

# The package `piton`\*

F. Pantigny  
fpantigny@wanadoo.fr

October 23, 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.<sup>1</sup>

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 4.9a of `piton`, at the date of 2025/10/23.

<sup>1</sup>Recall 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:" }

```

---

<sup>a</sup>Each 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`).

<sup>b</sup>The 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`.

<sup>c</sup>`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 command `\marginalia` of the package `marginalia` will be used for the margin notes created by the keys `paperclip` and `annotation`.

```

21 \RequirePackage { marginalia }

```

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

```

22 \RequirePackage { transparent }

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

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

67 \RequirePackage { luacode }

68 \@@_msg_new:nnn { piton.lua-not-found }
69 {
70   The-file-'piton.lua'-can't-be-found.\
71   This-error-is-fatal.\
72   If-you-want-to-know-how-to-retrieve-the-file-'piton.lua',-type-H<return>.
73 }
74 {
75   On-the-site-CTAN,-go-to-the-page-of-'piton':-https://ctan.org/pkg/piton.-
76   The-file-'README.md'-explains-how-to-retrieve-the-files-'piton.sty'-and-
77   'piton.lua'.
78 }

79 \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.

```
80 \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.

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

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

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

```

83 \keys_define:nn { piton }
84 {
85   footnote .bool_gset:N = \g_@@_footnote_bool ,
86   footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
87   footnote .usage:n = load ,
88   footnotehyper .usage:n = load ,
89
90   beamer .bool_gset:N = \g_@@_beamer_bool ,
91   beamer .default:n = true ,
92   beamer .usage:n = load ,
93
94   unknown .code:n = \@@_error:n { Unknown-key-for-package }
95 }
96 \@@_msg_new:nn { Unknown-key-for-package }
97 {
98   Unknown-key.\

```

```

99   You~have~used~the~key~'\l_keys_key_str'~when~loading~piton~
100  but~the~only~keys~available~here~are~'beamer',~'footnote'~
101  and~'footnotehyper'.~Other~keys~are~available~in~
102  \token_to_str:N \PitonOptions.\\
103  That~key~will~be~ignored.
104  }

```

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

```

105  \ProcessKeyOptions

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

109  \lua_now:e
110  {
111    piton = piton-or~{ }
112    piton.last_code = ''
113    piton.last_language = ''
114    piton.join = ''
115    piton.write = ''
116    piton.path_write = ''
117    \bool_if:NT \g_@@_beamer_bool { piton.beamer = true }
118  }

119  \RequirePackage { xcolor }

120  \@@_msg_new:nn { footnote-with-footnotehyper~package }
121  {
122    Footnote~forbidden.\\
123    You~can't~use~the~option~'footnote'~because~the~package~
124    footnotehyper~has~already~been~loaded.~
125    If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
126    within~the~environments~of~piton~will~be~extracted~with~the~tools~
127    of~the~package~footnotehyper.\\
128    If~you~go~on,~the~package~footnote~won't~be~loaded.
129  }

130  \@@_msg_new:nn { footnotehyper~with~footnote~package }
131  {
132    You~can't~use~the~option~'footnotehyper'~because~the~package~
133    footnote~has~already~been~loaded.~
134    If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
135    within~the~environments~of~piton~will~be~extracted~with~the~tools~
136    of~the~package~footnote.\\
137    If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
138  }

139  \bool_if:NT \g_@@_footnote_bool
140  {

```

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

```

141  \IfClassLoadedTF { beamer }
142  { \bool_gset_false:N \g_@@_footnote_bool }
143  {
144    \IfPackageLoadedTF { footnotehyper }
145    { \@@_error:n { footnote-with-footnotehyper~package } }
146    { \usepackage { footnote } }
147  }
148  }

149  \bool_if:NT \g_@@_footnotehyper_bool
150  {

```

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

```

151 \IfClassLoadedTF { beamer }
152   { \bool_gset_false:N \g_@@_footnote_bool }
153   {
154     \IfPackageLoadedTF { footnote }
155       { \@@_error:n { footnotehyper~with~footnote~package } }
156       { \usepackage { footnotehyper } }
157     \bool_gset_true:N \g_@@_footnote_bool
158   }
159 }

```

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

```

160 \dim_new:N \l_@@_rounded_corners_dim
161 \bool_new:N \l_@@_in_label_bool
162 \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`).

```

163 \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:`).

```

164 \box_new:N \g_@@_output_box

```

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

```

165 \str_new:N \l_piton_language_str
166 \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.

```

167 \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`).

```

168 \seq_new:N \l_@@_path_seq

```

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

```

169 \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`).

```

170 \str_new:N \l_@@_path_write_str

```

The following parameter corresponds to the key `tcolorbox`.

```

171 \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.

```

172 \dim_new:N \l_@@_tcb_margins_dim

```

The following parameter corresponds to the key `box`.

```

173 \str_new:N \l_@@_box_str

```

In order to have a better control over the keys.

```

174 \bool_new:N \l_@@_in_PitonOptions_bool

```

```
175 \bool_new:N \l_@@_in_PitonInputFile_bool
```

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

```
176 \tl_new:N \l_@@_font_command_tl
177 \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.

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

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

```
179 \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).

```
180 \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).

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

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

```
182 \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).

```
183 \tl_new:N \l_@@_split_separation_tl
184 \tl_set:Nn \l_@@_split_separation_tl
185 { \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`.

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

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

```
187 \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`).

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

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

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

```
191 \str_new:N \l_@@_begin_range_str
192 \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).

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

The argument of `\PitonInputFile`.

```
194 \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}`

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

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

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

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

```
198 \str_new:N \l_@@_join_str
199 \str_new:N \l_@@_join_separation_str
200 \str_set:Nn \l_@@_join_separation_str { }
```

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

```
201 \bool_new:N \l_@@_paperclip_bool
202 \str_new:N \l_@@_paperclip_str
203 \bool_new:N \l_@@_annotation_bool
```

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

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

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

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

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

```
206 \bool_new:N \l_@@_break_lines_in_Piton_bool
207 \bool_set_true:N \l_@@_break_lines_in_Piton_bool
208 \bool_new:N \l_@@_indent_broken_lines_bool
```

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

```
209 \tl_new:N \l_@@_continuation_symbol_tl
210 \tl_set:Nn \l_@@_continuation_symbol_tl { + }
```

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

```
211 \tl_new:N \l_@@_csoi_tl
212 \tl_set:Nn \l_@@_csoi_tl { $ \hookrightarrow \; $ }
```

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

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

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

```
215 \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).

```
216 \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.

```
217 \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 force<sup>2</sup>).

---

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

```
218 \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`:

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

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

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

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

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

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

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

```
223 \dim_new:N \l_@@_numbers_sep_dim
```

```
224 \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.

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

```
226 \int_new:N \l_@@_tab_size_int
```

```
227 \int_set:Nn \l_@@_tab_size_int { 4 }
```

```
228 \cs_new_protected:Npn \@@_tab:
```

```
229 {
```

```
230   \bool_if:NTF \l_@@_show_spaces_bool
```

```
231   {
```

```
232     \hbox_set:Nn \l_tmpa_box
```

```
233     { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
```

```
234     \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
```

```
235     \< \mathcolor { gray }
```

```
236     { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \>
```

```
237   }
```

```
238   { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
```

```
239   \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
```

```
240 }
```

The following integer corresponds to the key `gobble`.

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

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

```
242 \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.

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

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

```

244 \cs_new_protected:Npn \@@_label:n #1
245 {
246   \bool_if:NTF \l_@@_line_numbers_bool
247   {
248     \@bsphack
249     \protected@write \@auxout { }
250     {
251       \string \newlabel { #1 }
252       {
253         { \int_use:N \g_@@_visual_line_int }
254         { \thepage }
255         { }
256         { line.#1 }
257         { }
258       }
259     }
260     \@esphack
261     \IfPackageLoadedT { hyperref }
262     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
263   }
264   { \@@_error:n { label-with-lines-numbers } }
265 }

```

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.

```

266 \cs_new_protected:Npn \@@_zlabel:n #1
267 {
268   \bool_if:NTF \l_@@_line_numbers_bool
269   {
270     \@bsphack
271     \protected@write \@auxout { }
272     {
273       \string \zref@newlabel { #1 }
274       {
275         \string \default { \int_use:N \g_@@_visual_line_int }
276         \string \page { \thepage }
277         \string \zc@type { line }
278         \string \anchor { line.#1 }
279       }
280     }
281     \@esphack
282     \IfPackageLoadedT { hyperref }
283     { \Hy@raisedlink { \hyper@anchorstart { line.#1 } \hyper@anchorend } }
284   }
285   { \@@_error:n { label-with-lines-numbers } }
286 }

```

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

```

287 \NewDocumentCommand { \@@_rowcolor:n } { o m }
288 {
289   \tl_gset:ce
290   { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 }_ t1 }
291   { \tl_if_novalue:nTF { #1 } { #2 } { [ #1 ] { #2 } } }
292   \bool_gset_true:N \g_@@_rowcolor_inside_bool
293 }

```

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).

```

294 \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.

```
295 \cs_new:Npn \@@_marker_beginning:n #1 { }
296 \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.

```
297 \tl_new:N \g_@@_after_line_tl
```

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

```
298 \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`.

```
299 \bool_new:N \g_@@_color_is_none_bool
300 \bool_new:N \g_@@_next_color_is_none_bool
```

```
301 \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 *wihtout* the backlash.

```
302 \clist_new:N \l_@@_detected_commands_clist
303 \clist_new:N \l_@@_raw_detected_commands_clist
304 \clist_new:N \l_@@_beamer_commands_clist
305 \clist_set:Nn \l_@@_beamer_commands_clist
306   { uncover , only , visible , invisible , alert , action }
307 \clist_new:N \l_@@_beamer_environments_clist
308 \clist_set:Nn \l_@@_beamer_environments_clist
309   { 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.

```
310 \hook_gput_code:nmn { begindocument } { . }
311   {
312     \newtoks \PitonDetectedCommands
313     \newtoks \PitonRawDetectedCommands
314     \newtoks \PitonBeamerCommands
315     \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.

```

316 \exp_args:NV \PitonDetectedCommands \l_@@_detected_commands_clist
317 \exp_args:NV \PitonRawDetectedCommands \l_@@_raw_detected_commands_clist
318 \exp_args:NV \PitonBeamerCommands \l_@@_beamer_commands_clist
319 \exp_args:NV \PitonBeamerEnvironments \l_@@_beamer_environments_clist
320 }

```

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.

```

321 \tl_new:N \g_@@_def_vertical_commands_tl

322 \cs_new_protected:Npn \@@_vertical_commands:n #1
323 {
324   \clist_put_right:Nn \l_@@_raw_detected_commands_clist { #1 }
325   \clist_map_inline:nn { #1 }
326   {
327     \cs_set_eq:cc { @@ _ old _ ##1 : } { ##1 }
328     \cs_new_protected:cn { @@ _ new _ ##1 : n }
329     {
330       \bool_if:nTF
331       { \l_@@_tcolorbox_bool || ! \str_if_empty_p:N \l_@@_box_str }
332       {
333         \tl_gput_right:Nn \g_@@_after_line_tl
334         { \use:c { @@ _ old _ ##1 : } { ####1 } }
335       }
336       {
337         \cs_if_exist:cTF { g_@@_after_line _ \int_use:N \g_@@_line_int _ tl }
338         { \tl_gput_right:cn }
339         { \tl_gset:cn }
340         { g_@@_after_line _ \int_eval:n { \g_@@_line_int + 1 } _ tl }
341         { \use:c { @@ _ old _ ##1 : } { ####1 } }
342       }
343     }
344     \tl_gput_right:Nn \g_@@_def_vertical_commands_tl
345     { \cs_set_eq:cc { ##1 } { @@ _ new _ ##1 : n } }
346   }
347 }

```

## 2.0.4 Treatment of a line of code

```

348 \cs_new_protected:Npn \@@_replace_spaces:n #1
349 {
350   \tl_set:Nn \l_tmpa_tl { #1 }
351   \bool_if:NTF \l_@@_show_spaces_bool
352   {
353     \tl_set:Nn \l_@@_space_in_string_tl { } % U+2423
354     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl { } % U+2423
355   }
356   {

```

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

```

357   \bool_if:NT \l_@@_break_lines_in_Piton_bool

```

```

358     {
359         \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
360         { \tl_set_eq:NN \l_@@_space_in_string_tl \@@_breakable_space: }

```

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

The first implementation was using `\tl_regex_replace_all:nnN`  
`\tl_regex_replace_all:nnN { \x20 } { \c { @@_breakable_space: } } \l_tmpa_tl`  
but that programming was certainly slow.

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

```

361         \tl_replace_all:NVn \l_tmpa_tl
362         \c_catcode_other_space_tl
363         \@@_breakable_space:
364     }
365 }
366 \l_tmpa_tl
367 }
368 \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:`).

```

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

370 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
371 {
372     \group_begin:
373     \int_gzero:N \g_@@_indentation_int

```

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

```

374     \hbox_set:Nn \l_@@_line_box
375     {
376         \skip_horizontal:N \l_@@_left_margin_dim
377         \bool_if:NT \l_@@_line_numbers_bool
378         {

```

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

```

379         \int_set:Nn \l_tmpa_int
380         {
381             \lua_now:e
382             {
383                 tex.sprint
384                 (

```

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`.

```

385                 piton.empty_lines
386                 [ \int_eval:n { \g_@@_line_int + 1 } ]
387             )
388         }
389     }
390     \bool_lazy_or:nnT
391     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
392     { ! \l_@@_skip_empty_lines_bool }

```

```

393         { \int_gincr:N \g_@@_visual_line_int }
394     \bool_lazy_or:nnT
395         { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
396         { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
397     { \@@_print_number: }
398 }

```

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).

```

399     \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
400     {

```

... but if only if the key left-margin is not used !

```

401         \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
402         { \skip_horizontal:n { 0.5 em } }
403     }

```

```

404     \bool_if:NTF \l_@@_minimize_width_bool
405     {
406         \hbox_set:Nn \l_tmpa_box
407         {
408             \language = -1
409             \raggedright
410             \strut
411             \@@_replace_spaces:n { #1 }
412             \strut \hfil
413         }
414         \dim_compare:nNnTF { \box_wd:N \l_tmpa_box } < \l_@@_code_width_dim
415         { \box_use:N \l_tmpa_box }
416         { \@@_vtop_of_code:n { #1 } }
417     }
418     { \@@_vtop_of_code:n { #1 } }
419 }

```

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

```

420     \box_set_dp:Nn \l_@@_line_box { \box_dp:N \l_@@_line_box + 1.25 pt }
421     \box_set_ht:Nn \l_@@_line_box { \box_ht:N \l_@@_line_box + 1.25 pt }
422     \box_use_drop:N \l_@@_line_box
423     \group_end:
424     \g_@@_after_line_tl
425     \tl_gclear:N \g_@@_after_line_tl
426 }

```

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

```

427 \cs_new_protected:Npn \@@_vtop_of_code:n #1
428 {
429     \vbox_top:n
430     {
431         \hspace = \l_@@_code_width_dim
432         \language = -1
433         \raggedright
434         \strut
435         \@@_replace_spaces:n { #1 }
436         \strut \hfil
437     }
438 }

```

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:.`

```

439 \cs_new_protected:Npn \@@_add_background_to_line_and_use:
440 {

```

```

441 \vtop
442 {
443   \offinterlineskip
444   \hbox
445   {

```

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`.

```

446   \@@_compute_and_set_color:

```

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

```

447   \dim_set:Nn \l_tmpa_dim { \box_dp:N \l_@@_line_box }
448   \bool_if:NT \g_@@_next_color_is_none_bool
449   { \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.

```

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

```

Now, the colored panel.

```

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

```

```

494     { \skip_vertical:n { 2.5 pt } }
495     \skip_vertical:n { - \box_ht_plus_dp:N \l_@@_line_box }
496     \box_use_drop:N \l_@@_line_box
497   }
498 }

```

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.

```

499 \cs_set_protected:Npn \@@_compute_and_set_color:
500 {
501   \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
502     { \tl_set:Nn \l_tmpa_tl { none } }
503     {
504       \int_set:Nn \l_tmpb_int
505         { \int_mod:nn \g_@@_line_int \l_@@_bg_colors_int + 1 }
506       \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
507     }

```

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

```

508   \cs_if_exist:cT { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
509   {
510     \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).

```

511     \cs_undefine:c { g_@@_color_ \int_use:N \g_@@_line_int _ tl }
512   }
513   \tl_if_eq:NnTF \l_tmpa_tl { none }
514     { \bool_gset_true:N \g_@@_color_is_none_bool }
515     {
516       \bool_gset_false:N \g_@@_color_is_none_bool
517       \@@_color:o \l_tmpa_tl
518     }

```

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).

```

519   \int_compare:nNnTF { \g_@@_line_int + 1 } = \g_@@_nb_lines_int
520     { \bool_gset_false:N \g_@@_next_color_is_none_bool }
521     {
522       \int_compare:nNnTF \l_@@_bg_colors_int = \c_zero_int
523         { \tl_set:Nn \l_tmpa_tl { none } }
524         {
525           \int_set:Nn \l_tmpb_int
526             { \int_mod:nn { \g_@@_line_int + 1 } \l_@@_bg_colors_int + 1 }
527           \tl_set:Ne \l_tmpa_tl { \clist_item:Nn \l_@@_bg_color_clist \l_tmpb_int }
528         }
529       \cs_if_exist:cT { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
530       {
531         \tl_set_eq:Nc \l_tmpa_tl
532           { g_@@_color_ \int_eval:n { \g_@@_line_int + 1 } _ tl }
533       }
534       \tl_if_eq:NnTF \l_tmpa_tl { none }
535         { \bool_gset_true:N \g_@@_next_color_is_none_bool }
536         { \bool_gset_false:N \g_@@_next_color_is_none_bool }
537     }
538 }

```

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} }.

```

539 \cs_set_protected:Npn \@@_color:n #1
540 {
541   \tl_if_head_eq_meaning:nNTF { #1 } [

```

```

542     {
543     \tl_set:Nn \l_tmpa_tl { #1 }
544     \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
545     \exp_last_unbraced:No \color \l_tmpa_tl
546     }
547     { \color { #1 } }
548   }
549 \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.

```

550 \cs_new_protected:Npn \@@_par:
551   {

```

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

```

552   \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.

```

553   \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.

```

554   \kern -2.5 pt

```

Now, we control page breaks after the paragraph.

```

555   \@@_add_penalty_for_the_line:
556   }

```

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`.

```

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

```

```

579     }
580   }
581   { \hbox { ~ } }
582 }

```

## 2.0.5 PitonOptions

```

583 \bool_new:N \l_@@_line_numbers_bool
584 \bool_new:N \l_@@_skip_empty_lines_bool
585 \bool_set_true:N \l_@@_skip_empty_lines_bool
586 \bool_new:N \l_@@_line_numbers_absolute_bool
587 \tl_new:N \l_@@_line_numbers_format_tl
588 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
589 \bool_new:N \l_@@_label_empty_lines_bool
590 \bool_set_true:N \l_@@_label_empty_lines_bool
591 \int_new:N \l_@@_number_lines_start_int
592 \bool_new:N \l_@@_resume_bool
593 \bool_new:N \l_@@_split_on_empty_lines_bool
594 \bool_new:N \l_@@_splittable_on_empty_lines_bool
595 \bool_new:N \g_@@_label_as_zlabel_bool

596 \keys_define:nn { PitonOptions / marker }
597   {
598     beginning .cs_set:Np = \@@_marker_beginning:n #1 ,
599     beginning .value_required:n = true ,
600     end .cs_set:Np = \@@_marker_end:n #1 ,
601     end .value_required:n = true ,
602     include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
603     include-lines .default:n = true ,
604     unknown .code:n = \@@_error:n { Unknown-key-for-marker }
605   }

606 \keys_define:nn { PitonOptions / line-numbers }
607   {
608     true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
609     false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
610
611     start .code:n =
612       \bool_set_true:N \l_@@_line_numbers_bool
613       \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
614     start .value_required:n = true ,
615
616     skip-empty-lines .code:n =
617       \bool_if:NF \l_@@_in_PitonOptions_bool
618         { \bool_set_true:N \l_@@_line_numbers_bool }
619       \str_if_eq:nnTF { #1 } { false }
620         { \bool_set_false:N \l_@@_skip_empty_lines_bool }
621         { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
622     skip-empty-lines .default:n = true ,
623
624     label-empty-lines .code:n =
625       \bool_if:NF \l_@@_in_PitonOptions_bool
626         { \bool_set_true:N \l_@@_line_numbers_bool }
627       \str_if_eq:nnTF { #1 } { false }
628         { \bool_set_false:N \l_@@_label_empty_lines_bool }
629         { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
630     label-empty-lines .default:n = true ,
631
632     absolute .code:n =
633       \bool_if:NTF \l_@@_in_PitonOptions_bool
634         { \bool_set_true:N \l_@@_line_numbers_absolute_bool }

```

```

635     { \bool_set_true:N \l_@@_line_numbers_bool }
636 \bool_if:NT \l_@@_in_PitonInputFile_bool
637   {
638     \bool_set_true:N \l_@@_line_numbers_absolute_bool
639     \bool_set_false:N \l_@@_skip_empty_lines_bool
640   } ,
641 absolute .value_forbidden:n = true ,
642
643 resume .code:n =
644   \bool_set_true:N \l_@@_resume_bool
645   \bool_if:NF \l_@@_in_PitonOptions_bool
646   { \bool_set_true:N \l_@@_line_numbers_bool } ,
647 resume .value_forbidden:n = true ,
648
649 sep .dim_set:N = \l_@@_numbers_sep_dim ,
650 sep .value_required:n = true ,
651
652 format .tl_set:N = \l_@@_line_numbers_format_tl ,
653 format .value_required:n = true ,
654
655 unknown .code:n = \@@_error:n { Unknown~key~for~line~numbers }
656 }

```

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

```

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

```

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

```

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

```

```

694 detected-beamer-commands .usage:n = preamble ,
695 detected-beamer-environments .code:n =
696   \@@_error_if_not_in_beamer:
697   \clist_put_right:Nn \l_@@_beamer_environments_clist { #1 } ,
698 detected-beamer-environments .value_required:n = true ,
699 detected-beamer-environments .usage:n = preamble ,

```

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

```

700 begin-escape .code:n =
701   \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
702 begin-escape .value_required:n = true ,
703 begin-escape .usage:n = preamble ,
704
705 end-escape .code:n =
706   \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
707 end-escape .value_required:n = true ,
708 end-escape .usage:n = preamble ,
709
710 begin-escape-math .code:n =
711   \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
712 begin-escape-math .value_required:n = true ,
713 begin-escape-math .usage:n = preamble ,
714
715 end-escape-math .code:n =
716   \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
717 end-escape-math .value_required:n = true ,
718 end-escape-math .usage:n = preamble ,
719
720 comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
721 comment-latex .value_required:n = true ,
722 comment-latex .usage:n = preamble ,
723
724 label-as-zlabel .bool_gset:N = \g_@@_label_as_zlabel_bool ,
725 label-as-zlabel .default:n = true ,
726 label-as-zlabel .usage:n = preamble ,
727
728 math-comments .bool_gset:N = \g_@@_math_comments_bool ,
729 math-comments .default:n = true ,
730 math-comments .usage:n = preamble ,

```

Now, general keys.

