_required:n = true ,
882
883 env-used-by-split .code:n =
884   \lua_now:n { piton.env_used_by_split = '#1' } ,
885 env-used-by-split .initial:n = Piton ,
886
887 resume .meta:n = line-numbers/resume ,

```

```

888
889 unknown .code:n = \@@_error:n { Unknown-key-for-PitonOptions } ,
890
891 % deprecated
892 all-line-numbers .code:n =
893   \bool_set_true:N \l_@@_line_numbers_bool
894   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
895 rounded-corners .code:n =
896   \AtBeginDocument
897   {
898     \IfPackageLoadedTF { tikz }
899     { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
900     { \@@_err_rounded_corners_without_Tikz: }
901   } ,
902 rounded-corners .default:n = 4 pt
903 }
904 \hook_gput_code:nnn { begindocument } { . }
905 {
906   \IfPackageLoadedTF { tcolorbox }
907   {
908     \pgfkeysifdefined { / tcb / libload / breakable }
909     {
910       \keys_define:nn { PitonOptions }
911       {
912         tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
913         tcolorbox .default:n = true
914       }
915     }
916     {
917       \keys_define:nn { PitonOptions }
918       { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
919     }
920   }
921   {
922     \keys_define:nn { PitonOptions }
923     { tcolorbox .code:n = \@@_error:n { tcolorbox-not-loaded } }
924   }
925 }
926 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
927 {
928   \@@_error:n { rounded-corners-without-Tikz }
929   \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }
930 }
931 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
932 {
933   \bool_if:NTF \l_@@_in_PitonInputFile_bool
934   { #1 }
935   { \@@_error:n { Invalid-key } }
936 }
937 \NewDocumentCommand \PitonOptions { m }
938 {
939   \bool_set_true:N \l_@@_in_PitonOptions_bool
940   \keys_set:nn { PitonOptions } { #1 }
941   \bool_set_false:N \l_@@_in_PitonOptions_bool
942 }

```

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.

```
943 \NewDocumentCommand \@@_fake_PitonOptions { }
944   { \keys_set:nn { PitonOptions } }
```

2.0.6 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.

```
945 \int_new:N \g_@@_visual_line_int
946 \cs_new_protected:Npn \@@_incr_visual_line:
947   {
948     \bool_if:NF \l_@@_skip_empty_lines_bool
949     { \int_gincr:N \g_@@_visual_line_int }
950   }
951 \cs_new_protected:Npn \@@_print_number:
952   {
953     \hbox_overlap_left:n
954     {
955       {
956         \l_@@_line_numbers_format_tl
```

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

```
957     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
958     { \int_to_arabic:n \g_@@_visual_line_int }
959     \pdfextension literal { EMC }
960   }
961   \skip_horizontal:N \l_@@_numbers_sep_dim
962 }
963 }
```

2.0.7 The main commands and environments for the end user

```
964 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
965   {
966     \tl_if_novalue:nTF { #3 }
```

The last argument is provided by curryfication.

```
967     { \@@_NewPitonLanguage:nnn { #1 } { #2 } }
```

The two last arguments are provided by curryfication.

```
968     { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
969   }
```

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.

```
970 \prop_new:N \g_@@_languages_prop
971 \keys_define:nn { NewPitonLanguage }
972   {
973     morekeywords .code:n = ,
974     otherkeywords .code:n = ,
975     sensitive .code:n = ,
976     keywordsprefix .code:n = ,
977     moretexcs .code:n = ,
978     morestring .code:n = ,
979     morecomment .code:n = ,
980     moredelim .code:n = ,
```



```

981   moreredirectives .code:n = ,
982   tag .code:n = ,
983   alsodigit .code:n = ,
984   alsoletter .code:n = ,
985   alsoother .code:n = ,
986   unknown .code:n = \@_error:n { Unknown-key-NewPitonLanguage }
987 }

```

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

```

988 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
989 {

```

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}{...}`.

```

990   \tl_set:Ne \l_tmpa_tl
991   {
992     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
993     \str_lowercase:n { #2 }
994   }

```

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

```

995   \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.

```

996   \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).

```

997   \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
998 }

999 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1000 {
1001   \hook_gput_code:nnn { begindocument } { . }
1002   { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1003 }
1004 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

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

```

1005 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4 #5
1006 {

```

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}{...}`

```

1007   \tl_set:Ne \l_tmpa_tl
1008   {
1009     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1010     \str_lowercase:n { #4 }
1011   }

```

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`.

```

1012   \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

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

```

1013   { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
1014   { \@@_error:n { Language~not~defined } }
1015 }

```

```

1016 \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.

```

1017 { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1018 \cs_generate_variant:Nn \@@_NewPitonLanguage:mnnn { n n n o }

1019 \NewDocumentCommand { \piton } { }
1020 { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }

1021 \NewDocumentCommand { \@@_piton_standard } { m }
1022 {
1023   \group_begin:
1024   \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1025   {

```

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.

```

1026     \bool_lazy_or:nnT
1027     \l_@@_break_lines_in_piton_bool
1028     \l_@@_break_strings_anywhere_bool
1029     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1030   }

```

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

```

1031   \automaticallyphenmode = 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:

```

1032   \cs_set_eq:NN \ \ \c_backslash_str
1033   \cs_set_eq:NN \% \c_percent_str
1034   \cs_set_eq:NN \{ \c_left_brace_str
1035   \cs_set_eq:NN \} \c_right_brace_str
1036   \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.

```

1037   \cs_set_eq:cN { ~ } \space
1038   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1039   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1040   \tl_set:Ne \l_tmpa_tl
1041   {
1042     \lua_now:e
1043     { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
1044     { #1 }
1045   }
1046   \bool_if:NTF \l_@@_show_spaces_bool
1047   { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1048   {
1049     \bool_if:NT \l_@@_break_lines_in_piton_bool

```

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

```

1050     { \tl_replace_all:Nvn \l_tmpa_tl \c_catcode_other_space_tl \space }
1051   }

```

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

```

1052   \if_mode_math:
1053     \text { \l_@@_font_command_tl \l_tmpa_tl }
1054   \else:
1055     \l_@@_font_command_tl \l_tmpa_tl
1056   \fi:
1057   \group_end:
1058 }

```

```

1059 \NewDocumentCommand { \@@_piton_verbatim } { v }
1060 {
1061   \group_begin:
1062   \automatichyphenmode = 1
1063   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1064   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1065   \tl_set:Ne \l_tmpa_tl
1066   {
1067     \lua_now:e
1068     { piton.Parse('\l_piton_language_str',token.scan_string()) }
1069     { #1 }
1070   }
1071   \bool_if:NT \l_@@_show_spaces_bool
1072   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1073   \if_mode_math:
1074     \text { \l_@@_font_command_tl \l_tmpa_tl }
1075   \else:
1076     \l_@@_font_command_tl \l_tmpa_tl
1077   \fi:
1078   \group_end:
1079 }

```

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).

```

1080 \cs_new_protected:Npn \@@_piton:n #1
1081 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1082
1083 \cs_new_protected:Npn \@@_piton_i:n #1
1084 {
1085   \group_begin:
1086   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1087   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1088   \cs_set:cpn { pitonStyle _ Prompt } { }
1089   \cs_set_eq:NN \@@_leading_space: \space
1090   \cs_set_eq:NN \@@_trailing_space: \space
1091   \tl_set:Ne \l_tmpa_tl
1092   {
1093     \lua_now:e
1094     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1095     { #1 }
1096   }
1097   \bool_if:NT \l_@@_show_spaces_bool
1098   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1099   \@@_replace_spaces:o \l_tmpa_tl
1100   \group_end:
1101 }

```

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

```

1102 \cs_new_protected:Npn \@@_pre_composition:
1103 {
1104   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1105   {
1106     \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).

```

1107   \str_if_empty:NF \l_@@_box_str

```

```

1108     { \bool_set_true:N \l_@@_minimize_width_bool }
1109   }

```

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).

```

1110   \dim_set:Nn \l_@@_listing_width_dim
1111   {
1112     \bool_if:NTF \l_@@_tcolorbox_bool
1113     {
1114       \l_@@_width_dim -
1115       ( \kvtcb@left@rule
1116       + \kvtcb@left@upper
1117       + \kvtcb@boxsep * 2
1118       + \kvtcb@right@upper
1119       + \kvtcb@right@rule )
1120     }
1121     { \l_@@_width_dim }
1122   }
1123   \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1124   \automatichyphenmode = 1
1125   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1126   \g_@@_def_vertical_commands_tl
1127   \int_gzero:N \g_@@_line_int
1128   \int_gzero:N \g_@@_nb_lines_int
1129   \dim_zero:N \parindent
1130   \dim_zero:N \lineskip
1131   \dim_zero:N \parskip
1132   \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

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

```

1133   \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1134   { \bool_set_true:N \l_@@_bg_bool }
1135   \bool_gset_false:N \g_@@_rowcolor_inside_bool
1136   \IfPackageLoadedTF { zref-base }
1137   {
1138     \bool_if:NTF \g_@@_label_as_zlabel_bool
1139     { \cs_set_eq:NN \label \@@_zlabel:n }
1140     { \cs_set_eq:NN \label \@@_label:n }
1141     \cs_set_eq:NN \zlabel \@@_zlabel:n
1142   }
1143   { \cs_set_eq:NN \label \@@_label:n }
1144   \l_@@_font_command_tl
1145 }

```

If the end user has used both `left-margin=auto` and `line-numbers`, 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`.

```

1146 \cs_new_protected:Npn \@@_compute_left_margin:
1147 {
1148   \use:e
1149   {
1150     \bool_if:NTF \l_@@_skip_empty_lines_bool
1151     { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1152     { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1153     { \l_@@_listing_tl }
1154   }
1155   \hbox_set:Nn \l_tmpa_box
1156   {
1157     \l_@@_line_numbers_format_tl
1158     \int_to_arabic:n
1159     {
1160       \g_@@_visual_line_int
1161       +

```

```

1162         \bool_if:NTF \l_@@_skip_empty_lines_bool
1163         { \l_@@_nb_non_empty_lines_int }
1164         { \g_@@_nb_lines_int }
1165     }
1166 }
1167 \dim_set:Nn \l_@@_left_margin_dim
1168 { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1169 }

```

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:`.

```

1170 \cs_new_protected:Npn \@@_recompute_listing_width:
1171 {
1172     \dim_set:Nn \l_@@_listing_width_dim { \box_wd:N \g_@@_output_box }
1173     \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1174     {
1175         \dim_add:Nn \l_@@_listing_width_dim { 0.5 em }
1176         \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
1177         { \dim_add:Nn \l_@@_listing_width_dim { 0.5 em } }
1178         { \dim_add:Nn \l_@@_listing_width_dim \l_@@_left_margin_dim }
1179     }
1180     { \dim_add:Nn \l_@@_listing_width_dim \l_@@_left_margin_dim }
1181 }

```

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

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

```

1182 \cs_new_protected:Npn \@@_compute_code_width:
1183 {
1184     \dim_set_eq:NN \l_@@_code_width_dim \l_@@_listing_width_dim
1185     \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }

```

If there is a background (even a background with only the color `none`), we subtract 0.5 em for the margin on the right.

```

1186     {
1187         \dim_sub:Nn \l_@@_code_width_dim { 0.5 em }

```

And we subtract also for the left margin. If the key `left-margin` has been used (with a numerical value or with the special value `min`), `\l_@@_left_margin_dim` has a non-zero value³ and we use that value. Elsewhere, we use a value of 0.5 em.

```

1188         \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
1189         { \dim_sub:Nn \l_@@_code_width_dim { 0.5 em } }
1190         { \dim_sub:Nn \l_@@_code_width_dim \l_@@_left_margin_dim }
1191     }

```

If there is no background, we only subtract the left margin.

```

1192     { \dim_sub:Nn \l_@@_code_width_dim \l_@@_left_margin_dim }
1193 }

```

```

1194 \cs_new_protected:Npn \@@_store_body:n #1
1195 {

```

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

```

1196     \tl_set:Ne \obeyedline { \char_generate:nn { 13 } { 11 } }
1197     \tl_set:Ne \l_@@_listing_tl { #1 }
1198     \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1199 }

```

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

³If the key `left-margin` has been used with the special value `min`, the actual value of `\l_@@_left_margin_dim` has yet been computed when we use the current command.

```

1200 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nmnnn
1201 {
1202   \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1203   {
1204     \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1205     #4
1206     \@@_pre_composition:
1207     \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1208     {
1209       \int_gset:Nn \g_@@_visual_line_int
1210       { \l_@@_number_lines_start_int - 1 }
1211     }
1212     \bool_if:NT \g_@@_beamer_bool
1213     { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1214     \bool_if:NT \g_@@_footnote_bool \savenotes
1215     \@@_composition:
1216     \bool_lazy_or:nNT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1217     { \@@_create_paperclip_annotation: }
1218     \bool_if:NT \g_@@_footnote_bool \endsavenotes
1219     #5
1220   }
1221   { \ignorespacesafterend }
1222 }

```

```

1223 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1224 {
1225   \marginpar
1226   {
1227     \vspace* { - 0.8 em }
1228     \hbox:n
1229     {
1230       \bool_if:NT \l_@@_annotation_bool
1231       {
1232         \lua_now:n
1233         {

```

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!

```

1234         pdf.immediateobj
1235         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1236       }
1237     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1238     {
1239       /Subtype /Text
1240       /Contents~\pdf_object_ref_last:
1241       /Name /Note
1242       /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).

```

1243       /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1244       /F~512
1245       /C [0.8~0.8~0.8]
1246     }
1247     \hspace* { 7 mm }
1248   }
1249   \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1250 }
1251 }
1252 }

```

```

1253 \cs_new_protected:Npn \@@_create_paperclip:

```

```

1254 {
1255   \str_if_empty:NT \l_@@_paperclip_str
1256   {
1257     \int_gincr:N \g_@@_paperclip_int
1258     \str_set:Ne \l_@@_paperclip_str { listing_\int_use:N \g_@@_paperclip_int .txt }
1259   }

```

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

```

1260   \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1261   \box_move_down:nn
1262   { 10 pt }
1263   {
1264     \hbox:n
1265     {
1266       \pdfextension annot~width~10pt~height~20pt~depth~0pt
1267       {
1268         /Subtype /FileAttachment
1269         /Name /Paperclip
1270         /F-8 % no zoom

```

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

```

1271       /Contents (The~computer~listing)
1272       /FS <<
1273         /Type /Filespec
1274         /F (\l_@@_paperclip_str)
1275         /EF << /F~\pdf_object_ref_last: >>
1276         /AFRelationship /Supplement
1277       >>
1278     }
1279   }
1280 }
1281 }

```

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

```

1282 \NewDocumentCommand { \NewPitonEnvironment } { }
1283 { \@@_DefinePitonEnvironment:nnnnn { New } }
1284 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1285 { \@@_DefinePitonEnvironment:nnnnn { Declare } }
1286 \NewDocumentCommand { \RenewPitonEnvironment } { }
1287 { \@@_DefinePitonEnvironment:nnnnn { Renew } }
1288 \NewDocumentCommand { \ProvidePitonEnvironment } { }
1289 { \@@_DefinePitonEnvironment:nnnnn { Provide } }
1290 \cs_new_protected:Npn \@@_translate_beamer_env:n
1291 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1292 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }
1293 \cs_new_protected:Npn \@@_composition:
1294 {
1295   \str_if_empty:NT \l_@@_box_str
1296   {
1297     \mode_if_vertical:F
1298     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1299   }
1300   \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
1301   { \@@_compute_left_margin: }
1302   \lua_now:e
1303   {
1304     piton.join = "\l_@@_join_str"
1305     piton.write = "\l_@@_write_str"
1306     piton.path_write = "\l_@@_path_write_str"

```

```

1307     }
1308     \noindent
1309     \bool_if:NTF \l_@@_print_bool
1310     {

```

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.

