The package piton*

F. Pantigny
fpantigny@wanadoo.fr

December 6, 2024

Abstract

The package piton provides tools to typeset computer listings, with syntactic highlighting, by using the Lua library LPEG. It requires LuaLaTeX.

Since the version 4.0, the syntax of the absolute and relative paths used in `\PitonInputFile` has been changed: cf. part 6.1, p. 11.

1 Presentation

The package piton uses the Lua library LPEG[1] for parsing informatic listings and typesets them with syntactic highlighting. Since it uses the Lua of LuaLaTeX, it works with `lualatex` only (and won't work with the other engines: `latex`, `pdflatex` and `xelatex`). It does not use external program and the compilation does not require `--shell-escape` (except when the key `write` is used). The compilation is very fast since all the parsing is done by the library LPEG, written in C.

Here is an example of code typeset by piton, with the environment `{Piton}`.

```
from math import pi

def arctan(x,n=10):
    """Compute the mathematical value of arctan(x)

    n is the number of terms in the sum
    """
    if x < 0:
        return -arctan(-x) # recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) = π/2 for x > 0)[2]
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

The main alternatives to the package piton are probably the packages listings and minted.

The name of this extension (piton) has been chosen arbitrarily by reference to the pitons used by the climbers in alpinism.

*This document corresponds to the version 4.2a of piton, at the date of 2024/12/06.

[1]LPEG is a pattern-matching library for Lua, written in C, based on *parsing expression grammars*: http://www.inf.puc-rio.br/~roberto/lpeg/

[2]This LaTeX escape has been done by beginning the comment by `#>`.

2 Installation

The package piton is contained in two files: `piton.sty` and `piton.lua` (the LaTeX file `piton.sty` loaded by `\usepackage` will load the Lua file `piton.lua`). Both files must be in a repertory where LaTeX will be able to find them, for instance in a `texmf` tree. However, the best is to install piton with a TeX distribution such as MiKTeX, TeX Live or MacTeX.

3 Use of the package

The package piton must be used with LuaLaTeX exclusively: if another LaTeX engine (`latex`, `pdflatex`, `xelatex`,...) is used, a fatal error will be raised.

3.1 Loading the package

The package piton should be loaded by: `\usepackage{piton}`.

If, at the end of the preamble, the package xcolor has not been loaded (by the final user or by another package), piton loads xcolor with the instruction `\usepackage{xcolor}` (that is to say without any option). The package piton doesn't load any other package. It does not any exterior program.

3.2 Choice of the computer language

The package piton supports two kinds of languages:

- the languages natively supported by piton, which are Python, OCaml, C (in fact C++), SQL and a language called `minimal`[3];
- the languages defined by the final user by using the built-in command `\NewPitonLanguage` described p. 9 (the parsers of those languages can't be as precise as those of the languages supported natively by piton).

By default, the language used is Python.

It's possible to change the current language with the command `\PitonOptions` and its key `language`: `\PitonOptions{language = OCaml}`.

In fact, for piton, the names of the informatic languages are always **case-insensitive**. In this example, we might have written `Ocaml` or `ocaml`.

For the developers, let's say that the name of the current language is stored (in lower case) in the L3 public variable `\l_piton_language_str`.

In what follows, we will speak of Python, but the features described also apply to the other languages.

3.3 The tools provided to the user

The package piton provides several tools to typeset informatic codes: the command `\piton`, the environment `{Piton}` and the command `\PitonInputFile`.

- The command `\piton` should be used to typeset small pieces of code inside a paragraph. For example:

  `\piton{def square(x): return x*x}`   `def square(x): return x*x`

  The syntax and particularities of the command `\piton` are detailed below.
- The environment `{Piton}` should be used to typeset multi-lines code. Since it takes its argument in a verbatim mode, it can't be used within the argument of a LaTeX command. For sake of customization, it's possible to define new environments similar to the environment `{Piton}` with the command `\NewPitonEnvironment`: cf. 4.3 p. 8.
- The command `\PitonInputFile` is used to insert and typeset an external file: cf. 6.1 p. 11.

[3]That language `minimal` may be used to format pseudo-codes: cf. p. 31

3.4 The syntax of the command \piton

In fact, the command `\piton` is provided with a double syntax. It may be used as a standard command of LaTeX taking its argument between curly braces (`\piton{...}`) but it may also be used with a syntax similar to the syntax of the command `\verb`, that is to say with the argument delimited by two identical characters (e.g.: `\piton|...|`).

- Syntax `\piton{...}`

  When its argument is given between curly braces, the command `\piton` does not take its argument in verbatim mode. In particular:

  - several consecutive spaces will be replaced by only one space (and the also the character of end on line),
    but the command `\␣` is provided to force the insertion of a space;
  - it's not possible to use `%` inside the argument,
    but the command `\%` is provided to insert a `%`;
  - the braces must be appear by pairs correctly nested
    but the commands `\{` and `\}` are also provided for individual braces;
  - the LaTeX commands[4] are fully expanded and not executed,
    so it's possible to use `\\` to insert a backslash.

  The other characters (including `#`, `^`, `_`, `&`, `$` and `@`) must be inserted without backslash.

  Examples :

  ```
  \piton{MyString = '\\n'}
  \piton{def even(n): return n\%2==0}
  \piton{c="#"    # an affectation }
  \piton{c="#" \ \ \ # an affectation }
  \piton{MyDict = {'a': 3, 'b': 4 }}
  ```

  ```
  MyString = '\n'
  def even(n): return n%2==0
  c="#" # an affectation
  c="#"    # an affectation
  MyDict = {'a': 3, 'b': 4 }
  ```

  It's possible to use the command `\piton` in the arguments of a LaTeX command.[5]

  However, since the argument is expanded (in the TeX sens), one should take care not using in its argument *fragile* commands (that is to say commands which are neither *protected* nor *fully expandable*).

- Syntax `\piton|...|`

  When the argument of the command `\piton` is provided between two identical characters (all the characters are allowed except `%`, `\`, `#`, `{`, `}` and the space), that argument is taken in a *verbatim mode*. Therefore, with that syntax, the command `\piton` can't be used within the argument of another command.

  Examples :