```

731 language .code:n =
732   \str_set:Ne \l_piton_language_str { \str_lowercase:n { #1 } } ,
733 language .value_required:n = true ,
734 path .code:n =
735   \seq_clear:N \l_@@_path_seq
736   \clist_map_inline:nn { #1 }
737   {
738     \str_set:Nn \l_tmpa_str { ##1 }
739     \seq_put_right:No \l_@@_path_seq { \l_tmpa_str }
740   } ,
741 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.

```

742 path .initial:n = . ,
743 path-write .str_set:N = \l_@@_path_write_str ,
744 path-write .value_required:n = true ,
745 font-command .tl_set:N = \l_@@_font_command_tl ,
746 font-command .value_required:n = true ,
747 gobble .int_set:N = \l_@@_gobble_int ,
748 gobble .default:n = -1 ,
749 auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
750 auto-gobble .value_forbidden:n = true ,

```

```

751 env-gobble      .code:n          = \int_set:Nn \l_@@_gobble_int { -2 } ,
752 env-gobble      .value_forbidden:n = true ,
753 tabs-auto-gobble .code:n          = \int_set:Nn \l_@@_gobble_int { -3 } ,
754 tabs-auto-gobble .value_forbidden:n = true ,
755
756 splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
757 splittable-on-empty-lines .default:n = true ,
758
759 split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,
760 split-on-empty-lines .default:n = true ,
761
762 split-separation .tl_set:N          = \l_@@_split_separation_tl ,
763 split-separation .value_required:n = true ,
764
765 add-to-split-separation .code:n =
766   \tl_put_right:Nn \l_@@_split_separation_tl { #1 } ,
767 add-to-split-separation .value_required:n = true ,
768
769 marker .code:n =
770   \bool_lazy_or:nnTF
771     \l_@@_in_PitonInputFile_bool
772     \l_@@_in_PitonOptions_bool
773     { \keys_set:nn { PitonOptions / marker } { #1 } }
774     { \@@_error:n { Invalid-key } } ,
775 marker .value_required:n = true ,
776
777 line-numbers .code:n =
778   \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
779 line-numbers .default:n = true ,
780
781 splittable      .int_set:N          = \l_@@_splittable_int ,
782 splittable      .default:n          = 1 ,
783 background-color .code:n =
784   \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.

```

785   \int_set:Nn \l_@@_bg_colors_int { \clist_count:N \l_@@_bg_color_clist } ,
786 background-color .value_required:n = true ,
787 prompt-background-color .tl_set:N          = \l_@@_prompt_bg_color_tl ,
788 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.

```

789 print .bool_set:N = \l_@@_print_bool ,
790 print .value_required:n = true ,
791
792 width .code:n =
793   \str_if_eq:nnTF { #1 } { min }
794   {
795     \bool_set_true:N \l_@@_minimize_width_bool
796     \dim_zero:N \l_@@_width_dim
797   }
798   {
799     \bool_set_false:N \l_@@_minimize_width_bool
800     \dim_set:Nn \l_@@_width_dim { #1 }
801   } ,
802 width .value_required:n = true ,
803
804 max-width .code:n =
805   \bool_set_true:N \l_@@_minimize_width_bool
806   \dim_set:Nn \l_@@_width_dim { #1 } ,
807 max-width .value_required:n = true ,
808
809 paperclip .code:n =

```

```

810     \bool_set_true:N \l_@@_paperclip_bool
811     \tl_if_novalue:nTF { #1 }
812     { \str_set:Nn \l_@@_paperclip_str { } }
813     { \str_set:Nn \l_@@_paperclip_str { #1 } } ,
814
815     annotation .bool_set:N = \l_@@_annotation_bool ,
816     annotation .default:n = true ,
817
818     write .str_set:N = \l_@@_write_str ,
819     write .value_required:n = true ,
820     no-write .code:n = \str_set_eq:NN \l_@@_write_str \c_empty_str ,
821     no-write .value_forbidden:n = true ,
822     join .code:n =
823     \str_set:Nn \l_@@_join_str { #1 }
824     \seq_if_in:NnF \g_@@_join_seq { #1 }
825     { \seq_gput_right:No \g_@@_join_seq { #1 } } ,
826     join .value_required:n = true ,
827     join-separation .str_set:N = \l_@@_join_separation_str ,
828     join-separation .value_required:n = true ,
829     no-join .code:n = \str_set_eq:NN \l_@@_join_str \c_empty_str ,
830     no-join .value_forbidden:n = true ,
831     left-margin .code:n =
832     \str_if_eq:nnTF { #1 } { auto }
833     {
834         \dim_zero:N \l_@@_left_margin_dim
835         \bool_set_true:N \l_@@_left_margin_auto_bool
836     }
837     {
838         \dim_set:Nn \l_@@_left_margin_dim { #1 }
839         \bool_set_false:N \l_@@_left_margin_auto_bool
840     } ,
841     left-margin .value_required:n = true ,
842
843     tab-size .int_set:N = \l_@@_tab_size_int ,
844     tab-size .value_required:n = true ,
845     show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
846     show-spaces .value_forbidden:n = true ,
847     show-spaces-in-strings .code:n =
848     \tl_set:Nn \l_@@_space_in_string_tl { □ } , % U+2423
849     show-spaces-in-strings .value_forbidden:n = true ,
850     break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
851     break-lines-in-Piton .default:n = true ,
852     break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
853     break-lines-in-piton .default:n = true ,
854     break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
855     break-lines .value_forbidden:n = true ,
856     indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
857     indent-broken-lines .default:n = true ,
858     end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
859     end-of-broken-line .value_required:n = true ,
860     continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
861     continuation-symbol .value_required:n = true ,
862     continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
863     continuation-symbol-on-indentation .value_required:n = true ,
864
865     first-line .code:n = \@@_in_PitonInputFile:n
866     { \int_set:Nn \l_@@_first_line_int { #1 } } ,
867     first-line .value_required:n = true ,
868
869     last-line .code:n = \@@_in_PitonInputFile:n
870     { \int_set:Nn \l_@@_last_line_int { #1 } } ,
871     last-line .value_required:n = true ,
872

```

```

873 begin-range .code:n = \@@_in_PitonInputFile:n
874   { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
875 begin-range .value_required:n = true ,
876
877 end-range .code:n = \@@_in_PitonInputFile:n
878   { \str_set:Nn \l_@@_end_range_str { #1 } } ,
879 end-range .value_required:n = true ,
880
881 range .code:n = \@@_in_PitonInputFile:n
882   {
883     \str_set:Nn \l_@@_begin_range_str { #1 }
884     \str_set:Nn \l_@@_end_range_str { #1 }
885   } ,
886 range .value_required:n = true ,
887
888 env-used-by-split .code:n =
889   \lua_now:n { piton.env_used_by_split = '#1' } ,
890 env-used-by-split .initial:n = Piton ,
891
892 resume .meta:n = line-numbers/resume ,
893
894 unknown .code:n = \@@_error:n { Unknown-key-for-PitonOptions } ,
895
896 % deprecated
897 all-line-numbers .code:n =
898   \bool_set_true:N \l_@@_line_numbers_bool
899   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
900 rounded-corners .code:n =
901   \AtBeginDocument
902   {
903     \IfPackageLoadedTF { tikz }
904       { \dim_set:Nn \l_@@_rounded_corners_dim { #1 } }
905       { \@@_err_rounded_corners_without_Tikz: }
906   } ,
907 rounded-corners .default:n = 4 pt
908 }
909 \hook_gput_code:nnn { begindocument } { . }
910 {
911   \IfPackageLoadedTF { tcolorbox }
912   {
913     \pgfkeysifdefined { / tcb / libload / breakable }
914     {
915       \keys_define:nn { PitonOptions }
916       {
917         tcolorbox .bool_set:N = \l_@@_tcolorbox_bool ,
918         tcolorbox .default:n = true
919       }
920     }
921     {
922       \keys_define:nn { PitonOptions }
923       { tcolorbox .code:n = \@@_error:n { library-breakable-not-loaded } }
924     }
925   }
926   {
927     \keys_define:nn { PitonOptions }
928     { tcolorbox .code:n = \@@_error:n { tcolorbox-not-loaded } }
929   }
930 }
931 \cs_new_protected:Npn \@@_err_rounded_corners_without_Tikz:
932 {
933   \@@_error:n { rounded-corners-without-Tikz }
934   \cs_gset:Npn \@@_err_rounded_corners_without_Tikz: { }

```

```

935 }

936 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
937 {
938   \bool_if:NTF \l_@@_in_PitonInputFile_bool
939     { #1 }
940     { \@@_error:n { Invalid-key } }
941 }

942 \NewDocumentCommand \PitonOptions { m }
943 {
944   \bool_set_true:N \l_@@_in_PitonOptions_bool
945   \keys_set:nn { PitonOptions } { #1 }
946   \bool_set_false:N \l_@@_in_PitonOptions_bool
947 }

```

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.

```

948 \NewDocumentCommand \@@_fake_PitonOptions { }
949 { \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.

```

950 \int_new:N \g_@@_visual_line_int

951 \cs_new_protected:Npn \@@_incr_visual_line:
952 {
953   \bool_if:NF \l_@@_skip_empty_lines_bool
954     { \int_gincr:N \g_@@_visual_line_int }
955 }

956 \cs_new_protected:Npn \@@_print_number:
957 {
958   \hbox_overlap_left:n
959     {
960       {
961         \l_@@_line_numbers_format_tl

```

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

```

962     \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
963     { \int_to_arabic:n \g_@@_visual_line_int }
964     \pdfextension literal { EMC }
965   }
966   \skip_horizontal:N \l_@@_numbers_sep_dim
967 }
968 }

```

## 2.0.7 The main commands and environments for the end user

```

969 \NewDocumentCommand { \NewPitonLanguage } { 0 { } m ! o }
970 {
971   \tl_if_novalue:nTF { #3 }

```

The last argument is provided by curryfication.

```

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

```

The two last arguments are provided by curryfication.

```

973   { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
974   }

```

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.

```

975 \prop_new:N \g_@@_languages_prop

976 \keys_define:nn { NewPitonLanguage }
977   {
978     morekeywords .code:n = ,
979     otherkeywords .code:n = ,
980     sensitive .code:n = ,
981     keywordsprefix .code:n = ,
982     moretexcs .code:n = ,
983     morestring .code:n = ,
984     morecomment .code:n = ,
985     moredelim .code:n = ,
986     moredirectives .code:n = ,
987     tag .code:n = ,
988     alsodigit .code:n = ,
989     alsoletter .code:n = ,
990     alsoother .code:n = ,
991     unknown .code:n = \@@_error:n { Unknown-key-NewPitonLanguage }
992   }

```

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

```

993 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
994   {

```

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}{...}`.

```

995   \tl_set:Ne \l_tmpa_tl
996     {
997     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
998     \str_lowercase:n { #2 }
999   }

```

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

```

1000   \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.

```

1001   \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).

```

1002   \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
1003   }

1004 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
1005   {
1006   \hook_gput_code:nnn { begindocument } { . }
1007     { \lua_now:e { piton.new_language("#1","\lua_escape:n{#2}") } }
1008   }
1009 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }

```

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

```

1010 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4 #5
1011   {

```

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}{...}`

```

1012   \tl_set:Ne \l_tmpa_tl
1013   {
1014     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
1015     \str_lowercase:n { #4 }
1016   }

```

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`.

```

1017   \prop_get:NoNTF \g_@@_languages_prop \l_tmpa_tl \l_tmpb_tl

```

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

```

1018   { \@@_NewPitonLanguage:nno { #1 } { #2 } { #5 } \l_tmpb_tl }
1019   { \@@_error:n { Language~not~defined } }
1020 }

```

```

1021 \cs_new_protected:Npn \@@_NewPitonLanguage:nnon #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.

```

1022   { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
1023 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnon { n n n o }

```

```

1024 \NewDocumentCommand { \piton } { }
1025   { \peek_meaning:NTF \bgroup { \@@_piton_standard } { \@@_piton_verbatim } }
1026 \NewDocumentCommand { \@@_piton_standard } { m }
1027   {
1028     \group_begin:
1029     \tl_if_eq:NnF \l_@@_space_in_string_tl { \_ }
1030   }

```

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.

```

1031     \bool_lazy_or:nnT
1032     \l_@@_break_lines_in_piton_bool
1033     \l_@@_break_strings_anywhere_bool
1034     { \tl_set:Nn \l_@@_space_in_string_tl { \exp_not:N \space } }
1035   }

```

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

```

1036   \automaticallyhyphenmode = 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:

```

1037   \cs_set_eq:NN \\ \c_backslash_str
1038   \cs_set_eq:NN \% \c_percent_str
1039   \cs_set_eq:NN \{ \c_left_brace_str
1040   \cs_set_eq:NN \} \c_right_brace_str
1041   \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.

```

1042   \cs_set_eq:cN { ~ } \space
1043   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1044   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1045   \tl_set:Ne \l_tmpa_tl
1046   {
1047     \lua_now:e
1048     { piton.ParseBis('\l_piton_language_str',token.scan_string()) }

```

```

1049     { #1 }
1050   }
1051   \bool_if:NTF \l_@@_show_spaces_bool
1052   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1053   {
1054     \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.

```

1055     { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl \space }
1056   }

```

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

```

1057   \if_mode_math:
1058     \text { \l_@@_font_command_tl \l_tmpa_tl }
1059   \else:
1060     \l_@@_font_command_tl \l_tmpa_tl
1061   \fi:
1062   \group_end:
1063 }

```

```

1064 \NewDocumentCommand { \@@_piton_verbatim } { v }
1065 {
1066   \group_begin:
1067   \automatichyphenmode = 1
1068   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:

```

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

```

1069   \cs_set_eq:NN \rowcolor \@@_noop_rowcolor
1070   \tl_set:Ne \l_tmpa_tl
1071   {
1072     \lua_now:e
1073     { piton.Parse('\l_piton_language_str', token.scan_string()) }
1074     { #1 }
1075   }
1076   \bool_if:NT \l_@@_show_spaces_bool
1077   { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1078   \if_mode_math:
1079     \text { \l_@@_font_command_tl \l_tmpa_tl }
1080   \else:
1081     \l_@@_font_command_tl \l_tmpa_tl
1082   \fi:
1083   \group_end:
1084 }

```

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).

```

1085 \cs_new_protected:Npn \@@_piton:n #1
1086 { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
1087
1088 \cs_new_protected:Npn \@@_piton_i:n #1
1089 {
1090   \group_begin:
1091   \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
1092   \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
1093   \cs_set:cpn { pitonStyle _ Prompt } { }
1094   \cs_set_eq:NN \@@_leading_space: \space
1095   \cs_set_eq:NN \@@_trailing_space: \space
1096   \tl_set:Ne \l_tmpa_tl
1097   {
1098     \lua_now:e

```

```

1099     { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
1100     { #1 }
1101   }
1102   \bool_if:NT \l_@@_show_spaces_bool
1103     { \tl_replace_all:NvN \l_tmpa_tl \c_catcode_other_space_tl { \_ } } % U+2423
1104   \@@_replace_spaces:o \l_tmpa_tl
1105   \group_end:
1106 }

```

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

```

1107 \cs_new_protected:Npn \@@_pre_composition:
1108 {
1109   \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
1110   {
1111     \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).

```

1112     \str_if_empty:NF \l_@@_box_str
1113     { \bool_set_true:N \l_@@_minimize_width_bool }
1114   }

```

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).

```

1115   \dim_set:Nn \l_@@_listing_width_dim
1116   {
1117     \bool_if:NTF \l_@@_tcolorbox_bool
1118     {
1119       \l_@@_width_dim -
1120       ( \kvtcb@left@rule
1121       + \kvtcb@left@upper
1122       + \kvtcb@boxsep * 2
1123       + \kvtcb@right@upper
1124       + \kvtcb@right@rule )
1125     }
1126     { \l_@@_width_dim }
1127   }
1128   \legacy_if:nT { @inlabel } { \bool_set_true:N \l_@@_in_label_bool }
1129   \automatichyphenmode = 1
1130   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
1131   \g_@@_def_vertical_commands_tl
1132   \int_gzero:N \g_@@_line_int
1133   \int_gzero:N \g_@@_nb_lines_int
1134   \dim_zero:N \parindent
1135   \dim_zero:N \lineskip
1136   \dim_zero:N \parskip
1137   \cs_set_eq:NN \rowcolor \@@_rowcolor:n

```

For efficiency, we keep in \l\_@@\_bg\_colors\_int the length of \l\_@@\_bg\_color\_clist.

```

1138   \int_compare:nNnT \l_@@_bg_colors_int > { \c_zero_int }
1139   { \bool_set_true:N \l_@@_bg_bool }
1140   \bool_gset_false:N \g_@@_rowcolor_inside_bool
1141   \IfPackageLoadedTF { zref-base }
1142   {
1143     \bool_if:NTF \g_@@_label_as_zlabel_bool
1144     { \cs_set_eq:NN \label \@@_zlabel:n }
1145     { \cs_set_eq:NN \label \@@_label:n }
1146     \cs_set_eq:NN \zlabel \@@_zlabel:n
1147   }
1148   { \cs_set_eq:NN \label \@@_label:n }
1149   \l_@@_font_command_tl

```

```
1150 }
```

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`.

```
1151 \cs_new_protected:Npn \@@_compute_left_margin:
1152 {
1153   \use:e
1154   {
1155     \bool_if:NTF \l_@@_skip_empty_lines_bool
1156       { \lua_now:n { piton.CountNonEmptyLines(token.scan_argument()) } }
1157       { \lua_now:n { piton.CountLines(token.scan_argument()) } }
1158     { \l_@@_listing_tl }
1159   }
1160   \hbox_set:Nn \l_tmpa_box
1161   {
1162     \l_@@_line_numbers_format_tl
1163     \int_to_arabic:n
1164     {
1165       \g_@@_visual_line_int
1166       +
1167       \bool_if:NTF \l_@@_skip_empty_lines_bool
1168         { \l_@@_nb_non_empty_lines_int }
1169         { \g_@@_nb_lines_int }
1170     }
1171   }
1172   \dim_set:Nn \l_@@_left_margin_dim
1173   { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
1174 }
```

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:`.

```
1175 \cs_new_protected:Npn \@@_recompute_listing_width:
1176 {
1177   \dim_set:Nn \l_@@_listing_width_dim { \box_wd:N \g_@@_output_box }
1178   \int_compare:nNnTF \l_@@_bg_colors_int > { \c_zero_int }
1179   {
1180     \dim_add:Nn \l_@@_listing_width_dim { 0.5 em }
1181     \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
1182       { \dim_add:Nn \l_@@_listing_width_dim { 0.5 em } }
1183       { \dim_add:Nn \l_@@_listing_width_dim \l_@@_left_margin_dim }
1184   }
1185   { \dim_add:Nn \l_@@_listing_width_dim \l_@@_left_margin_dim }
1186 }
```

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

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

```
1187 \cs_new_protected:Npn \@@_compute_code_width:
1188 {
1189   \dim_set_eq:NN \l_@@_code_width_dim \l_@@_listing_width_dim
1190   \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.