```

1311     \bool_if:NTF \l_@@_split_on_empty_lines_bool
1312     { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1313     {
1314         \@@_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).

```

1315     \bool_if:NTF \l_@@_tcolorbox_bool
1316     {
1317         \str_if_empty:NTF \l_@@_box_str
1318         { \@@_composition_iii: }
1319         { \@@_composition_iv: }
1320     }
1321     {
1322         \str_if_empty:NTF \l_@@_box_str
1323         { \@@_composition_i: }
1324         { \@@_composition_ii: }
1325     }
1326 }
1327 }
1328 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1329 }

```

`\@@_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=...`

```

1330 \cs_new_protected:Npn \@@_composition_i:
1331 {

```

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.

```

1332     \box_clear:N \g_tmpa_box

```

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

```

1333     \box_clear_new:N \g_@@_line_box

```

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

```

1334     \vbox_set:Nn \l_tmpa_box
1335     {
1336         \vbox_unpack_drop:N \g_@@_output_box
1337         \bool_gset_false:N \g_tmpa_bool
1338         \unskip \unskip
1339         \bool_gset_false:N \g_tmpa_bool
1340         \bool_do_until:nn \g_tmpa_bool
1341         {
1342             \unskip \unskip \unskip
1343             \unpenalty \unkern
1344             \box_set_to_last:N \l_@@_line_box
1345             \box_if_empty:NTF \l_@@_line_box
1346             { \bool_gset_true:N \g_tmpa_bool }
1347             {
1348                 \vbox_gset:Nn \g_tmpa_box
1349                 {
1350                     \vbox_unpack:N \g_tmpa_box

```



```

1351             \box_use:N \l_@@_line_box
1352         }
1353     }
1354 }
1355 }

```

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

```

1356 \bool_gset_false:N \g_tmpa_bool
1357 \int_zero:N \g_@@_line_int
1358 \bool_do_until:nn \g_tmpa_bool
1359 {

```

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`.

```

1360     \vbox_gset:Nn \g_tmpa_box
1361     {
1362         \vbox_unpack_drop:N \g_tmpa_box
1363         \box_gset_to_last:N \g_@@_line_box
1364     }

```

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.

```

1365     \box_if_empty:NTF \g_@@_line_box
1366     { \bool_gset_true:N \g_tmpa_bool }
1367     {
1368         \box_use:N \g_@@_line_box
1369         \int_gincr:N \g_@@_line_int
1370         \par
1371         \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`.

```

1372     \@@_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.

```

1373     \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 }
1374     { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 } }
1375     \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1376     { \mode_leave_vertical: }
1377 }
1378 }
1379 \skip_vertical:n { 2.5 pt }
1380 }

```

`\@@_composition_ii`: will be used when the key `box` is in force.

```

1381 \cs_new_protected:Npn \@@_composition_ii:
1382 {
1383     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1384     { \l_@@_listing_width_dim }

```

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

```

1385     \vbox_unpack:N \g_@@_output_box

```

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

```

1386     \kern 2.5 pt
1387     \end { minipage }
1388 }

```

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

```

1389 \cs_new_protected:Npn \@@_composition_iii:
1390 {
1391     \use:e
1392     {
1393         \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.

```

1394     [ breakable , text-width = \l_@@_listing_width_dim ]
1395   }
1396   \par
1397   \vbox_unpack:N \g_@@_output_box
1398   \end { tcolorbox }
1399 }

```

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

```

1400 \cs_new_protected:Npn \@@_composition_iv:
1401 {
1402   \use:e
1403   {
1404     \begin { tcolorbox }
1405       [
1406         hbox ,
1407         text-width = \l_@@_listing_width_dim ,
1408         nobeforeafter ,
1409         box-align =
1410         \str_case:Nn \l_@@_box_str
1411         {
1412           t { top }
1413           b { bottom }
1414           c { center }
1415           m { center }
1416         }
1417       ]
1418     }
1419     \box_use:N \g_@@_output_box
1420     \end { tcolorbox }
1421   }

```

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.

```

1422 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1423 {
1424   \int_case:nn
1425   {
1426     \lua_now:e
1427     {
1428       tex.sprint
1429       ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1430     }
1431   }
1432   { 1 { \penalty 100 } 2 \nobreak }
1433 }

```

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

It creates (and modify when there are backgrounds) `\g_@@_output_box`.

```

1434 \cs_new_protected:Npn \@@_create_output_box:
1435 {
1436   \@@_compute_code_width:
1437   \vbox_gset:Nn \g_@@_output_box
1438   { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1439   \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1440   \bool_lazy_or:nnT
1441   { \int_compare_p:nNn \l_@@_bg_colors_int > { \c_zero_int } }
1442   { \g_@@_rowcolor_inside_bool }
1443   { \@@_add_backgrounds_to_output_box: }
1444 }

```

We add the backgrounds after the composition of the box `\g_@@_output_box` by a loop over the lines in that box. The backgrounds will have a width equal to `\l_@@_listing_width_dim`. That command will be used only once, in `\@@_create_output_box:`.

```

1445 \cs_new_protected:Npn \@@_add_backgrounds_to_output_box:
1446   {
1447     \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`.

```

1448     \vbox_set:Nn \l_tmpa_box
1449     {
1450       \vbox_unpack_drop:N \g_@@_output_box

```

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

```

1451       \bool_gset_false:N \g_tmpa_bool
1452       \unskip \unskip

```

We begin the loop.

```

1453     \bool_do_until:nn \g_tmpa_bool
1454     {
1455       \unskip \unskip \unskip
1456       \int_set_eq:NN \l_tmpa_int \lastpenalty
1457       \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...

```

1458       \box_set_to_last:N \l_@@_line_box
1459       \box_if_empty:NTF \l_@@_line_box
1460         { \bool_gset_true:N \g_tmpa_bool }
1461       {

```

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

```

1462         \vbox_gset:Nn \g_@@_output_box
1463         {

```

The command `\@@_add_background_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`.

```

1464         \@@_add_background_to_line_and_use:
1465         \kern -2.5 pt
1466         \penalty \l_tmpa_int
1467         \vbox_unpack:N \g_@@_output_box
1468         }
1469       }
1470       \int_gdecr:N \g_@@_line_int
1471     }
1472   }
1473 }

```

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

```

1474 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1475   {
1476     \lua_now:e
1477     {
1478       piton.GobbleParseNoPrint
1479       (
1480         '\l_piton_language_str' ,
1481         \int_use:N \l_@@_gobble_int ,
1482         token.scan_argument ( )
1483       )
1484     }
1485   }
1486 \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.

```

1487 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1488 {
1489   \lua_now:e
1490   {
1491     piton.RetrieveGobbleParse
1492     (
1493       '\l_piton_language_str' ,
1494       \int_use:N \l_@@_gobble_int ,
1495       \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1496         { \int_eval:n { - \l_@@_splittable_int } }
1497         { \int_use:N \l_@@_splittable_int } ,
1498       token.scan_argument ( )
1499     )
1500   }
1501 }
1502 \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.

```

1503 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1504 {
1505   \lua_now:e
1506   {
1507     piton.RetrieveGobbleSplitParse
1508     (
1509       '\l_piton_language_str' ,
1510       \int_use:N \l_@@_gobble_int ,
1511       \int_use:N \l_@@_splittable_int ,
1512       token.scan_argument ( )
1513     )
1514   }
1515 }
1516 \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`.

```

1517 \bool_if:NTF \g_@@_beamer_bool
1518 {
1519   \NewPitonEnvironment { Piton } { D < > { .- } 0 { } }
1520   {
1521     \keys_set:nn { PitonOptions } { #2 }
1522     \begin { actionenv } < #1 >
1523   }
1524   { \end { actionenv } }
1525 }
1526 {
1527   \NewPitonEnvironment { Piton } { 0 { } }
1528   { \keys_set:nn { PitonOptions } { #1 } }
1529   { }
1530 }

```

```

1531 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1532 {
1533   \group_begin:
1534   \seq_concat:NNN
1535     \l_file_search_path_seq
1536     \l_@@_path_seq
1537     \l_file_search_path_seq
1538   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1539   {
1540     \@@_input_file:nn { #1 } { #2 }

```

```

1541     #4
1542   }
1543   { #5 }
1544   \group_end:
1545 }

1546 \cs_new_protected:Npn \@@_unknown_file:n #1
1547   { \msg_error:nnn { piton } { Unknown-file } { #1 } }
1548 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1549   {
1550     \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
1551     {

```

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

```

1552     \iow_log:n { No-file-#3 }
1553     \@@_unknown_file:n { #3 }
1554   }
1555 }
1556 \NewDocumentCommand { \PitonInputFileT } { d < > 0 { } m m }
1557   {
1558     \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1559     {

```

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

```

1560     \iow_log:n { No-file-#3 }
1561     \@@_unknown_file:n { #3 }
1562   }
1563 }
1564 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1565   { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1566 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1567   {

```

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 >).

```

1568   \tl_if_novalue:nF { #1 }
1569   {
1570     \bool_if:NTF \g_@@_beamer_bool
1571     { \begin { uncoverenv } < #1 > }
1572     { \@_error_or_warning:n { overlay~without~beamer } }
1573   }
1574   \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.

```

1575   \iow_log:e { (\l_@@_file_name_str) }
1576   \int_zero_new:N \l_@@_first_line_int
1577   \int_zero_new:N \l_@@_last_line_int
1578   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1579   \bool_set_true:N \l_@@_in_PitonInputFile_bool
1580   \keys_set:nn { PitonOptions } { #2 }
1581   \bool_if:NT \l_@@_line_numbers_absolute_bool
1582     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1583   \bool_if:NTF
1584     {
1585       (
1586         \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1587         || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1588       )
1589       && ! \str_if_empty_p:N \l_@@_begin_range_str
1590     }
1591     {
1592       \@_error_or_warning:n { bad-range-specification }
1593       \int_zero:N \l_@@_first_line_int

```

```

1594     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1595   }
1596   {
1597     \str_if_empty:NF \l_@@_begin_range_str
1598     {
1599       \@@_compute_range:
1600       \bool_lazy_or:nnT
1601         \l_@@_marker_include_lines_bool
1602         { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1603       {
1604         \int_decr:N \l_@@_first_line_int
1605         \int_incr:N \l_@@_last_line_int
1606       }
1607     }
1608   }
1609   \@@_pre_composition:
1610   \bool_if:NT \l_@@_line_numbers_absolute_bool
1611     { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1612   \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1613     {
1614       \int_gset:Nn \g_@@_visual_line_int
1615         { \l_@@_number_lines_start_int - 1 }
1616     }

```

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

```

1617   \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1618     { \int_gzero:N \g_@@_visual_line_int }
1619   \lua_now:e
1620   {

```

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

```

1621     piton.ReadFile(
1622       '\l_@@_file_name_str' ,
1623       \int_use:N \l_@@_first_line_int ,
1624       \int_use:N \l_@@_last_line_int )
1625   }
1626   \@@_composition:
1627   \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.

```

1628   \tl_if_novalue:nF { #1 }
1629     { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1630   }

```

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

```

1631   \cs_new_protected:Npn \@@_compute_range:
1632   {

```

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

```

1633   \str_set:Nc \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1634   \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`.

```

1635   \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1636   \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1637   \lua_now:e
1638   {
1639     piton.ComputeRange
1640     ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1641   }

```

```
1642 }
```

2.0.8 The styles

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

```
1643 \NewDocumentCommand { \PitonStyle } { m }
1644 {
1645   \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1646   { \use:c { pitonStyle _ #1 } }
1647 }
```

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”.

```
1648 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1649 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1650 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1651 {
1652   \str_clear_new:N \l_@@_SetPitonStyle_option_str
1653   \str_set:Ne \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1654   \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1655   { \str_set:eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1656   \keys_set:mn { piton / Styles } { #2 }
1657 }
```

```
1658 \cs_new_protected:Npn \@@_math_scantokens:n #1
1659 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }
```

```
1660 \clist_new:N \g_@@_styles_clist
1661 \clist_gset:Nn \g_@@_styles_clist
1662 {
1663   Comment ,
1664   Comment.Internal ,
1665   Comment.LaTeX ,
1666   Discard ,
1667   Exception ,
1668   FormattingType ,
1669   Identifier.Internal ,
1670   Identifier ,
1671   InitialValues ,
1672   Interpol.Inside ,
1673   Keyword ,
1674   Keyword.Governing ,
1675   Keyword.Constant ,
1676   Keyword2 ,
1677   Keyword3 ,
1678   Keyword4 ,
1679   Keyword5 ,
1680   Keyword6 ,
1681   Keyword7 ,
1682   Keyword8 ,
1683   Keyword9 ,
1684   Name.Builtin ,
1685   Name.Class ,
1686   Name.Constructor ,
1687   Name.Decorator ,
1688   Name.Field ,
1689   Name.Function ,
1690   Name.Module ,
1691   Name.Namespace ,
```

```

1692 Name.Table ,
1693 Name.Type ,
1694 Number ,
1695 Number.Internal ,
1696 Operator ,
1697 Operator.Word ,
1698 Preproc ,
1699 Prompt ,
1700 String.Doc ,
1701 String.Doc.Internal ,
1702 String.Interpol ,
1703 String.Long ,
1704 String.Long.Internal ,
1705 String.Short ,
1706 String.Short.Internal ,
1707 Tag ,
1708 TypeParameter ,
1709 UserFunction ,

```

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

```

1710 TypeExpression ,

```

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

```

1711 Directive
1712 }
1713 \clist_map_inline:Nn \g_@@_styles_clist
1714 {
1715   \keys_define:nn { piton / Styles }
1716   {
1717     #1 .value_required:n = true ,
1718     #1 .code:n =
1719       \tl_set:cn
1720       {
1721         pitonStyle _
1722         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1723         { \l_@@_SetPitonStyle_option_str _ }
1724         #1
1725       }
1726     { ##1 }
1727   }
1728 }
1729
1730 \keys_define:nn { piton / Styles }
1731 {
1732   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1733   String      .value_required:n = true ,
1734   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1735   Comment.Math .value_required:n = true ,
1736   unknown     .code:n = \@@_unknown_style:
1737 }

```

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.

```

1738 \cs_new_protected:Npn \@@_unknown_style:
1739 {
1740   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1741   {
1742     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1743     \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`.

```

1744     \bool_lazy_and:nnTF
1745     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }

```



```

1746     {
1747         \str_if_eq_p:Vn \l_tmpa_str { Module }
1748         ||
1749         \str_if_eq_p:Vn \l_tmpa_str { Type }
1750     }

```

Now, we will create a new style.

```

1751     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1752     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1753 }
1754 { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1755 }

```

```

1756 \SetPitonStyle[OCaml]
1757 {
1758   TypeExpression =
1759   {
1760     \SetPitonStyle [ OCaml ] { Identifier = \PitonStyle { Name.Type } }
1761     \@@_piton:n
1762   }
1763 }

```

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`.

```

1764 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that clist.