  ```
  \piton|MyString = '\n'|
  \piton!def even(n): return n%2==0!
  \piton+c="#"    # an affectation +
  \piton?MyDict = {'a': 3, 'b': 4}?
  ```

  ```
  MyString = '\n'
  def even(n): return n%2==0
  c="#"    # an affectation
  MyDict = {'a': 3, 'b': 4}
  ```

[4] That concerns the commands beginning with a backslash but also the active characters (with catcode equal to 13).
[5] For example, it's possible to use the command `\piton` in a footnote. Example : `s = 123`.

4 Customization

4.1 The keys of the command \PitonOptions

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.[6]
These keys may also be applied to an individual environment `{Piton}` (between square brackets).

- The key `language` specifies which computer language is considered (that key is case-insensitive). It's possible to use the name of the six built-in languages (`Python`, `OCaml`, `C`, `SQL`, `minimal` and `verbatim`) or the name of a language defined by the user with `\NewPitonLanguage` (cf. part 5, p. 9).

  The initial value is `Python`.

- **New 4.0**

  The key `font-command` contains instructions of font which will be inserted at the beginning of all the elements composed by piton (without surprise, these instructions are not used for the so-called "LaTeX comments").

  The initial value is `\ttfamily` and, thus, piton uses by default the current monospaced font.

- The key `gobble` takes in as value a positive integer n: the first n characters are discarded (before the process of highlighting of the code) for each line of the environment `{Piton}`. These characters are not necessarily spaces.

- When the key `auto-gobble` is in force, the extension piton computes the minimal value n of the number of consecutive spaces beginning each (non empty) line of the environment `{Piton}` and applies `gobble` with that value of n.

- When the key `env-gobble` is in force, piton analyzes the last line of the environment `{Piton}`, that is to say the line which contains `\end{Piton}` and determines whether that line contains only spaces followed by the `\end{Piton}`. If we are in that situation, piton computes the number n of spaces on that line and applies `gobble` with that value of n. The name of that key comes from *environment gobble*: the effect of gobble is set by the position of the commands `\begin{Piton}` and `\end{Piton}` which delimit the current environment.

- The key `write` takes in as argument a name of file (with its extension) and write the content[7] of the current environment in that file. At the first use of a file by piton, it is erased.

  This key requires a compilation with `lualatex -shell-escape`.

- The key `path-write` specifies a path where the files written by the key `write` will be written.

- The key `line-numbers` activates the line numbering in the environments `{Piton}` and in the listings resulting from the use of `\PitonInputFile`.

  In fact, the key `line-numbers` has several subkeys.

  - With the key `line-numbers/skip-empty-lines`, the empty lines (which contains only spaces) are considered as non existent for the line numbering (if the key `/absolute`, described below, is in force, the key `/skip-empty-lines` is no-op in `\PitonInputFile`). The initial value of that key is `true` (and not `false`).[8]
  - With the key `line-numbers/label-empty-lines`, the labels (that is to say the numbers) of the empty lines are displayed. If the key `/skip-empty-line` is in force, the clé `/label-empty-lines` is no-op. The initial value of that key is `true`.[9]

[6] We remind that a LaTeX environment is, in particular, a TeX group.
[7] In fact, it's not exactly the body of the environment but the value of `piton.get_last_code()` which is the body without the overwritten LaTeX formatting instructions (cf. the part 7, p. 23).
[8] For the language Python, the empty lines in the docstrings are taken into account (by design).
[9] When the key `split-on-empty-lines` is in force, the labels of the empty are never printed.

– With the key `line-numbers/absolute`, in the listings generated in `\PitonInputFile`, the numbers of the lines displayed are *absolute* (that is to say: they are the numbers of the lines in the file). That key may be useful when `\PitonInputFile` is used to insert only a part of the file (cf. part 6.1.2, p. 11). The key `/absolute` is no-op in the environments `{Piton}` and those created by `\NewPitonEnvironment`.

– The key `line-numbers/start` requires that the line numbering begins to the value of the key.

– With the key `line-numbers/resume`, the counter of lines is not set to zero at the beginning of each environment `{Piton}` or use of `\PitonInputFile` as it is otherwise. That allows a numbering of the lines across several environments.

– The key `line-numbers/sep` is the horizontal distance between the numbers of lines (inserted by `line-numbers`) and the beginning of the lines of code. The initial value is 0.7 em.

– The key `line-numbers/format` is a list of tokens which are inserted before the number of line in order to format it. It's possible to put, *at the end* of the list, a LaTeX command with one argument, such as, for example, `\fbox`.

The initial value is `\footnotesize\color{gray}`.

For convenience, a mechanism of factorisation of the prefix `line-numbers` is provided. That means that it is possible, for instance, to write:

```
\PitonOptions
  {
    line-numbers =
      {
        skip-empty-lines = false ,
        label-empty-lines = false ,
        sep = 1 em ,
        format = \footnotesize \color{blue}
      }
  }
```

- The key `left-margin` corresponds to a margin on the left. That key may be useful in conjunction with the key `line-numbers` if one does not want the numbers in an overlapping position on the left.

  It's possible to use the key `left-margin` with the value `auto`. With that value, if the key `line-numbers` is in force, a margin will be automatically inserted to fit the numbers of lines. See an example part 8.1 on page 24.

- The key `background-color` sets the background color of the environments `{Piton}` and the listings produced by `\PitonInputFile` (it's possible to fix the width of that background with the key `width` described below).

  The key `background-color` supports also as value a *list* of colors. In this case, the successive rows are colored by using the colors of the list in a cyclic way.

  Example : `\PitonOptions{background-color = {gray!15,white}}`

  The key `background-color` accepts a color defined «on the fly». For example, it's possible to write `background-color = [cmyk]{0.1,0.05,0,0}`.

- With the key `prompt-background-color`, piton adds a color background to the lines beginning with the prompt "`>>>`" (and its continuation "`...`") characteristic of the Python consoles with REPL (*read-eval-print loop*).

- The key `width` will fix the width of the listing. That width applies to the colored backgrounds specified by `background-color` and `prompt-background-color` but also for the automatic breaking of the lines (when required by `break-lines`: cf. 6.2.1, p. 13).

That key may take in as value a numeric value but also the special value `min`. With that value, the width will be computed from the maximal width of the lines of code. Caution: the special value `min` requires two compilations with LuaLaTeX[10].

For an example of use of `width=min`, see the section 8.2, p. 24.

- When the key `show-spaces-in-strings` is activated, the spaces in the strings of characters[11] are replaced by the character ␣ (U+2423 : OPEN BOX). Of course, that character U+2423 must be present in the monospaced font which is used.[12]

  Example : `my_string = 'Very␣good␣answer'`

  With the key `show-spaces`, all the spaces are replaced by U+2423 (and no line break can occur on those "visible spaces", even when the key `break-lines`[13] is in force). By the way, one should remark that all the trailing spaces (at the end of a line) are deleted by piton. The tabulations at the beginning of the lines are represented by arrows.

```
\begin{Piton}[language=C,line-numbers,auto-gobble,background-color = gray!15]
    void bubbleSort(int arr[], int n) {
        int temp;
        int swapped;
        for (int i = 0; i < n-1; i++) {
            swapped = 0;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = 1;
                }
            }
            if (!swapped) break;
        }
    }
\end{Piton}
```

```
void bubbleSort(int arr[], int n) {
    int temp;
    int swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}
```

The command `\PitonOptions` provides in fact several other keys which will be described further (see in particular the "Pages breaks and line breaks" p. 13).

[10]The maximal width is computed during the first compilation, written on the `aux` file and re-used during the second compilation. Several tools such as `latexmk` (used by Overleaf) do automatically a sufficient number of compilations.

[11]With the language Python that feature applies only to the short strings (delimited by `'` or `"`). In OCaml, that feature does not apply to the *quoted strings*.

[12]The initial value of `font-command` is and, thus, by default, piton merely uses the current monospaced font.

[13]cf. 6.2.1 p. 13

4.2 The styles

4.2.1 Notion of style

The package piton provides the command `\SetPitonStyle` to customize the different styles used to format the syntactic elements of the informatic listings. The customizations done by that command are limited to the current TeX group.[14]

The command `\SetPitonStyle` takes in as argument a comma-separated list of *key=value* pairs. The keys are names of styles and the value are LaTeX formatting instructions.

These LaTeX instructions must be formatting instructions such as `\color{...}`, `\bfseries`, `\slshape`, etc. (the commands of this kind are sometimes called *semi-global* commands). It's also possible to put, *at the end of the list of instructions*, a LaTeX command taking exactly one argument.

Here an example which changes the style used to highlight, in the definition of a Python function, the name of the function which is defined. That code uses the command `\highLight` of lua-ul (that package requires also the package luacolor).

```
\SetPitonStyle{ Name.Function = \bfseries \highLight[red!30] }
```

In that example, `\highLight[red!30]` must be considered as the name of a LaTeX command which takes in exactly one argument, since, usually, it is used with `\highLight[red!30]{...}`.

With that setting, we will have : `def cube(x) : return x * x * x`

The different styles, and their use by piton in the different languages which it supports (Python, OCaml, C, SQL, "`minimal`" and "`verbatim`"), are described in the part 9, starting at the page 27.

The command `\PitonStyle` takes in as argument the name of a style and allows to retrieve the value (as a list of LaTeX instructions) of that style.

For example, it's possible to write `{\PitonStyle{Keyword}{function}}` and we will have the word **function** formatted as a keyword.

The syntax `{\PitonStyle{`*`style`*`}{...}}` is mandatory in order to be able to deal both with the semi-global commands and the commands with arguments which may be present in the definition of the style *`style`*.

4.2.2 Global styles and local styles

A style may be defined globally with the command `\SetPitonStyle`. That means that it will apply to all the informatic languages that use that style.

For example, with the command

```
\SetPitonStyle{Comment = \color{gray}}
```

all the comments will be composed in gray in all the listings, whatever informatic language they use (Python, C, OCaml, etc. or a language defined by the command `\NewPitonLanguage`).

But it's also possible to define a style locally for a given informatic language by providing the name of that language as optional argument (between square brackets) to the command `\SetPitonStyle`.[15]

For example, with the command

```
\SetPitonStyle[SQL]{Keyword = \color[HTML]{006699} \bfseries \MakeUppercase}
```

the keywords in the SQL listings will be composed in capital letters, even if they appear in lower case in the LaTeX source (we recall that, in SQL, the keywords are case-insensitive).

As expected, if an informatic language uses a given style and if that style has no local definition for that language, the global version is used. That notion of "global style" has no link with the notion of global definition in TeX (the notion of *group* in TeX).[16]

[14] We remind that a LaTeX environment is, in particular, a TeX group.
[15] We recall, that, in the package piton, the names of the informatic languages are case-insensitive.
[16] As regards the TeX groups, the definitions done by `\SetPitonStyle` are always local.

The package piton itself (that is to say the file `piton.sty`) defines all the styles globally.

4.2.3 The style UserFunction

The extension piton provides a special style called `UserFunction`. That style applies to the names of the functions previously defined by the user (for example, in Python, these names are those following the keyword `def` in a previous Python listing). The initial value of that style `\PitonStyle{Identifier}` and, therefore, the names of the functions are formatted like the other identifiers (that is to say, by default, with no special formatting except the features provided in `font-command`). However, it's possible to change the value of that style, as any other style, with the command `\SetPitonStyle`.

In the following example, we tune the styles `Name.Function` and `UserFunction` so as to have clickable names of functions linked to the (informatic) definition of the function.