```
1191 {
1192   \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 value<sup>3</sup> and we use that

---

<sup>3</sup>If 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.

value. Elsewhere, we use a value of 0.5 em.

```

1193     \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
1194     { \dim_sub:Nn \l_@@_code_width_dim { 0.5 em } }
1195     { \dim_sub:Nn \l_@@_code_width_dim \l_@@_left_margin_dim }
1196   }

```

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

```

1197   { \dim_sub:Nn \l_@@_code_width_dim \l_@@_left_margin_dim }
1198 }

```

```

1199 \cs_new_protected:Npn \@@_store_body:n #1
1200 {

```

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

```

1201   \tl_set:Ne \obeyedline { \char_generate:nn { 13 } { 11 } }
1202   \tl_set:Ne \l_@@_listing_tl { #1 }
1203   \tl_set_eq:NN \ProcessedArgument \l_@@_listing_tl
1204 }

```

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

```

1205 \cs_new_protected:Nn \@@_DefinePitonEnvironment:nnnnn
1206 {
1207   \use:c { #1 DocumentEnvironment } { #2 } { #3 > { \@@_store_body:n } c }
1208   {
1209     \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
1210     #4
1211     \@@_pre_composition:
1212     \int_compare:nNnT { \l_@@_number_lines_start_int } > { \c_zero_int }
1213     {
1214       \int_gset:Nn \g_@@_visual_line_int
1215       { \l_@@_number_lines_start_int - 1 }
1216     }
1217     \bool_if:NT \g_@@_beamer_bool
1218     { \@@_translate_beamer_env:o { \l_@@_listing_tl } }
1219     \bool_if:NT \g_@@_footnote_bool \savenotes
1220     \@@_composition:
1221     \bool_lazy_or:nnT { \l_@@_paperclip_bool } { \l_@@_annotation_bool }
1222     { \@@_create_paperclip_annotation: }
1223     \bool_if:NT \g_@@_footnote_bool \endsavenotes
1224     #5
1225   }
1226   { \ignorespacesafterend }
1227 }

```

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

```

1228 \cs_new_protected:Npn \@@_create_paperclip_annotation:
1229 {
1230   \marginalia
1231   {
1232     \vspace* { - 0.8 em }
1233     \hbox:n
1234     {
1235       \vrule-height~0~pt~depth~12~pt~width~0~pt
1236       \bool_if:NT \l_@@_annotation_bool
1237       {
1238         \lua_now:n
1239         {

```

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!

```

1240         pdf.immediateobj
1241         ( "<" .. piton.utf16 ( piton.get_last_code ( ) ) .. ">" )
1242     }

```

```

1243     \pdfextension annot~width~5pt~height~10pt~depth~0pt
1244     {
1245         /Subtype /Text
1246         /Contents~\pdf_object_ref_last:
1247         /Name /Note
1248         /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).

```

1249         /ReplyType /Group

```

Adds the bit 10 which means LockedContents.

```

1250         /F~512
1251         /C [0.8~0.8~0.8]
1252     }
1253     \hspace* { 7 mm }
1254 }
1255 \bool_if:NT \l_@@_paperclip_bool { \@@_create_paperclip: }
1256 }
1257 }
1258 }

```

```

1259 \cs_new_protected:Npn \@@_create_paperclip:
1260 {
1261     \str_if_empty:NT \l_@@_paperclip_str
1262     {
1263         \int_gincr:N \g_@@_paperclip_int
1264         \str_set:Ne \l_@@_paperclip_str { listing\_int_use:N \g_@@_paperclip_int .txt }
1265     }

```

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

```

1266     \lua_now:n { pdf.immediateobj ( "stream" , tostring ( piton.get_last_code() ) ) }
1267     \box_move_down:nn
1268     { 10 pt }
1269     {
1270         \hbox:n
1271         {
1272             \pdfextension annot~width~10pt~height~20pt~depth~0pt
1273             {
1274                 /Subtype /FileAttachment
1275                 /Name /Paperclip
1276                 /F~8 % no zoom

```

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

```

1277         /Contents (The~computer~listing)
1278         /FS <<
1279             /Type /Filespec
1280             /F (\l_@@_paperclip_str)
1281             /EF << /F~\pdf_object_ref_last: >>
1282             /AFRelationship /Supplement
1283         >>
1284     }
1285 }
1286 }
1287 }

```

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

```

1288 \NewDocumentCommand { \NewPitonEnvironment } { }
1289 { \@@_DefinePitonEnvironment:nnnnn { New } }
1290 \NewDocumentCommand { \DeclarePitonEnvironment } { }
1291 { \@@_DefinePitonEnvironment:nnnnn { Declare } }
1292 \NewDocumentCommand { \RenewPitonEnvironment } { }
1293 { \@@_DefinePitonEnvironment:nnnnn { Renew } }
1294 \NewDocumentCommand { \ProvidePitonEnvironment } { }

```

```

1295 { \@@_DefinePitonEnvironment:nnnnn { Provide } }

1296 \cs_new_protected:Npn \@@_translate_beamer_env:n
1297 { \lua_now:e { piton.TranslateBeamerEnv(token.scan_argument ( ) ) } }
1298 \cs_generate_variant:Nn \@@_translate_beamer_env:n { o }

1299 \cs_new_protected:Npn \@@_composition:
1300 {
1301   \str_if_empty:NT \l_@@_box_str
1302   {
1303     \mode_if_vertical:F
1304     { \bool_if:NF \l_@@_in_PitonInputFile_bool { \newline } }
1305   }
1306   \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
1307   { \@@_compute_left_margin: }
1308   \lua_now:e
1309   {
1310     piton.join_separation = "\l_@@_join_separation_str"
1311     piton.join = "\l_@@_join_str"
1312     piton.write = "\l_@@_write_str"
1313     piton.path_write = "\l_@@_path_write_str"
1314   }
1315   \noindent
1316   \bool_if:NTF \l_@@_print_bool
1317   {

```

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.

```

1318   \bool_if:NTF \l_@@_split_on_empty_lines_bool
1319   { \par \@@_retrieve_gobble_split_parse:o \l_@@_listing_tl }
1320   {
1321     \@@_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).

```

1322   \bool_if:NTF \l_@@_tcolorbox_bool
1323   {
1324     \str_if_empty:NTF \l_@@_box_str
1325     { \@@_composition_iii: }
1326     { \@@_composition_iv: }
1327   }
1328   {
1329     \str_if_empty:NTF \l_@@_box_str
1330     { \@@_composition_i: }
1331     { \@@_composition_ii: }
1332   }
1333 }
1334 }
1335 { \@@_gobble_parse_no_print:o \l_@@_listing_tl }
1336 }

```

`\@@_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=...`

```

1337 \cs_new_protected:Npn \@@_composition_i:
1338 {

```

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.

```
1339 \box_clear:N \g_tmpa_box
```

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

```
1340 \box_clear_new:N \g_@@_line_box
```

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

```
1341 \vbox_set:Nn \l_tmpa_box
1342 {
1343   \vbox_unpack_drop:N \g_@@_output_box
1344   \bool_gset_false:N \g_tmpa_bool
1345   \unskip \unskip
1346   \bool_gset_false:N \g_tmpa_bool
1347   \bool_do_until:nn \g_tmpa_bool
1348   {
1349     \unskip \unskip \unskip
1350     \unpenalty \unkern
1351     \box_set_to_last:N \l_@@_line_box
1352     \box_if_empty:NTF \l_@@_line_box
1353     { \bool_gset_true:N \g_tmpa_bool }
1354     {
1355       \vbox_gset:Nn \g_tmpa_box
1356       {
1357         \vbox_unpack:N \g_tmpa_box
1358         \box_use:N \l_@@_line_box
1359       }
1360     }
1361   }
1362 }
```

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

```
1363 \bool_gset_false:N \g_tmpa_bool
1364 \int_zero:N \g_@@_line_int
1365 \bool_do_until:nn \g_tmpa_bool
1366 {
```

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`.

```
1367   \vbox_gset:Nn \g_tmpa_box
1368   {
1369     \vbox_unpack_drop:N \g_tmpa_box
1370     \box_gset_to_last:N \g_@@_line_box
1371   }
```

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.

```
1372   \box_if_empty:NTF \g_@@_line_box
1373   { \bool_gset_true:N \g_tmpa_bool }
1374   {
1375     \box_use:N \g_@@_line_box
1376     \int_gincr:N \g_@@_line_int
1377     \par
1378     \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`.

```
1379     \@@_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.

```
1380     \cs_if_exist_use:cT { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 }
1381     { \cs_undefine:c { g_@@_after_line _ \int_use:N \g_@@_line_int _ t1 } }
1382     \int_compare:nNnT \g_@@_line_int < \g_@@_nb_lines_int
1383     { \mode_leave_vertical: }
1384   }
1385 }
1386 \skip_vertical:n { 2.5 pt }
1387 }
```

\@@\_composition\_ii: will be used when the key box is in force.

```
1388 \cs_new_protected:Npn \@@_composition_ii:
1389   {
1390     \use:e { \begin { minipage } [ \l_@@_box_str ] }
1391     { \l_@@_listing_width_dim }
```

Here, \vbox\_unpack:N, instead of \box\_use:N is mandatory for the vertical position of the box.

```
1392   \vbox_unpack:N \g_@@_output_box
```

\kern is mandatory here (\skip\_vertical:n won't work).

```
1393   \kern 2.5 pt
1394   \end { minipage }
1395 }
```

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

```
1396 \cs_new_protected:Npn \@@_composition_iii:
1397   {
1398     \use:e
1399     {
1400       \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.

```
1401       [ breakable , text~width = \l_@@_listing_width_dim ]
1402     }
1403     \par
1404     \vbox_unpack:N \g_@@_output_box
1405     \end { tcolorbox }
1406   }
```

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

```
1407 \cs_new_protected:Npn \@@_composition_iv:
1408   {
1409     \use:e
1410     {
1411       \begin { tcolorbox }
1412         [
1413           hbox ,
1414           text~width = \l_@@_listing_width_dim ,
1415           nobeforeafter ,
1416           box~align =
1417             \str_case:Nn \l_@@_box_str
1418             {
1419               t { top }
1420               b { bottom }
1421               c { center }
1422               m { center }
1423             }
1424         ]
1425       }
1426       \box_use:N \g_@@_output_box
1427       \end { tcolorbox }
1428     }
```

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.

```
1429 \cs_new_protected:Npn \@@_add_penalty_for_the_line:
1430   {
1431     \int_case:nn
1432     {
1433       \lua_now:e
1434     }
```

```

1435         tex.sprint
1436         ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
1437     }
1438 }
1439 { 1 { \penalty 100 } 2 \nobreak }
1440 }

```

\@@\_create\_output\_box: is used only once, in \@@\_composition:.  
It creates (and modify when there are backgrounds) \g\_@@\_output\_box.

```

1441 \cs_new_protected:Npn \@@_create_output_box:
1442 {
1443     \@@_compute_code_width:
1444     \vbox_gset:Nn \g_@@_output_box
1445     { \@@_retrieve_gobble_parse:o \l_@@_listing_tl }
1446     \bool_if:NT \l_@@_minimize_width_bool { \@@_recompute_listing_width: }
1447     \bool_lazy_or:nnT
1448     { \int_compare_p:nNn \l_@@_bg_colors_int > { \c_zero_int } }
1449     { \g_@@_rowcolor_inside_bool }
1450     { \@@_add_backgrounds_to_output_box: }
1451 }

```

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:.

```

1452 \cs_new_protected:Npn \@@_add_backgrounds_to_output_box:
1453 {
1454     \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.

```

1455     \vbox_set:Nn \l_tmpa_box
1456     {
1457         \vbox_unpack_drop:N \g_@@_output_box

```

We will raise \g\_tmpa\_bool to exit the loop \bool\_do\_until:nn below.

```

1458     \bool_gset_false:N \g_tmpa_bool
1459     \unskip \unskip

```

We begin the loop.

```

1460     \bool_do_until:nn \g_tmpa_bool
1461     {
1462         \unskip \unskip \unskip
1463         \int_set_eq:NN \l_tmpa_int \lastpenalty
1464         \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...

```

1465         \box_set_to_last:N \l_@@_line_box
1466         \box_if_empty:NTF \l_@@_line_box
1467         { \bool_gset_true:N \g_tmpa_bool }
1468         {

```

\g\_@@\_line\_int will be used in \@@\_add\_background\_to\_line\_and\_use:.

```

1469         \vbox_gset:Nn \g_@@_output_box
1470         {

```

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.

```

1471         \@@_add_background_to_line_and_use:
1472         \kern -2.5 pt
1473         \penalty \l_tmpa_int
1474         \vbox_unpack:N \g_@@_output_box
1475     }
1476 }
1477 \int_gdecr:N \g_@@_line_int

```

```

1478     }
1479   }
1480 }

```

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

```

1481 \cs_new_protected:Npn \@@_gobble_parse_no_print:n
1482 {
1483   \lua_now:e
1484   {
1485     piton.GobbleParseNoPrint
1486     (
1487       '\l_piton_language_str' ,
1488       \int_use:N \l_@@_gobble_int ,
1489       token.scan_argument ( )
1490     )
1491   }
1492 }
1493 \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.

```

1494 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
1495 {
1496   \lua_now:e
1497   {
1498     piton.RetrieveGobbleParse
1499     (
1500       '\l_piton_language_str' ,
1501       \int_use:N \l_@@_gobble_int ,
1502       \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1503         { \int_eval:n { - \l_@@_splittable_int } }
1504         { \int_use:N \l_@@_splittable_int } ,
1505       token.scan_argument ( )
1506     )
1507   }
1508 }
1509 \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.

```

1510 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1511 {
1512   \lua_now:e
1513   {
1514     piton.RetrieveGobbleSplitParse
1515     (
1516       '\l_piton_language_str' ,
1517       \int_use:N \l_@@_gobble_int ,
1518       \int_use:N \l_@@_splittable_int ,
1519       token.scan_argument ( )
1520     )
1521   }
1522 }
1523 \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`.

```

1524 \bool_if:NTF \g_@@_beamer_bool
1525 {
1526   \NewPitonEnvironment { Piton } { D < > { .- } O { } }

```

```

1527     {
1528       \keys_set:nn { PitonOptions } { #2 }
1529       \begin { actionenv } < #1 >
1530     }
1531     { \end { actionenv } }
1532   }
1533   {
1534     \NewPitonEnvironment { Piton } { 0 { } }
1535     { \keys_set:nn { PitonOptions } { #1 } }
1536     { }
1537   }

1538 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1539 {
1540   \mode_if_vertical:F { \par }
1541   \group_begin:
1542   \seq_concat:NNN
1543     \l_file_search_path_seq
1544     \l_@@_path_seq
1545     \l_file_search_path_seq
1546   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1547   {
1548     \@@_input_file:nn { #1 } { #2 }
1549     #4
1550   }
1551   { #5 }
1552   \group_end:
1553 }

1554 \cs_new_protected:Npn \@@_unknown_file:n #1
1555 { \msg_error:nnn { piton } { Unknown-file } { #1 } }

1556 \NewDocumentCommand { \PitonInputFile } { d < > 0 { } m }
1557 {
1558   \PitonInputFileTF < #1 > [ #2 ] { #3 } { }
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 { \PitonInputFileT } { d < > 0 { } m m }
1565 {
1566   \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 }
1567 }

```

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

```

1568   \iow_log:n { No-file-#3 }
1569   \@@_unknown_file:n { #3 }
1570 }
1571 }
1572 \NewDocumentCommand { \PitonInputFileF } { d < > 0 { } m m }
1573 { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1574 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1575 {

```

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 >).

```

1576   \tl_if_novalue:nF { #1 }
1577   {
1578     \bool_if:NTF \g_@@_beamer_bool
1579     { \begin { uncoverenv } < #1 > }
1580     { \@@_error_or_warning:n { overlay-without-beamer } }

```

```

1581     }
1582     \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.
1583     \iow_log:e { (\l_@@_file_name_str) }
1584     \int_zero_new:N \l_@@_first_line_int
1585     \int_zero_new:N \l_@@_last_line_int
1586     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1587     \bool_set_true:N \l_@@_in_PitonInputFile_bool
1588     \keys_set:nn { PitonOptions } { #2 }
1589     \bool_if:NT \l_@@_line_numbers_absolute_bool
1590     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1591     \bool_if:nTF
1592     {
1593     (
1594         \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1595         || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1596     )
1597     && ! \str_if_empty_p:N \l_@@_begin_range_str
1598     }
1599     {
1600     \@@_error_or_warning:n { bad-range-specification }
1601     \int_zero:N \l_@@_first_line_int
1602     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1603     }
1604     {
1605     \str_if_empty:NF \l_@@_begin_range_str
1606     {
1607     \@@_compute_range:
1608     \bool_lazy_or:nnT
1609     \l_@@_marker_include_lines_bool
1610     { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1611     {
1612     \int_decr:N \l_@@_first_line_int
1613     \int_incr:N \l_@@_last_line_int
1614     }
1615     }
1616     }
1617     \@@_pre_composition:
1618     \bool_if:NT \l_@@_line_numbers_absolute_bool
1619     { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1620     \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
1621     {
1622     \int_gset:Nn \g_@@_visual_line_int
1623     { \l_@@_number_lines_start_int - 1 }
1624     }

```

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

```

1625     \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1626     { \int_gzero:N \g_@@_visual_line_int }
1627     \lua_now:e
1628     {

```

The following command will store the content of the file (or only a part of that file) in \l\_@@\_listing\_tl.

```

1629     piton.ReadFile(
1630     '\l_@@_file_name_str' ,
1631     \int_use:N \l_@@_first_line_int ,
1632     \int_use:N \l_@@_last_line_int )
1633     }
1634     \@@_composition:
1635     \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.

```

1636 \tl_if_novalue:nF { #1 }
1637 { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1638 }

```

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

```

1639 \cs_new_protected:Npn \@@_compute_range:
1640 {

```

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

```

1641 \str_set:Ne \l_tmpa_str { \@@_marker_beginning:n { \l_@@_begin_range_str } }
1642 \str_set:Ne \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`.

```

1643 \tl_replace_all:Nee \l_tmpa_str { \c_backslash_str \c_hash_str } \c_hash_str
1644 \tl_replace_all:Nee \l_tmpb_str { \c_backslash_str \c_hash_str } \c_hash_str
1645 \lua_now:e
1646 {
1647     piton.ComputeRange
1648     ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1649 }
1650 }

```

## 2.0.8 The styles

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

```

1651 \NewDocumentCommand { \PitonStyle } { m }
1652 {
1653     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1654     { \use:c { pitonStyle _ #1 } }
1655 }

```

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”.

```

1656 \NewDocumentCommand { \OptionalLocalPitonStyle } { m }
1657 { \cs_if_exist_use:c { pitonStyle _ \l_piton_language_str _ #1 } }

1658 \NewDocumentCommand { \SetPitonStyle } { 0 { } m }
1659 {
1660     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1661     \str_set:Ne \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1662     \str_if_eq:onT { \l_@@_SetPitonStyle_option_str } { current-language }
1663     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1664     \keys_set:nn { piton / Styles } { #2 }
1665 }

```

```

1666 \cs_new_protected:Npn \@@_math_scantokens:n #1
1667 { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

```

```

1668 \clist_new:N \g_@@_styles_clist
1669 \clist_gset:Nn \g_@@_styles_clist
1670 {
1671     Comment ,
1672     Comment.Internal ,
1673     Comment.LaTeX ,
1674     Discard ,
1675     Exception ,
1676     FormattingType ,

```

```

1677 Identifier.Internal ,
1678 Identifier ,
1679 InitialValues ,
1680 Interpol.Inside ,
1681 Keyword ,
1682 Keyword.Governing ,
1683 Keyword.Constant ,
1684 Keyword2 ,
1685 Keyword3 ,
1686 Keyword4 ,
1687 Keyword5 ,
1688 Keyword6 ,
1689 Keyword7 ,
1690 Keyword8 ,
1691 Keyword9 ,
1692 Name.Builtin ,
1693 Name.Class ,
1694 Name.Constructor ,
1695 Name.Decorator ,
1696 Name.Field ,
1697 Name.Function ,
1698 Name.Module ,
1699 Name.Namespace ,
1700 Name.Table ,
1701 Name.Type ,
1702 Number ,
1703 Number.Internal ,
1704 Operator ,
1705 Operator.Word ,
1706 Preproc ,
1707 Prompt ,
1708 String.Doc ,
1709 String.Doc.Internal ,
1710 String.Interpol ,
1711 String.Long ,
1712 String.Long.Internal ,
1713 String.Short ,
1714 String.Short.Internal ,
1715 Tag ,
1716 TypeParameter ,
1717 UserFunction ,

```

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

```

1718 TypeExpression ,

```

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

```

1719 Directive
1720 }
1721 \clist_map_inline:Nn \g_@@_styles_clist
1722 {
1723   \keys_define:nm { piton / Styles }
1724   {
1725     #1 .value_required:n = true ,
1726     #1 .code:n =
1727       \tl_set:cn
1728       {
1729         pitonStyle _
1730         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1731         { \l_@@_SetPitonStyle_option_str _ }
1732         #1
1733       }
1734     { ##1 }
1735   }
1736 }

```

```

1737
1738 \keys_define:nn { piton / Styles }
1739 {
1740   String      .meta:n = { String.Long = #1 , String.Short = #1 } ,
1741   String      .value_required:n = true ,
1742   Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1743   Comment.Math .value_required:n = true ,
1744   unknown     .code:n = \@@_unknown_style:
1745 }

```

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.

```

1746 \cs_new_protected:Npn \@@_unknown_style:
1747 {
1748   \str_if_eq:eeTF \l_@@_SetPitonStyle_option_str { expl }
1749   {
1750     \seq_set_split:Nne \l_tmpa_seq { . } \l_keys_key_str
1751     \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`.