```

1765 \clist_gsort:Nn \g_@@_styles_clist
1766 {
1767   \str_compare:nNnTF { #1 } < { #2 }
1768   \sort_return_same:
1769   \sort_return_swapped:
1770 }

```

```

1771 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1772
1773 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1774
1775 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1776 {
1777   \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`.

```

1778   \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1779   \seq_clear:N \l_tmpa_seq
1780   \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1781   \seq_use:Nn \l_tmpa_seq { \- }
1782 }

```

```

1783 \cs_new_protected:Npn \@@_comment:n #1
1784 {
1785   \PitonStyle { Comment }
1786   {
1787     \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1788     {
1789       \tl_set:Nn \l_tmpa_tl { #1 }
1790       \tl_replace_all:NVn \l_tmpa_tl
1791         \c_catcode_other_space_tl
1792         \@@_breakable_space:

```

```

1793         \l_tmpa_tl
1794     }
1795     { #1 }
1796 }
1797 }

1798 \cs_new_protected:Npn \@@_string_long:n #1
1799 {
1800     \PitonStyle { String.Long }
1801     {
1802         \bool_if:NTF \l_@@_break_strings_anywhere_bool
1803         { \@@_actually_break_anywhere:n { #1 } }
1804     }

```

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.

```

1805         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1806         {
1807             \tl_set:Nn \l_tmpa_tl { #1 }
1808             \tl_replace_all:NVn \l_tmpa_tl
1809             \c_catcode_other_space_tl
1810             \@@_breakable_space:
1811             \l_tmpa_tl
1812         }
1813         { #1 }
1814     }
1815 }
1816 }

1817 \cs_new_protected:Npn \@@_string_short:n #1
1818 {
1819     \PitonStyle { String.Short }
1820     {
1821         \bool_if:NT \l_@@_break_strings_anywhere_bool
1822         { \@@_actually_break_anywhere:n }
1823         { #1 }
1824     }
1825 }

1826 \cs_new_protected:Npn \@@_string_doc:n #1
1827 {
1828     \PitonStyle { String.Doc }
1829     {
1830         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1831         {
1832             \tl_set:Nn \l_tmpa_tl { #1 }
1833             \tl_replace_all:NVn \l_tmpa_tl
1834             \c_catcode_other_space_tl
1835             \@@_breakable_space:
1836             \l_tmpa_tl
1837         }
1838         { #1 }
1839     }
1840 }

1841 \cs_new_protected:Npn \@@_number:n #1
1842 {
1843     \PitonStyle { Number }
1844     {
1845         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1846         { \@@_actually_break_anywhere:n }
1847         { #1 }
1848     }

```

```
1849 }
```

2.0.9 The initial styles

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

```
1850 \SetPitonStyle
1851 {
1852   Comment                = \color [ HTML ] { 0099FF } \itshape ,
1853   Comment.Internal       = \@_comment:n ,
1854   Exception              = \color [ HTML ] { CC0000 } ,
1855   Keyword                = \color [ HTML ] { 006699 } \bfseries ,
1856   Keyword.Governing      = \color [ HTML ] { 006699 } \bfseries ,
1857   Keyword.Constant       = \color [ HTML ] { 006699 } \bfseries ,
1858   Name.Builtin           = \color [ HTML ] { 336666 } ,
1859   Name.Decorator         = \color [ HTML ] { 9999FF } ,
1860   Name.Class             = \color [ HTML ] { 00AA88 } \bfseries ,
1861   Name.Function          = \color [ HTML ] { CC00FF } ,
1862   Name.Namespace        = \color [ HTML ] { 00CCFF } ,
1863   Name.Constructor       = \color [ HTML ] { 006000 } \bfseries ,
1864   Name.Field             = \color [ HTML ] { AA6600 } ,
1865   Name.Module            = \color [ HTML ] { 0060A0 } \bfseries ,
1866   Name.Table             = \color [ HTML ] { 309030 } ,
1867   Number                 = \color [ HTML ] { FF6600 } ,
1868   Number.Internal       = \@_number:n ,
1869   Operator               = \color [ HTML ] { 555555 } ,
1870   Operator.Word         = \bfseries ,
1871   String                 = \color [ HTML ] { CC3300 } ,
1872   String.Long.Internal  = \@_string_long:n ,
1873   String.Short.Internal = \@_string_short:n ,
1874   String.Doc.Internal   = \@_string_doc:n ,
1875   String.Doc            = \color [ HTML ] { CC3300 } \itshape ,
1876   String.Interpol       = \color [ HTML ] { AA0000 } ,
1877   Comment.LaTeX         = \normalfont \color [ rgb ] { .468, .532, .6 } ,
1878   Name.Type             = \color [ HTML ] { 336666 } ,
1879   InitialValues         = \@_piton:n ,
1880   Interpol.Inside       = { \l_@_font_command_tl \@_piton:n } ,
1881   TypeParameter         = \color [ HTML ] { 336666 } \itshape ,
1882   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).

```
1883   Identifier.Internal   = \@_identifier:n ,
1884   Identifier            = ,
1885   Directive            = \color [ HTML ] { AA6600 } ,
1886   Tag                  = \colorbox { gray!10 } ,
1887   UserFunction         = \PitonStyle { Identifier } ,
1888   Prompt               = ,
1889   Discard              = \use_none:n
1890 }
```

2.0.10 Styles specific to the language expl

```
1891 \clist_new:N \g_@_expl_styles_clist
1892 \clist_gset:Nn \g_@_expl_styles_clist
1893 {
1894   Scope.l ,
1895   Scope.g ,
1896   Scope.c
1897 }
```

```

1898 \clist_map_inline:Nn \g_@@_expl_styles_clist
1899 {
1900   \keys_define:nm { piton / Styles }
1901   {
1902     #1 .value_required:n = true ,
1903     #1 .code:n =
1904       \tl_set:cn
1905       {
1906         pitonStyle _
1907         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1908         { \l_@@_SetPitonStyle_option_str _ }
1909         #1
1910       }
1911     { ##1 }
1912   }
1913 }
1914 \SetPitonStyle [ expl ]
1915 {
1916   Scope.l           = ,
1917   Scope.g           = \bfseries ,
1918   Scope.c           = \slshape ,
1919   Type.bool         = \color [ HTML ] { AA6600 } ,
1920   Type.box          = \color [ HTML ] { 267910 } ,
1921   Type.clist        = \color [ HTML ] { 309030 } ,
1922   Type.fp           = \color [ HTML ] { FF3300 } ,
1923   Type.int          = \color [ HTML ] { FF6600 } ,
1924   Type.seq          = \color [ HTML ] { 309030 } ,
1925   Type.skip         = \color [ HTML ] { OCC060 } ,
1926   Type.str          = \color [ HTML ] { CC3300 } ,
1927   Type.tl           = \color [ HTML ] { AA2200 } ,
1928   Module.bool       = \color [ HTML ] { AA6600 } ,
1929   Module.box        = \color [ HTML ] { 267910 } ,
1930   Module.cs         = \bfseries \color [ HTML ] { 006699 } ,
1931   Module.exp        = \bfseries \color [ HTML ] { 404040 } ,
1932   Module.hbox       = \color [ HTML ] { 267910 } ,
1933   Module.prg        = \bfseries ,
1934   Module.clist      = \color [ HTML ] { 309030 } ,
1935   Module.fp         = \color [ HTML ] { FF3300 } ,
1936   Module.int        = \color [ HTML ] { FF6600 } ,
1937   Module.seq        = \color [ HTML ] { 309030 } ,
1938   Module.skip       = \color [ HTML ] { OCC060 } ,
1939   Module.str        = \color [ HTML ] { CC3300 } ,
1940   Module.tl         = \color [ HTML ] { AA2200 } ,
1941   Module.vbox       = \color [ HTML ] { 267910 }
1942 }

```

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).

```

1943 \hook_gput_code:nnn { begindocument } { . }
1944 {
1945   \bool_if:NT \g_@@_math_comments_bool
1946     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
1947 }

```

2.0.11 Highlighting some identifiers

```

1948 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
1949 {
1950   \clist_set:Nn \l_tmpa_clist { #2 }
1951   \tl_if_novalue:nTF { #1 }
1952     {
1953       \clist_map_inline:Nn \l_tmpa_clist

```

```

1954     { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
1955   }
1956   {
1957     \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
1958     \str_if_eq:onT \l_tmpa_str { current-language }
1959     { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
1960     \clist_map_inline:Nn \l_tmpa_clist
1961     { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
1962   }
1963 }
1964 \cs_new_protected:Npn \@@_identifier:n #1
1965 {
1966   \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }
1967   {
1968     \cs_if_exist_use:cF { PitonIdentifier _ #1 }
1969     { \PitonStyle { Identifier } }
1970   }
1971   { #1 }
1972 }

```

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 (you define it directly and you short-cut the function `\SetPitonStyle`).

```

1973 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
1974 {

```

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

```

1975   { \PitonStyle { Name.Function } { #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}`.

```

1976   \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ #1 }
1977   { \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`.**

```

1978   \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
1979   { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
1980   \seq_gput_right:cn { g_@@_functions _ \l_piton_language_str _ seq } { #1 }

```

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

```

1981   \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }
1982   { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
1983 }

```

```

1984 \NewDocumentCommand \PitonClearUserFunctions { ! o }
1985 {
1986   \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.

```

1987   { \@@_clear_all_functions: }
1988   { \@@_clear_list_functions:n { #1 } }
1989 }

```

```

1990 \cs_new_protected:Npn \@@_clear_list_functions:n #1
1991 {
1992   \clist_set:Nn \l_tmpa_clist { #1 }
1993   \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
1994   \clist_map_inline:nn { #1 }

```

```

1995     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
1996 }

```

```

1997 \cs_new_protected:Npn \@@_clear_functions_i:n #1
1998 { \@@_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).

```

1999 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2000 {
2001   \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2002   {
2003     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2004     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2005     \seq_gclear:c { g_@@_functions _ #1 _ seq }
2006   }
2007 }
2008 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

```

```

2009 \cs_new_protected:Npn \@@_clear_functions:n #1
2010 {
2011   \@@_clear_functions_i:n { #1 }
2012   \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2013 }

```

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

```

2014 \cs_new_protected:Npn \@@_clear_all_functions:
2015 {
2016   \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2017   \seq_gclear:N \g_@@_languages_seq
2018 }

```

```

2019 \AtEndDocument
2020 {

```

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

```

2021   \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.

```

2022   \IfPDFManagementActiveTF
2023   { \@@_join_files: }
2024   { \@@_join_files_legacy: }
2025 }

```

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

```

2026 % \cs_new_protected:Npn \@@_join_files:
2027 % {
2028 %   \seq_map_inline:Nn \g_@@_join_seq
2029 %   {
2030 %

```

The group is for the modifications of the the dictionary `l_pdffile/Filespec`

```

2031 %   \group_begin:
2032 %

```

We create a new PDF object but, in fact, it won't be really used.

```

2033 %   \pdf_object_new:n { piton / join / ##1 }
2034 %

```

The stream of the file is created by Lua (in the Lua side of LuaLaTeX) but the file itself will be added to the catalog of the PDF file in the LaTeX part of LuaLaTeX by using the command `\pdfmanagement_add:nne` of `pdfmanagement`.

```

2035 %   \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2036 %

```

The value of the key /AFRelationship must be a name of PDF (beginning with a solidus).

```
2037 %         \pdfdict_put:nnn { l_pdffile / Filespec } { AFRelationship } { /Supplement }
2038 %
```

The value of the key /Desc must be a string of PDF (between parenthesis).

```
2039 %         \pdfdict_put:nnn { l_pdffile / Filespec } { Desc } { (Computer~listing) }
2040 %         \pdffile_filespec:nnn { piton / join / ##1 } { ##1 } { \pdf_object_ref_last: }
2041 %
```

It's mandatory to use \pdfmanagement_add:nne to add to the catalog of the PDF in order to avoid clashes with other extensions writing to the catalog.

```
2042 %         \pdfmanagement_add:nne
2043 %         { Catalog / Names }
2044 %         { EmbeddedFiles }
2045 %         { \pdf_object_ref_last: }
2046 %     \group_end:
2047 % }
2048 % }
2049 %
```

Here is a version of the previous code with a direct code for the PDF dictionary /FileSpec.

```
2050 \cs_new_protected:Npn \@@_join_files:
2051 {
2052     \seq_map_inline:Nn \g_@@_join_seq
2053     {
2054         \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2055         \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2056         \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2057         {
2058             <<
2059                 /Type /Filespec
2060                 /UF <\l_tmpa_str>
2061                 /EF << /F~\pdf_object_ref_last: >>
2062                 /Desc (Computer~listing)
2063                 /AFRelationship /Supplement
2064             >>
2065         }
2066     }
2067 }
```

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 techniques.

```
2068 \cs_new_protected:Npn \@@_join_files_legacy:
2069 {
2070     \seq_map_inline:Nn \g_@@_join_seq
2071     {
2072         \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2073         \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2074         \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.

```
2075     {
2076         /Subtype /FileAttachment
2077         /F~2
2078         /Name /Paperclip
2079         /Contents (Computer~listing)
2080         /FS <<
2081             /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).

```
2082         /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`).

```
2083         /EF << /F~\pdf_object_ref_last: >>
2084         /AFRelationship /Supplement
2085     >>
2086     }
2087 }
2088 }
```

2.0.12 Spaces of indentation

```
2089 \cs_new_protected:Npn \@@_define_leading_space_normal:
2090 {
2091     \cs_set_protected:Npn \@@_leading_space:
2092     {
2093         \int_gincr:N \g_@@_indentation_int
```

Be careful: the `\hbox:n` is mandatory.

```
2094         \hbox:n { ~ }
2095     }
2096 }
```

```
2097 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2098 {
2099     \cs_set_protected:Npn \@@_leading_space:
2100     {
2101         \int_gincr:N \g_@@_indentation_int
2102         \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2103         {
2104             \color { white }
2105             \transparent { 0 }
2106             . % previously : □ U+2423
2107         }
2108         \pdfextension literal { EMC }
2109     }
2110 }
2111 \@@_define_leading_space_Foxit:
```

2.0.13 Security

```
2112 \AddToHook { env / piton / before }
2113 { \@@_fatal:n { No~environment~piton } }
```

2.0.14 The error messages of the package

```
2114 \@@_msg_new:nn { No~environment~piton }
2115 {
2116     There-is~no~environment~piton!\\
2117     There-is~an~environment~{Piton}~and~a~command~
2118     \token_to_str:N \piton\ but~there-is-no~environment~
2119     {piton}.~This~error-is~fatal.
2120 }
2121 \@@_msg_new:nn { rounded-corners~without~Tikz }
2122 {
2123     TikZ-not~used \\
2124     You~can't~use~the~key~'rounded-corners'~because~
2125     you~have~not~loaded~the~package~TikZ. \\
2126     If~you~go~on,~that~key~will~be~ignored. \\
```



```

2127     You-won't-have-similar-error-till-the-end-of-the-document.
2128 }
2129 \@@_msg_new:nn { tcolorbox-not-loaded }
2130 {
2131     tcolorbox-not-loaded \\
2132     You-can't-use-the-key~'tcolorbox'~because~
2133     you-have-not-loaded-the-package~tcolorbox. \\
2134     Use~\token_to_str:N \usepackage[breakable]{tcolorbox}. \\
2135     If~you-go-on,~that~key~will~be~ignored.
2136 }
2137 \@@_msg_new:nn { library-breakable-not-loaded }
2138 {
2139     breakable-not-loaded \\
2140     You-can't-use-the-key~'tcolorbox'~because~
2141     you-have-not-loaded-the-library~'breakable'~of~tcolorbox'. \\
2142     Use~\token_to_str:N \tcbuselibrary{breakable}~in-the-preamble~
2143     of~your~document.\\
2144     If~you-go-on,~that~key~will~be~ignored.
2145 }
2146 \@@_msg_new:nn { Language-not-defined }
2147 {
2148     Language-not-defined \\
2149     The~language~'\l_tmpa_tl'~has-not-been-defined~previously.\\
2150     If~you-go-on,~your~command~\token_to_str:N \NewPitonLanguage\
2151     will~be~ignored.
2152 }
2153 \@@_msg_new:nn { bad-version-of-piton.lua }
2154 {
2155     Bad-number-version-of~'piton.lua'\\
2156     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2157     version~as~the~file~'piton.sty'~.~You~can~go~on~but~you~should~
2158     address~that~issue.
2159 }
2160 \@@_msg_new:nn { Unknown-key-NewPitonLanguage }
2161 {
2162     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
2163     The~key~'\l_keys_key_str'~is~unknown.\\
2164     This~key~will~be~ignored.\\
2165 }
2166 \@@_msg_new:nn { Unknown-key-for-SetPitonStyle }
2167 {
2168     The~style~'\l_keys_key_str'~is~unknown.\\
2169     This~setting~will~be~ignored.\\
2170     The~available~styles~are~(in~alphabetic~order):~
2171     \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2172 }
2173 \@@_msg_new:nn { Invalid-key }
2174 {
2175     Wrong-use-of~key.\\
2176     You~can't~use~the~key~'\l_keys_key_str'~here.\\
2177     That~key~will~be~ignored.
2178 }
2179 \@@_msg_new:nn { Unknown-key-for-line-numbers }
2180 {
2181     Unknown~key. \\
2182     The~key~'line-numbers / \l_keys_key_str'~is~unknown.\\
2183     The~available~keys~of~the~family~'line-numbers'~are~(in~
2184     alphabetic~order):~
2185     absolute,~false,~label-empty-lines,~resume,~skip-empty-lines,~
2186     sep,~start~and~true.\\

```

```

2187     That~key~will~be~ignored.
2188   }
2189 \@@_msg_new:nn { Unknown~key~for~marker }
2190   {
2191     Unknown~key. \\
2192     The~key~'marker / \l_keys_key_str'~is~unknown.\\
2193     The~available~keys~of~the~family~'marker'~are~(in~
2194     alphabetic~order):~ beginning,~end~and~include~lines.\\
2195     That~key~will~be~ignored.
2196   }
2197 \@@_msg_new:nn { bad~range~specification }
2198   {
2199     Incompatible~keys.\\
2200     You~can't~specify~the~range~of~lines~to~include~by~using~both~
2201     markers~and~explicit~number~of~lines.\\
2202     Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2203   }
2204 \cs_new_nopar:Nn \@@_thepage:
2205   {
2206     \thepage
2207     \cs_if_exist:NT \insertframenummer
2208       {
2209         ~(frame~\insertframenummer
2210         \cs_if_exist:NT \beamer@slidenummer { ,~slide~\insertslidenummer }
2211         )
2212       }
2213   }

```

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`.