```
\NewDocumentCommand{\MyDefFunction}{m}
   {\hypertarget{piton:#1}{\color[HTML]{CC00FF}{#1}}}
\NewDocumentCommand{\MyUserFunction}{m}{\hyperlink{piton:#1}{#1}}

\SetPitonStyle{Name.Function = \MyDefFunction, UserFunction = \MyUserFunction}
```

```
def transpose(v,i,j):
    x = v[i]
    v[i] = v[j]
    v[j] = x

def passe(v):
    for in in range(0,len(v)-1):
        if v[i] > v[i+1]:
            transpose(v,i,i+1)
```

(Some PDF viewers display a frame around the clickable word `transpose` but other do not.)

Of course, the list of the names of Python functions previously défined is kept in the memory of LuaLaTeX (in a global way, that is to say independently of the TeX groups). The extension piton provides a command to clear that list : it's the command `\PitonClearUserFunctions`. When it is used without argument, that command is applied to all the informatic languages used by the user but it's also possible to use it with an optional argument (between square brackets) which is a list of informatic languages to which the command will be applied.[17]

4.3 Creation of new environments

Since the environment `{Piton}` has to catch its body in a special way (more or less as verbatim text), it's not possible to construct new environments directly over the environment `{Piton}` with the classical commands `\newenvironment` (of standard LaTeX) or `\NewDocumentEnvironment` (of LaTeX3).
That's why piton provides a command `\NewPitonEnvironment`. That command takes in three mandatory arguments.
That command has the same syntax as the classical environment `\NewDocumentEnvironment`.[18]

With the following instruction, a new environment `{Python}` will be constructed with the same behaviour as `{Piton}`:

```
\NewPitonEnvironment{Python}{O{}}{\PitonOptions{#1}}{}
```

If one wishes to format Python code in a box of tcolorbox, it's possible to define an environment `{Python}` with the following code (of course, the package tcolorbox must be loaded).

[17] We remind that, in piton, the name of the informatic languages are case-insensitive.

[18] However, the specifier of argument `b` (used to catch the body of the environment as a LaTeX argument) is not allowed.

```
\NewPitonEnvironment{Python}{}
  {\begin{tcolorbox}}
  {\end{tcolorbox}}
```

With this new environment `{Python}`, it's possible to write:

```
\begin{Python}
def square(x):
    """Compute the square of a number"""
    return x*x
\end{Python}
```

```
def square(x):
    """Compute the square of a number"""
    return x*x
```

5 Definition of new languages with the syntax of listings

The package listings is a famous LaTeX package to format informatic listings.
That package provides a command `\lstdefinelanguage` which allows the user to define new languages. That command is also used by listings itself to provide the definition of the predefined languages in listings (in fact, for this task, listings uses a command called `\lst@definelanguage` but that command has the same syntax as `\lstdefinelanguage`).

The package piton provides a command `\NewPitonLanguage` to define new languages (available in `\piton`, `{Piton}`, etc.) with a syntax which is almost the same as the syntax of `\lstdefinelanguage`. Let's precise that piton does *not* use that command to define the languages provided natively (Python, OCaml, C, SQL, `minimal` and `verbatim`), which allows more powerful parsers.

For example, in the file `lstlang1.sty`, which is one of the definition files of listings, we find the following instructions (in version 1.10a).

```
\lstdefinelanguage{Java}%
  {morekeywords={abstract,boolean,break,byte,case,catch,char,class,%
      const,continue,default,do,double,else,extends,false,final,%
      finally,float,for,goto,if,implements,import,instanceof,int,%
      interface,label,long,native,new,null,package,private,protected,%
      public,return,short,static,super,switch,synchronized,this,throw,%
      throws,transient,true,try,void,volatile,while},%
   sensitive,%
   morecomment=[l]//,%
   morecomment=[s]{/*}{*/},%
   morestring=[b]",%
   morestring=[b]',%
  }[keywords,comments,strings]
```

In order to define a language called `Java` for piton, one has only to write the following code **where the last argument of `\lst@definelanguage`, between square brackets, has been discarded** (in fact, the symbols `%` may be deleted without any problem).

```
\NewPitonLanguage{Java}%
  {morekeywords={abstract,boolean,break,byte,case,catch,char,class,%
      const,continue,default,do,double,else,extends,false,final,%
      finally,float,for,goto,if,implements,import,instanceof,int,%
      interface,label,long,native,new,null,package,private,protected,%
      public,return,short,static,super,switch,synchronized,this,throw,%
      throws,transient,true,try,void,volatile,while},%
   sensitive,%
```

```
  morecomment=[l]//,%
  morecomment=[s]{/*}{*/},%
  morestring=[b]",%
  morestring=[b]',%
 }
```

It's possible to use the language Java like any other language defined by piton.
Here is an example of code formatted in an environment `{Piton}` with the key `language=Java`.[19]

```
public class Cipher {  // Caesar cipher
    public static void main(String[] args) {
        String str = "The quick brown fox Jumped over the lazy Dog";
        System.out.println( Cipher.encode( str, 12 ));
        System.out.println( Cipher.decode( Cipher.encode( str, 12), 12 ));
    }

    public static String decode(String enc, int offset) {
        return encode(enc, 26-offset);
    }

    public static String encode(String enc, int offset) {
        offset = offset % 26 + 26;
        StringBuilder encoded = new StringBuilder();
        for (char i : enc.toCharArray()) {
            if (Character.isLetter(i)) {
                if (Character.isUpperCase(i)) {
                    encoded.append((char) ('A' + (i - 'A' + offset) % 26 ));
                } else {
                    encoded.append((char) ('a' + (i - 'a' + offset) % 26 ));
                }
            } else {
                encoded.append(i);
            }
        }
        return encoded.toString();
    }
}
```

The keys of the command `\lstdefinelanguage` of listings supported by `\NewPitonLanguage` are: `morekeywords`, `otherkeywords`, `sensitive`, `keywordsprefix`, `moretexcs`, `morestring` (with the letters `b`, `d`, `s` and `m`), `morecomment` (with the letters `i`, `l`, `s` and `n`), `moredelim` (with the letters `i`, `l`, `s`, `*` and `**`), `moredirectives`, `tag`, `alsodigit`, `alsoletter` and `alsoother`.
For the description of those keys, we redirect the reader to the documentation of the package listings (type `texdoc listings` in a terminal).

For example, here is a language called "LaTeX" to format LaTeX chunks of codes:

```
\NewPitonLanguage{LaTeX}{keywordsprefix = \ , alsoletter = _ }
```

Initially, the characters `@` and `_` are considered as letters because, in many informatic languages, they are allowed in the keywords and the names of the identifiers. With `alsoletter = @_`, we retrieve them from the category of the letters.

[19] We recall that, for piton, the names of the informatic languages are case-insensitive. Hence, it's possible to write, for instance, `language=java`.

6 Advanced features

6.1 Insertion of a file

6.1.1 The command \PitonInputFile

The command `\PitonInputFile` includes the content of the file specified in argument (or only a part of that file: see below). The extension piton also provides the commands `\PitonInputFileT`, `\PitonInputFileF` and `\PitonInputFileTF` with supplementary arguments corresponding to the letters `T` and `F`. Those arguments will be executed if the file to include has been found (letter `T`) or not found (letter `F`).

Modification 4.0

The syntax for the absolute and relative paths has been changed in order to be conform to the traditionnal usages. However, it's possible to use the key `old-PitonInputFile` at load-time (that is to say with the `\usepackage`) in order to have the old behaviour (though, that key will be deleted in a future version of piton!).

Now, the syntax is the following one:

- The paths beginning by `/` are absolute.

  Example : `\PitonInputFile{/Users/joe/Documents/program.py}`

- The paths which do not begin with `/` are relative to the current repertory.

  Example : `\PitonInputFile{my_listings/program.py}`

The key `path` of the command `\PitonOptions` specifies a *list* of paths where the files included by `\PitonInputFile` will be searched. That list is comma separated.
As previously, the absolute paths must begin with `/`.

6.1.2 Insertion of a part of a file

The command `\PitonInputFile` inserts (with formatting) the content of a file. In fact, it's possible to insert only *a part* of that file. Two mechanisms are provided in this aim.

- It's possible to specify the part that we want to insert by the numbers of the lines (in the original file).

- It's also possible to specify the part to insert with textual markers.

In both cases, if we want to number the lines with the numbers of the lines in the file, we have to use the key `line-numbers/absolute`.

With line numbers
The command `\PitonInputFile` supports the keys `first-line` and `last-line` in order to insert only the part of file between the corresponding lines. Not to be confused with the key `line-numbers/start` which fixes the first line number for the line numbering. In a sens, `line-numbers/start` deals with the output whereas `first-line` and `last-line` deal with the input.

With textual markers
In order to use that feature, we first have to specify the format of the markers (for the beginning and the end of the part to include) with the keys `marker-beginning` and `marker-end` (usually with the command `\PitonOptions`).

Let us take a practical example.

We assume that the file to include contains solutions to exercises of programmation on the following model.

```
#[Exercise 1] Iterative version
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
            w = u+v
            u = v
            v = w
        return v
#<Exercise 1>
```

The markers of the beginning and the end are the strings `#[Exercise 1]` and `#<Exercise 1>`. The string "`Exercise 1`" will be called the *label* of the exercise (or of the part of the file to be included). In order to specify such markers in `piton`, we will use the keys `marker/beginning` and `marker/end` with the following instruction (the character `#` of the comments of Python must be inserted with the protected form `\#`).

```
\PitonOptions{ marker/beginning = \#[#1] , marker/end = \#<#1> }
```

As one can see, `marker/beginning` is an expression corresponding to the mathematical function which transforms the label (here `Exercise 1`) into the the beginning marker (in the example `#[Exercise 1]`). The string `#1` corresponds to the occurrences of the argument of that function, which the classical syntax in TeX. Idem for `marker/end`.

Now, you only have to use the key `range` of `\PitonInputFile` to insert a marked content of the file.

```
\PitonInputFile[range = Exercise 1]{file_name}
```

```
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
            w = u+v
            u = v
            v = w
        return v
```

The key `marker/include-lines` requires the insertion of the lines containing the markers.

```
\PitonInputFile[marker/include-lines,range = Exercise 1]{file_name}
```

```
#[Exercise 1] Iterative version
def fibo(n):
    if n==0: return 0
    else:
        u=0
        v=1
        for i in range(n-1):
            w = u+v
            u = v
            v = w
        return v
#<Exercise 1>
```

In fact, there exist also the keys `begin-range` and `end-range` to insert several marked contents at the same time.
For example, in order to insert the solutions of the exercises 3 to 5, we will write (if the file has the correct structure!):

```
\PitonInputFile[begin-range = Exercise 3, end-range = Exercise 5]{file_name}
```

6.2 Page breaks and line breaks

6.2.1 Line breaks

By default, the elements produced by piton can't be broken by an end on line. However, there are keys to allow such breaks (the possible breaking points are the spaces, even the spaces which appear in the strings of the informatic languages).

- With the key `break-lines-in-piton`, the line breaks are allowed in the command `\piton{...}` (but not in the command `\piton|...|`, that is to say the command `\piton` in verbatim mode).
- With the key `break-lines-in-Piton`, the line breaks are allowed in the environment `{Piton}` (hence the capital letter `P` in the name) and in the listings produced by `\PitonInputFile`.
- The key `break-lines` is a conjunction of the two previous keys.

The package piton provides also several keys to control the appearance on the line breaks allowed by `break-lines-in-Piton`.

- With the key `indent-broken-lines`, the indentation of a broken line is respected at carriage return (on the condition that the used font is a monospaced font and this is the case by default since the initial value of `font-command` is `\ttfamily`).
- The key `end-of-broken-line` corresponds to the symbol placed at the end of a broken line. The initial value is: `\hspace*{0.5em}\textbackslash`.
- The key `continuation-symbol` corresponds to the symbol placed at each carriage return. The initial value is: `+\;` (the command `\;` inserts a small horizontal space).
- The key `continuation-symbol-on-indentation` corresponds to the symbol placed at each carriage return, on the position of the indentation (only when the key `indent-broken-line` is in force). The initial value is: `$\hookrightarrow\;$`.

The following code has been composed with the following tuning:

```
\PitonOptions{width=12cm,break-lines,indent-broken-lines,background-color=gray!15}
```

```
def dict_of_list(l):
    """Converts a list of subrs and descriptions of glyphs in \
+   ↪ a dictionary"""
    our_dict = {}
    for list_letter in l:
        if (list_letter[0][0:3] == 'dup'): # if it's a subr
            name = list_letter[0][4:-3]
            print("We treat the subr of number " + name)
        else:
            name = list_letter[0][1:-3] # if it's a glyph
            print("We treat the glyph of number " + name)
        our_dict[name] = [treat_Postscript_line(k) for k in \
+       ↪ list_letter[1:-1]]
    return dict
```

New 4.1

With the key `break-strings-anywhere`, the strings may be broken anywhere (and not only on the spaces).

New 4.2

With the key `break-numbers-anywhere`, the numbers may be broken anywhere.

6.2.2 Page breaks

By default, the listings produced by the environment `{Piton}` and the command `\PitonInputFile` are not breakable.
However, piton provides the keys `splittable-on-empty-lines` and `splittable` to allow such breaks.

- The key `splittable-on-empty-lines` allows breaks on the empty lines. The "empty lines" are in fact the lines which contains only spaces.

- Of course, the key `splittable-on-empty-lines` may not be sufficient and that's why piton provides the key `splittable`.

  When the key `splittable` is used with the numeric value n (which must be a positive integer) the listing, or each part of the listing delimited by empty lines (when `split-on-empty-lines` is in force) may be broken anywhere with the restriction that no break will occur within the n first lines of the listing or within the n last lines.[20]

  For example, a tuning with `splittable = 4` may be a good choice.

  When used without value, the key `splittable` is equivalent to `splittable = 1` and the listings may be broken anywhere (it's probably not recommandable).

  The initial value of the key `splittable` is equal to 100 (by default, the listings are not breakable at all).

Even with a background color (set by the key `background-color`), the pages breaks are allowed, as soon as the key `split-on-empty-lines` or the key `splittable` is in force.[21]

6.3 Splitting of a listing in sub-listings

The extension piton provides the key `split-on-empty-lines`, which should not be confused with the key `splittable-on-empty-lines` previously defined.

In order to understand the behaviour of the key `split-on-empty-lines`, one should imagine that he has to compose an informatic listing which contains several definitions of informatic functions. Usually, in the informatic languages, those definitions of functions are separated by empty lines.

The key `split-on-empty-lines` splits the listings on the empty lines. Several empty lines are deleted and replaced by the content of the parameter corresponding to the key `split-separation`.

- That parameter must contain elements allowed to be inserted in *vertical mode* of TeX. For example, it's possible to put the TeX primitive `\hrule`.

- The initial value of this parameter is `\vspace{\baselineskip}\vspace{-1.25pt}` which corresponds eventually to an empty line in the final PDF (this vertical space is deleted if it occurs on a page break). If the key `background-color` is in force, no background color is added to that empty line.

[20]Remark that we speak of the lines of the original informatic listing and such line may be composed on several lines in the final PDF when the key `break-lines-in-Piton` is in force.

[21]With the key `splittable`, the environments `{Piton}` are breakable, even within a (breakable) environment of tcolorbox. Remind that an environment of tcolorbox included in another environment of tcolorbox is *not* breakable, even when both environments use the key `breakable` of tcolorbox.

New 4.0

Each chunk of the informatic listing is composed in an environment whose name is given by the key `env-used-by-split`. The initial value of that parameter is, not surprisingly, `Piton` and, hence, the different chunks are composed in several environments `{Piton}`. If one decides to change the value of `env-used-by-split`, he should use the name of an environment created by `\NewPitonEnvironment` (cf. part 4.3, p. 8).

Each chunk of the informatic listing is formated in its own environment. Therefore, it has its own line numbering (if the key `line-numbers` is in force) and its own colored background (when the key `background-color` is in force), separated from the background color of the other chunks. When used, the key `splittable` applies in each chunk (independently of the other chunks). Of course, a page break may occur between the chunks of code, regardless of the value of `splittable`.

```
\begin{Piton}[split-on-empty-lines,background-color=gray!15,line-numbers]
def square(x):
    """Computes the square of x"""
    return x*x

def cube(x):
    """Calcule the cube of x"""
    return x*x*x
\end{Piton}
```

```
def square(x):
    """Computes the square of x"""
    return x*x
```

```
def cube(x):
    """Calcule the cube of x"""
    return x*x*x
```

Caution: Since each chunk is treated independently of the others, the commands specified by `detected-commands` and the commands and environments of Beamer automatically detected by piton must not cross the enmpty lines of the original listing.

6.4 Highlighting some identifiers

The command `\SetPitonIdentifier` allows to change the formatting of some identifiers.

That command takes in three arguments:

- The optional argument (within square brackets) specifies the informatic language. If this argument is not present, the tunings done by `\SetPitonIdentifier` will apply to all the informatic languages of piton.[22]
- The first mandatory argument is a comma-separated list of names of identifiers.
- The second mandatory argument is a list of LaTeX instructions of the same type as piton "styles" previously presented (cf. 4.2 p. 7).

Caution: Only the identifiers may be concerned by that key. The keywords and the built-in functions won't be affected, even if their name appear in the first argument of the command `\SetPitonIdentifier`.

[22]We recall, that, in the package piton, the names of the informatic languages are case-insensitive.

```
\SetPitonIdentifier{l1,l2}{\color{red}}
\begin{Piton}
def tri(l):
    """Segmentation sort"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
\end{Piton}
```

```
def tri(l):
    """Segmentation sort"""
    if len(l) <= 1:
        return l
    else:
        a = l[0]
        l1 = [ x for x in l[1:] if x < a  ]
        l2 = [ x for x in l[1:] if x >= a ]
        return tri(l1) + [a] + tri(l2)
```

By using the command `\SetPitonIdentifier`, it's possible to add other built-in functions (or other new keywords, etc.) that will be detected by piton.