```

1752     \bool_lazy_and:nnTF
1753     { \int_compare_p:nNn { \seq_count:N \l_tmpa_seq } > { 1 } }
1754     {
1755       \str_if_eq_p:Vn \l_tmpa_str { Module }
1756       ||
1757       \str_if_eq_p:Vn \l_tmpa_str { Type }
1758     }

```

Now, we will create a new style.

```

1759     { \tl_set:co { pitonStyle _ expl _ \l_keys_key_str } \l_keys_value_tl }
1760     { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1761   }
1762   { \@@_error:n { Unknown-key-for-SetPitonStyle } }
1763 }

```

```

1764 \SetPitonStyle[OCaml]
1765 {
1766   TypeExpression =
1767   {
1768     \SetPitonStyle [ OCaml ] { Identifier = \PitonStyle { Name.Type } }
1769     \@@_piton:n
1770   }
1771 }

```

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`.

```

1772 \clist_gput_left:Nn \g_@@_styles_clist { String }

```

Of course, we sort that `clist`.

```

1773 \clist_gsort:Nn \g_@@_styles_clist
1774 {
1775   \str_compare:nNnTF { #1 } < { #2 }
1776   \sort_return_same:
1777   \sort_return_swapped:
1778 }

```

```

1779 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1780
1781 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1782

```

```

1783 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1784 {
1785     \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`.

```

1786     \tl_replace_all:NVn \l_tmpa_tl \c_catcode_other_space_tl \space
1787     \seq_clear:N \l_tmpa_seq
1788     \tl_map_inline:Nn \l_tmpa_tl { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1789     \seq_use:Nn \l_tmpa_seq { \- }
1790 }

```

```

1791 \cs_new_protected:Npn \@@_comment:n #1
1792 {
1793     \PitonStyle { Comment }
1794     {
1795         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1796         {
1797             \tl_set:Nn \l_tmpa_tl { #1 }
1798             \tl_replace_all:NVn \l_tmpa_tl
1799             \c_catcode_other_space_tl
1800             \@@_breakable_space:
1801             \l_tmpa_tl
1802         }
1803         { #1 }
1804     }
1805 }

```

```

1806 \cs_new_protected:Npn \@@_string_long:n #1
1807 {
1808     \PitonStyle { String.Long }
1809     {
1810         \bool_if:NTF \l_@@_break_strings_anywhere_bool
1811         { \@@_actually_break_anywhere:n { #1 } }
1812         {

```

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.

```

1813         \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1814         {
1815             \tl_set:Nn \l_tmpa_tl { #1 }
1816             \tl_replace_all:NVn \l_tmpa_tl
1817             \c_catcode_other_space_tl
1818             \@@_breakable_space:
1819             \l_tmpa_tl
1820         }
1821         { #1 }
1822     }
1823 }
1824 }
1825 \cs_new_protected:Npn \@@_string_short:n #1
1826 {
1827     \PitonStyle { String.Short }
1828     {
1829         \bool_if:NT \l_@@_break_strings_anywhere_bool
1830         { \@@_actually_break_anywhere:n }
1831         { #1 }
1832     }
1833 }

```

```

1834 \cs_new_protected:Npn \@@_string_doc:n #1
1835 {
1836   \PitonStyle { String.Doc }
1837   {
1838     \bool_if:NTF \l_@@_break_lines_in_Piton_bool
1839     {
1840       \tl_set:Nn \l_tmpa_tl { #1 }
1841       \tl_replace_all:NVn \l_tmpa_tl
1842         \c_catcode_other_space_tl
1843         \@@_breakable_space:
1844       \l_tmpa_tl
1845     }
1846     { #1 }
1847   }
1848 }
1849 \cs_new_protected:Npn \@@_number:n #1
1850 {
1851   \PitonStyle { Number }
1852   {
1853     \bool_if:NT \l_@@_break_numbers_anywhere_bool
1854     { \@@_actually_break_anywhere:n }
1855     { #1 }
1856   }
1857 }

```

## 2.0.9 The initial styles

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

```

1858 \SetPitonStyle
1859 {
1860   Comment           = \color [ HTML ] { 0099FF } \itshape ,
1861   Comment.Internal  = \@@_comment:n ,
1862   Exception         = \color [ HTML ] { CC0000 } ,
1863   Keyword           = \color [ HTML ] { 006699 } \bfseries ,
1864   Keyword.Governing = \color [ HTML ] { 006699 } \bfseries ,
1865   Keyword.Constant  = \color [ HTML ] { 006699 } \bfseries ,
1866   Name.Builtin      = \color [ HTML ] { 336666 } ,
1867   Name.Decorator    = \color [ HTML ] { 9999FF } ,
1868   Name.Class        = \color [ HTML ] { 00AA88 } \bfseries ,
1869   Name.Function     = \color [ HTML ] { CC00FF } ,
1870   Name.Namespace   = \color [ HTML ] { 00CCFF } ,
1871   Name.Constructor  = \color [ HTML ] { 006000 } \bfseries ,
1872   Name.Field        = \color [ HTML ] { AA6600 } ,
1873   Name.Module       = \color [ HTML ] { 0060A0 } \bfseries ,
1874   Name.Table        = \color [ HTML ] { 309030 } ,
1875   Number           = \color [ HTML ] { FF6600 } ,
1876   Number.Internal  = \@@_number:n ,
1877   Operator          = \color [ HTML ] { 555555 } ,
1878   Operator.Word     = \bfseries ,
1879   String            = \color [ HTML ] { CC3300 } ,
1880   String.Long.Internal = \@@_string_long:n ,
1881   String.Short.Internal = \@@_string_short:n ,
1882   String.Doc.Internal = \@@_string_doc:n ,
1883   String.Doc        = \color [ HTML ] { CC3300 } \itshape ,
1884   String.Interpol   = \color [ HTML ] { AA0000 } ,
1885   Comment.LaTeX     = \normalfont \color [ rgb ] { .468, .532, .6 } ,
1886   Name.Type         = \color [ HTML ] { 336666 } ,
1887   InitialValues     = \@@_piton:n ,
1888   Interpol.Inside   = { \l_@@_font_command_tl \@@_piton:n } ,
1889   TypeParameter     = \color [ HTML ] { 336666 } \itshape ,
1890   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).

```

1891 Identifier.Internal = \@@_identifier:n ,
1892 Identifier         = ,
1893 Directive         = \color [ HTML ] { AA6600} ,
1894 Tag                = \colorbox { gray!10 } ,
1895 UserFunction       = \PitonStyle { Identifier } ,
1896 Prompt            = ,
1897 Discard            = \use_none:n
1898 }

```

## 2.0.10 Styles specific to the language expl

```

1899 \clist_new:N \g_@@_expl_styles_clist
1900 \clist_gset:Nn \g_@@_expl_styles_clist
1901 {
1902   Scope.l ,
1903   Scope.g ,
1904   Scope.c
1905 }

1906 \clist_map_inline:Nn \g_@@_expl_styles_clist
1907 {
1908   \keys_define:nm { piton / Styles }
1909   {
1910     #1 .value_required:n = true ,
1911     #1 .code:n =
1912       \tl_set:cn
1913       {
1914         pitonStyle _
1915         \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1916         { \l_@@_SetPitonStyle_option_str _ }
1917         #1
1918       }
1919     { ##1 }
1920   }
1921 }

1922 \SetPitonStyle [ expl ]
1923 {
1924   Scope.l      = ,
1925   Scope.g      = \bfseries ,
1926   Scope.c      = \slshape ,
1927   Type.bool    = \color [ HTML ] { AA6600} ,
1928   Type.box     = \color [ HTML ] { 267910 } ,
1929   Type.clist   = \color [ HTML ] { 309030 } ,
1930   Type.fp      = \color [ HTML ] { FF3300 } ,
1931   Type.int     = \color [ HTML ] { FF6600 } ,
1932   Type.seq     = \color [ HTML ] { 309030 } ,
1933   Type.skip    = \color [ HTML ] { 0CC060 } ,
1934   Type.str     = \color [ HTML ] { CC3300 } ,
1935   Type.tl      = \color [ HTML ] { AA2200 } ,
1936   Module.bool  = \color [ HTML ] { AA6600} ,
1937   Module.box   = \color [ HTML ] { 267910 } ,
1938   Module.cs    = \bfseries \color [ HTML ] { 006699 } ,
1939   Module.exp   = \bfseries \color [ HTML ] { 404040 } ,
1940   Module.hbox  = \color [ HTML ] { 267910 } ,
1941   Module.prg   = \bfseries ,
1942   Module.clist = \color [ HTML ] { 309030 } ,
1943   Module.fp    = \color [ HTML ] { FF3300 } ,
1944   Module.int   = \color [ HTML ] { FF6600 } ,
1945   Module.seq   = \color [ HTML ] { 309030 } ,

```

```

1946 Module.skip      = \color [ HTML ] { OCC060 } ,
1947 Module.str       = \color [ HTML ] { CC3300 } ,
1948 Module.tl        = \color [ HTML ] { AA2200 } ,
1949 Module.vbox      = \color [ HTML ] { 267910 }
1950 }

```

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).

```

1951 \hook_gput_code:nnn { begindocument } { . }
1952 {
1953   \bool_if:NT \g_@@_math_comments_bool
1954     { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
1955 }

```

## 2.0.11 Highlighting some identifiers

```

1956 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
1957 {
1958   \clist_set:Nn \l_tmpa_clist { #2 }
1959   \tl_if_novalue:nTF { #1 }
1960     {
1961       \clist_map_inline:Nn \l_tmpa_clist
1962         { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
1963     }
1964     {
1965       \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
1966       \str_if_eq:onT \l_tmpa_str { current-language }
1967         { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
1968       \clist_map_inline:Nn \l_tmpa_clist
1969         { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
1970     }
1971 }
1972 \cs_new_protected:Npn \@@_identifier:n #1
1973 {
1974   \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }
1975     {
1976       \cs_if_exist_use:cF { PitonIdentifier _ #1 }
1977         { \PitonStyle { Identifier } }
1978     }
1979   { #1 }
1980 }

```

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`).

```

1981 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
1982 {

```

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

```

1983   { \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}`.

```

1984   \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ #1 }
1985     { \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`.**

```

1986   \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }

```

```

1987     { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
1988     \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.

```

1989     \seq_if_in:NoF \g_@@_languages_seq { \l_piton_language_str }
1990     { \seq_gput_left:No \g_@@_languages_seq { \l_piton_language_str } }
1991 }

```

```

1992 \NewDocumentCommand \PitonClearUserFunctions { ! o }
1993 {
1994     \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.

```

1995     { \@@_clear_all_functions: }
1996     { \@@_clear_list_functions:n { #1 } }
1997 }

```

```

1998 \cs_new_protected:Npn \@@_clear_list_functions:n #1
1999 {
2000     \clist_set:Nn \l_tmpa_clist { #1 }
2001     \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
2002     \clist_map_inline:nn { #1 }
2003     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
2004 }

```

```

2005 \cs_new_protected:Npn \@@_clear_functions_i:n #1
2006 { \@@_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).

```

2007 \cs_new_protected:Npn \@@_clear_functions_ii:n #1
2008 {
2009     \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
2010     {
2011         \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
2012         { \cs_undefine:c { PitonIdentifier _ #1 _ ##1 } }
2013         \seq_gclear:c { g_@@_functions _ #1 _ seq }
2014     }
2015 }
2016 \cs_generate_variant:Nn \@@_clear_functions_ii:n { e }

```

```

2017 \cs_new_protected:Npn \@@_clear_functions:n #1
2018 {
2019     \@@_clear_functions_i:n { #1 }
2020     \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
2021 }

```

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

```

2022 \cs_new_protected:Npn \@@_clear_all_functions:
2023 {
2024     \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
2025     \seq_gclear:N \g_@@_languages_seq
2026 }

```

```

2027 \AtEndDocument
2028 {

```

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

```

2029     \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.

```

2030 \IfPDFManagementActiveTF
2031   { \@@_join_files: }
2032   { \@@_join_files_legacy: }
2033 }

```

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

```

2034 % \cs_new_protected:Npn \@@_join_files:
2035 %   {
2036 %     \seq_map_inline:Nn \g_@@_join_seq
2037 %     {
2038 %

```

The group is for the modifications of the the dictionary l\_pdffile/Filespec

```

2039 %       \group_begin:
2040 %

```

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

```

2041 %       \pdf_object_new:n { piton / join / ##1 }
2042 %

```

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.

```

2043 %       \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2044 %

```

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

```

2045 %       \pdfdict_put:nnn { l_pdffile / Filespec } { AFRelationship } { /Supplement }
2046 %

```

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

```

2047 %       \pdfdict_put:nnn { l_pdffile / Filespec } { Desc } { (Computer~listing) }
2048 %       \pdffile_filespec:nnn { piton / join / ##1 } { ##1 } { \pdf_object_ref_last: }
2049 %

```

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.

```

2050 %       \pdfmanagement_add:nne
2051 %       { Catalog / Names }
2052 %       { EmbeddedFiles }
2053 %       { \pdf_object_ref_last: }
2054 %     \group_end:
2055 %   }
2056 % }
2057 %

```

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

```

2058 \cs_new_protected:Npn \@@_join_files:
2059   {
2060     \seq_map_inline:Nn \g_@@_join_seq
2061     {
2062       \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2063       \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2064       \pdfmanagement_add:nne { Catalog / Names } { EmbeddedFiles }
2065       {
2066         <<
2067           /Type /Filespec
2068           /UF <\l_tmpa_str>
2069           /EF << /F~\pdf_object_ref_last: >>
2070           /Desc (Computer~listing)
2071           /AFRelationship /Supplement
2072         >>
2073       }
2074     }
2075   }

```

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.

```

2076 \cs_new_protected:Npn \@@_join_files_legacy:
2077   {
2078     \seq_map_inline:Nn \g_@@_join_seq
2079     {
2080       \str_set_convert:Nnnn \l_tmpa_str { ##1 } { } { utf16/hex }
2081       \lua_now:n { pdf.immediateobj ( "stream" , piton.join_files["##1"] ) }
2082       \pdfextension annot~width~Opt~height~Opt~depth~Opt

```

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 Opt height Opt depth Opt`.

```

2083     {
2084       /Subtype /FileAttachment
2085       /F~2
2086       /Name /Paperclip
2087       /Contents (Computer~listing)
2088       /FS <<
2089         /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).

```

2090       /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`).

```

2091       /EF << /F~\pdf_object_ref_last: >>
2092       /AFRelationship /Supplement
2093     >>
2094   }
2095 }
2096 }

```

## 2.0.12 Spaces of indentation

```

2097 \cs_new_protected:Npn \@@_define_leading_space_normal:
2098   {
2099     \cs_set_protected:Npn \@@_leading_space:
2100     {
2101       \int_gincr:N \g_@@_indentation_int

```

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

```

2102     \hbox:n { ~ }
2103   }
2104 }

```

```

2105 \cs_new_protected:Npn \@@_define_leading_space_Foxit:
2106   {
2107     \cs_set_protected:Npn \@@_leading_space:
2108     {
2109       \int_gincr:N \g_@@_indentation_int
2110       \pdfextension literal { /Artifact << /ActualText (\space) >> BDC }
2111       {
2112         \color { white }
2113         \transparent { 0 }
2114         . % previously : ◻ U+2423
2115       }
2116       \pdfextension literal { EMC }
2117     }
2118 }

```

```
2119 \@@_define_leading_space_Foxit:
```

## 2.0.13 Security

```
2120 \AddToHook { env / piton / before }
2121 { \@@_fatal:n { No~environment~piton } }
```

## 2.0.14 The error messages of the package

```
2122 \@@_msg_new:nn { No~environment~piton }
2123 {
2124   There-is~no~environment~piton!\
2125   There-is~an~environment~{Piton}~and~a~command~
2126   \token_to_str:N \piton\ but~there-is~no~environment~
2127   {piton}.~This~error-is~fatal.
2128 }

2129 \@@_msg_new:nn { rounded-corners-without~Tikz }
2130 {
2131   TikZ-not~used \
2132   You~can't~use~the~key~'rounded-corners'~because~
2133   you~have~not~loaded~the~package~TikZ. \
2134   If~you~go~on,~that~key~will~be~ignored. \
2135   You~won't~have~similar~error~till~the~end~of~the~document.
2136 }

2137 \@@_msg_new:nn { tcolorbox-not~loaded }
2138 {
2139   tcolorbox-not~loaded \
2140   You~can't~use~the~key~'tcolorbox'~because~
2141   you~have~not~loaded~the~package~tcolorbox. \
2142   Use~\token_to_str:N \usepackage[breakable]{tcolorbox}. \
2143   If~you~go~on,~that~key~will~be~ignored.
2144 }

2145 \@@_msg_new:nn { library~breakable~not~loaded }
2146 {
2147   breakable~not~loaded \
2148   You~can't~use~the~key~'tcolorbox'~because~
2149   you~have~not~loaded~the~library~'breakable'~of~tcolorbox'. \
2150   Use~\token_to_str:N \tcbuselibrary{breakable}~in~the~preamble~
2151   of~your~document.\
2152   If~you~go~on,~that~key~will~be~ignored.
2153 }

2154 \@@_msg_new:nn { Language~not~defined }
2155 {
2156   Language~not~defined \
2157   The~language~'\l_tmpa_tl'~has~not~been~defined~previously.\
2158   If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
2159   will~be~ignored.
2160 }

2161 \@@_msg_new:nn { bad~version~of~piton.lua }
2162 {
2163   Bad~number~version~of~'piton.lua'\
2164   The~file~'piton.lua'~loaded~has~not~the~same~number~of~
2165   version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
2166   address~that~issue.
2167 }

2168 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
2169 {
2170   Unknown~key~for~\token_to_str:N \NewPitonLanguage.\
2171   The~key~'\l_keys_key_str'~is~unknown.\
2172   This~key~will~be~ignored.\
2173 }
```

```

2174 \@@_msg_new:nn { Unknown-key-for-SetPitonStyle }
2175 {
2176   The~style~'\l_keys_key_str'~is~unknown.\\
2177   This~setting~will~be~ignored.\\
2178   The~available~styles~are~(in~alphabetic~order):~
2179   \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
2180 }

2181 \@@_msg_new:nn { Invalid~key }
2182 {
2183   Wrong~use~of~key.\\
2184   You~can't~use~the~key~'\l_keys_key_str'~here.\\
2185   That~key~will~be~ignored.
2186 }

2187 \@@_msg_new:nn { Unknown~key~for~line~numbers }
2188 {
2189   Unknown~key. \\
2190   The~key~'line-numbers / \l_keys_key_str'~is~unknown.\\
2191   The~available~keys~of~the~family~'line-numbers'~are~(in~
2192   alphabetic~order):~
2193   absolute,~false,~label-empty-lines,~resume,~skip-empty-lines,~
2194   sep,~start~and~true.\\
2195   That~key~will~be~ignored.
2196 }

2197 \@@_msg_new:nn { Unknown~key~for~marker }
2198 {
2199   Unknown~key. \\
2200   The~key~'marker / \l_keys_key_str'~is~unknown.\\
2201   The~available~keys~of~the~family~'marker'~are~(in~
2202   alphabetic~order):~ beginning,~end~and~include-lines.\\
2203   That~key~will~be~ignored.
2204 }

2205 \@@_msg_new:nn { bad~range~specification }
2206 {
2207   Incompatible~keys.\\
2208   You~can't~specify~the~range~of~lines~to~include~by~using~both~
2209   markers~and~explicit~number~of~lines.\\
2210   Your~whole~file~'\l_@@_file_name_str'~will~be~included.
2211 }

2212 \cs_new_nopar:Nn \@@_thepage:
2213 {
2214   \thepage
2215   \cs_if_exist:NT \insertframenummer
2216     {
2217       ~(frame~\insertframenummer
2218         \cs_if_exist:NT \beamer@slidenummer { ,~slide~\insertslidenummer }
2219       )
2220     }
2221 }

```

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`.