```

2214 \@@_msg_new:nn { SyntaxError }
2215   {
2216     Syntax~Error~on~page~\@@_thepage:. \\
2217     Your~code~of~the~language~'\l_piton_language_str'~is~not~
2218     syntactically~correct.\\
2219     It~won't~be~printed~in~the~PDF~file.
2220   }
2221 \@@_msg_new:nn { FileError }
2222   {
2223     File~Error.\\
2224     It's~not~possible~to~write~on~the~file~'#1' \\
2225     \sys_if_shell_unrestricted:F
2226     { (try~to~compile~with~'lualatex~--shell-escape'). \\ }
2227     If~you~go~on,~nothing~will~be~written~on~that~file.
2228   }
2229 \@@_msg_new:nn { InexistentDirectory }
2230   {
2231     Inexistent~directory.\\
2232     The~directory~'\l_@@_path_write_str'~
2233     given~in~the~key~'path-write'~does~not~exist.\\
2234     Nothing~will~be~written~on~'\l_@@_write_str'.
2235   }
2236 \@@_msg_new:nn { begin~marker~not~found }
2237   {
2238     Marker~not~found.\\
2239     The~range~'\l_@@_begin_range_str'~provided~to~the~
2240     command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2241     The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2242   }
2243 \@@_msg_new:nn { end~marker~not~found }

```

```

2244 {
2245   Marker~not~found.\\
2246   The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2247   provided~to~the~command~\token_to_str:N \PitonInputFile\
2248   has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2249   be~inserted~till~the~end.
2250 }
2251 \@@_msg_new:nn { Unknown~file }
2252 {
2253   Unknown~file. \\
2254   The~file~'#1'~is~unknown.\\
2255   Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
2256 }
2257 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2258 {
2259   \bool_if:NF \g_@@_beamer_bool
2260     { \@@_error_or_warning:n { Without~beamer } }
2261 }
2262 \@@_msg_new:nn { Without~beamer }
2263 {
2264   Key~'\l_keys_key_str'~without~Beamer.\\
2265   You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2266   are~not~in~Beamer.\\
2267   However,~you~can~go~on.
2268 }
2269 \@@_msg_new:nn { rowcolor~in~detected~commands }
2270 {
2271   'rowcolor'~forbidden~in~'detected~commands'.\\
2272   You~should~put~'rowcolor'~in~'raw~detected~commands'.\\
2273   That~key~will~be~ignored.
2274 }
2275 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2276 {
2277   Unknown~key. \\
2278   The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2279   It~will~be~ignored.\\
2280   For~a~list~of~the~available~keys,~type~H~<return>.
2281 }
2282 {
2283   The~available~keys~are~(in~alphabetic~order):~
2284   annotation,~
2285   add-to-split-separation,~
2286   auto-gobble,~
2287   background-color,~
2288   begin-range,~
2289   box,~
2290   break-lines,~
2291   break-lines-in-piton,~
2292   break-lines-in-Piton,~
2293   break-numbers-anywhere,~
2294   break-strings-anywhere,~
2295   continuation-symbol,~
2296   continuation-symbol-on-indentation,~
2297   detected-beamer-commands,~
2298   detected-beamer-environments,~
2299   detected-commands,~
2300   end-of-broken-line,~
2301   end-range,~
2302   env-gobble,~
2303   env-used-by-split,~
2304   font-command,~
2305   gobble,~

```

```

2306 indent-broken-lines,~
2307 join,~
2308 label-as-zlabel,~
2309 language,~
2310 left-margin,~
2311 line-numbers/,~
2312 marker/,~
2313 math-comments,~
2314 no-join,~
2315 no-write,~
2316 path,~
2317 path-write,~
2318 print,~
2319 prompt-background-color,~
2320 raw-detected-commands,~
2321 resume,~
2322 rounded-corners,~
2323 show-spaces,~
2324 show-spaces-in-strings,~
2325 splittable,~
2326 splittable-on-empty-lines,~
2327 split-on-empty-lines,~
2328 split-separation,~
2329 tabs-auto-gobble,~
2330 tab-size,~
2331 tcolorbox,~
2332 varwidth,~
2333 vertical-detected-commands,~
2334 width-and-write.
2335 }

2336 \@@_msg_new:nn { label-with-lines-numbers }
2337 {
2338   You-can't-use-the-command~\token_to_str:N \label\
2339   or~\token_to_str:N \zlabel\ because-the-key-'line-numbers'
2340   ~is-not-active.\
2341   If-you-go-on,~that-command-will-ignored.
2342 }

2343 \@@_msg_new:nn { overlay-without-beamer }
2344 {
2345   You-can't-use-an-argument~<...>~for-your-command~
2346   \token_to_str:N \PitonInputFile\ because-you-are-not~
2347   in-Beamer.\
2348   If-you-go-on,~that-argument-will-be-ignored.
2349 }

2350 \@@_msg_new:nn { label-as-zlabel-needs-zref-package }
2351 {
2352   The-key-'label-as-zlabel'~requires-the-package-'zref'.~
2353   Please-load-the-package-'zref'~before-setting-the-key.\
2354   This-error-is-fatal.
2355 }
2356 \hook_gput_code:nnn { begindocument } { . }
2357 {
2358   \bool_if:NT \g_@@_label_as_zlabel_bool
2359   {
2360     \IfPackageLoadedF { zref-base }
2361     { \@@_fatal:n { label-as-zlabel-needs-zref-package } }
2362   }
2363 }

```

2.0.15 We load `piton.lua`

```
2364 \cs_new_protected:Npn \@@_test_version:n #1
2365   {
2366     \str_if_eq:onF \PitonFileVersion { #1 }
2367     { \@@_error:n { bad-version-of-piton.lua } }
2368   }

2369 \hook_gput_code:nnn { begindocument } { . }
2370   {
2371     \lua_load_module:n { piton }
2372     \lua_now:n
2373     {
2374       tex.sprint ( luatexbase.catcodetables.expl ,
2375                   [[\@@_test_version:n {}] .. piton_version .. "]" )
2376     }
2377   }
</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`.

```
2378 ⟨*LUA⟩
2379 piton.comment_latex = piton.comment_latex or ">"
2380 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`.

```
2381 piton.write_files = { }
2382 piton.join_files = { }

2383 local sprintL3
2384 function sprintL3 ( s )
2385   tex.sprint ( luatexbase.catcodetables.expl , s )
2386 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.

```
2387 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2388 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2389 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2390 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”).

```
2391 local Q
2392 function Q ( pattern )
2393   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2394 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.

```
2395 local L
2396 function L ( pattern ) return
2397   Ct ( C ( pattern ) )
2398 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.

```
2399 local Lc
2400 function Lc ( string ) return
2401   Cc ( { luatexbase.catcodetables.expl , string } )
2402 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)

```
2403 local K
2404 function K ( style , pattern ) return
2405   Lc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] )
2406   * Q ( pattern )
2407   * Lc "}"
2408 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.

```
2409 local WithStyle
2410 function WithStyle ( style , pattern ) return
2411   Ct ( Cc "Open" * Cc ( [ [ {\PitonStyle{ }} .. style .. "}{" ] ) * Cc "}" )
2412   * pattern
2413   * Ct ( Cc "Close" )
2414 end
```

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

```
2415 Escape = P ( false )
2416 EscapeClean = P ( false )
```

```

2417 if piton.begin_escape then
2418     Escape =
2419         P ( piton.begin_escape )
2420         * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2421         * 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).

```

2422     EscapeClean =
2423         P ( piton.begin_escape )
2424         * ( 1 - P ( piton.end_escape ) ) ^ 1
2425         * P ( piton.end_escape )
2426     end

2427     EscapeMath = P ( false )
2428     if piton.begin_escape_math then
2429         EscapeMath =
2430             P ( piton.begin_escape_math )
2431             * Lc "$"
2432             * L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
2433             * Lc "$"
2434             * P ( piton.end_escape_math )
2435     end

```

The basic syntactic LPEG

```

2436 local alpha , digit = lpeg.alpha , lpeg.digit
2437 local space = P " "

```

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

```

2438 local letter = alpha + "_" + "â" + "ã" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
2439                 + "ô" + "û" + "ü" + "À" + "Á" + "Ç" + "É" + "È" + "Ê" + "Ë"
2440                 + "Ī" + "Ī" + "Ū" + "Ū" + "Ū"
2441
2442 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).

```

2443 local identifier = letter * alphanum ^ 0

```

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

```

2444 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.⁴

```

2445 local allow_underscores_except_first
2446 function allow_underscores_except_first ( p )
2447     return p * ( P "_" + p ) ^ 0
2448 end

2449 local allow_underscores
2450 function allow_underscores ( p )
2451     return ( P "_" + p ) ^ 0
2452 end

2453 local digits_to_number
2454 function digits_to_number(prefix, digits)

```

⁴The edge cases such as

```

2455 -- The edge cases of what is allowed in number literals is modelled after
2456 -- OCaml numbers, which seems to be the most permissive language
2457 -- in this regard (among C, OCaml, Python & SQL).
2458 return prefix
2459     * allow_underscores_except_first(digits^1)
2460     * (P "." * #(1 - P ".") * allow_underscores(digits))^-1
2461     * (S "eE" * S "+-"-1 * allow_underscores_except_first(digits^1))^-1
2462 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).

```

2463 local Number =
2464   K ( 'Number.Internal' ,
2465     digits_to_number (P "0x" + P "0X", R "af" + R "AF" + digit)
2466     + digits_to_number (P "0o" + P "0O", R "07")
2467     + digits_to_number (P "0b" + P "0B", R "01")
2468     + digits_to_number ( "" , digit )
2469   )

```

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?).

```

2470 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`.

```

2471 if piton.begin_escape then
2472   lpeg_central = lpeg_central - piton.begin_escape
2473 end
2474 if piton.begin_escape_math then
2475   lpeg_central = lpeg_central - piton.begin_escape_math
2476 end
2477 local Word = Q ( lpeg_central ^ 1 )

```

```

2478 local Space = Q " " ^ 1

```

```

2479 local SkipSpace = Q " " ^ 0

```

```

2480

```

```

2481 local Punct = Q ( S ".,:;!" )

```

```

2482

```

```

2483 local Tab = "\t" * Lc [[ \@@_tab: ]]

```

```

2484 local LeadingSpace = Lc [[ \@@_leading_space: ]] * P " "

```

```

2485 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.

```

2486 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).

```
2487 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ', ' )
2488 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ', ' )
2489 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ', ' )
2490 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.

```
2491 local detectedCommands = P ( false )
2492 for _ , x in ipairs ( detected_commands ) do
2493   detectedCommands = detectedCommands + P ( "\\\" .. x )
2494 end
```

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

```
2495 local rawDetectedCommands = P ( false )
2496 for _ , x in ipairs ( raw_detected_commands ) do
2497   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
2498 end
2499 local beamerCommands = P ( false )
2500 for _ , x in ipairs ( beamer_commands ) do
2501   beamerCommands = beamerCommands + P ( "\\\" .. x )
2502 end
2503 local beamerEnvironments = P ( false )
2504 for _ , x in ipairs ( beamer_environments ) do
2505   beamerEnvironments = beamerEnvironments + P ( x )
2506 end
```

Several tools for the construction of the main LPEG

```
2507 local LPEGO = { }
2508 local LPEG1 = { }
2509 local LPEG2 = { }
2510 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*.

```
2511 local Compute_braces
2512 function Compute_braces ( lpeg_string ) return
2513   P { "E" ,
2514     E =
2515       (
2516         "{ * V "E" * }"
2517         +
2518         lpeg_string
```

```

2519         +
2520         ( 1 - S "{" )
2521         ) ^ 0
2522     }
2523 end

```

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

```

2524 local Compute_DetectedCommands
2525 function Compute_DetectedCommands ( lang , braces ) return
2526     Ct (
2527         Cc "Open"
2528         * C ( detectedCommands * space ^ 0 * P "{" )
2529         * Cc "}"
2530     )
2531     * ( braces
2532         / ( function ( s )
2533             if s ~= '' then return
2534                 LPEG1[lang] : match ( s )
2535             end
2536         end )
2537     )
2538     * P "}"
2539     * Ct ( Cc "Close" )
2540 end

2541 local Compute_RawDetectedCommands
2542 function Compute_RawDetectedCommands ( lang , braces ) return
2543     Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2544 end

2545 local Compute_LPEG_cleaner
2546 function Compute_LPEG_cleaner ( lang , braces ) return
2547     Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
2548         * ( braces
2549             / ( function ( s )
2550                 if s ~= '' then return
2551                     LPEG_cleaner[lang] : match ( s )
2552                 end
2553             end )
2554         )
2555         * "}"
2556         + EscapeClean
2557         + C ( P ( 1 ) )
2558         ) ^ 0 ) / table.concat
2559 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`.

```

2560 local ParseAgain
2561 function ParseAgain ( code )
2562     if code ~= '' then return

```

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

```

2563     LPEG1[piton.language] : match ( code )
2564 end
2565 end

```

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

```
2566 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.

```
2567 local Compute_Beamer
2568 function Compute_Beamer ( lang , braces )
```

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

```
2569 local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
2570 lpeg = lpeg +
2571   Ct ( Cc "Open"
2572     * C ( beamerCommands
2573       * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2574       * P "{"
2575       )
2576     * Cc "]"
2577   )
2578   * ( braces /
2579     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2580   * "]"
2581   * Ct ( Cc "Close" )
```

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

```
2582 lpeg = lpeg +
2583   L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2584   * ( braces /
2585     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2586   * L ( P "}" )
2587   * ( braces /
2588     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2589   * L ( P "]" )
```

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

```
2590 lpeg = lpeg +
2591   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2592   * ( braces
2593     / ( function ( s )
2594       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2595   * L ( P "}" )
2596   * ( braces
2597     / ( function ( s )
2598       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2599   * L ( P "}" )
2600   * ( braces
2601     / ( function ( s )
2602       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2603   * L ( P "]" )
```

Now, the environments of Beamer.

```
2604 for _ , x in ipairs ( beamer_environments ) do
2605   lpeg = lpeg +
2606     Ct ( Cc "Open"
2607       * C (
2608         P ( [[\begin{]] .. x .. "]" )
2609         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2610       )
2611       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
2612       * Cc ( [[\end{]] .. x .. "]" )
```

```

2613     )
2614     * (
2615         ( ( 1 - P ( [[\end{}} .. x .. "]" ) ) ^ 0 )
2616         / ( function ( s )
2617             if s ~= '' then return
2618                 LPEG1[lang] : match ( s )
2619             end
2620         end )
2621     )
2622     * P ( [[\end{}} .. x .. "]" )
2623     * Ct ( Cc "Close" )
2624 end

```

Now, you can return the value we have computed.

```

2625     return lpeg
2626 end

```

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

```

2627 local CommentMath =
2628     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`.

```

2629 local Prompt =
2630     K ( 'Prompt' , ( P ">>>" + "... " ) * P " " ^ -1 )
2631     * Lc [[ \rowcolor { \l_@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

2632 local EOL =
2633     P "\r"
2634     *
2635     (
2636         space ^ 0 * -1
2637         +
2638         Cc "EOL"
2639     )
2640     * ( LeadingSpace ^ 0 * # ( 1 - S "\r" ) ) ^ -1

```

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

```

2641 local CommentLaTeX =
2642     P ( piton.comment_latex )
2643     * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
2644     * L ( ( 1 - P "\r" ) ^ 0 )
2645     * Lc "}"
2646     * ( 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).

```

2647 --python Python
2648 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.

```

2649 local Operator =
2650     K ( 'Operator' ,
2651         P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "/" + "*"
2652         + S "--+/*%=<>&.@|" )
2653
2654 local OperatorWord =
2655     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`.

```

2656 local For = K ( 'Keyword' , P "for" )
2657     * Space
2658     * Identifier
2659     * Space
2660     * K ( 'Keyword' , P "in" )
2661
2662 local Keyword =
2663     K ( 'Keyword' ,
2664         P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
2665         "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
2666         "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
2667         "try" + "while" + "with" + "yield" + "yield from" )
2668     + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
2669
2670 local Builtin =
2671     K ( 'Name.Builtin' ,
2672         P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
2673         "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
2674         "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
2675         "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
2676         "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
2677         "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
2678         + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
2679         "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
2680         "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
2681         "vars" + "zip" )
2682
2683 local Exception =
2684     K ( 'Exception' ,
2685         P "ArithmeticError" + "AssertionError" + "AttributeError" +
2686         "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
2687         "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
2688         "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
2689         "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
2690         "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
2691         "NotImplementedError" + "OSError" + "OverflowError" +
2692         "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
2693         "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
2694         "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
2695         + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
2696         "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
2697         "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
2698         "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
2699         "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
2700         "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
2701         "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
2702         "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
2703         "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
2704         "RecursionError" )
2705
2706 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.