```
\SetPitonIdentifier[Python]
  {cos, sin, tan, floor, ceil, trunc, pow, exp, ln, factorial}
  {\PitonStyle{Name.Builtin}}

\begin{Piton}
from math import *
cos(pi/2)
factorial(5)
ceil(-2.3)
floor(5.4)
\end{Piton}
```

```
from math import *
cos(pi/2)
factorial(5)
ceil(-2.3)
floor(5.4)
```

6.5 Mechanisms to escape to LaTeX

The package piton provides several mechanisms for escaping to LaTeX:

- It's possible to compose comments entirely in LaTeX.
- It's possible to have the elements between `$` in the comments composed in LateX mathematical mode.
- It's possible to ask piton to detect automatically some LaTeX commands, thanks to the key `detected-commands`.
- It's also possible to insert LaTeX code almost everywhere in a Python listing.

One should also remark that, when the extension piton is used with the class beamer, piton detects in `{Piton}` many commands and environments of Beamer: cf. 6.6 p. 20.

6.5.1 The "LaTeX comments"

In this document, we call "LaTeX comments" the comments which begins by `#>`. The code following those characters, until the end of the line, will be composed as standard LaTeX code. There is two tools to customize those comments.

- It's possible to change the syntactic mark (which, by default, is `#>`). For this purpose, there is a key `comment-latex` available only in the preamble of the document, allows to choice the characters which, preceded by `#`, will be the syntactic marker.

  For example, if the preamble contains the following instruction:

  ```
  \PitonOptions{comment-latex = LaTeX}
  ```

  the LaTeX comments will begin by `#LaTeX`.

  If the key `comment-latex` is used with the empty value, all the Python comments (which begins by `#`) will, in fact, be "LaTeX comments".

- It's possible to change the formatting of the LaTeX comment itself by changing the piton style `Comment.LaTeX`.

  For example, with `\SetPitonStyle{Comment.LaTeX = \normalfont\color{blue}}`, the LaTeX comments will be composed in blue.

  If you want to have a character `#` at the beginning of the LaTeX comment in the PDF, you can use set `Comment.LaTeX` as follows:

  ```
  \SetPitonStyle{Comment.LaTeX = \color{gray}\#\normalfont\space }
  ```

  For other examples of customization of the LaTeX comments, see the part 8.2 p. 24

If the user has required line numbers (with the key `line-numbers`), it's possible to refer to a number of line with the command `\label` used in a LaTeX comment.[23]

6.5.2 The key "math-comments"

It's possible to request that, in the standard Python comments (that is to say those beginning by `#` and not `#>`), the elements between `$` be composed in LaTeX mathematical mode (the other elements of the comment being composed verbatim).
That feature is activated by the key `math-comments`, *which is available only in the preamble of the document.*

Here is a example, where we have assumed that the preamble of the document contains the instruction `\PitonOptions{math-comment}`:

```
\begin{Piton}
def square(x):
    return x*x # compute $x^2$
\end{Piton}
```

```
def square(x):
    return x*x # compute x²
```

[23]That feature is implemented by using a redefinition of the standard command `\label` in the environments `{Piton}`. Therefore, incompatibilities may occur with extensions which redefine (globally) that command `\label` (for example: varioref, refcheck, showlabels, etc.)

6.5.3 The key "detected-commands"

The key `detected-commands` of `\PitonOptions` allows to specify a (comma-separated) list of names of LaTeX commands that will be detected directly by piton.

- The key `detected-commands` must be used in the preamble of the LaTeX document.
- The names of the LaTeX commands must appear without the leading backslash (eg. `detected-commands = { emph, textbf }`).
- These commands must be LaTeX commands with only one (mandatory) argument between braces (and these braces must appear explicitly in the informatic listing).

In the following example, which is a recursive programmation of the factorial function, we decide to highlight the recursive call. The command `\highLight` of lua-ul[24] directly does the job with the easy syntax `\highLight{...}`.

We assume that the preamble of the LaTeX document contains the following line:

```
\PitonOptions{detected-commands = highLight}
```

Then, it's possible to write directly:

```
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        \highLight{return n*fact(n-1)}
\end{Piton}
```

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

6.5.4 The mechanism "escape"

It's also possible to overwrite the informatic listings to insert LaTeX code almost everywhere (but between lexical units, of course). By default, piton does not fix any delimiters for that kind of escape. In order to use this mechanism, it's necessary to specify the delimiters which will delimit the escape (one for the beginning and one for the end) by using the keys `begin-escape` and `end-escape`, *available only in the preamble of the document.*

We consider once again the previous example of a recursive programmation of the factorial. We want to highlight in pink the instruction containing the recursive call. With the package lua-ul, we can use the syntax `\highLight[LightPink]{...}`. Because of the optional argument between square brackets, it's not possible to use the key `detected-commands` but it's possible to achieve our goal with the more general mechanism "escape".

We assume that the preamble of the document contains the following instruction:

```
\PitonOptions{begin-escape=!,end-escape=!}
```

Then, it's possible to write:

[24] The package lua-ul requires itself the package luacolor.