```

2222 \@@_msg_new:nn { SyntaxError }
2223 {
2224   Syntax~Error~on~page~\@@_thepage:.\\
2225   Your~code~of~the~language~'\l_piton_language_str'~is~not~
2226   syntactically~correct.\\
2227   It~won't~be~printed~in~the~PDF~file.
2228 }

2229 \@@_msg_new:nn { FileError }
2230 {
2231   File~Error.\\

```

```

2232 It's~not~possible~to~write~on~the~file~'#1'~\
2233 \sys_if_shell_unrestricted:F
2234 { (try~to~compile~with~'lua~l~a~t~e~x~--shell~e~s~c~a~p~e')~\
2235 If~you~go~on,~nothing~will~be~written~on~that~file.
2236 }
2237 \@@_msg_new:nn { InexistentDirectory }
2238 {
2239 Inexistent~directory.\
2240 The~directory~'\l_@@_path_write_str'~
2241 given~in~the~key~'path~write'~does~not~exist.\
2242 Nothing~will~be~written~on~'\l_@@_write_str'.
2243 }
2244 \@@_msg_new:nn { begin~marker~not~found }
2245 {
2246 Marker~not~found.\
2247 The~range~'\l_@@_begin_range_str'~provided~to~the~
2248 command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
2249 The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
2250 }
2251 \@@_msg_new:nn { end~marker~not~found }
2252 {
2253 Marker~not~found.\
2254 The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
2255 provided~to~the~command~\token_to_str:N \PitonInputFile\
2256 has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
2257 be~inserted~till~the~end.
2258 }
2259 \@@_msg_new:nn { Unknown~file }
2260 {
2261 Unknown~file.~\
2262 The~file~'#1'~is~unknown.\
2263 Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
2264 }
2265 \cs_new_protected:Npn \@@_error_if_not_in_beamer:
2266 {
2267 \bool_if:NF \g_@@_beamer_bool
2268 { \@@_error_or_warning:n { Without~beamer } }
2269 }
2270 \@@_msg_new:nn { Without~beamer }
2271 {
2272 Key~'\l_keys_key_str'~without~Beamer.\
2273 You~should~not~use~the~key~'\l_keys_key_str'~since~you~
2274 are~not~in~Beamer.\
2275 However,~you~can~go~on.
2276 }
2277 \@@_msg_new:nn { rowcolor~in~detected~commands }
2278 {
2279 'rowcolor'~forbidden~in~'detected~commands'.\
2280 You~should~put~'rowcolor'~in~'raw~detected~commands'.\
2281 That~key~will~be~ignored.
2282 }
2283 \@@_msg_new:nnn { Unknown~key~for~PitonOptions }
2284 {
2285 Unknown~key.~\
2286 The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
2287 It~will~be~ignored.\
2288 For~a~list~of~the~available~keys,~type~H~<return>.
2289 }
2290 {
2291 The~available~keys~are~(in~alphabetic~order):~

```

```

2292 annotation,~
2293 add-to-split-separation,~
2294 auto-gobble,~
2295 background-color,~
2296 begin-range,~
2297 box,~
2298 break-lines,~
2299 break-lines-in-piton,~
2300 break-lines-in-Piton,~
2301 break-numbers-anywhere,~
2302 break-strings-anywhere,~
2303 continuation-symbol,~
2304 continuation-symbol-on-indentation,~
2305 detected-beamer-commands,~
2306 detected-beamer-environments,~
2307 detected-commands,~
2308 end-of-broken-line,~
2309 end-range,~
2310 env-gobble,~
2311 env-used-by-split,~
2312 font-command,~
2313 gobble,~
2314 indent-broken-lines,~
2315 join,~
2316 label-as-zlabel,~
2317 language,~
2318 left-margin,~
2319 line-numbers/,~
2320 marker/,~
2321 math-comments,~
2322 no-join,~
2323 no-write,~
2324 path,~
2325 path-write,~
2326 print,~
2327 prompt-background-color,~
2328 raw-detected-commands,~
2329 resume,~
2330 rounded-corners,~
2331 show-spaces,~
2332 show-spaces-in-strings,~
2333 splittable,~
2334 splittable-on-empty-lines,~
2335 split-on-empty-lines,~
2336 split-separation,~
2337 tabs-auto-gobble,~
2338 tab-size,~
2339 tcolorbox,~
2340 varwidth,~
2341 vertical-detected-commands,~
2342 width-and-write.
2343 }

2344 \@_msg_new:nn { label-with-lines-numbers }
2345 {
2346   You-can't-use-the-command~\token_to_str:N \label\
2347   or~\token_to_str:N \zlabel\ because-the-key-'line-numbers'
2348   ~is-not-active.\\
2349   If~you-go~on,~that~command~will~ignored.
2350 }

2351 \@_msg_new:nn { overlay-without-beamer }
2352 {

```

```

2353   You~can't~use~an~argument~<...>~for~your~command~
2354   \token_to_str:N \PitonInputFile\ because~you~are~not~
2355   in~Beamer.\\
2356   If~you~go~on,~that~argument~will~be~ignored.
2357 }

2358 \@@_msg_new:nn { label~as~zlabel~needs~zref~package }
2359 {
2360   The~key~'label-as-zlabel'~requires~the~package~'zref'.~
2361   Please~load~the~package~'zref'~before~setting~the~key.\\
2362   This~error~is~fatal.
2363 }
2364 \hook_gput_code:nnn { begindocument } { . }
2365 {
2366   \bool_if:NT \g_@@_label_as_zlabel_bool
2367   {
2368     \IfPackageLoadedF { zref-base }
2369     { \@@_fatal:n { label~as~zlabel~needs~zref~package } }
2370   }
2371 }

```

## 2.0.15 We load piton.lua

```

2372 \cs_new_protected:Npn \@@_test_version:n #1
2373 {
2374   \str_if_eq:onF \PitonFileVersion { #1 }
2375   { \@@_error:n { bad~version~of~piton.lua } }
2376 }

2377 \hook_gput_code:nnn { begindocument } { . }
2378 {
2379   \lua_load_module:n { piton }
2380   \lua_now:n
2381   {
2382     tex.sprint ( luatexbase.catcodetables.expl ,
2383                 [[\@@_test_version:n {}] .. piton_version .. "]" )
2384   }
2385 }
</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`.

```

2386 <*LUA>
2387 piton.comment_latex = piton.comment_latex or ">"
2388 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`.

```

2389 piton.write_files = { }
2390 piton.join_files = { }

2391 local sprintL3
2392 function sprintL3 ( s )
2393   tex.sprint ( luatexbase.catcodetables.expl , s )
2394 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.

```
2395 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
2396 local Cg, Cmt, Cb = lpeg.Cg, lpeg.Cmt, lpeg.Cb
2397 local B, R = lpeg.B, lpeg.R
```

The following line is mandatory.

```
2398 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”).

```
2399 local Q
2400 function Q ( pattern )
2401   return Ct ( Cc ( luatexbase.catcodetables.other ) * C ( pattern ) )
2402 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.

```
2403 local L
2404 function L ( pattern ) return
2405   Ct ( C ( pattern ) )
2406 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.

```
2407 local Lc
2408 function Lc ( string ) return
2409   Cc ( { luatexbase.catcodetables.expl, string } )
2410 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)

```
2411 local K
2412 function K ( style, pattern ) return
2413   Lc ( [[ {\PitonStyle{ }} .. style .. "}{" )
2414   * Q ( pattern )
2415   * Lc "}}}"
2416 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.

```

2417 local WithStyle
2418 function WithStyle ( style , pattern ) return
2419     Ct ( Cc "Open" * Cc ( [{"\PitonStyle{}}] .. style .. "}{" ) * Cc "}" )
2420     * pattern
2421     * Ct ( Cc "Close" )
2422 end

```

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

```

2423 Escape = P ( false )
2424 EscapeClean = P ( false )
2425 if piton.begin_escape then
2426     Escape =
2427         P ( piton.begin_escape )
2428         * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
2429         * 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).

```

2430 EscapeClean =
2431     P ( piton.begin_escape )
2432     * ( 1 - P ( piton.end_escape ) ) ^ 1
2433     * P ( piton.end_escape )
2434 end

2435 EscapeMath = P ( false )
2436 if piton.begin_escape_math then
2437     EscapeMath =
2438         P ( piton.begin_escape_math )
2439         * Lc "$"
2440         * L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
2441         * Lc "$"
2442         * P ( piton.end_escape_math )
2443 end

```

## The basic syntactic LPEG

```

2444 local alpha , digit = lpeg.alpha , lpeg.digit
2445 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.

```

2446 local letter = alpha + "_" + "â" + "ã" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
2447                 + "ô" + "û" + "ü" + "À" + "Á" + "Ç" + "É" + "Ê" + "Ë" + "Ï"
2448                 + "Ī" + "Ĳ" + "Ū" + "Ū" + "Ū"
2449
2450 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).

```

2451 local identifier = letter * alphanum ^ 0

```

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

```

2452 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.<sup>4</sup>

```
2453 local allow_underscores_except_first
2454 function allow_underscores_except_first ( p )
2455     return p * ( P "_" + p )^0
2456 end
2457 local allow_underscores
2458 function allow_underscores ( p )
2459     return ( P "_" + p )^0
2460 end
2461 local digits_to_number
2462 function digits_to_number(prefix, digits)
2463     -- The edge cases of what is allowed in number literals is modelled after
2464     -- OCaml numbers, which seems to be the most permissive language
2465     -- in this regard (among C, OCaml, Python & SQL).
2466     return prefix
2467         * allow_underscores_except_first(digits^1)
2468         * ( P "." * #(1 - P ".") * allow_underscores(digits))^~1
2469         * ( S "eE" * S "+-""^~1 * allow_underscores_except_first(digits^1))^~1
2470 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).

```
2471 local Number =
2472     K ( 'Number.Internal' ,
2473         digits_to_number ( P "0x" + P "OX", R "af" + R "AF" + digit )
2474         + digits_to_number ( P "0o" + P "OO", R "07" )
2475         + digits_to_number ( P "0b" + P "OB", R "01" )
2476         + digits_to_number ( "" , digit )
2477     )
```

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?).

```
2478 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`.

```
2479 if piton.begin_escape then
2480     lpeg_central = lpeg_central - piton.begin_escape
2481 end
2482 if piton.begin_escape_math then
2483     lpeg_central = lpeg_central - piton.begin_escape_math
2484 end
2485 local Word = Q ( lpeg_central ^ 1 )

2486 local Space = Q " " ^ 1
2487 local SkipSpace = Q " " ^ 0
2488
2489 local Punct = Q ( S ".,:;! " )
2490
2491 local Tab = "\t" * Lc [ [ \@_tab: ] ]
```

---

<sup>4</sup>The edge cases such as

```
2492 local LeadingSpace = Lc [[ \@@_leading_space: ]] * P " "
```

```
2493 local Delim = Q ( S "[({})]" )
```

The following LPEG catches a space (U+0020) and replaces it by `\l_@@_space_in_string_tl`. It will be used in the strings. Usually, `\l_@@_space_in_string_tl` 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_tl` will contain `␣` (U+2423) in order to visualize the spaces.

```
2494 local SpaceInString = space * Lc [[ \l_@@_space_in_string_tl ]]
```

### 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).

```
2495 local detected_commands = tex.toks.PitonDetectedCommands : explode ( ',' )
```

```
2496 local raw_detected_commands = tex.toks.PitonRawDetectedCommands : explode ( ',' )
```

```
2497 local beamer_commands = tex.toks.PitonBeamerCommands : explode ( ',' )
```

```
2498 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.

```
2499 local detectedCommands = P ( false )
```

```
2500 for _ , x in ipairs ( detected_commands ) do
```

```
2501   detectedCommands = detectedCommands + P ( "\\\" .. x )
```

```
2502 end
```

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

```
2503 local rawDetectedCommands = P ( false )
```

```
2504 for _ , x in ipairs ( raw_detected_commands ) do
```

```
2505   rawDetectedCommands = rawDetectedCommands + P ( "\\\" .. x )
```

```
2506 end
```

```
2507 local beamerCommands = P ( false )
```

```
2508 for _ , x in ipairs ( beamer_commands ) do
```

```
2509   beamerCommands = beamerCommands + P ( "\\\" .. x )
```

```
2510 end
```

```
2511 local beamerEnvironments = P ( false )
```

```
2512 for _ , x in ipairs ( beamer_environments ) do
```

```
2513   beamerEnvironments = beamerEnvironments + P ( x )
```

```
2514 end
```

## Several tools for the construction of the main LPEG

```
2515 local LPEG0 = { }
2516 local LPEG1 = { }
2517 local LPEG2 = { }
2518 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*.

```
2519 local Compute_braces
2520 function Compute_braces ( lpeg_string ) return
2521   P { "E" ,
2522     E =
2523       (
2524         "{ " * V "E" * "}"
2525         +
2526         lpeg_string
2527         +
2528         ( 1 - S "{" )
2529       ) ^ 0
2530   }
2531 end
```

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

```
2532 local Compute_DetectedCommands
2533 function Compute_DetectedCommands ( lang , braces ) return
2534   Ct (
2535     Cc "Open"
2536     * C ( detectedCommands * space ^ 0 * P "{" )
2537     * Cc "}"
2538   )
2539   * ( braces
2540     / ( function ( s )
2541         if s ~= '' then return
2542           LPEG1[lang] : match ( s )
2543         end
2544       end )
2545   )
2546   * P "}"
2547   * Ct ( Cc "Close" )
2548 end
2549 local Compute_RawDetectedCommands
2550 function Compute_RawDetectedCommands ( lang , braces ) return
2551   Ct ( C ( rawDetectedCommands * space ^ 0 * P "{" * braces * P "}" ) )
2552 end
2553 local Compute_LPEG_cleaner
2554 function Compute_LPEG_cleaner ( lang , braces ) return
2555   Ct ( ( ( detectedCommands + rawDetectedCommands ) * "{"
2556     * ( braces
2557       / ( function ( s )
2558           if s ~= '' then return
2559             LPEG_cleaner[lang] : match ( s )
2560           end
2561         end )
2562     )
2563     * "}"
2564     + EscapeClean
```

```

2565         + C ( P ( 1 ) )
2566     ) ^ 0 ) / table.concat
2567 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`.

```

2568 local ParseAgain
2569 function ParseAgain ( code )
2570     if code ~= '' then return

```

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

```

2571     LPEG1[piton.language] : match ( code )
2572 end
2573 end

```

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

```

2574 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.

```

2575 local Compute_Beamer
2576 function Compute_Beamer ( lang , braces )

```

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

```

2577     local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
2578     lpeg = lpeg +
2579         Ct ( Cc "Open"
2580             * C ( beamerCommands
2581                 * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2582                 * P "{"
2583             )
2584             * Cc "}"
2585         )
2586     * ( braces /
2587         ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2588     * "]"
2589     * Ct ( Cc "Close" )

```

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

```

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

```

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

```

2598 lpeg = lpeg +
2599   L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
2600   * ( braces
2601     / ( function ( s )
2602       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2603   * L ( P "}" )
2604   * ( braces
2605     / ( function ( s )
2606       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2607   * L ( P "}" )
2608   * ( braces
2609     / ( function ( s )
2610       if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
2611   * L ( P "}" )

```

Now, the environments of Beamer.

```

2612 for _ , x in ipairs ( beamer_environments ) do
2613   lpeg = lpeg +
2614     Ct ( Cc "Open"
2615       * C (
2616         P ( [[\begin{]] .. x .. "]" )
2617         * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
2618       )
2619       * space ^ 0 * ( P "\r" ) ^ 1 -- added 25/08/23
2620       * Cc ( [[\end{]] .. x .. "]" )
2621     )
2622   * (
2623     ( ( 1 - P ( [[\end{]] .. x .. "]" ) ) ^ 0 )
2624     / ( function ( s )
2625       if s ~= '' then return
2626         LPEG1[lang] : match ( s )
2627       end
2628     end )
2629   )
2630   * P ( [[\end{]] .. x .. "]" )
2631   * Ct ( Cc "Close" )
2632 end

```

Now, you can return the value we have computed.

```

2633 return lpeg
2634 end

```

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

```

2635 local CommentMath =
2636   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`.

```

2637 local Prompt =
2638   K ( 'Prompt' , ( P ">>>" + "...") * P " " ^ -1 )
2639   * Lc [[ \rowcolor { \l_@@_prompt_bg_color_tl } ]]

```

The following LPEG EOL is for the end of lines.

```

2640 local EOL =
2641   P "\r"
2642   *
2643   (
2644     space ^ 0 * -1

```

```

2645     +
2646     Cc "EOL"
2647   )
2648   * ( LeadingSpace ^ 0 * # ( 1 - S "\r" ) ) ^ -1

```

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

```

2649 local CommentLaTeX =
2650   P ( piton.comment_latex )
2651   * Lc [[{\PitonStyle{Comment.LaTeX}{\ignorespaces}}]
2652   * L ( ( 1 - P "\r" ) ^ 0 )
2653   * Lc "}}"
2654   * ( 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).

```

2655 --python Python
2656 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.

```

2657 local Operator =
2658   K ( 'Operator' ,
2659     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + "!=" + "://" + "**"
2660     + S "--+/*%=<>&.@|" )
2661
2662 local OperatorWord =
2663   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`.

```

2664 local For = K ( 'Keyword' , P "for" )
2665     * Space
2666     * Identifier
2667     * Space
2668     * K ( 'Keyword' , P "in" )
2669
2670 local Keyword =
2671   K ( 'Keyword' ,
2672     P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
2673     "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
2674     "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
2675     "try" + "while" + "with" + "yield" + "yield from" )
2676   + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
2677
2678 local Builtin =
2679   K ( 'Name.Builtin' ,
2680     P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
2681     "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
2682     "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
2683     "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
2684     "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
2685     "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
2686     + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
2687     "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
2688     "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
2689     "vars" + "zip" )
2690
2691 local Exception =
2692   K ( 'Exception' ,

```

```

2693 P "ArithmeticError" + "AssertionError" + "AttributeError" +
2694 "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
2695 "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
2696 "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
2697 "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
2698 "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
2699 "NotImplementedError" + "OSError" + "OverflowError" +
2700 "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
2701 "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
2702 "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
2703 + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
2704 "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
2705 "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
2706 "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
2707 "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
2708 "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
2709 "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
2710 "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
2711 "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
2712 "RecursionError" )
2713
2714 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.

```

2715 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:`

```

2716 local DefClass =
2717 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`

```

2718 local ImportAs =
2719 K ( 'Keyword' , "import" )
2720 * Space
2721 * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
2722 * (
2723 ( Space * K ( 'Keyword' , "as" ) * Space
2724 * K ( 'Name.Namespace' , identifier ) )
2725 +
2726 ( SkipSpace * Q "," * SkipSpace
2727 * K ( 'Name.Namespace' , identifier ) ) ^ 0
2728 )

```

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`

```
2729 local FromImport =
2730   K ( 'Keyword' , "from" )
2731   * Space * K ( 'Name.Namespace' , identifier )
2732   * 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 instruction<sup>5</sup> in that interpolation:

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

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

```
2733 local PercentInterpol =
2734   K ( 'String.Interpol' ,
2735     P "%"
2736     * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2737     * ( S "-#0 +" ) ^ 0
2738     * ( digit ^ 1 + "*" ) ^ -1
2739     * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2740     * ( S "HLL" ) ^ -1
2741     * S "sdfFeExXorgiGauc%"
2742   )
```

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.<sup>6</sup>

```
2743 local SingleShortString =
2744   WithStyle ( 'String.Short.Internal' ,
```

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

```
2745     Q ( P "f'" + "F'" )
2746     * (
2747       K ( 'String.Interpol' , "{" )
2748       * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0 )
2749       * Q ( P ":" * ( 1 - S "}':" ) ^ 0 ) ^ -1
2750       * K ( 'String.Interpol' , "}" )
2751     +
2752     SpaceInString
2753     +
2754     Q ( ( P "\\'" + "\\\\" + "{{" + "}" ) + 1 - S " {}'" ) ^ 1 )
2755     ) ^ 0
2756     * Q ""
2757     +
```

<sup>5</sup>There 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`.

<sup>6</sup>The interpolations are formatted with the piton style `Interpol.Inside`. The initial value of that style is `\\_\\_piton:` which means that the interpolations are parsed once again by piton.

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

```

2758     Q ( P "" + "r" + "R" )
2759     * ( Q ( ( P "\\\" + "\\\" + 1 - S " \"r%\" ) ^ 1 )
2760         + SpaceInString
2761         + PercentInterpol
2762         + Q \"%\"
2763     ) ^ 0
2764     * Q "" )
2765 local DoubleShortString =
2766     WithStyle ( 'String.Short.Internal' ,
2767         Q ( P "f\" + "F\" )
2768         * (
2769             K ( 'String.Interpol' , "{" )
2770             * K ( 'Interpol.Inside' , ( 1 - S "}\" ) ^ 0 )
2771             * ( K ( 'String.Interpol' , ":" ) * Q ( ( 1 - S "}:\") ^ 0 ) ) ^ -1
2772             * K ( 'String.Interpol' , "}" )
2773         +
2774         SpaceInString
2775         +
2776         Q ( ( P "\\\" + "\\\" + "{" + "}" + 1 - S " {}\" ) ^ 1 )
2777     ) ^ 0
2778     * Q "\"
2779 +
2780     Q ( P "\" + "r\" + "R\" )
2781     * ( Q ( ( P "\\\" + "\\\" + 1 - S " \"r%\" ) ^ 1 )
2782         + SpaceInString
2783         + PercentInterpol
2784         + Q \"%\"
2785     ) ^ 0
2786     * Q "\" )
2787
2788 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.

```

2789 local braces =
2790     Compute_braces
2791     (
2792         ( P "\" + "r\" + "R\" + "f\" + "F\" )
2793         * ( P '\\\" + 1 - S "\" ) ^ 0 * "\"
2794     +
2795         ( P '\\" + 'r\' + 'R\' + 'f\' + 'F\' )
2796         * ( P '\\\' + 1 - S '\\" ) ^ 0 * '\\"
2797     )
2798
2799 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

### Detected commands

```

2800 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
2801     + Compute_RawDetectedCommands ( 'python' , braces )

```

### LPEG\_cleaner

```

2802 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )

```

## The long strings

```

2803 local SingleLongString =
2804     WithStyle ( 'String.Long.Internal' ,
2805         ( Q ( S "fF" * P "'''''' )
2806             * (
2807                 K ( 'String.Interpol' , "{" )
2808                 * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "'''''' ) ^ 0 )
2809                 * Q ( P ":" * ( 1 - S "}:\\r" - "'''''' ) ^ 0 ) ^ -1
2810                 * K ( 'String.Interpol' , "}" )
2811             +
2812             Q ( ( 1 - P "'''''' - S "{}\\r" ) ^ 1 )
2813             +
2814             EOL
2815         ) ^ 0
2816     +
2817     Q ( ( S "rR" ) ^ -1 * "'''''' )
2818     * (
2819         Q ( ( 1 - P "'''''' - S "\\r%" ) ^ 1 )
2820         +
2821         PercentInterpol
2822         +
2823         P "%"
2824         +
2825         EOL
2826     ) ^ 0
2827 )
2828 * Q "'''''' )

2829 local DoubleLongString =
2830     WithStyle ( 'String.Long.Internal' ,
2831         (
2832             Q ( S "fF" * "\\\"\\\"\\\" )
2833             * (
2834                 K ( 'String.Interpol' , "{" )
2835                 * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\\\"\\\"\\\" ) ^ 0 )
2836                 * Q ( ":" * ( 1 - S "}:\\r" - "\\\"\\\"\\\" ) ^ 0 ) ^ -1
2837                 * K ( 'String.Interpol' , "}" )
2838             +
2839             Q ( ( 1 - S "{}\\r" - "\\\"\\\"\\\" ) ^ 1 )
2840             +
2841             EOL
2842         ) ^ 0
2843     +
2844     Q ( S "rR" ^ -1 * "\\\"\\\"\\\" )
2845     * (
2846         Q ( ( 1 - P "\\\"\\\"\\\" - S "%\\r" ) ^ 1 )
2847         +
2848         PercentInterpol
2849         +
2850         P "%"
2851         +
2852         EOL
2853     ) ^ 0
2854 )
2855 * Q "\\\"\\\"\\\" )

2856 )

2857 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).

```

2858 local StringDoc =
2859     K ( 'String.Doc.Internal' , P "r" ^ -1 * "\\\"\\\"\\\" )
2860     * ( K ( 'String.Doc.Internal' , ( 1 - P "\\\"\\\"\\\" - "\\r" ) ^ 0 ) * EOL

```

```

2861         * Tab ^ 0
2862     ) ^ 0
2863     * 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.