```
2707 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:`

```
2708 local DefClass =
2709 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`

```
2710 local ImportAs =
2711 K ( 'Keyword' , "import" )
2712 * Space
2713 * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
2714 * (
2715 ( Space * K ( 'Keyword' , "as" ) * Space
2716 * K ( 'Name.Namespace' , identifier ) )
2717 +
2718 ( SkipSpace * Q "," * SkipSpace
2719 * K ( 'Name.Namespace' , identifier ) ) ^ 0
2720 )
```

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`

```
2721 local FromImport =
2722 K ( 'Keyword' , "from" )
2723 * Space * K ( 'Name.Namespace' , identifier )
2724 * 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'''	"""text"""

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 instruction⁵ in that interpolation:

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

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

```
2725 local PercentInterpol =
2726   K ( 'String.Interpol' ,
2727     P "%"
2728     * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2729     * ( S "-#0 +" ) ^ 0
2730     * ( digit ^ 1 + "*" ) ^ -1
2731     * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2732     * ( S "HLL" ) ^ -1
2733     * S "sdfFeExXorgiGauc%"
2734   )
```

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.⁶

```
2735 local SingleShortString =
2736   WithStyle ( 'String.Short.Internal' ,
```

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

```
2737     Q ( P "f" + "F" )
2738     * (
2739       K ( 'String.Interpol' , "{" )
2740       * K ( 'Interpol.Outside' , ( 1 - S "}'" ) ^ 0 )
2741       * Q ( P ":" * ( 1 - S "}'" ) ^ 0 ) ^ -1
2742       * K ( 'String.Interpol' , "}" )
2743     +
2744     SpaceInString
2745     +
2746     Q ( ( P "\\\"" + "\\\"" + "{{" + "}}" + 1 - S " {}" ) ^ 1 )
2747     ) ^ 0
2748     * Q ""
2749     +
```

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

```
2750     Q ( P "" + "r" + "R" )
2751     * ( Q ( ( P "\\\"" + "\\\"" + 1 - S " 'r%" ) ^ 1 )
2752         + SpaceInString
2753         + PercentInterpol
2754         + Q "%"
2755     ) ^ 0
2756     * Q "" )

2757 local DoubleShortString =
2758   WithStyle ( 'String.Short.Internal' ,
2759     Q ( P "f\"" + "F\"" )
2760     * (
2761       K ( 'String.Interpol' , "{" )
2762       * K ( 'Interpol.Outside' , ( 1 - S "}'" ) ^ 0 )
2763       * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}'" ) ^ 0 ) ) ^ -1
2764       * K ( 'String.Interpol' , "}" )
2765     +
2766     SpaceInString
2767     +
2768     Q ( ( P "\\\"" + "\\\"" + "{{" + "}}" + 1 - S " {}\"" ) ^ 1 )
```

⁵There 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`.

⁶The interpolations are formatted with the piton style `Interpol.Outside`. The initial value of that style is `\\@_piton:n` which means that the interpolations are parsed once again by piton.

```

2769         ) ^ 0
2770     * Q "\""
2771 +
2772     Q ( P "\"" + "r\"" + "R\"" )
2773     * ( Q ( ( P "\\\"" + "\\\"" + 1 - S "\r%" ) ^ 1 )
2774         + SpaceInString
2775         + PercentInterpol
2776         + Q "%"
2777     ) ^ 0
2778     * Q " " )
2779
2780 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.

```

2781 local braces =
2782     Compute_braces
2783     (
2784         ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
2785         * ( P "\\\"" + 1 - S "\"" ) ^ 0 * "\""
2786     +
2787         ( P "'" + "r'" + "R'" + "f'" + "F'" )
2788         * ( P "\\'" + 1 - S "'" ) ^ 0 * "'"
2789     )
2790
2791 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```

2792 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
2793 + Compute_RawDetectedCommands ( 'python' , braces )

```

LPEG_cleaner

```

2794 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

The long strings

```

2795 local SingleLongString =
2796     WithStyle ( 'String.Long.Internal' ,
2797         ( Q ( S "fF" * P "'''" )
2798             * (
2799                 K ( 'String.Interpol' , "{" )
2800                 * K ( 'Interpol.Outside' , ( 1 - S "};\r" - "'''" ) ^ 0 )
2801                 * Q ( P ":" * ( 1 - S "};\r" - "'''" ) ^ 0 ) ^ -1
2802                 * K ( 'String.Interpol' , "}" )
2803             +
2804                 Q ( ( 1 - P "'''" - S "{'}\r" ) ^ 1 )
2805             +
2806                 EOL
2807             ) ^ 0
2808         +
2809         Q ( ( S "rR" ) ^ -1 * "'''" )
2810         * (
2811             Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
2812             +
2813             PercentInterpol
2814             +

```



```

2815         P "%"
2816         +
2817         EOL
2818     ) ^ 0
2819 )
2820 * Q "'''" )
2821 local DoubleLongString =
2822     WithStyle ( 'String.Long.Internal' ,
2823     (
2824         Q ( S "fF" * "\"\\\"" )
2825         * (
2826             K ( 'String.Interpol', "{" )
2827             * K ( 'Interpol.Inside' , ( 1 - S "}:\"r" - "\"\\\"" ) ^ 0 )
2828             * Q ( ":" * ( 1 - S "}:\"r" - "\"\\\"" ) ^ 0 ) ^ -1
2829             * K ( 'String.Interpol' , "}" )
2830             +
2831             Q ( ( 1 - S "{}\"r" - "\"\\\"" ) ^ 1 )
2832             +
2833             EOL
2834         ) ^ 0
2835         +
2836         Q ( S "rR" ^ -1 * "\"\\\"" )
2837         * (
2838             Q ( ( 1 - P "\"\\\"" - S "%\"r" ) ^ 1 )
2839             +
2840             PercentInterpol
2841             +
2842             P "%"
2843             +
2844             EOL
2845         ) ^ 0
2846     )
2847     * Q "\"\\\""
2848 )
2849 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).

```

2850 local StringDoc =
2851     K ( 'String.Doc.Internal' , P "r" ^ -1 * "\"\\\"" )
2852     * ( K ( 'String.Doc.Internal' , ( 1 - P "\"\\\"" - "\"r" ) ^ 0 ) * EOL
2853         * Tab ^ 0
2854     ) ^ 0
2855     * 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.

```

2856 local Comment =
2857     WithStyle
2858     ( 'Comment.Internal' ,
2859     Q "#" * ( CommentMath + Q ( ( 1 - S "$\"r" ) ^ 1 ) ) ^ 0 -- $
2860     )
2861     * ( 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).

```

2862 local expression =
2863   P { "E" ,
2864     E = ( "' * ( P "\\'" + 1 - S "'\r" ) ^ 0 * "'
2865           + "\" * ( P "\\\"" + 1 - S "\"\r" ) ^ 0 * "\"
2866           + "{" * V "F" * "}"
2867           + "(" * V "F" * ")"
2868           + "[" * V "F" * "]"
2869           + ( 1 - S "{}()[]\r," ) ^ 0 ,
2870     F = ( "{" * V "F" * "}"
2871           + "(" * V "F" * ")"
2872           + "[" * V "F" * "]"
2873           + ( 1 - S "{}()[]\r\''" ) ^ 0
2874   }

```

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`.

```

2875 local Params =
2876   P { "E" ,
2877     E = ( V "F" * ( Q " , " * V "F" ) ^ 0 ) ^ -1 ,
2878     F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
2879         * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
2880         * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
2881   }

```

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`...

```

2882 local DefFunction =
2883   K ( 'Keyword' , "def" )
2884   * Space
2885   * K ( 'Name.Function.Internal' , identifier )
2886   * SkipSpace
2887   * Q "(" * Params * Q ")"
2888   * SkipSpace
2889   * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
2890   * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
2891   * Q ":"
2892   * ( SkipSpace
2893       * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
2894       * Tab ^ 0
2895       * SkipSpace
2896       * StringDoc ^ 0 -- there may be additional docstrings
2897   ) ^ -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

```

2898 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

The main LPEG for the language Python

```
2899 local EndKeyword
2900     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
2901     EscapeMath + -1
```

First, the main loop :

```
2902 local Main =
2903     space ^ 0 * EOL -- faut-il le mettre en commentaire ?
2904     + Space
2905     + Tab
2906     + Escape + EscapeMath
2907     + Beamer
2908     + CommentLaTeX
2909     + DetectedCommands
2910     + Prompt
2911     + LongString
2912     + Comment
2913     + ExceptionInConsole
2914     + Delim
2915     + Operator
2916     + OperatorWord * EndKeyword
2917     + ShortString
2918     + Punct
2919     + FromImport
2920     + RaiseException
2921     + DefFunction
2922     + DefClass
2923     + For
2924     + Keyword * EndKeyword
2925     + Decorator
2926     + Builtin * EndKeyword
2927     + Identifier
2928     + Number
2929     + Word
```

Here, we must not put local, of course.

```
2930 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:`⁷.

```
2931 LPEG2.python =
2932     Ct (
2933         ( space ^ 0 * "\r" ) ^ -1
2934         * Lc [[ \@@_begin_line: ]]
2935         * LeadingSpace ^ 0
2936         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2937         * -1
2938         * Lc [[ \@@_end_line: ]]
2939     )
```

End of the Lua scope for the language Python.

```
2940 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:`

3.5 The language OCaml

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

```
2941 --ocaml Ocaml OCaml
2942 do

2943   local SkipSpace = ( Q " " + EOL ) ^ 0
2944   local Space = ( Q " " + EOL ) ^ 1

2945   local braces = Compute_braces ( '\"' * ( 1 - S "\"" ) ^ 0 * '\"' )

2946   if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
2947   DetectedCommands =
2948     Compute_DetectedCommands ( 'ocaml' , braces )
2949     + Compute_RawDetectedCommands ( 'ocaml' , braces )
2950   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`.

```
2951   function Q ( pattern, strict )
2952     if strict ~= nil then
2953       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2954     else
2955       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2956         + Beamer + DetectedCommands + EscapeMath + Escape
2957     end
2958   end

2959   local K
2960   function K ( style , pattern, strict ) return
2961     Lc ( [[ {\PitonStyle{}} .. style .. "}{"} ]
2962       * Q ( pattern, strict )
2963       * Lc "}}")
2964   end

2965   local WithStyle
2966   function WithStyle ( style , pattern ) return
2967     Ct ( Cc "Open" * Cc ( [[ {\PitonStyle{}} ] .. style .. "}{"} ) * Cc "}}")
2968     * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
2969     * Ct ( Cc "Close" )
2970   end
```

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

```
2971   local balanced_parens =
2972     P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }
```

The strings of OCaml

```
2973   local ocaml_string =
2974     P "\"\"
2975     * (
2976       P " "
2977       +
2978       P ( ( 1 - S "\\r" ) ^ 1 )
2979       +
2980       EOL -- ?
2981     ) ^ 0
2982     * P "\"\"
```

```

2983 local String =
2984   WithStyle
2985     ( 'String.Long.Internal' ,
2986       Q "\""
2987       * (
2988         SpaceInString
2989         +
2990         Q ( ( 1 - S "\r" ) ^ 1 )
2991         +
2992         EOL
2993       ) ^ 0
2994       * Q "\""
2995     )

```

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 programming, see the paragraphe *Lua’s long strings* in www.inf.puc-rio.br/~roberto/lpeg.

```

2996 local ext = ( R "az" + "_" ) ^ 0
2997 local open = "{" * Cg ( ext , 'init' ) * "|"
2998 local close = "|" * C ( ext ) * "}"
2999 local closeeq =
3000   Cmt ( close * Cb ( 'init' ) ,
3001     function ( s , i , a , b ) return a == b end )

```

The LPEG QuotedStringBis will do the second analysis.

```

3002 local QuotedStringBis =
3003   WithStyle ( 'String.Long.Internal' ,
3004     (
3005       Space
3006       +
3007       Q ( ( 1 - S "\r" ) ^ 1 )
3008       +
3009       EOL
3010     ) ^ 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.

```

3011 local QuotedString =
3012   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3013   ( 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).

```

3014 local comment =
3015   P {
3016     "A" ,
3017     A = Q "(*"
3018       * ( V "A"
3019         + Q ( ( 1 - S "\r$" - "(*" - "*" ) ^ 1 ) -- $
3020         + ocaml_string
3021         + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3022         + EOL
3023       ) ^ 0
3024       * Q "*" )
3025   }
3026 local Comment = WithStyle ( 'Comment.Internal' , comment )

```

Some standard LPEG

```
3027 local Delim = Q ( P "[" + "]" + S "[]" )
3028 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.

```
3029 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.

```
3030 local Constructor =
3031   P "::"
3032   * Lc (
3033     [[ {\PitonStyle{Name.Constructor}}] ] . .
3034     [[{\hspace{0.1em}:\hspace{-0.2em}:\hspace{0.1em}}] ]
3035   )
3036   +
3037   P "[]"
3038   * Lc ( [[{\PitonStyle{Name.Constructor}{\hspace{-0.1em}{\hspace{0.1em}}}] ] ] )
3039   K ( 'Name.Constructor' ,
3040     Q "`" ^ -1 * cap_identifier
3041     + Q ( "[" , true ) * SkipSpace * Q ( "]" , true )
3042
3043   local OperatorWord =
3044     K ( 'Operator.Word' ,
3045       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )
```

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

```
3046 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3047   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3048   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3049   "struct" + "type" + "val"
3050
3051 local Keyword =
3052   K ( 'Keyword' ,
3053     P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3054     + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3055     + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3056     + "virtual" + "when" + "while" + "with" )
3057   + K ( 'Keyword.Constant' , P "true" + "false" )
3058   + K ( 'Keyword.Governing' , governing_keyword )
3059
3060 local EndKeyword
3061   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3062   + EscapeMath + -1
```

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

```
3061 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3062   - ( 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`.

```
3063 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

```

3064 local ocaml_char =
3065   P "' *
3066   (
3067     ( 1 - S "'\\\" )
3068     + "\\\"
3069     * ( S "\\ntbr \"\"
3070         + digit * digit * digit
3071         + P "x" * ( digit + R "af" + R "AF" )
3072                 * ( digit + R "af" + R "AF" )
3073                 * ( digit + R "af" + R "AF" )
3074         + P "o" * R "03" * R "07" * R "07" )
3075   )
3076   * "'
3077 local Char =
3078   K ( 'String.Short.Internal', ocaml_char )

```

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

```

3079 local TypeParameter =
3080   K ( 'TypeParameter' ,
3081     "' * 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.

```

3082 local DotNotation =
3083   (
3084     K ( 'Name.Module' , cap_identifier )
3085       * Q "."
3086       * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3087     +
3088     Identifier
3089     * Q "."
3090     * K ( 'Name.Field' , identifier )
3091   )
3092   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```

The records

```

3093 local expression_for_fields_type =
3094   P { "E" ,
3095     E = ( "{ * V "F" * }"
3096           + "(" * V "F" * )"
3097           + TypeParameter
3098           + ( 1 - S "{() []\r;" ) ) ^ 0 ,
3099     F = ( "{ * V "F" * }"
3100           + "(" * V "F" * )"
3101           + ( 1 - S "{() []\r\"" ) + TypeParameter ) ^ 0
3102   }

```

```

3103 local expression_for_fields_value =
3104   P { "E" ,
3105     E = ( "{ * V "F" * }"
3106           + "(" * V "F" * )"
3107           + "[" * V "F" * "]"
3108           + ocaml_string + ocaml_char
3109           + ( 1 - S "{() [];" ) ) ^ 0 ,
3110     F = ( "{ * V "F" * }"
3111           + "(" * V "F" * )"
3112           + "[" * V "F" * "]"
3113           + ocaml_string + ocaml_char
3114           + ( 1 - S "{() []\\"" ) ) ^ 0
3115   }

```

```

3116 local OneFieldDefinition =
3117   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3118   * K ( 'Name.Field' , identifier ) * SkipSpace
3119   * Q ":" * SkipSpace
3120   * K ( 'TypeExpression' , expression_for_fields_type )
3121   * SkipSpace

```

```

3122 local OneField =
3123   K ( 'Name.Field' , identifier ) * SkipSpace
3124   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3125   * ( C ( expression_for_fields_value ) / ParseAgain )
3126   * SkipSpace

```

The records.

```

3127 local RecordVal =
3128   Q "{" * SkipSpace
3129   *
3130   (
3131     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3132   ) ^ -1
3133   *
3134   (
3135     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3136   )
3137   * SkipSpace
3138   * Q ";" ^ -1
3139   * SkipSpace
3140   * Comment ^ -1
3141   * SkipSpace
3142   * Q "}"
3143 local RecordType =
3144   Q "{" * SkipSpace
3145   *
3146   (
3147     OneFieldDefinition
3148     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3149   )
3150   * SkipSpace
3151   * Q ";" ^ -1
3152   * SkipSpace
3153   * Comment ^ -1
3154   * SkipSpace
3155   * Q "}"
3156 local Record = RecordType + RecordVal

```

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.

```

3157 local DotNotation =
3158   (
3159     K ( 'Name.Module' , cap_identifier )
3160     * Q "."
3161     * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3162   +
3163     Identifier
3164     * Q "."
3165     * K ( 'Name.Field' , identifier )
3166   )
3167   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

```



```

3168 local Operator =
3169   P "||" *
3170   Lc([[{\PitonStyle{Operator}{\hspace{0.1em}}|\hspace{-0.2em}}|\hspace{0.1em}}]])
3171   +
3172   K ( 'Operator' ,
3173     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3174     "//" + "*" + ";" + "->" + "+." + "-." + "*." + "/" +
3175     + S "--+/*%=<>&@|" )

3176 local Builtin =
3177   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

3178 local Exception =
3179   K ( 'Exception' ,
3180     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3181     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3182     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

3183 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).

```

3184 local pattern_part =
3185   ( 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).

```
3186 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.

```

3187   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3188   *

```

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

```

3189   (
3190     K ( 'Identifier.Internal' , identifier )
3191     +
3192     Q "(" * SkipSpace
3193     * ( C ( pattern_part ) / ParseAgain )
3194     * SkipSpace

```

Of course, the specification of type is optional.

```

3195     * ( Q ":" * #(1- P"=")
3196       * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3197     ) ^ -1
3198     * Q ")"
3199   )

```

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

```

3200 local DefFunction =
3201   K ( 'Keyword.Governing' , "let open" )
3202   * Space
3203   * K ( 'Name.Module' , cap_identifier )
3204   +
3205   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )

```

```

3206 * Space
3207 * K ( 'Name.Function.Internal' , identifier )
3208 * Space
3209 * (

```

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).

```

3210 Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3211 +
3212 Argument * ( SkipSpace * Argument ) ^ 0
3213 * (
3214   SkipSpace
3215   * Q ":" * # ( 1 - P "=" )
3216   * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3217 ) ^ -1
3218 )

```

DefModule

```

3219 local DefModule =
3220 K ( 'Keyword.Governing' , "module" ) * Space
3221 *
3222 (
3223   K ( 'Keyword.Governing' , "type" ) * Space
3224   * K ( 'Name.Type' , cap_identifier )
3225 +
3226 K ( 'Name.Module' , cap_identifier ) * SkipSpace
3227 *
3228 (
3229   Q "(" * SkipSpace
3230   * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3231   * Q ":" * # ( 1 - P "=" ) * SkipSpace
3232   * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3233   *
3234   (
3235     Q "," * SkipSpace
3236     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3237     * Q ":" * # ( 1 - P "=" ) * SkipSpace
3238     * K ( 'Name.Type' , cap_identifier ) * SkipSpace
3239   ) ^ 0
3240   * Q ")"
3241 ) ^ -1
3242 *
3243 (
3244   Q "=" * SkipSpace
3245   * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3246   * Q "("
3247   * K ( 'Name.Module' , cap_identifier ) * SkipSpace
3248   *
3249   (
3250     Q ","
3251     *
3252     K ( 'Name.Module' , cap_identifier ) * SkipSpace
3253   ) ^ 0
3254   * Q ")"
3255 ) ^ -1
3256 )
3257 +
3258 K ( 'Keyword.Governing' , P "include" + "open" )
3259 * Space
3260 * K ( 'Name.Module' , cap_identifier )

```

DefType

```

3261 local DefType =

```

```

3262 K ( 'Keyword.Governing' , "type" )
3263 * Space
3264 * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3265 * SkipSpace
3266 * ( Q "+=" + Q "=" )
3267 * SkipSpace
3268 * (
3269     RecordType
3270     +

```

The following lines are a suggestion of Y. Salmon.

```

3271     WithStyle
3272     (
3273         'TypeExpression' ,
3274         (
3275             (
3276                 EOL
3277                 + comment
3278                 + Q ( 1
3279                     - P ";;"
3280                     - P "type"
3281                     - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3282                 )
3283             ) ^ 0
3284             *
3285             (
3286                 # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3287                 + Q ";;"
3288                 + -1
3289             )
3290         )
3291     )
3292 )

3293 local prompt =
3294   Q "utop[" * digit^1 * Q "]"> "
3295 local start_of_line = P(function(subject, position)
3296   if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3297     return position
3298   end
3299   return nil
3300 end)
3301 local Prompt = #start_of_line * K( 'Prompt', prompt )
3302 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3303               * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3304               * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

The main LPEG for the language OCaml

```

3305 local Main =
3306   space ^ 0 * EOL
3307   + Space
3308   + Tab
3309   + Escape + EscapeMath
3310   + Beamer
3311   + DetectedCommands
3312   + TypeParameter
3313   + String + QuotedString + Char
3314   + Comment
3315   + Prompt + Answer

```

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

```

3316     + Q "~" * Identifier * ( Q ":" ) ^ -1
3317     + Q ":" * # ( 1 - P ":" ) * SkipSpace
3318     * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3319     + Exception
3320     + DefType
3321     + DefFunction
3322     + DefModule
3323     + Record
3324     + Keyword * EndKeyword
3325     + OperatorWord * EndKeyword
3326     + Builtin * EndKeyword
3327     + DotNotation
3328     + Constructor
3329     + Identifier
3330     + Punct
3331     + Delim -- Delim is before Operator for a correct analysis of [| et |]
3332     + Operator
3333     + Number
3334     + Word

```

Here, we must not put local, of course.

```

3335     LPEG1.ocaml = Main ^ 0

```

```

3336     LPEG2.ocaml =
3337     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).

```

3338     ( P ":" + ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3339     * Identifier * SkipSpace * Q ":" )
3340     * # ( 1 - S "!=" )
3341     * SkipSpace
3342     * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 )
3343     +
3344     ( space ^ 0 * "\r" ) ^ -1
3345     * Lc [[ \@@_begin_line: ]]
3346     * LeadingSpace ^ 0
3347     * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
3348     + space ^ 0 * EOL
3349     + Main
3350     ) ^ 0
3351     * -1
3352     * Lc [[ \@@_end_line: ]]
3353     )

```

End of the Lua scope for the language OCaml.

```

3354 end

```

3.6 The language C

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

```

3355 --c C c++ C++
3356 do

3357     local Delim = Q ( S "{[()]}")
3358     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.

```

3359 local identifier = letter * alphanum ^ 0
3360
3361 local Operator =
3362   K ( 'Operator' ,
3363     P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3364     + S "--+/*%=<>&.@|!" )
3365
3366 local Keyword =
3367   K ( 'Keyword' ,
3368     P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3369     "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3370     "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3371     "register" + "restricted" + "return" + "static" + "static_assert" +
3372     "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3373     "union" + "using" + "virtual" + "volatile" + "while"
3374   )
3375   + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3376
3377 local Builtin =
3378   K ( 'Name.Builtin' ,
3379     P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3380
3381 local Type =
3382   K ( 'Name.Type' ,
3383     P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3384     "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3385     "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3386     "void" + "wchar_t" ) * Q "*" ^ 0
3387
3388 local DefFunction =
3389   Type
3390   * Space
3391   * Q "*" ^ -1
3392   * K ( 'Name.Function.Internal' , identifier )
3393   * SkipSpace
3394   * # 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:`

```

3395 local DefClass =
3396   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).

```

3397 local Character =
3398   K ( 'String.Short' ,
3399     P [['\']] + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )

```

The strings of C

```

3400 String =
3401   WithStyle ( 'String.Long.Internal' ,
3402     Q "\""
3403     * ( SpaceInString
3404       + K ( 'String.Interpol' ,

```

```

3405         "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3406     )
3407     + Q ( ( P "\\\" + 1 - S " \" ) ^ 1 )
3408 ) ^ 0
3409 * Q "\"
3410 )

```

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

```

3411 local braces = Compute_braces ( "\" * ( 1 - S "\" ) ^ 0 * "\" )
3412 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
3413 DetectedCommands =
3414     Compute_DetectedCommands ( 'c' , braces )
3415     + Compute_RawDetectedCommands ( 'c' , braces )
3416 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

3417 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.

```

3418 local Comment =
3419     WithStyle ( 'Comment.Internal' ,
3420         Q "//" * ( CommentMath + Q ( ( 1 - S "$r" ) ^ 1 ) ) ^ 0 ) -- $
3421         * ( EOL + -1 )
3422
3423 local LongComment =
3424     WithStyle ( 'Comment.Internal' ,
3425         Q "/*"
3426         * ( CommentMath + Q ( ( 1 - P "*/" - S "$r" ) ^ 1 ) + EOL ) ^ 0
3427         * Q "*/"
3428         ) -- $

```

The main LPEG for the language C

```

3429 local EndKeyword
3430     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3431     EscapeMath + -1

```

First, the main loop :

```

3432 local Main =
3433     space ^ 0 * EOL
3434     + Space
3435     + Tab
3436     + Escape + EscapeMath
3437     + CommentLaTeX
3438     + Beamer
3439     + DetectedCommands
3440     + Preproc
3441     + Comment + LongComment
3442     + Delim
3443     + Operator
3444     + Character
3445     + String
3446     + Punct

```

```

3447     + DefFunction
3448     + DefClass
3449     + Type * ( Q "*" ^ -1 + EndKeyword )
3450     + Keyword * EndKeyword
3451     + Builtin * EndKeyword
3452     + Identifier
3453     + Number
3454     + Word

```

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

```

3455     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:`⁸.

```

3456     LPEG2.c =
3457     Ct (
3458         ( space ^ 0 * P "\r" ) ^ -1
3459         * Lc [[ \@@_begin_line: ]]
3460         * LeadingSpace ^ 0
3461         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3462         * -1
3463         * Lc [[ \@@_end_line: ]]
3464     )

```

End of the Lua scope for the language C.

```

3465 end

```

3.7 The language SQL

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

```

3466 --sql SQL
3467 do

3468     local LuaKeyword
3469     function LuaKeyword ( name ) return
3470         Lc [[ {\PitonStyle{Keyword}}{ } ]]
3471         * Q ( Cmt (
3472             C ( letter * alphanum ^ 0 ) ,
3473             function ( _ , _ , a ) return a : upper ( ) == name end
3474         )
3475     )
3476     * Lc "}"
3477 end

```

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

```

3478     local identifier =
3479         letter * ( alphanum + "-" ) ^ 0
3480         + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"

3481     local Operator =
3482         K ( 'Operator' , P "=" + "!=" + "<>" + ">=" + ">" + "<=" + "<" + S "*/" )

```

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:`

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).

```

3483 local Set
3484 function Set ( list )
3485     local set = { }
3486     for _ , l in ipairs ( list ) do set[l] = true end
3487     return set
3488 end

```

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.

```

3489 local set_keywords = Set
3490 {
3491     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3492     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3493     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3494     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3495     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3496     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3497     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3498     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3499     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3500     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3501     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3502     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3503     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3504     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3505     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3506     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3507     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3508     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3509     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3510     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3511 }
3512 local set_builtins = Set
3513 {
3514     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3515     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3516     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3517 }

```

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.

```

3518 local Identifier =
3519     C ( identifier ) /
3520     (
3521         function ( s )
3522             if set_keywords [ s : upper ( ) ] then return

```

Remind that, in Lua, it's possible to return *several* values.

```

3523         { [[{\PitonStyle{Keyword}{}}] } ,
3524         { luatexbase.catcodetables.other , s } ,
3525         { "}" } }
3526     else
3527         if set_builtins [ s : upper ( ) ] then return
3528         { [[{\PitonStyle{Name.Builtin}{}}] } ,

```



```

3529         { luatexbase.catcodetables.other , s } ,
3530         { "}" }
3531     else return
3532         { [[{\PitonStyle{Name.Field}{}}] } ,
3533         { luatexbase.catcodetables.other , s } ,
3534         { "}" }
3535     end
3536 end
3537 end
3538 )

```

The strings of SQL

```

3539 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.

```

3540 local braces = Compute_braces ( "" * ( 1 - P "" ) ^ 1 * "" )
3541 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
3542 DetectedCommands =
3543     Compute_DetectedCommands ( 'sql' , braces )
3544     + Compute_RawDetectedCommands ( 'sql' , braces )
3545 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.

```

3546 local Comment =
3547     WithStyle ( 'Comment.Internal' ,
3548         Q "--" -- syntax of SQL92
3549         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3550     * ( EOL + -1 )
3551
3552 local LongComment =
3553     WithStyle ( 'Comment.Internal' ,
3554         Q "/*"
3555         * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3556         * Q "*/"
3557     ) -- $

```

The main LPEG for the language SQL

```

3558 local EndKeyword
3559     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3560     EscapeMath + -1
3561
3562 local TableField =
3563     K ( 'Name.Table' , identifier )
3564     * Q "."
3565     * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
3566
3567 local OneField =
3568     (
3569         Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
3570         +
3571         K ( 'Name.Table' , identifier )
3572         * Q "."

```

```

3572     * K ( 'Name.Field' , identifier )
3573     +
3574     K ( 'Name.Field' , identifier )
3575 )
3576 * (
3577     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
3578 ) ^ -1
3579 * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
3580
3581 local OneTable =
3582     K ( 'Name.Table' , identifier )
3583     * (
3584         Space
3585         * LuaKeyword "AS"
3586         * Space
3587         * K ( 'Name.Table' , identifier )
3588     ) ^ -1
3589
3590 local WeCatchTableNames =
3591     LuaKeyword "FROM"
3592     * ( Space + EOL )
3593     * OneTable * ( SkipSpace * Q " ," * SkipSpace * OneTable ) ^ 0
3594     + (
3595         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
3596         + LuaKeyword "TABLE"
3597     )
3598     * ( Space + EOL ) * OneTable
3599
3600 local EndKeyword
3601     = Space + Punct + Delim + EOL + Beamer
3602     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

3602 local Main =
3603     space ^ 0 * EOL
3604     + Space
3605     + Tab
3606     + Escape + EscapeMath
3607     + CommentLaTeX
3608     + Beamer
3609     + DetectedCommands
3610     + Comment + LongComment
3611     + Delim
3612     + Operator
3613     + String
3614     + Punct
3615     + WeCatchTableNames
3616     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
3617     + Number
3618     + Word

```

Here, we must not put local, of course.

```

3619 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:`⁹.

```

3620 LPEG2.sql =
3621     Ct (
3622         ( space ^ 0 * "\r" ) ^ -1

```

⁹Remember 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:`

```

3623     * Lc [[ \@_begin_line: ]]
3624     * LeadingSpace ^ 0
3625     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3626     * -1
3627     * Lc [[ \@_end_line: ]]
3628 )

```

End of the Lua scope for the language SQL.

```

3629 end

```

3.8 The language “Minimal”

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

```

3630 --minimal Minimal
3631 do
3632   local Punct = Q ( S " , ; ! \ " )
3633
3634   local Comment =
3635     WithStyle ( 'Comment.Internal' ,
3636               Q "#"
3637               * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3638             )
3639     * ( EOL + -1 )
3640
3641   local String =
3642     WithStyle ( 'String.Short.Internal' ,
3643               Q "\""
3644               * ( SpaceInString
3645                 + Q ( ( P "[\]" + 1 - S " \" " ) ^ 1 )
3646                 ) ^ 0
3647               * Q "\""
3648             )

```

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

```

3649   local braces = Compute_braces ( P "\" * ( P "\\\"" + 1 - P "\" " ) ^ 1 * "\" " )
3650
3651   if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
3652
3653   DetectedCommands =
3654     Compute_DetectedCommands ( 'minimal' , braces )
3655     + Compute_RawDetectedCommands ( 'minimal' , braces )
3656
3657   LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
3658
3659   local identifier = letter * alphanum ^ 0
3660
3661   local Identifier = K ( 'Identifier.Internal' , identifier )
3662
3663   local Delim = Q ( S "{[(]}" )
3664
3665   local Main =
3666     space ^ 0 * EOL
3667     + Space
3668     + Tab
3669     + Escape + EscapeMath
3670     + CommentLaTeX
3671     + Beamer
3672     + DetectedCommands
3673     + Comment
3674     + Delim

```

```

3675     + String
3676     + Punct
3677     + Identifier
3678     + Number
3679     + Word

```

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

```

3680     LPEG1.minimal = Main ^ 0
3681
3682     LPEG2.minimal =
3683     Ct (
3684         ( space ^ 0 * "\r" ) ^ -1
3685         * Lc [[ \@@_begin_line: ]]
3686         * LeadingSpace ^ 0
3687         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3688         * -1
3689         * Lc [[ \@@_end_line: ]]
3690     )

```

End of the Lua scope for the language “Minimal”.