```

2864     local Comment =
2865         WithStyle
2866         ( 'Comment.Internal' ,
2867         Q "#" * ( CommentMath + Q ( ( 1 - S "$r\" ) ^ 1 ) ) ^ 0 -- $
2868         )
2869     * ( 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).

```

2870     local expression =
2871     P { "E" ,
2872         E = ( "" * ( P "\'" + 1 - S "'r\" ) ^ 0 * ""
2873             + "" * ( P "\\\" + 1 - S "\\r\" ) ^ 0 * ""
2874             + "{" * V "F" * "}"
2875             + "(" * V "F" * ")"
2876             + "[" * V "F" * "]"
2877             + ( 1 - S "{}()[]\r,\" ) ) ^ 0 ,
2878         F = ( "{" * V "F" * "}"
2879             + "(" * V "F" * ")"
2880             + "[" * V "F" * "]"
2881             + ( 1 - S "{}()[]\r\" ) ) ^ 0
2882     }

```

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`.

```

2883     local Params =
2884     P { "E" ,
2885         E = ( V "F" * ( Q ",\" * V "F" ) ^ 0 ) ^ -1 ,
2886         F = SkipSpace * ( Identifier + Q "*args\" + Q "**kwargs\" ) * SkipSpace
2887             * ( Q ":" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
2888             * ( SkipSpace * K ( 'InitialValues' , "=" * SkipSpace * expression ) ) ^ -1
2889     }

```

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`...

```

2890     local DefFunction =
2891     K ( 'Keyword' , "def\" )
2892     * Space
2893     * K ( 'Name.Function.Internal' , identifier )
2894     * SkipSpace
2895     * Q "(" * Params * Q ")"
2896     * SkipSpace
2897     * ( Q "->\" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1

```

```

2898 * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
2899 * Q ":"
2900 * ( SkipSpace
2901   * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
2902   * Tab ^ 0
2903   * SkipSpace
2904   * StringDoc ^ 0 -- there may be additional docstrings
2905 ) ^ -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

```

2906 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL

```

## The main LPEG for the language Python

```

2907 local EndKeyword
2908   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
2909   EscapeMath + -1

```

First, the main loop :

```

2910 local Main =
2911   space ^ 0 * EOL -- faut-il le mettre en commentaire ?
2912   + Space
2913   + Tab
2914   + Escape + EscapeMath
2915   + Beamer
2916   + CommentLaTeX
2917   + DetectedCommands
2918   + Prompt
2919   + LongString
2920   + Comment
2921   + ExceptionInConsole
2922   + Delim
2923   + Operator
2924   + OperatorWord * EndKeyword
2925   + ShortString
2926   + Punct
2927   + FromImport
2928   + RaiseException
2929   + DefFunction
2930   + DefClass
2931   + For
2932   + Keyword * EndKeyword
2933   + Decorator
2934   + Builtin * EndKeyword
2935   + Identifier
2936   + Number
2937   + Word

```

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

```

2938 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:`<sup>7</sup>.

---

<sup>7</sup>Remember 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:`

```

2939 LPEG2.python =
2940   Ct (
2941     ( space ^ 0 * "\r" ) ^ -1
2942     * Lc [[ \@@_begin_line: ]]
2943     * LeadingSpace ^ 0
2944     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2945     * -1
2946     * Lc [[ \@@_end_line: ]]
2947   )

```

End of the Lua scope for the language Python.

```
2948 end
```

### 3.5 The language OCaml

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

```

2949 --ocaml Ocaml OCaml
2950 do
2951   local SkipSpace = ( Q " " + EOL ) ^ 0
2952   local Space = ( Q " " + EOL ) ^ 1
2953   local braces = Compute_braces ( '\"' * ( 1 - S "\"" ) ^ 0 * '\"' )
2954   if piton.beamer then Beamer = Compute_Beamer ( 'ocaml' , braces ) end
2955   DetectedCommands =
2956     Compute_DetectedCommands ( 'ocaml' , braces )
2957     + Compute_RawDetectedCommands ( 'ocaml' , braces )
2958   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`.

```

2959   function Q ( pattern, strict )
2960     if strict ~= nil then
2961       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2962     else
2963       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2964         + Beamer + DetectedCommands + EscapeMath + Escape
2965     end
2966   end
2967   local K
2968   function K ( style , pattern, strict ) return
2969     Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
2970     * Q ( pattern, strict )
2971     * Lc "}"
2972   end
2973   local WithStyle
2974   function WithStyle ( style , pattern ) return
2975     Ct ( Cc "Open" * Cc ( [[{\PitonStyle{]] .. style .. "}{" ) * Cc "}" )
2976     * ( pattern + Beamer + DetectedCommands + EscapeMath + Escape )
2977     * Ct ( Cc "Close" )
2978   end

```

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

```

2979   local balanced_parens =
2980     P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "(" ) ) ^ 0 }

```

## The strings of OCaml

```
2981 local ocaml_string =
2982   P "\""
2983 * (
2984   P " "
2985   +
2986   P ( ( 1 - S " \"\r" ) ^ 1 )
2987   +
2988   EOL -- ?
2989 ) ^ 0
2990 * P "\""
2991
2992 local String =
2993   WithStyle
2994     ( 'String.Long.Internal' ,
2995       Q "\""
2996       * (
2997         SpaceInString
2998         +
2999         Q ( ( 1 - S " \"\r" ) ^ 1 )
3000         +
3001         EOL
3002       ) ^ 0
3003     )
3004     * Q "\""
3005   )
```

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

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

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

```
3004 local ext = ( R "az" + "_" ) ^ 0
3005 local open = "{" * Cg ( ext , 'init' ) * "/"
3006 local close = "/" * C ( ext ) * "}"
3007 local closeeq =
3008   Cmt ( close * Cb ( 'init' ) ,
3009     function ( s , i , a , b ) return a == b end )
```

The LPEG `QuotedStringBis` will do the second analysis.

```
3010 local QuotedStringBis =
3011   WithStyle ( 'String.Long.Internal' ,
3012     (
3013       Space
3014       +
3015       Q ( ( 1 - S " \r" ) ^ 1 )
3016       +
3017       EOL
3018     ) ^ 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.

```
3019 local QuotedString =
3020   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
3021   ( 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).

```
3022 local comment =
3023   P {
```

```

3024     "A" ,
3025     A = Q "(*"
3026         * ( V "A"
3027             + Q ( ( 1 - S "\r$" - "(*" - "*" ) ^ 1 ) -- $
3028             + ocaml_string
3029             + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
3030             + EOL
3031             ) ^ 0
3032         * Q "*" )
3033     }
3034     local Comment = WithStyle ( 'Comment.Internal' , comment )

```

### Some standard LPEG

```

3035     local Delim = Q ( P "[" + "]" + S "()" )
3036     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.

```

3037     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.

```

3038     local Constructor =
3039         P "::"

```

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

```

3040     * Lc [[{\PitonStyle{Name.Constructor}{\kern0.1em:\kern-0.2em:\kern0.1em}}]]
3041     +
3042     P "[]"
3043     * Lc ([{\PitonStyle{Name.Constructor}{\kern-0.1em[\kern0.1em}}]])
3044     K ( 'Name.Constructor' ,
3045         Q "`" ^ -1 * cap_identifier
3046         + Q ( "[" , true ) * SkipSpace * Q ( "]" , true ) )
3047     local ModuleType = K ( 'Name.Type' , cap_identifier )

```

```

3048     local OperatorWord =
3049         K ( 'Operator.Word' ,
3050             P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )

```

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

```

3051     local governing_keyword = P "and" + "begin" + "class" + "constraint" +
3052         "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
3053         "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
3054         "struct" + "type" + "val"
3055     local Keyword =
3056         K ( 'Keyword' ,
3057             P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
3058             + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
3059             + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
3060             + "virtual" + "when" + "while" + "with" )
3061         + K ( 'Keyword.Constant' , P "true" + "false" )
3062         + K ( 'Keyword.Governing' , governing_keyword )
3063     local EndKeyword
3064         = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
3065         + EscapeMath + -1

```

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

```
3066 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
3067                   - ( 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`.

```
3068 local Identifier = K ( 'Identifier.Internal' , identifier )
```

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

```
3069 local ocaml_char =
3070   P "' '*
3071   (
3072     ( 1 - S "\\\" )
3073     + "\\\"
3074     * ( S "\\ntbr \"
3075         + digit * digit * digit
3076         + P "x" * ( digit + R "af" + R "AF" )
3077                 * ( digit + R "af" + R "AF" )
3078                 * ( digit + R "af" + R "AF" )
3079         + P "o" * R "03" * R "07" * R "07" )
3080   )
3081   * "'*
3082 local Char =
3083   K ( 'String.Short.Internal' , ocaml_char )
```

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

```
3084 local TypeParameter =
3085   K ( 'TypeParameter' ,
3086       "'* 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.

```
3087 local DotNotation =
3088   (
3089     K ( 'Name.Module' , cap_identifier )
3090     * Q "."
3091     * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3092     +
3093     Identifier
3094     * Q "."
3095     * K ( 'Name.Field' , identifier )
3096   )
3097   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0
```

## The records

```
3098 local expression_for_fields_type =
3099   P { "E" ,
3100     E = ( "{ * V "F" * }"
3101           + "(" * V "F" * ")"
3102           + TypeParameter
3103           + ( 1 - S "{}() []\r;" ) ) ^ 0 ,
3104     F = ( "{ * V "F" * }"
3105           + "(" * V "F" * ")"
3106           + ( 1 - S "{}() []\r\"" ) + TypeParameter ) ^ 0
3107   }
```

```

3108 local expression_for_fields_value =
3109   P { "E" ,
3110     E = (   "{" * V "F" * "}"
3111           + "(" * V "F" * ")"
3112           + "[" * V "F" * "]"
3113           + ocaml_string + ocaml_char
3114           + ( 1 - S "{}()[];" ) ^ 0 ,
3115     F = (   "{" * V "F" * "}"
3116           + "(" * V "F" * ")"
3117           + "[" * V "F" * "]"
3118           + ocaml_string + ocaml_char
3119           + ( 1 - S "{}()[]\\"" ) ^ 0
3120   }

3121 local OneFieldDefinition =
3122   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
3123   * K ( 'Name.Field' , identifier ) * SkipSpace
3124   * Q ":" * SkipSpace
3125   * K ( 'TypeExpression' , expression_for_fields_type )
3126   * SkipSpace

3127 local OneField =
3128   K ( 'Name.Field' , identifier ) * SkipSpace
3129   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

3130   * ( C ( expression_for_fields_value ) / ParseAgain )
3131   * SkipSpace

```

The records.

```

3132 local RecordVal =
3133   Q "{" * SkipSpace
3134   *
3135   (
3136     (Identifier + DotNotation) * Space * K('Keyword', "with") * Space
3137   ) ^ -1
3138   *
3139   (
3140     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
3141   )
3142   * SkipSpace
3143   * Q ";" ^ -1
3144   * SkipSpace
3145   * Comment ^ -1
3146   * SkipSpace
3147   * Q "}"

3148 local RecordType =
3149   Q "{" * SkipSpace
3150   *
3151   (
3152     OneFieldDefinition
3153     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
3154   )
3155   * SkipSpace
3156   * Q ";" ^ -1
3157   * SkipSpace
3158   * Comment ^ -1
3159   * SkipSpace
3160   * Q "}"

3161 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.

```

3162 local DotNotation =
3163   (
3164     K ( 'Name.Module' , cap_identifier )
3165       * Q "."
3166       * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
3167     +
3168     Identifier
3169       * Q "."
3170       * K ( 'Name.Field' , identifier )
3171   )
3172   * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

3173 local Operator =
3174   P "||" *

```

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

```

3175   Lc([[{\PitonStyle{Operator}}{\kern0.1em/\kern-0.2em/\kern0.1em}]]))
3176   +
3177   K ( 'Operator' ,
3178     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":@" + "&&" +
3179     "//" + "*" + ";" + "->" + "+," + "-." + "*." + "/" +
3180     + S "--+/*%=<>&@|" )

```

```

3181 local Builtin =
3182   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

```

```

3183 local Exception =
3184   K ( 'Exception' ,
3185     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
3186     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
3187     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

```

```

3188 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).

```

3189 local pattern_part =
3190   ( 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).

```
3191 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.

```

3192   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
3193   *

```

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

```

3194   (
3195     K ( 'Identifier.Internal' , identifier )
3196     +
3197     Q "(" * SkipSpace
3198     * ( C ( pattern_part ) / ParseAgain )
3199     * SkipSpace

```

Of course, the specification of type is optional.

```

3200     * ( Q ":" * #(1- P"=")
3201         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace
3202     ) ^ -1
3203     * Q ")"
3204 )

```

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

```

3205 local DefFunction =
3206   K ( 'Keyword.Governing' , "let open" )
3207   * Space
3208   * K ( 'Name.Module' , cap_identifier )
3209   +
3210   K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
3211   * Space
3212   * K ( 'Name.Function.Internal' , identifier )
3213   * Space
3214   * (

```

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).

```

3215     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
3216     +
3217     Argument * ( SkipSpace * Argument ) ^ 0
3218     * (
3219         SkipSpace
3220         * Q ":" * # ( 1 - P "=" )
3221         * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
3222     ) ^ -1
3223 )

```

## DefModule

```

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

```

```

3254         (
3255             Q ", "
3256             *
3257             K ( 'Name.Module' , cap_identifier ) * SkipSpace
3258             ) ^ 0
3259             * Q ")"
3260         ) ^ -1
3261     )
3262 +
3263 K ( 'Keyword.Governing' , P "include" + "open" )
3264 * Space
3265 * K ( 'Name.Module' , cap_identifier )

```

## DefType

```

3266 local DefType =
3267     K ( 'Keyword.Governing' , "type" )
3268     * Space
3269     * K ( 'TypeExpression' , Q ( 1 - P "=" - P "+=" ) ^ 1 )
3270     * SkipSpace
3271     * ( Q "+=" + Q "=" )
3272     * SkipSpace
3273     * (
3274         RecordType
3275         +

```

The following lines are a suggestion of Y. Salmon.

```

3276     WithStyle
3277     (
3278         'TypeExpression' ,
3279         (
3280             (
3281                 EOL
3282                 + comment
3283                 + Q ( 1
3284                     - P ";;"
3285                     - P "type"
3286                     - ( ( Space + EOL ) * governing_keyword * EndKeyword )
3287                 )
3288             ) ^ 0
3289             *
3290             (
3291                 # ( P "type" + ( Space + EOL ) * governing_keyword * EndKeyword )
3292                 + Q ";;"
3293                 + -1
3294             )
3295         )
3296     )
3297 )

3298 local prompt =
3299     Q "utop[" * digit^1 * Q "> "
3300 local start_of_line = P(function(subject, position)
3301 if position == 1 or subject:sub(position - 1, position - 1) == "\r" then
3302     return position
3303 end
3304 return nil
3305 end)
3306 local Prompt = #start_of_line * K( 'Prompt', prompt )
3307 local Answer = #start_of_line * (Q "-" + Q "val" * Space * Identifier )
3308             * SkipSpace * Q ":" * #(1- P"=") * SkipSpace
3309             * ( K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 ) ) * SkipSpace * Q "="

```

## The main LPEG for the language OCaml

```
3310 local Main =
3311     space ^ 0 * EOL
3312     + Space
3313     + Tab
3314     + Escape + EscapeMath
3315     + Beamer
3316     + DetectedCommands
3317     + TypeParameter
3318     + String + QuotedString + Char
3319     + Comment
3320     + Prompt + Answer
```

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

```
3321     + Q "~" * Identifier * ( Q ":" ) ^ -1
3322     + Q ":" * # ( 1 - P ":" ) * SkipSpace
3323         * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
3324     + Exception
3325     + DefType
3326     + DefFunction
3327     + DefModule
3328     + Record
3329     + Keyword * EndKeyword
3330     + OperatorWord * EndKeyword
3331     + Builtin * EndKeyword
3332     + DotNotation
3333     + Constructor
3334     + Identifier
3335     + Punct
3336     + Delim -- Delim is before Operator for a correct analysis of [| et |]
3337     + Operator
3338     + Number
3339     + Word
```

Here, we must not put local, of course.

```
3340 LPEG1.ocaml = Main ^ 0
```

```
3341 LPEG2.ocaml =
```

```
3342 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).

```
3343     ( P ":" + ( K ( 'Name.Module' , cap_identifier ) * Q "." ) ^ -1
3344       * Identifier * SkipSpace * Q ":" )
3345     * # ( 1 - S "!=" )
3346     * SkipSpace
3347     * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 )
3348     +
3349     ( space ^ 0 * "\r" ) ^ -1
3350     * Lc [ [ \@@_begin_line: ] ]
3351     * LeadingSpace ^ 0
3352     * ( ( space * Lc [ [ \@@_trailing_space: ] ] ) ^ 1 * -1
3353       + space ^ 0 * EOL
3354       + Main
3355     ) ^ 0
3356     * -1
3357     * Lc [ [ \@@_end_line: ] ]
3358 )
```

End of the Lua scope for the language OCaml.

```
3359 end
```

## 3.6 The language C

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

```
3360 --c C c++ C++
3361 do

3362     local Delim = Q ( S "{[()]} " )
3363     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.

```
3364     local identifier = letter * alphanum ^ 0
3365
3366     local Operator =
3367         K ( 'Operator' ,
3368             P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
3369             + S "--+/*%=<>&.@|!" )
3370
3371     local Keyword =
3372         K ( 'Keyword' ,
3373             P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
3374             "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
3375             "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
3376             "register" + "restricted" + "return" + "static" + "static_assert" +
3377             "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
3378             "union" + "using" + "virtual" + "volatile" + "while"
3379         )
3380         + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
3381
3382     local Builtin =
3383         K ( 'Name.Builtin' ,
3384             P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
3385
3386     local Type =
3387         K ( 'Name.Type' ,
3388             P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" +
3389             "int8_t" + "int16_t" + "int32_t" + "int64_t" + "uint8_t" + "uint16_t" +
3390             "uint32_t" + "uint64_t" + "int" + "long" + "short" + "signed" + "unsigned" +
3391             "void" + "wchar_t" ) * Q "*" ^ 0
3392
3393     local DefFunction =
3394         Type
3395         * Space
3396         * Q "*" ^ -1
3397         * K ( 'Name.Function.Internal' , identifier )
3398         * SkipSpace
3399         * # 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:`

```
3400     local DefClass =
3401         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).