```

3691 end

```

3.9 The language “Verbatim”

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

```

3692 --verbatim Verbatim
3693 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”).

```

3694     local braces =
3695         P { "E" ,
3696           E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3697         }
3698
3699     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
3700
3701     DetectedCommands =
3702         Compute_DetectedCommands ( 'verbatim' , braces )
3703         + Compute_RawDetectedCommands ( 'verbatim' , braces )
3704
3705     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

3706     local lpeg_central = 1 - S "\\r"
3707     if piton.begin_escape then
3708         lpeg_central = lpeg_central - piton.begin_escape
3709     end
3710     if piton.begin_escape_math then
3711         lpeg_central = lpeg_central - piton.begin_escape_math
3712     end
3713     local Word = Q ( lpeg_central ^ 1 )
3714
3715     local Main =
3716         space ^ 0 * EOL
3717         + Space
3718         + Tab
3719         + Escape + EscapeMath
3720         + Beamer
3721         + DetectedCommands
3722         + Q [[\]]
3723         + Word

```

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

```

3724 LPEG1.verbatim = Main ^ 0
3725
3726 LPEG2.verbatim =
3727   Ct (
3728     ( space ^ 0 * "\r" ) ^ -1
3729     * Lc [[ \@@_begin_line: ]]
3730     * LeadingSpace ^ 0
3731     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3732     * -1
3733     * Lc [[ \@@_end_line: ]]
3734   )

```

End of the Lua scope for the language “verbatim”.

```

3735 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).

```

3736 --EXPL expl
3737 do
3738   local Comment =
3739     WithStyle
3740     ( 'Comment.Internal' ,
3741       Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3742     )
3743     * ( EOL + -1 )

```

First, we begin with a special function to analyse the “keywords”, that is to say the control sequences beginning by “\”.

```

3744 local analyze_cs
3745 function analyze_cs ( s )
3746   local i = s : find ( ":" )
3747   if i then

```

First, the case of what might be called a “function” in `expl`, for instance, `\tl_set:Nn` or `\int_compare:nNnTF`.

```

3748     local name = s : sub ( 2 , i - 1 )
3749     local parts = name : explode ( "_" )
3750     local module = parts[1]
3751     if module == "" then module = parts[3] end

```

Remind that, in Lua, we can return *several* values.

```

3752     return
3753     { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3754       { luatexbase.catcodetables.other , s } ,
3755       { "}" } }
3756   else
3757     local p = s : sub ( 1 , 3 )
3758     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`.

```

3759     local scope = s : sub(2,2)
3760     local parts = s : explode ( "_" )
3761     local module = parts[2]
3762     if module == "" then module = parts[3] end
3763     local type = parts[#parts]
3764     return
3765     { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
3766       { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3767       { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,

```

```

3768         { luatexbase.catcodetables.other , s } ,
3769         { "}}}}}}" }
3770     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.

```

3771         return { luatexbase.catcodetables.other , s }
3772     end
3773 end
3774 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`).

```

3775 local braces =
3776     P { "E" ,
3777         E = ( "{ " * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3778     }
3779
3780 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
3781
3782 DetectedCommands =
3783     Compute_DetectedCommands ( 'expl' , braces )
3784     + Compute_RawDetectedCommands ( 'expl' , braces )
3785
3786 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
3787
3788 local control_sequence = P "\\ " * ( R "Az" + "_" + ":" + "@" ) ^ 1
3789 local ControlSequence = C ( control_sequence ) / analyze_cs
3790
3791 local def_function
3792 = P [ [ \cs_ ] ]
3793 * ( P "set" + "new" )
3794 * ( P "_protected" ) ^ -1
3795 * P ":N" * ( P "p" ) ^ -1 * "n"
3796
3797 local DefFunction =
3798     C ( def_function ) / analyze_cs
3799     * Space
3800     * Lc ( [ [ { \PitonStyle{Name.Function}{ } ] ] )
3801     * ControlSequence -- Q ( ControlSequence ) ?
3802     * Lc "}" ] ]
3803
3804 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
3805
3806 local Main =
3807     space ^ 0 * EOL
3808     + Space
3809     + Tab
3810     + Escape + EscapeMath
3811     + Beamer
3812     + Comment
3813     + DetectedCommands
3814     + DefFunction
3815     + ControlSequence
3816     + Word

```

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

```

3813 LPEG1.expl = Main ^ 0
3814
3815 LPEG2.expl =
3816     Ct (
3817         ( space ^ 0 * " \r" ) ^ -1
3818         * Lc [ [ \@@_begin_line: ] ]
3819         * LeadingSpace ^ 0
3820         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0

```

```

3821         * -1
3822         * Lc [[ \@@_end_line: ]]
3823     )

```

End of the Lua scope for the language `expl` of LaTeX3.

```

3824 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.

```

3825 function piton.Parse ( language , code )

```

The variable `piton.language` will be used by the function `ParseAgain`.

```

3826     piton.language = language
3827     local t = LPEG2[language] : match ( code )
3828     if not t then
3829         sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
3830         return -- to exit in force the function
3831     end
3832     local left_stack = {}
3833     local right_stack = {}
3834     for _ , one_item in ipairs ( t ) do
3835         if one_item == "EOL" then
3836             for i = #right_stack, 1, -1 do
3837                 tex.sprint ( right_stack[i] )
3838             end

```

We remind that the `\@@_end_line:` must be explicit since it's the marker of end of the command `\@@_begin_line:`.

```

3839         sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
3840         tex.sprint ( table.concat ( left_stack ) )
3841     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}`.

```

3842     if one_item[1] == "Open" then
3843         tex.sprint ( one_item[2] )
3844         table.insert ( left_stack , one_item[2] )
3845         table.insert ( right_stack , one_item[3] )
3846     else
3847         if one_item[1] == "Close" then
3848             tex.sprint ( right_stack[#right_stack] )
3849             left_stack[#left_stack] = nil
3850             right_stack[#right_stack] = nil
3851         else
3852             tex.tprint ( one_item )
3853         end
3854     end
3855 end
3856 end
3857 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).

```

3858 local my_file_lines
3859 function my_file_lines ( filename )
3860     local f = io.open ( filename , 'rb' )
3861     local s = f : read ( '*a' )
3862     f : close ( )

```

À la fin, on doit bien mettre `(.-)` et pas `(.*)`.

```

3863     return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.-)\n' )
3864 end

```

Recall that, in Lua, `gmatch` returns an *iterator*.

```

3865 function piton.ReadFile ( name , first_line , last_line )
3866     local s = ''
3867     local i = 0
3868     for line in my_file_lines ( name ) do
3869         i = i + 1
3870         if i >= first_line then
3871             s = s .. '\r' .. line
3872         end
3873         if i >= last_line then break end
3874     end

```

We extract the BOM of utf-8, if present.

```

3875 if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
3876     s = s : sub ( 5 , -1 )
3877 end

```

```

3878 sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
3879 tex.sprint ( luatexbase.catcodetables.other , s )
3880 sprintL3 ( "}" )
3881 end

```

```

3882 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
3883     local s
3884     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
3885     piton.GobbleParse ( lang , n , splittable , s )
3886 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 `#`.

```

3887 function piton.ParseBis ( lang , code )
3888     return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
3889 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.

```

3890 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:`.

```

3891 return piton.Parse
3892     (
3893         lang ,
3894         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
3895     )
3896 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.

```

3897 local AutoGobbleLPEG =
3898     ( (
3899         P " " ^ 0 * "\r"
3900         +
3901         Ct ( C " " ^ 0 ) / table.getn
3902         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
3903     ) ^ 0
3904     * ( Ct ( C " " ^ 0 ) / table.getn
3905         * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3906 ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

3907 local TabsAutoGobbleLPEG =
3908     ( (
3909         P "\t" ^ 0 * "\r"
3910         +
3911         Ct ( C "\t" ^ 0 ) / table.getn
3912         * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
3913     ) ^ 0
3914     * ( Ct ( C "\t" ^ 0 ) / table.getn
3915         * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3916 ) / 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).

```

3918 local EnvGobbleLPEG =
3919     ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
3920     * 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.

```

3921 function piton.Gobble ( n , code )
3922     if n == 0 then return
3923         code
3924     else
3925         if n == -1 then
3926             n = AutoGobbleLPEG : match ( code )

```

for the case of an empty environment (only blank lines)

```
3927     if tonumber(n) then else n = 0 end
3928 else
3929     if n == -2 then
3930         n = EnvGobbleLPEG : match ( code )
3931     else
3932         if n == -3 then
3933             n = TabsAutoGobbleLPEG : match ( code )
3934         if tonumber(n) then else n = 0 end
3935         end
3936     end
3937 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...

```
3938     if n == 0 then return
3939         code
3940     else return
```

We will now use a LPEG that we have to compute dynamically because it depends on the value of `n`.

```
3941     ( Ct (
3942         ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3943         * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3944         ) ^ 0 )
3945     / table.concat
3946     ) : match ( code )
3947 end
3948 end
3949 end
```

In the following code, `n` is the value of `\l_@@_gobble_int`.
`splittable` is the value of `\l_@@_splittable_int`.

```
3950 function piton.GobbleParse ( lang , n , splittable , code )
3951     piton.ComputeLinesStatus ( code , splittable )
3952     piton.last_code = piton.Gobble ( n , code )
3953     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`.

```
3954     piton.CountLines ( piton.last_code )
3955     piton.Parse ( lang , piton.last_code )
3956     piton.join_and_write ( )
3957 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`.

```
3958 function piton.join_and_write ( )
3959     if piton.join ~= '' then
3960         if not piton.join_files [ piton.join ] then
3961             piton.join_files [ piton.join ] = piton.get_last_code ( )
3962         else
3963             piton.join_files [ piton.join ] =
3964                 piton.join_files [ piton.join ] .. "\r\n" .. piton.get_last_code ( )
3965         end
3966     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.

```
3967     if piton.write ~= '' then
```

We will write on `file_name` the full name (with the path) of the file in which we will write.

```

3968   local file_name = ''
3969   if piton.path_write == '' then
3970     file_name = piton.write
3971   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.

```

3972     local attr = lfs.attributes ( piton.path_write )
3973     if attr and attr.mode == "directory" then
3974       file_name = piton.path_write .. "/" .. piton.write
3975     else

```

If the directory does *not* exist, you raise an (non-fatal) error since TeX is not able to create a new directory.

```

3976     sprintL3 [[ \@_error_or_warning:n { InexistentDirectory } ]]
3977     end
3978   end
3979   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`).

```

3980   if not piton.write_files [ file_name ] then
3981     piton.write_files [ file_name ] = piton.get_last_code ( )
3982   else
3983     piton.write_files [ file_name ] =
3984     piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
3985   end
3986 end
3987 end
3988 end

```

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

```

3989 function piton.GobbleParseNoPrint ( lang , n , code )
3990   piton.last_code = piton.Gobble ( n , code )
3991   piton.last_language = lang
3992   piton.join_and_write ( )
3993 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).

```

3994 function piton.GobbleSplitParse ( lang , n , splittable , code )
3995   local chunks
3996   chunks =
3997     (
3998     Ct (
3999     (
4000     P " " ^ 0 * "\r"
4001     +
4002     C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4003     - ( P " " ^ 0 * ( P "\r" + -1 ) )
4004     ) ^ 1
4005     )
4006     ) ^ 0
4007     )
4008     ) : match ( piton.Gobble ( n , code ) )
4009   sprintL3 [[ \begingroup ]]

```

```

4010  sprintL3
4011  (
4012    [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, }]
4013    .. "language = " .. lang .. ","
4014    .. "splittable = " .. splittable .. "]"
4015  )
4016  for k , v in pairs ( chunks ) do
4017    if k > 1 then
4018      sprintL3 ( [[ \l_@@_split_separation_tl ]] )
4019    end
4020    tex.print
4021    (
4022      [[\begin{]} .. piton.env_used_by_split .. "}\r"
4023      .. v
4024      .. [[\end{]} .. piton.env_used_by_split .. "}\r"
4025    )
4026  end
4027  sprintL3 [[ \endgroup ]]
4028  end

4029  function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4030    local s
4031    s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4032    piton.GobbleSplitParse ( lang , n , splittable , s )
4033  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.

```

4034  piton.string_between_chunks =
4035  [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4036  .. [[ \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.

```

4037  function piton.get_last_code ( )
4038    return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4039    : gsub ( '\r\n?' , '\n' )
4040  end

```

3.14 To count the number of lines

```

4041  local CountBeamerEnvironments
4042  function CountBeamerEnvironments ( code ) return
4043  (
4044    Ct (
4045      (
4046        P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"
4047        +
4048        ( 1 - P "\r" ) ^ 0 * "\r"
4049      ) ^ 0
4050      * ( 1 - P "\r" ) ^ 0
4051      * -1
4052    ) / table.getn
4053  ) : match ( code )
4054  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 `\l_@@_composition`: (in some rare circumstances). Be careful. We have tried a version with `string.gsub` without success.

```

4055 function piton.CountLines ( code )
4056   local count
4057   count =
4058     ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4059       *
4060       (
4061         space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4062         + space ^ 0
4063       ) ^ -1
4064       * -1
4065     ) / table.getn
4066   ) : match ( code )
4067   if piton.beamer then
4068     count = count - 2 * CountBeamerEnvironments ( code )
4069   end
4070   sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4071 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).

```

4072 function piton.CountNonEmptyLines ( code )
4073   local count = 0

```

The following code is not clear. We should try to replace it by use of the `string` library of Lua.

```

4074   count =
4075     ( Ct ( ( P " " ^ 0 * "\r"
4076       + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4077       * ( 1 - P "\r" ) ^ 0
4078       * -1
4079     ) / table.getn
4080   ) : match ( code )
4081   count = count + 1
4082   if piton.beamer then
4083     count = count - 2 * CountBeamerEnvironments ( code )
4084   end
4085   sprintL3
4086   ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4087 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.

```

4088 function piton.ComputeRange ( s , t , file_name )
4089   local first_line = -1
4090   local count = 0
4091   local last_found = false
4092   for line in io.lines ( file_name ) do
4093     if first_line == -1 then
4094       if line : sub ( 1 , #s ) == s then
4095         first_line = count
4096       end
4097     else
4098       if line : sub ( 1 , #t ) == t then
4099         last_found = true
4100         break
4101       end
4102     end
4103     count = count + 1
4104   end

```

```

4105 if first_line == -1 then
4106   sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4107 else
4108   if not last_found then
4109     sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4110   end
4111 end
4112 sprintL3 (
4113   [[ \int_set:Nn \l_@@_first_line_int { } ]] .. first_line .. ' + 2 ]'
4114   .. [[ \global \l_@@_last_line_int = ]] .. count )
4115 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`.

```

4116 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).

```

4117 local lpeg_line_beamer
4118 if piton.beamer then
4119   lpeg_line_beamer =
4120     space ^ 0
4121     * P [[\begin{}} * beamerEnvironments * "]"
4122     * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4123   +
4124     space ^ 0
4125     * P [[\end{}} * beamerEnvironments * "]"
4126 else
4127   lpeg_line_beamer = P ( false )
4128 end
4129 local lpeg_empty_lines =
4130   Ct (
4131     ( lpeg_line_beamer * "\r"
4132       +
4133       P " " ^ 0 * "\r" * Cc ( 0 )
4134       +
4135       ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4136     ) ^ 0
4137     *
4138     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4139   )
4140   * -1
4141 local lpeg_all_lines =
4142   Ct (
4143     ( lpeg_line_beamer * "\r"
4144       +

```

```

4145         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4146     ) ^ 0
4147     *
4148     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4149 )
4150 * -1

```

We begin with the computation of `piton.empty_lines`. It will be used in conjunction with `line-numbers`.

```

4151 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`).

```

4152 local lines_status
4153 local s = splittable
4154 if splittable < 0 then s = - splittable end
4155 if splittable > 0 then
4156     lines_status = lpeg_all_lines : match ( code )
4157 else

```

Here, we should try to copy `piton.empty_lines` but it's not easy.

```

4158     lines_status = lpeg_empty_lines : match ( code )
4159     for i , x in ipairs ( lines_status ) do
4160         if x == 0 then
4161             for j = 1 , s - 1 do
4162                 if i + j > #lines_status then break end
4163                 if lines_status[i+j] == 0 then break end
4164                 lines_status[i+j] = 2
4165             end
4166             for j = 1 , s - 1 do
4167                 if i - j == 1 then break end
4168                 if lines_status[i-j-1] == 0 then break end
4169                 lines_status[i-j-1] = 2
4170             end
4171         end
4172     end
4173 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.

```

4174     for j = 1 , s - 1 do
4175         if j > #lines_status then break end
4176         if lines_status[j] == 0 then break end
4177         lines_status[j] = 2
4178     end

```

Now, from the end of the code.

```

4179     for j = 1 , s - 1 do
4180         if #lines_status - j == 0 then break end
4181         if lines_status[#lines_status - j] == 0 then break end
4182         lines_status[#lines_status - j] = 2
4183     end

```

```

4184 piton.lines_status = lines_status
4185 end

```

```

4186 function piton.TranslateBeamerEnv ( code )
4187     local s
4188     s =
4189     (
4190         Ct (

```

```

4191      (
4192          space ^ 0
4193          * C (
4194              ( P "\\begin{" + "\\end{" )
4195              * beamerEnvironments * "]" * ( 1 - P "\r" ) ^ 0 * "\r"
4196          )
4197          + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4198      ) ^ 0
4199      *
4200      (
4201          (
4202              space ^ 0
4203              * C (
4204                  ( P "\\begin{" + "\\end{" )
4205                  * beamerEnvironments * "]" * ( 1 - P "\r" ) ^ 0 * -1
4206              )
4207              + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4208          ) ^ -1
4209      )
4210      ) ^ -1 / table.concat
4211      ) : match ( code )
4212      sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4213      tex.sprint ( luatexbase.catcodetables.other , s )
4214      sprintL3 ( "]" )
4215  end

```

3.16 To create new languages with the syntax of listings

```

4216 function piton.new_language ( lang , definition )
4217     lang = lang : lower ( )

4218     local alpha , digit = lpeg.alpha , lpeg.digit
4219     local extra_letters = { "@" , "_" , "$" } --

```

The command `add_to_letter` (triggered by the key `alsoother`) 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.

```

4220     function add_to_letter ( c )
4221         if c ~= " " then table.insert ( extra_letters , c ) end
4222     end

```

For the digits, it's straitforward.

```

4223     function add_to_digit ( c )
4224         if c ~= " " then digit = digit + c end
4225     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 `}`).

```

4226     local other = S " :_@+*/<>!?.() []~^=#&\"'\\\$" --
4227     local extra_others = { }
4228     function add_to_other ( c )
4229         if c ~= " " then

```

We will use `extra_others` to retrieve further these characters from the list of the letters.

```

4230         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 `</...>`.

```

4231     other = other + P ( c )

```



```

4232     end
4233 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`.

```

4234 local def_table
4235 if ( S ", " ^ 0 * -1 ) : match ( definition ) then
4236     def_table = {}
4237 else
4238     local strict_braces =
4239         P { "E" ,
4240             E = ( "{" * V "F" * "}" + ( 1 - S ",{" }" ) ) ^ 0 ,
4241             F = ( "{" * V "F" * "}" + ( 1 - S "{" }" ) ) ^ 0
4242         }
4243     local cut_definition =
4244         P { "E" ,
4245             E = Ct ( V "F" * ( ", " * V "F" ) ^ 0 ) ,
4246             F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4247                 * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4248         }
4249     def_table = cut_definition : match ( definition )
4250 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.).

```

4251 local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4252 local tex_arg = tex_braced_arg + C ( 1 )
4253 local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4254 local args_for_tag
4255     = tex_option_arg
4256     * space ^ 0
4257     * tex_arg
4258     * space ^ 0
4259     * tex_arg
4260 local args_for_morekeywords
4261     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4262     * space ^ 0
4263     * tex_option_arg
4264     * space ^ 0
4265     * tex_arg
4266     * space ^ 0
4267     * ( tex_braced_arg + Cc ( nil ) )
4268 local args_for_moredelims
4269     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4270     * args_for_morekeywords
4271 local args_for_morecomment
4272     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4273     * space ^ 0
4274     * tex_option_arg
4275     * space ^ 0
4276     * 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`.

```

4277 local sensitive = true
4278 local style_tag , left_tag , right_tag
4279 for _ , x in ipairs ( def_table ) do
4280     if x[1] == "sensitive" then

```

```

4281     if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4282         sensitive = true
4283     else
4284         if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4285     end
4286 end
4287 if x[1] == "alsodigit" then x[2] : gsub ( ".", add_to_digit ) end
4288 if x[1] == "alsoletter" then x[2] : gsub ( ".", add_to_letter ) end
4289 if x[1] == "alsoother" then x[2] : gsub ( ".", add_to_other ) end
4290 if x[1] == "tag" then
4291     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4292     style_tag = style_tag or [[\PitonStyle{Tag}]]
4293 end
4294 end

```

Now, the LPEG for the numbers. Of course, it uses digit previously computed.

```

4295 local Number =
4296     K ( 'Number.Internal' ,
4297         ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4298             + digit ^ 0 * "." * digit ^ 1
4299             + digit ^ 1 )
4300         * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4301         + digit ^ 1
4302     )
4303 local string_extra_letters = ""
4304 for _ , x in ipairs ( extra_letters ) do
4305     if not ( extra_others[x] ) then
4306         string_extra_letters = string_extra_letters .. x
4307     end
4308 end
4309 local letter = alpha + S ( string_extra_letters )
4310     + P "â" + "à" + "ç" + "é" + "ê" + "ë" + "ï" + "î"
4311     + "ô" + "û" + "ü" + "À" + "Á" + "Ç" + "É" + "Ê" + "Ë" + "Ï"
4312     + "Î" + "Ï" + "Ï" + "Û" + "Ü"
4313 local alphanum = letter + digit
4314 local identifier = letter * alphanum ^ 0
4315 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.

```

4316 local split_clist =
4317     P { "E" ,
4318         E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4319             * ( P "{" ) ^ 1
4320             * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4321             * ( P "}" ) ^ 1 * space ^ 0 ,
4322         F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4323     }

```

The following function will be used if the keywords are not case-sensitive.

```

4324 local keyword_to_lpeg
4325 function keyword_to_lpeg ( name ) return
4326     Q ( Cmt (
4327         C ( identifier ) ,
4328         function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4329     end
4330     )
4331 )
4332 end
4333 local Keyword = P ( false )
4334 local PrefixedKeyword = P ( false )

```

Now, we actually treat all the keywords and also the key `moredirectives`.

```

4335 for _ , x in ipairs ( def_table )

```

```

4336 do if x[1] == "morekeywords"
4337     or x[1] == "otherkeywords"
4338     or x[1] == "moredirectives"
4339     or x[1] == "moretexcs"
4340 then
4341     local keywords = P ( false )
4342     local style = [[\PitonStyle{Keyword}]]
4343     if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
4344     style = tex_option_arg : match ( x[2] ) or style
4345     local n = tonumber ( style )
4346     if n then
4347         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4348     end
4349     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4350         if x[1] == "moretexcs" then
4351             keywords = Q ( [[\]] .. word ) + keywords
4352         else
4353             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*.

```

4354         then keywords = Q ( word ) + keywords
4355         else keywords = keyword_to_lpeg ( word ) + keywords
4356     end
4357 end
4358 end
4359 Keyword = Keyword +
4360     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4361 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.

```

4362     if x[1] == "keywordsprefix" then
4363         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4364         PrefixedKeyword = PrefixedKeyword
4365             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4366     end
4367 end

```

Now, we scan the definition of the language (i.e. the table `def_table`) for the strings.

```

4368     local long_string = P ( false )
4369     local Long_string = P ( false )
4370     local LongString = P ( false )
4371     local central_pattern = P ( false )
4372     for _ , x in ipairs ( def_table ) do
4373         if x[1] == "morestring" then
4374             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4375             arg2 = arg2 or [[\PitonStyle{String.Long}]]
4376             if arg1 ~= "s" then
4377                 arg4 = arg3
4378             end
4379             central_pattern = 1 - S ( " \r" .. arg4 )
4380             if arg1 : match "b" then
4381                 central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4382             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...

```

4383     if arg1 : match "d" or arg1 == "m" then
4384         central_pattern = P ( arg3 .. arg3 ) + central_pattern
4385     end
4386     if arg1 == "m"
4387     then prefix = B ( 1 - letter - "]" - "]" )
4388     else prefix = P ( true )
4389     end

```

First, a pattern *without captures* (needed to compute braces).

```

4390     long_string = long_string +
4391         prefix
4392         * arg3
4393         * ( space + central_pattern ) ^ 0
4394         * arg4

```

Now a pattern *with captures*.

```

4395     local pattern =
4396         prefix
4397         * Q ( arg3 )
4398         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4399         * Q ( arg4 )

```

We will need `Long_string` in the nested comments.

```

4400     Long_string = Long_string + pattern
4401     LongString = LongString +
4402         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4403         * pattern
4404         * Ct ( Cc "Close" )
4405     end
4406 end

```

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

```

4407     local braces = Compute_braces ( long_string )
4408     if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4409
4410     DetectedCommands =
4411         Compute_DetectedCommands ( lang , braces )
4412         + Compute_RawDetectedCommands ( lang , braces )
4413
4414     LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

4415     local CommentDelim = P ( false )
4416
4417     for _ , x in ipairs ( def_table ) do
4418         if x[1] == "morecomment" then
4419             local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4420             arg2 = arg2 or [[\PitonStyle{Comment}]]

```

If the letter `i` is present in the first argument (eg: `morecomment = [si]{(*){*}}`), then the corresponding comments are discarded.

```

4421         if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4422         if arg1 : match "l" then
4423             local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4424                 : match ( other_args )
4425             if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4426             if arg3 == [[\%]] then arg3 = "%" end -- mandatory
4427             CommentDelim = CommentDelim +
4428                 Ct ( Cc "Open"
4429                     * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4430                     * Q ( arg3 )

```

```

4431         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4432         * Ct ( Cc "Close" )
4433         * ( EOL + -1 )
4434     else
4435         local arg3 , arg4 =
4436             ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4437         if arg1 : match "s" then
4438             CommentDelim = CommentDelim +
4439                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
4440                 * Q ( arg3 )
4441                 * (
4442                     CommentMath
4443                     + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4444                     + EOL
4445                     ) ^ 0
4446                 * Q ( arg4 )
4447                 * Ct ( Cc "Close" )
4448         end
4449         if arg1 : match "n" then
4450             CommentDelim = CommentDelim +
4451                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
4452                 * P { "A" ,
4453                     A = Q ( arg3 )
4454                     * ( V "A"
4455                         + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4456                             - S "\r$" ) ^ 1 ) -- $
4457                         + long_string
4458                         + "$" -- $
4459                         * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4460                         * "$" -- $
4461                         + EOL
4462                     ) ^ 0
4463                     * Q ( arg4 )
4464                 }
4465                 * Ct ( Cc "Close" )
4466         end
4467     end
4468 end

```

For the keys `moredelim`, we have to add another argument in first position, equal to `*` or `**`.

```

4469 if x[1] == "moredelim" then
4470     local arg1 , arg2 , arg3 , arg4 , arg5
4471     = args_for_moredelims : match ( x[2] )
4472     local MyFun = Q
4473     if arg1 == "*" or arg1 == "**" then
4474         function MyFun ( x )
4475             if x ~= '' then return
4476                 LPEG1[lang] : match ( x )
4477             end
4478         end
4479     end
4480     local left_delim
4481     if arg2 : match "i" then
4482         left_delim = P ( arg4 )
4483     else
4484         left_delim = Q ( arg4 )
4485     end
4486     if arg2 : match "l" then
4487         CommentDelim = CommentDelim +
4488             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )
4489             * left_delim
4490             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4491             * Ct ( Cc "Close" )
4492             * ( EOL + -1 )

```

```

4493     end
4494     if arg2 : match "s" then
4495         local right_delim
4496         if arg2 : match "i" then
4497             right_delim = P ( arg5 )
4498         else
4499             right_delim = Q ( arg5 )
4500         end
4501         CommentDelim = CommentDelim +
4502             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" ) * Cc "}" )
4503             * left_delim
4504             * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4505             * right_delim
4506             * Ct ( Cc "Close" )
4507     end
4508 end
4509 end
4510
4511 local Delim = Q ( S "{[()]}")
4512 local Punct = Q ( S "=:;!\\" )
4513
4514 local Main =
4515     space ^ 0 * EOL
4516     + Space
4517     + Tab
4518     + Escape + EscapeMath
4519     + CommentLaTeX
4520     + Beamer
4521     + DetectedCommands
4522     + CommentDelim

```

We must put LongString before Delim because, in PostScript, the strings are delimited by parenthesis and those parenthesis would be caught by Delim.

```

4522     + LongString
4523     + Delim
4524     + PrefixedKeyword
4525     + Keyword * ( -1 + # ( 1 - alphanum ) )
4526     + Punct
4527     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4528     + Number
4529     + 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.

```

4530 LPEG1[lang] = Main ^ 0

```

The LPEG LPEG2[lang] is used to format general chunks of code.

```

4531 LPEG2[lang] =
4532     Ct (
4533         ( space ^ 0 * P "\r" ) ^ -1
4534         * Lc [[ \@@_begin_line: ]]
4535         * LeadingSpace ^ 0
4536         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
4537         * -1
4538         * Lc [[ \@@_end_line: ]]
4539     )

```

If the key tag has been used. Of course, this feature is designed for the languages such as HTML and XML.

```

4540     if left_tag then
4541         local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
4542             * Q ( left_tag * other ^ 0 ) -- $
4543             * ( ( 1 - P ( right_tag ) ) ^ 0 )
4544             / ( function ( x ) return LPEGO[lang] : match ( x ) end ) )

```

```

4545         * Q ( right_tag )
4546         * Ct ( Cc "Close" )
4547 MainWithoutTag
4548     = space ^ 1 * -1
4549     + space ^ 0 * EOL
4550     + Space
4551     + Tab
4552     + Escape + EscapeMath
4553     + CommentLaTeX
4554     + Beamer
4555     + DetectedCommands
4556     + CommentDelim
4557     + Delim
4558     + LongString
4559     + PrefixedKeyword
4560     + Keyword * ( -1 + # ( 1 - alphanum ) )
4561     + Punct
4562     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4563     + Number
4564     + Word
4565 LPEG0[lang] = MainWithoutTag ^ 0
4566 local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
4567               + Beamer + DetectedCommands + CommentDelim + Tag
4568 MainWithTag
4569     = space ^ 1 * -1
4570     + space ^ 0 * EOL
4571     + Space
4572     + LPEGaux
4573     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
4574 LPEG1[lang] = MainWithTag ^ 0
4575 LPEG2[lang] =
4576     Ct (
4577         ( space ^ 0 * P "\r" ) ^ -1
4578         * Lc [[ \@@_begin_line: ]]
4579         * Beamer
4580         * LeadingSpace ^ 0
4581         * LPEG1[lang]
4582         * -1
4583         * Lc [[ \@@_end_line: ]]
4584     )
4585 end
4586 end

```

3.17 We write the files (key 'write') and join the files in the PDF (key 'join')

```

4587 function piton.write_files_now ( )
4588     for file_name , file_content in pairs ( piton.write_files ) do
4589         local file = io.open ( file_name , "w" )
4590         if file then
4591             file : write ( file_content )
4592             file : close ( )
4593         else
4594             sprintL3
4595                 ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "]" )
4596         end
4597     end
4598 end

4599 function piton.utf16 ( str )
4600     local hex = { "FEFF" } -- BOM UTF-16BE
4601     for _, codepoint in utf8.codes(str) do
4602         table.insert(hex, string.format("%04X", codepoint))

```

```
4603 end
4604 return table.concat(hex)
4605 end
4606 </LUA>
```