```
3402     local Character =
3403         K ( 'String.Short' ,
3404             P "[[\\']]" + P "'" * ( 1 - P "'" ) ^ 0 * P "'" )
```

## The strings of C

```
3405 String =
3406   WithStyle ( 'String.Long.Internal' ,
3407     Q "\""
3408     * ( SpaceInString
3409       + K ( 'String.Interpol' ,
3410         "%" * ( S "difcspXou" + "ld" + "li" + "hd" + "hi" )
3411       )
3412       + Q ( ( P "\\\"" + 1 - S " \" " ) ^ 1 )
3413     ) ^ 0
3414     * Q "\""
3415   )
```

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

```
3416 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
3417 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
3418 DetectedCommands =
3419   Compute_DetectedCommands ( 'c' , braces )
3420   + Compute_RawDetectedCommands ( 'c' , braces )
3421 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )
```

## The directives of the preprocessor

```
3422 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.

```
3423 local Comment =
3424   WithStyle ( 'Comment.Internal' ,
3425     Q "/*" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3426     * ( EOL + -1 )
3427
3428 local LongComment =
3429   WithStyle ( 'Comment.Internal' ,
3430     Q "/*"
3431     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3432     * Q "*/"
3433   ) -- $
```

## The main LPEG for the language C

```
3434 local EndKeyword
3435   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3436     EscapeMath + -1
```

First, the main loop :

```
3437 local Main =
3438   space ^ 0 * EOL
3439   + Space
3440   + Tab
3441   + Escape + EscapeMath
3442   + CommentLaTeX
3443   + Beamer
3444   + DetectedCommands
3445   + Preproc
```

```

3446     + Comment + LongComment
3447     + Delim
3448     + Operator
3449     + Character
3450     + String
3451     + Punct
3452     + DefFunction
3453     + DefClass
3454     + Type * ( Q "*" ^ -1 + EndKeyword )
3455     + Keyword * EndKeyword
3456     + Builtin * EndKeyword
3457     + Identifier
3458     + Number
3459     + Word

```

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

```

3460     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:`<sup>8</sup>.

```

3461     LPEG2.c =
3462     Ct (
3463         ( space ^ 0 * P "\r" ) ^ -1
3464         * Lc [[ \@@_begin_line: ]]
3465         * LeadingSpace ^ 0
3466         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3467         * -1
3468         * Lc [[ \@@_end_line: ]]
3469     )

```

End of the Lua scope for the language C.

```

3470 end

```

### 3.7 The language SQL

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

```

3471 --sql SQL
3472 do
3473     local LuaKeyword
3474     function LuaKeyword ( name ) return
3475         Lc [[ {\PitonStyle{Keyword}}{ ]]
3476         * Q ( Cmt (
3477             C ( letter * alphanum ^ 0 ) ,
3478             function ( _ , _ , a ) return a : upper ( ) == name end
3479         )
3480         )
3481         * Lc "}"
3482     end

```

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

```

3483     local identifier =
3484         letter * ( alphanum + "-" ) ^ 0
3485         + P "'" * ( ( 1 - P "'" ) ^ 1 ) * "'"

```

---

<sup>8</sup>Remember 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:`

```

3486 local Operator =
3487 K ( 'Operator' , P "=" + "!=" + "<>" + ">=" + ">" + "<=" + "<" + S "*/" )

```

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

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

```

3488 local Set
3489 function Set ( list )
3490     local set = { }
3491     for _ , l in ipairs ( list ) do set[l] = true end
3492     return set
3493 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](https://sqlite.org/lang_keywords.html).

```

3494 local set_keywords = Set
3495 {
3496     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
3497     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
3498     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
3499     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
3500     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
3501     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
3502     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
3503     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
3504     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
3505     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
3506     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
3507     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
3508     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
3509     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
3510     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
3511     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
3512     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
3513     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
3514     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
3515     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
3516 }
3517 local set_builtins = Set
3518 {
3519     "AVG", "COUNT", "CHAR_LENGTH", "CONCAT", "CURDATE", "CURRENT_DATE",
3520     "DATE_FORMAT", "DAY", "LOWER", "LTRIM", "MAX", "MIN", "MONTH", "NOW",
3521     "RANK", "ROUND", "RTRIM", "SUBSTRING", "SUM", "UPPER", "YEAR"
3522 }

```

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.

```

3523 local Identifier =
3524 C ( identifier ) /
3525 (
3526     function ( s )
3527         if set_keywords [ s : upper ( ) ] then return

```

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

```

3528     { [[{\PitonStyle{Keyword}{}}] } ,
3529     { luatexbase.catcodetables.other , s } ,
3530     { "}" } }
3531 else

```

```

3532         if set_builtins [ s : upper ( ) ] then return
3533             { [[{\PitonStyle{Name.Builtin}{}}] } ,
3534             { luatexbase.catcodetables.other , s } ,
3535             { "}" } }
3536         else return
3537             { [[{\PitonStyle{Name.Field}{}}] } ,
3538             { luatexbase.catcodetables.other , s } ,
3539             { "}" } }
3540         end
3541     end
3542 end
3543 )

```

## The strings of SQL

```

3544 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.

```

3545 local braces = Compute_braces ( "" * ( 1 - P "" ) ^ 1 * "" )
3546 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
3547 DetectedCommands =
3548     Compute_DetectedCommands ( 'sql' , braces )
3549     + Compute_RawDetectedCommands ( 'sql' , braces )
3550 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.

```

3551 local Comment =
3552     WithStyle ( 'Comment.Internal' ,
3553         Q "--" -- syntax of SQL92
3554         * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
3555     * ( EOL + -1 )
3556
3557 local LongComment =
3558     WithStyle ( 'Comment.Internal' ,
3559         Q "/*"
3560         * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
3561         * Q "*/"
3562     ) -- $

```

## The main LPEG for the language SQL

```

3563 local EndKeyword
3564     = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
3565     EscapeMath + -1
3566 local TableField =
3567     K ( 'Name.Table' , identifier )
3568     * Q "."
3569     * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
3570
3571 local OneField =
3572     (
3573     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
3574     +

```

```

3575     K ( 'Name.Table' , identifier )
3576     * Q "."
3577     * K ( 'Name.Field' , identifier )
3578     +
3579     K ( 'Name.Field' , identifier )
3580 )
3581 * (
3582     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
3583 ) ^ -1
3584 * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
3585
3586 local OneTable =
3587     K ( 'Name.Table' , identifier )
3588     * (
3589         Space
3590         * LuaKeyword "AS"
3591         * Space
3592         * K ( 'Name.Table' , identifier )
3593     ) ^ -1
3594
3595 local WeCatchTableNames =
3596     LuaKeyword "FROM"
3597     * ( Space + EOL )
3598     * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
3599     + (
3600         LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
3601         + LuaKeyword "TABLE"
3602     )
3603     * ( Space + EOL ) * OneTable
3604
3604 local EndKeyword
3605     = Space + Punct + Delim + EOL + Beamer
3606     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

3607 local Main =
3608     space ^ 0 * EOL
3609     + Space
3610     + Tab
3611     + Escape + EscapeMath
3612     + CommentLaTeX
3613     + Beamer
3614     + DetectedCommands
3615     + Comment + LongComment
3616     + Delim
3617     + Operator
3618     + String
3619     + Punct
3620     + WeCatchTableNames
3621     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
3622     + Number
3623     + Word

```

Here, we must not put local, of course.

```

3624 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:`<sup>9</sup>.

```

3625 LPEG2.sql =

```

---

<sup>9</sup>Remember 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:`

```

3626 Ct (
3627     ( space ^ 0 * "\r" ) ^ -1
3628     * Lc [[ \@@_begin_line: ]]
3629     * LeadingSpace ^ 0
3630     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3631     * -1
3632     * Lc [[ \@@_end_line: ]]
3633 )

```

End of the Lua scope for the language SQL.

```

3634 end

```

### 3.8 The language “Minimal”

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

```

3635 --minimal Minimal
3636 do
3637     local Punct = Q ( S ",:;!\" )
3638
3639     local Comment =
3640         WithStyle ( 'Comment.Internal' ,
3641             Q "#"
3642             * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3643             )
3644             * ( EOL + -1 )
3645
3646     local String =
3647         WithStyle ( 'String.Short.Internal' ,
3648             Q "\"
3649             * ( SpaceInString
3650                 + Q ( ( P [[\]] + 1 - S " \" ) ^ 1 )
3651                 ) ^ 0
3652             * Q "\"
3653             )

```

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

```

3654     local braces = Compute_braces ( P "\" * ( P "\\\"" + 1 - P "\" ) ^ 1 * "\" )
3655
3656     if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
3657
3658     DetectedCommands =
3659         Compute_DetectedCommands ( 'minimal' , braces )
3660         + Compute_RawDetectedCommands ( 'minimal' , braces )
3661
3662     LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
3663
3664     local identifier = letter * alphanum ^ 0
3665
3666     local Identifier = K ( 'Identifier.Internal' , identifier )
3667
3668     local Delim = Q ( S "{[()]}" )
3669
3670     local Main =
3671         space ^ 0 * EOL
3672         + Space
3673         + Tab
3674         + Escape + EscapeMath
3675         + CommentLaTeX
3676         + Beamer
3677         + DetectedCommands

```

```

3678     + Comment
3679     + Delim
3680     + String
3681     + Punct
3682     + Identifier
3683     + Number
3684     + Word

```

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

```

3685     LPEG1.minimal = Main ^ 0
3686
3687     LPEG2.minimal =
3688     Ct (
3689         ( space ^ 0 * "\r" ) ^ -1
3690         * Lc [[ \@@_begin_line: ]]
3691         * LeadingSpace ^ 0
3692         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3693         * -1
3694         * Lc [[ \@@_end_line: ]]
3695     )

```

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

```

3696 end

```

### 3.9 The language “Verbatim”

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

```

3697 --verbatim Verbatim
3698 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”).

```

3699     local braces =
3700     P { "E" ,
3701         E = ( "{" * V "E" * "}" + ( 1 - S "{" ) ) ^ 0
3702     }
3703
3704     if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
3705
3706     DetectedCommands =
3707     Compute_DetectedCommands ( 'verbatim' , braces )
3708     + Compute_RawDetectedCommands ( 'verbatim' , braces )
3709
3710     LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )

```

Now, you will construct the LPEG Word.

```

3711     local lpeg_central = 1 - S "\\r"
3712     if piton.begin_escape then
3713         lpeg_central = lpeg_central - piton.begin_escape
3714     end
3715     if piton.begin_escape_math then
3716         lpeg_central = lpeg_central - piton.begin_escape_math
3717     end
3718     local Word = Q ( lpeg_central ^ 1 )
3719
3720     local Main =
3721     space ^ 0 * EOL
3722     + Space
3723     + Tab
3724     + Escape + EscapeMath
3725     + Beamer

```

```

3726     + DetectedCommands
3727     + Q [[\]]
3728     + Word

```

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

```

3729 LPEG1.verbatim = Main ^ 0
3730
3731 LPEG2.verbatim =
3732   Ct (
3733     ( space ^ 0 * "\r" ) ^ -1
3734     * Lc [[ \@@_begin_line: ]]
3735     * LeadingSpace ^ 0
3736     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3737     * -1
3738     * Lc [[ \@@_end_line: ]]
3739   )

```

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

```

3740 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).

```

3741 --EXPL expl
3742 do
3743   local Comment =
3744     WithStyle
3745     ( 'Comment.Internal' ,
3746       Q "%" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
3747     )
3748     * ( EOL + -1 )

```

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

```

3749 local analyze_cs
3750 function analyze_cs ( s )
3751   local i = s : find ( ":" )
3752   if i then

```

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

```

3753     local name = s : sub ( 2 , i - 1 )
3754     local parts = name : explode ( "_" )
3755     local module = parts[1]
3756     if module == "" then module = parts[3] end

```

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

```

3757     return
3758     { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{"} } ,
3759     { luatexbase.catcodetables.other , s } ,
3760     { "}" } }
3761   else
3762     local p = s : sub ( 1 , 3 )
3763     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`.

```

3764     local scope = s : sub(2,2)
3765     local parts = s : explode ( "_" )
3766     local module = parts[2]
3767     if module == "" then module = parts[3] end

```

```

3768     local type = parts[#parts]
3769     return
3770     { [[{\OptionalLocalPitonStyle{Scope.}] .. scope .. "}{" } ,
3771       { [[{\OptionalLocalPitonStyle{Module.}] .. module .. "}{" } ,
3772       { [[{\OptionalLocalPitonStyle{Type.}] .. type .. "}{" } ,
3773       { luatexbase.catcodetables.other , s } ,
3774       { "}}}}}" }
3775     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.

```

3776         return { luatexbase.catcodetables.other , s }
3777     end
3778 end
3779 end

```

Here, we don’t use braces as done with the other languages because we don’t have have to take into account the strings (there is no string in the language `expl`).

```

3780 local braces =
3781   P { "E" ,
3782     E = ( "{ " * V "E" * "}" + ( 1 - S "{ " ) ) ^ 0
3783   }
3784
3785 if piton.beamer then Beamer = Compute_Beamer ( 'expl' , braces ) end
3786
3787 DetectedCommands =
3788   Compute_DetectedCommands ( 'expl' , braces )
3789   + Compute_RawDetectedCommands ( 'expl' , braces )
3790
3791 LPEG_cleaner.expl = Compute_LPEG_cleaner ( 'expl' , braces )
3792
3793 local control_sequence = P "\\ " * ( R "Az" + "_" + ":" + "@" ) ^ 1
3794 local ControlSequence = C ( control_sequence ) / analyze_cs
3795
3796 local def_function
3797   = P [[\cs_]
3798     * ( P "set" + "new" )
3799     * ( P "_protected" ) ^ -1
3800     * P ":N" * ( P "p" ) ^ -1 * "n"
3801
3802 local DefFunction =
3803   C ( def_function ) / analyze_cs
3804   * Space
3805   * Lc ( [[ {\PitonStyle{Name.Function}{ }} ] ] )
3806   * ControlSequence -- Q ( ControlSequence ) ?
3807   * Lc "}"
3808
3809 local Word = Q ( ( 1 - S " \r" ) ^ 1 )
3810
3811 local Main =
3812   space ^ 0 * EOL
3813   + Space
3814   + Tab
3815   + Escape + EscapeMath
3816   + Beamer
3817   + Comment
3818   + DetectedCommands
3819   + DefFunction
3820   + ControlSequence
3821   + Word

```

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

```

3818 LPEG1.expl = Main ^ 0
3819
3820 LPEG2.expl =

```

```

3821 Ct (
3822   ( space ^ 0 * "\r" ) ^ -1
3823   * Lc [[ \@@_begin_line: ]]
3824   * LeadingSpace ^ 0
3825   * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3826   * -1
3827   * Lc [[ \@@_end_line: ]]
3828 )

```

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

```
3829 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.

```
3830 function piton.Parse ( language , code )
```

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

```

3831 piton.language = language
3832 local t = LPEG2[language] : match ( code )
3833 if not t then
3834   sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
3835   return -- to exit in force the function
3836 end
3837 local left_stack = {}
3838 local right_stack = {}
3839 for _ , one_item in ipairs ( t ) do
3840   if one_item == "EOL" then
3841     for i = #right_stack, 1, -1 do
3842       tex.sprint ( right_stack[i] )
3843     end

```

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

```

3844   sprintL3 ( [[ \@@_end_line: \@@_par: \@@_begin_line: ]] )
3845   tex.sprint ( table.concat ( left_stack ) )
3846 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}`.

```

3847   if one_item[1] == "Open" then
3848     tex.sprint ( one_item[2] )
3849     table.insert ( left_stack , one_item[2] )
3850     table.insert ( right_stack , one_item[3] )
3851   else
3852     if one_item[1] == "Close" then
3853       tex.sprint ( right_stack[#right_stack] )
3854       left_stack[#left_stack] = nil
3855       right_stack[#right_stack] = nil
3856     else
3857       tex.tprint ( one_item )
3858     end
3859   end

```

```

3860     end
3861   end
3862 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).

```

3863 local my_file_lines
3864 function my_file_lines ( filename )
3865   local f = io.open ( filename , 'rb' )
3866   local s = f : read ( '*a' )
3867   f : close ( )

```

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

```

3868   return ( s .. '\n' ) : gsub( '\r\n?' , '\n' ) : gmatch ( '(.)\n' )
3869 end

```

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

```

3870 function piton.ReadFile ( name , first_line , last_line )
3871   local s = ''
3872   local i = 0
3873   for line in my_file_lines ( name ) do
3874     i = i + 1
3875     if i >= first_line then
3876       s = s .. '\r' .. line
3877     end
3878     if i >= last_line then break end
3879   end

```

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

```

3880   if s : sub ( 1 , 4 ) == string.char ( 13 , 239 , 187 , 191 ) then
3881     s = s : sub ( 5 , -1 )
3882   end
3883   sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { }}] )
3884   tex.sprint ( luatexbase.catcodetables.other , s )
3885   sprintL3 ( "}" )
3886 end

```

```

3887 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
3888   local s
3889   s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
3890   piton.GobbleParse ( lang , n , splittable , s )
3891 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 `#`.

```

3892 function piton.ParseBis ( lang , code )
3893   return piton.Parse ( lang , code : gsub ( '##' , '#' ) )
3894 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.

```

3895 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:`.

```

3896   return piton.Parse
3897       (
3898         lang ,
3899         code : gsub ( [[\@@_breakable_space: ]] , ' ' )
3900       )
3901   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.

```

3902 local AutoGobbleLPEG =
3903   ( (
3904     P " " ^ 0 * "\r"
3905     +
3906     Ct ( C " " ^ 0 ) / table.getn
3907     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
3908   ) ^ 0
3909   * ( Ct ( C " " ^ 0 ) / table.getn
3910     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3911 ) / math.min

```

The following LPEG is similar but works with the tabulations.

```

3912 local TabsAutoGobbleLPEG =
3913   ( (
3914     P "\t" ^ 0 * "\r"
3915     +
3916     Ct ( C "\t" ^ 0 ) / table.getn
3917     * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
3918   ) ^ 0
3919   * ( Ct ( C "\t" ^ 0 ) / table.getn
3920     * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3921 ) / math.min
3922

```

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).

```

3923 local EnvGobbleLPEG =
3924   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
3925   * 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.

```

3926 function piton.Gobble ( n , code )
3927   if n == 0 then return
3928     code
3929   else
3930     if n == -1 then
3931       n = AutoGobbleLPEG : match ( code )

```

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

```

3932     if tonumber(n) then else n = 0 end
3933 else
3934     if n == -2 then
3935         n = EnvGobbleLPEG : match ( code )
3936     else
3937         if n == -3 then
3938             n = TabsAutoGobbleLPEG : match ( code )
3939             if tonumber(n) then else n = 0 end
3940         end
3941     end
3942 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...

```

3943     if n == 0 then return
3944         code
3945     else return

```

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

```

3946     ( Ct (
3947         ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3948         * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3949         ) ^ 0 )
3950     / table.concat
3951     ) : match ( code )
3952 end
3953 end
3954 end

```

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

```

3955 function piton.GobbleParse ( lang , n , splittable , code )
3956     piton.ComputeLinesStatus ( code , splittable )
3957     piton.last_code = piton.Gobble ( n , code )
3958     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`.

```

3959     piton.CountLines ( piton.last_code )
3960     piton.Parse ( lang , piton.last_code )
3961     piton.join_and_write ( )
3962 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`.

```

3963 function piton.join_and_write ( )
3964     if piton.join ~= '' then
3965         if not piton.join_files [ piton.join ] then
3966             piton.join_files [ piton.join ] = piton.get_last_code ( )
3967         else
3968             if piton.join_separation == '' then
3969                 piton.join_files [ piton.join ] =
3970                 piton.join_files [ piton.join ]
3971                 .. "\r\n"
3972                 .. piton.get_last_code ( )
3973             else
3974                 piton.join_files [ piton.join ] =
3975                 piton.join_files [ piton.join ]
3976                 .. "\r\n"
3977                 .. ( piton.join_separation : gsub ( '##' , '#' ) )
3978                 .. "\r\n"
3979                 .. piton.get_last_code ( )

```

```

3980     end
3981     end
3982 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.

```

3983 if piton.write ~= '' then

```

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

```

3984     local file_name = ''
3985     if piton.path_write == '' then
3986         file_name = piton.write
3987     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.

```

3988         local attr = lfs.attributes ( piton.path_write )
3989         if attr and attr.mode == "directory" then
3990             file_name = piton.path_write .. "/" .. piton.write
3991         else

```

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

```

3992             sprintL3 [[ \@@_error_or_warning:n { InexistentDirectory } ]]
3993         end
3994     end
3995     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`).

```

3996     if not piton.write_files [ file_name ] then
3997         piton.write_files [ file_name ] = piton.get_last_code ( )
3998     else
3999         piton.write_files [ file_name ] =
4000         piton.write_files [ file_name ] .. "\n" .. piton.get_last_code ( )
4001     end
4002 end
4003 end
4004 end

```

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

```

4005 function piton.GobbleParseNoPrint ( lang , n , code )
4006     piton.last_code = piton.Gobble ( n , code )
4007     piton.last_language = lang
4008     piton.join_and_write ( )
4009 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).

```

4010 function piton.GobbleSplitParse ( lang , n , splittable , code )
4011     local chunks
4012     chunks =
4013     (
4014         Ct (

```

```

4015         (
4016             P " " ^ 0 * "\r"
4017         +
4018             C ( ( ( 1 - P "\r" ) ^ 1 * ( P "\r" + -1 )
4019                 - ( P " " ^ 0 * ( P "\r" + -1 ) )
4020                 ) ^ 1
4021             )
4022         ) ^ 0
4023     )
4024     ) : match ( piton.Gobble ( n , code ) )
4025     sprintL3 [[ \begingroup ]]
4026     sprintL3
4027     (
4028         [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
4029         .. "language = " .. lang .. ","
4030         .. "splittable = " .. splittable .. "]"
4031     )
4032     for k , v in pairs ( chunks ) do
4033         if k > 1 then
4034             sprintL3 ( [[ \l_@@_split_separation_tl ]] )
4035         end
4036         tex.print
4037         (
4038             [[\begin{]] .. piton.env_used_by_split .. "}\r"
4039             .. v
4040             .. [[\end{]] .. piton.env_used_by_split .. "}\r"
4041         )
4042     end
4043     sprintL3 [[ \endgroup ]]
4044 end

4045 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
4046     local s
4047     s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
4048     piton.GobbleSplitParse ( lang , n , splittable , s )
4049 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.

```

4050 piton.string_between_chunks =
4051 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
4052 .. [[ \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.

```

4053 function piton.get_last_code ( )
4054     return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
4055         : gsub ( '\r\n?', '\n' )
4056 end

```

### 3.14 To count the number of lines

```

4057 local CountBeamerEnvironments
4058 function CountBeamerEnvironments ( code ) return
4059     (
4060         Ct (
4061             (
4062                 P "\\begin{" * beamerEnvironments * ( 1 - P "\r" ) ^ 0 * C "\r"

```

```

4063         +
4064         ( 1 - P "\r" ) ^ 0 * "\r"
4065     ) ^ 0
4066     * ( 1 - P "\r" ) ^ 0
4067     * -1
4068     ) / table.getn
4069 ) : match ( code )
4070 end

```

The following function counts the lines of code except the lines which contains only instructions for the environments of Beamer.

It is used in `GobbleParse` and at the beginning of `\@@_composition:` (in some rare circumstances). Be careful. We have tried a version with `string.gsub` without success.

```

4071 function piton.CountLines ( code )
4072     local count
4073     count =
4074         ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4075             *
4076             (
4077                 space ^ 0 * ( 1 - P "\r" - space ) * ( 1 - P "\r" ) ^ 0 * Cc "\r"
4078                 + space ^ 0
4079             ) ^ -1
4080             * -1
4081         ) / table.getn
4082     ) : match ( code )
4083     if piton.beamer then
4084         count = count - 2 * CountBeamerEnvironments ( code )
4085     end
4086     sprintL3 ( [[ \int_gset:Nn \g_@@_nb_lines_int { ]] .. count .. "]" )
4087 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).

```

4088 function piton.CountNonEmptyLines ( code )
4089     local count = 0

```

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

```

4090     count =
4091         ( Ct ( ( P " " ^ 0 * "\r"
4092             + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
4093             * ( 1 - P "\r" ) ^ 0
4094             * -1
4095         ) / table.getn
4096     ) : match ( code )
4097     count = count + 1
4098     if piton.beamer then
4099         count = count - 2 * CountBeamerEnvironments ( code )
4100     end
4101     sprintL3
4102     ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { ]] .. count .. "]" )
4103 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.

```

4104 function piton.ComputeRange ( s , t , file_name )
4105     local first_line = -1
4106     local count = 0
4107     local last_found = false
4108     for line in io.lines ( file_name ) do

```

```

4109   if first_line == -1 then
4110       if line : sub ( 1 , #s ) == s then
4111           first_line = count
4112       end
4113   else
4114       if line : sub ( 1 , #t ) == t then
4115           last_found = true
4116           break
4117       end
4118   end
4119   count = count + 1
4120 end
4121 if first_line == -1 then
4122     sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
4123 else
4124     if not last_found then
4125         sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
4126     end
4127 end
4128 sprintL3 (
4129     [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 }'
4130     .. [[ \global \l_@@_last_line_int = ]] .. count )
4131 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`.

```

4132 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).

```

4133   local lpeg_line_beamer
4134   if piton.beamer then
4135       lpeg_line_beamer =
4136           space ^ 0
4137           * P [[\begin{]] * beamerEnvironments * "]"
4138           * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
4139           +
4140           space ^ 0
4141           * P [[\end{]] * beamerEnvironments * "]"
4142   else
4143       lpeg_line_beamer = P ( false )
4144   end
4145   local lpeg_empty_lines =
4146       Ct (
4147           ( lpeg_line_beamer * "\r"
4148             +
4149             P " " ^ 0 * "\r" * Cc ( 0 )

```

```

4150         +
4151         ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4152     ) ^ 0
4153     *
4154     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4155 )
4156 * -1

4157 local lpeg_all_lines =
4158 Ct (
4159     ( lpeg_line_beamer * "\r"
4160     +
4161     ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
4162     ) ^ 0
4163     *
4164     ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
4165 )
4166 * -1

```

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

```

4167 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 `\1_@@_splittable_int` (provided by the *absolute value* of the argument `splittable`).

```

4168 local lines_status
4169 local s = splittable
4170 if splittable < 0 then s = - splittable end
4171 if splittable > 0 then
4172     lines_status = lpeg_all_lines : match ( code )
4173 else

```

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

```

4174     lines_status = lpeg_empty_lines : match ( code )
4175     for i , x in ipairs ( lines_status ) do
4176         if x == 0 then
4177             for j = 1 , s - 1 do
4178                 if i + j > #lines_status then break end
4179                 if lines_status[i+j] == 0 then break end
4180                 lines_status[i+j] = 2
4181             end
4182             for j = 1 , s - 1 do
4183                 if i - j == 1 then break end
4184                 if lines_status[i-j-1] == 0 then break end
4185                 lines_status[i-j-1] = 2
4186             end
4187         end
4188     end
4189 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.

```

4190     for j = 1 , s - 1 do
4191         if j > #lines_status then break end
4192         if lines_status[j] == 0 then break end
4193         lines_status[j] = 2
4194     end

```

Now, from the end of the code.

```

4195     for j = 1 , s - 1 do
4196         if #lines_status - j == 0 then break end
4197         if lines_status[#lines_status - j] == 0 then break end

```

```

4198     lines_status[#lines_status - j] = 2
4199 end

4200 piton.lines_status = lines_status
4201 end

4202 function piton.TranslateBeamerEnv ( code )
4203     local s
4204     s =
4205     (
4206         Ct (
4207             (
4208                 space ^ 0
4209                 * C (
4210                     ( P "\\begin{" + "\\end{" )
4211                     * beamerEnvironments * "]" * ( 1 - P "\r" ) ^ 0 * "\r"
4212                 )
4213                 + C ( ( 1 - P "\r" ) ^ 0 * "\r" )
4214             ) ^ 0
4215             *
4216             (
4217                 (
4218                     space ^ 0
4219                     * C (
4220                         ( P "\\begin{" + "\\end{" )
4221                         * beamerEnvironments * "]" * ( 1 - P "\r" ) ^ 0 * -1
4222                     )
4223                     + C ( ( 1 - P "\r" ) ^ 1 ) * -1
4224                 ) ^ -1
4225             )
4226         ) ^ -1 / table.concat
4227     ) : match ( code )
4228     sprintL3 ( [[ \tl_set:Nn \l_@@_listing_tl { ]] )
4229     tex.sprint ( luatexbase.catcodetables.other , s )
4230     sprintL3 ( "]" )
4231 end

```

### 3.16 To create new languages with the syntax of listings

```

4232 function piton.new_language ( lang , definition )
4233     lang = lang : lower ( )

4234     local alpha , digit = lpeg.alpha , lpeg.digit
4235     local extra_letters = { "@", "_", "$" } --

```

The command `add_to_letter` (triggered by the key `)` don't write right away in the LPEG pattern of the letters in an intermediate `extra_letters` because we may have to retrieve letters from that "list" if there appear in a key alsoother.

```

4236     function add_to_letter ( c )
4237         if c ~= " " then table.insert ( extra_letters , c ) end
4238     end

```

For the digits, it's straitforward.

```

4239     function add_to_digit ( c )
4240         if c ~= " " then digit = digit + c end
4241     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 }).

```

4242 local other = S "._@+*/<>!?.() []~^=#&\"'\\\$" --
4243 local extra_others = { }
4244 function add_to_other ( c )
4245     if c ~= " " then

```

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

```

4246     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 `</...>`.

```

4247     other = other + P ( c )
4248 end
4249 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`.

```

4250 local def_table
4251 if ( S " , " ^ 0 * -1 ) : match ( definition ) then
4252     def_table = {}
4253 else
4254     local strict_braces =
4255         P { "E" ,
4256             E = ( "{" * V "F" * "}" + ( 1 - S " , {"} " ) ) ^ 0 ,
4257             F = ( "{" * V "F" * "}" + ( 1 - S "{" } " ) ) ^ 0
4258         }
4259     local cut_definition =
4260         P { "E" ,
4261             E = Ct ( V "F" * ( " , " * V "F" ) ^ 0 ) ,
4262             F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
4263                 * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
4264         }
4265     def_table = cut_definition : match ( definition )
4266 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.).

```

4267 local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
4268 local tex_arg = tex_braced_arg + C ( 1 )
4269 local tex_option_arg = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
4270 local args_for_tag
4271     = tex_option_arg
4272     * space ^ 0
4273     * tex_arg
4274     * space ^ 0
4275     * tex_arg
4276 local args_for_morekeywords
4277     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4278     * space ^ 0
4279     * tex_option_arg
4280     * space ^ 0
4281     * tex_arg
4282     * space ^ 0
4283     * ( tex_braced_arg + Cc ( nil ) )
4284 local args_for_moredelims
4285     = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
4286     * args_for_morekeywords
4287 local args_for_morecomment
4288     = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
4289     * space ^ 0

```

```

4290     * tex_option_arg
4291     * space ^ 0
4292     * 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`.

```

4293     local sensitive = true
4294     local style_tag , left_tag , right_tag
4295     for _ , x in ipairs ( def_table ) do
4296         if x[1] == "sensitive" then
4297             if x[2] == nil or ( P "true" ) : match ( x[2] ) then
4298                 sensitive = true
4299             else
4300                 if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
4301             end
4302         end
4303         if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
4304         if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
4305         if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
4306         if x[1] == "tag" then
4307             style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
4308             style_tag = style_tag or [[\PitonStyle{Tag}]]
4309         end
4310     end

```

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

```

4311     local Number =
4312         K ( 'Number.Internal' ,
4313             ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
4314               + digit ^ 0 * "." * digit ^ 1
4315               + digit ^ 1 )
4316             * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
4317             + digit ^ 1
4318         )
4319     local string_extra_letters = ""
4320     for _ , x in ipairs ( extra_letters ) do
4321         if not ( extra_others[x] ) then
4322             string_extra_letters = string_extra_letters .. x
4323         end
4324     end
4325     local letter = alpha + S ( string_extra_letters )
4326                   + P "â" + "à" + "ç" + "é" + "ê" + "ë" + "ï" + "î"
4327                   + "ô" + "û" + "ü" + "Ë" + "Ê" + "É" + "È" + "Ë"
4328                   + "ï" + "î" + "Û" + "Û" + "Û"
4329     local alphanum = letter + digit
4330     local identifier = letter * alphanum ^ 0
4331     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.

```

4332     local split_clist =
4333         P { "E" ,
4334             E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
4335                 * ( P "{" ) ^ 1
4336                 * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
4337                 * ( P "}" ) ^ 1 * space ^ 0 ,
4338             F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
4339         }

```

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

```

4340     local keyword_to_lpeg
4341     function keyword_to_lpeg ( name ) return

```

```

4342   Q ( Cmt (
4343         C ( identifier ) ,
4344         function ( _ , _ , a ) return a : upper ( ) == name : upper ( )
4345         end
4346     )
4347 )
4348 end
4349 local Keyword = P ( false )
4350 local PrefixedKeyword = P ( false )

```

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

```

4351 for _ , x in ipairs ( def_table )
4352 do if x[1] == "morekeywords"
4353     or x[1] == "otherkeywords"
4354     or x[1] == "moreredirectives"
4355     or x[1] == "moretexcs"
4356 then
4357     local keywords = P ( false )
4358     local style = [[\PitonStyle{Keyword}]]
4359     if x[1] == "moreredirectives" then style = [[\PitonStyle{Directive}]] end
4360     style = tex_option_arg : match ( x[2] ) or style
4361     local n = tonumber ( style )
4362     if n then
4363         if n > 1 then style = [[\PitonStyle{Keyword}] .. style .. "]" end
4364     end
4365     for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
4366         if x[1] == "moretexcs" then
4367             keywords = Q ( [[\]] .. word ) + keywords
4368         else
4369             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*.

```

4370         then keywords = Q ( word ) + keywords
4371         else keywords = keyword_to_lpeg ( word ) + keywords
4372         end
4373     end
4374 end
4375 Keyword = Keyword +
4376     Lc ( "{" .. style .. "{" ) * keywords * Lc "}"
4377 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.

```

4378     if x[1] == "keywordsprefix" then
4379         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
4380         PrefixedKeyword = PrefixedKeyword
4381             + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
4382     end
4383 end

```

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

```

4384 local long_string = P ( false )
4385 local Long_string = P ( false )
4386 local LongString = P ( false )

```

```

4387 local central_pattern = P ( false )
4388 for _ , x in ipairs ( def_table ) do
4389   if x[1] == "morestring" then
4390     arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
4391     arg2 = arg2 or [[\PitonStyle{String.Long}]]
4392     if arg1 ~= "s" then
4393       arg4 = arg3
4394     end
4395     central_pattern = 1 - S ( " \r" .. arg4 )
4396     if arg1 : match "b" then
4397       central_pattern = P ( [[\]] .. arg3 ) + central_pattern
4398     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...

```

4399   if arg1 : match "d" or arg1 == "m" then
4400     central_pattern = P ( arg3 .. arg3 ) + central_pattern
4401   end
4402   if arg1 == "m"
4403   then prefix = B ( 1 - letter - ")" - "]" )
4404   else prefix = P ( true )
4405   end

```

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

```

4406   long_string = long_string +
4407     prefix
4408     * arg3
4409     * ( space + central_pattern ) ^ 0
4410     * arg4

```

Now a pattern *with captures*.

```

4411   local pattern =
4412     prefix
4413     * Q ( arg3 )
4414     * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
4415     * Q ( arg4 )

```

We will need `Long_string` in the nested comments.

```

4416   Long_string = Long_string + pattern
4417   LongString = LongString +
4418     Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4419     * pattern
4420     * Ct ( Cc "Close" )
4421   end
4422 end

```

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

```

4423 local braces = Compute_braces ( long_string )
4424 if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
4425
4426 DetectedCommands =
4427   Compute_DetectedCommands ( lang , braces )
4428   + Compute_RawDetectedCommands ( lang , braces )
4429
4430 LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

4431 local CommentDelim = P ( false )
4432
4433 for _ , x in ipairs ( def_table ) do
4434   if x[1] == "morecomment" then
4435     local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
4436     arg2 = arg2 or [[\PitonStyle{Comment}]]

```

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

```

4437     if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
4438     if arg1 : match "l" then
4439         local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
4440                     : match ( other_args )
4441         if arg3 == [[\#]] then arg3 = "#" end -- mandatory
4442         if arg3 == [[\%]] then arg3 = "%" end -- mandatory"
4443         CommentDelim = CommentDelim +
4444             Ct ( Cc "Open"
4445                 * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4446                 * Q ( arg3 )
4447                 * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
4448                 * Ct ( Cc "Close" )
4449                 * ( EOL + -1 )
4450     else
4451         local arg3 , arg4 =
4452             ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
4453         if arg1 : match "s" then
4454             CommentDelim = CommentDelim +
4455                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4456                 * Q ( arg3 )
4457                 * (
4458                     CommentMath
4459                     + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
4460                     + EOL
4461                     ) ^ 0
4462                 * Q ( arg4 )
4463                 * Ct ( Cc "Close" )
4464         end
4465         if arg1 : match "n" then
4466             CommentDelim = CommentDelim +
4467                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}" )
4468                 * P { "A" ,
4469                     A = Q ( arg3 )
4470                     * ( V "A"
4471                         + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
4472                             - S "\r$" ) ^ 1 ) -- $
4473                         + long_string
4474                         + "$" -- $
4475                         * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
4476                         * "$" -- $
4477                         + EOL
4478                         ) ^ 0
4479                     * Q ( arg4 )
4480                 }
4481                 * Ct ( Cc "Close" )
4482         end
4483     end
4484 end

```

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

```

4485     if x[1] == "moredelim" then
4486         local arg1 , arg2 , arg3 , arg4 , arg5
4487             = args_for_moredelims : match ( x[2] )
4488         local MyFun = Q
4489         if arg1 == "*" or arg1 == "**" then
4490             function MyFun ( x )
4491                 if x ~= '' then return
4492                     LPEG1[lang] : match ( x )
4493                 end
4494             end
4495         end
4496         local left_delim

```

```

4497     if arg2 : match "i" then
4498         left_delim = P ( arg4 )
4499     else
4500         left_delim = Q ( arg4 )
4501     end
4502     if arg2 : match "l" then
4503         CommentDelim = CommentDelim +
4504             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4505             * left_delim
4506             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
4507             * Ct ( Cc "Close" )
4508             * ( EOL + -1 )
4509     end
4510     if arg2 : match "s" then
4511         local right_delim
4512         if arg2 : match "i" then
4513             right_delim = P ( arg5 )
4514         else
4515             right_delim = Q ( arg5 )
4516         end
4517         CommentDelim = CommentDelim +
4518             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}" )
4519             * left_delim
4520             * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
4521             * right_delim
4522             * Ct ( Cc "Close" )
4523     end
4524     end
4525 end
4526
4527 local Delim = Q ( S "{[()]}")
4528 local Punct = Q ( S "=",:;!\\'\'" )
4529
4529 local Main =
4530     space ^ 0 * EOL
4531     + Space
4532     + Tab
4533     + Escape + EscapeMath
4534     + CommentLaTeX
4535     + Beamer
4536     + DetectedCommands
4537     + CommentDelim

```

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

```

4538     + LongString
4539     + Delim
4540     + PrefixedKeyword
4541     + Keyword * ( -1 + # ( 1 - alphanum ) )
4542     + Punct
4543     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4544     + Number
4545     + 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.

```

4546 LPEG1[lang] = Main ^ 0

```

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

```

4547 LPEG2[lang] =
4548     Ct (
4549         ( space ^ 0 * P "\r" ) ^ -1
4550         * Lc [[ \@@_begin_line: ]]
4551         * LeadingSpace ^ 0

```

```

4552     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
4553     * -1
4554     * Lc [[ \@@_end_line: ]]
4555 )

```

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

```

4556 if left_tag then
4557   local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" ) * Cc "}" )
4558     * Q ( left_tag * other ^ 0 ) -- $
4559     * ( ( ( 1 - P ( right_tag ) ) ^ 0 )
4560       / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
4561     * Q ( right_tag )
4562     * Ct ( Cc "Close" )
4563   MainWithoutTag
4564     = space ^ 1 * -1
4565     + space ^ 0 * EOL
4566     + Space
4567     + Tab
4568     + Escape + EscapeMath
4569     + CommentLaTeX
4570     + Beamer
4571     + DetectedCommands
4572     + CommentDelim
4573     + Delim
4574     + LongString
4575     + PrefixedKeyword
4576     + Keyword * ( -1 + # ( 1 - alphanum ) )
4577     + Punct
4578     + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
4579     + Number
4580     + Word
4581   LPEG0[lang] = MainWithoutTag ^ 0
4582   local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
4583     + Beamer + DetectedCommands + CommentDelim + Tag
4584   MainWithTag
4585     = space ^ 1 * -1
4586     + space ^ 0 * EOL
4587     + Space
4588     + LPEGaux
4589     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
4590   LPEG1[lang] = MainWithTag ^ 0
4591   LPEG2[lang] =
4592     Ct (
4593       ( space ^ 0 * P "\r" ) ^ -1
4594       * Lc [[ \@@_begin_line: ]]
4595       * Beamer
4596       * LeadingSpace ^ 0
4597       * LPEG1[lang]
4598       * -1
4599       * Lc [[ \@@_end_line: ]]
4600     )
4601 end
4602 end

```

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

```

4603 function piton.write_files_now ( )
4604   for file_name , file_content in pairs ( piton.write_files ) do
4605     local file = io.open ( file_name , "w" )
4606     if file then
4607       file : write ( file_content )

```

```

4608     file : close ( )
4609     else
4610         sprintL3
4611         ( [[ \@@_error_or_warning:nn { FileError } { ]] .. file_name .. "}" )
4612     end
4613 end
4614 end

```

### 3.18 Conversion from utf8 to utf16

Caution: the following function should be considered as public.

```

4615 function piton.utf16 ( str )
4616     local hex = { "FEFF" } -- BOM UTF-16BE
4617     for _, codepoint in utf8.codes(str) do
4618         table.insert(hex, string.format("%04X", codepoint))
4619     end
4620     return table.concat(hex)
4621 end
4622 </LUA>

```