```
\begin{Piton}
def fact(n):
    if n==0:
        return 1
    else:
        !\highLight[LightPink]{!return n*fact(n-1)!}!
\end{Piton}
```

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Caution : The mechanism "escape" is not active in the strings nor in the Python comments (however, it's possible to have a whole Python comment composed in LaTeX by beginning it with `#>`; such comments are merely called "LaTeX comments" in this document).

6.5.5 The mechanism "escape-math"

The mechanism "escape-math" is very similar to the mechanism "escape" since the only difference is that the elements sent to LaTeX are composed in the math mode of LaTeX.
This mechanism is activated with the keys `begin-escape-math` and `end-escape-math` (*which are available only in the preamble of the document*).
Despite the technical similarity, the use of the the mechanism "escape-math" is in fact rather different from that of the mechanism "escape". Indeed, since the elements are composed in a mathematical mode of LaTeX, they are, in particular, composed within a TeX group and, therefore, they can't be used to change the formatting of other lexical units.
In the languages where the character `$` does not play a important role, it's possible to activate that mechanism "escape-math" with the character `$`:

```
\PitonOptions{begin-escape-math=$,end-escape-math=$}
```

Remark that the character `$` must *not* be protected by a backslash.

However, it's probably more prudent to use `\(` et `\)`, which are delimiters of the mathematical mode provided by LaTeX.

```
\PitonOptions{begin-escape-math=\(,end-escape-math=\)}
```

Here is an example of utilisation.

```
\begin{Piton}[line-numbers]
def arctan(x,n=10):
    if \(x < 0\) :
        return \(-\arctan(-x)\)
    elif \(x > 1\) :
        return \(\pi/2 - \arctan(1/x)\)
    else:
        s = \(0\)
        for \(k\) in range(\(n\)): s += \(\smash{\frac{(-1)^k}{2k+1} x^{2k+1}}\)
        return s
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0 :
        return −arctan(−x)
    elif x > 1 :
        return π/2 − arctan(1/x)
    else:
        s = 0
        for k in range(n): s += (−1)^k/(2k+1) x^(2k+1)
        return s
```

6.6 Behaviour in the class Beamer

First remark

Since the environment `{Piton}` catches its body with a verbatim mode, it's necessary to use the environments `{Piton}` within environments `{frame}` of Beamer protected by the key `fragile`, i.e. beginning with `\begin{frame}[fragile]`.[25]

When the package piton is used within the class beamer[26], the behaviour of piton is slightly modified, as described now.

6.6.1 {Piton} et \PitonInputFile are "overlay-aware"

When piton is used in the class beamer, the environment `{Piton}` and the command `\PitonInputFile` accept the optional argument `<...>` of Beamer for the overlays which are involved.
For example, it's possible to write:

```
\begin{Piton}<2-5>
...
\end{Piton}
```

and

```
\PitonInputFile<2-5>{my_file.py}
```

6.6.2 Commands of Beamer allowed in {Piton} and \PitonInputFile

When piton is used in the class beamer , the following commands of beamer (classified upon their number of arguments) are automatically detected in the environments `{Piton}` (and in the listings processed by `\PitonInputFile`):

- no mandatory argument : `\pause`[27]. ;
- one mandatory argument : `\action`, `\alert`, `\invisible`, `\only`, `\uncover` and `\visible` ;
  It's possible to add new commands to that list with the key `detected-beamer-commands` (the names of the commands must *not* be preceded by a backslash).
- two mandatory arguments : `\alt` ;
- three mandatory arguments : `\temporal`.

[25] Remind that for an environment `{frame}` of Beamer using the key `fragile`, the instruction `\end{frame}` must be alone on a single line (except for any leading whitespace).

[26] The extension piton detects the class beamer and the package beamerarticle if it is loaded previously but, if needed, it's also possible to activate that mechanism with the key `beamer` provided by piton at load-time: `\usepackage[beamer]{piton}`

[27] One should remark that it's also possible to use the command `\pause` in a "LaTeX comment", that is to say by writing `#> \pause`. By this way, if the Python code is copied, it's still executable by Python

These commands must be used preceded and following by a space. In the mandatory arguments of these commands, the braces must be balanced. However, the braces included in short strings[28] of Python are not considered.

Regarding the functions `\alt` and `\temporal` there should be no carriage returns in the mandatory arguments of these functions.

Here is a complete example of file:

```
\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def string_of_list(l):
    """Convert a list of numbers in string"""
    \only<2->{s = "{" + str(l[0])}
    \only<3->{for x in l[1:]: s = s + "," + str(x)}
    \only<4->{s = s + "}"}
    return s
\end{Piton}
\end{frame}
\end{document}
```

In the previous example, the braces in the Python strings `"{"` and `"}"` are correctly interpreted (without any escape character).

6.6.3 Environments of Beamer allowed in {Piton} and \PitonInputFile

When piton is used in the class beamer, the following environments of Beamer are directly detected in the environments `{Piton}` (and in the listings processed by `\PitonInputFile`): `{actionenv}`, `{alertenv}`, `{invisibleenv}`, `{onlyenv}`, `{uncoverenv}` and `{visibleenv}`.

It's possible to add new environments to that list with the key `detected-beamer-environments`.

However, there is a restriction: these environments must contain only *whole lines of Python code* in their body. The instructions `\begin{...}` and `\end{...}` must be alone on their lines.

Here is an example:

```
\documentclass{beamer}
\usepackage{piton}
\begin{document}
\begin{frame}[fragile]
\begin{Piton}
def square(x):
    """Compure the square of its argument"""
    \begin{uncoverenv}<2>
    return x*x
    \end{uncoverenv}
\end{Piton}
\end{frame}
\end{document}
```

Remark concerning the command \alert and the environment {alertenv} of Beamer

Beamer provides an easy way to change the color used by the environment `{alertenv}` (and by the command `\alert` which relies upon it) to highlight its argument. Here is an example:

[28]The short strings of Python are the strings delimited by characters `'` or the characters `"` and not `'''` nor `"""`. In Python, the short strings can't extend on several lines.

```
\setbeamercolor{alerted text}{fg=blue}
```

However, when used inside an environment `{Piton}`, such tuning will probably not be the best choice because piton will, by design, change (most of the time) the color the different elements of text. One may prefer an environment `{alertenv}` that will change the background color for the elements to be highlighted.

Here is a code that will do that job and add a yellow background. That code uses the command `\@highLight` of lua-ul (that extension requires also the package luacolor).

```
\setbeamercolor{alerted text}{bg=yellow!50}
\makeatletter
\AddToHook{env/Piton/begin}
  {\renewenvironment<>{alertenv}{\only#1{\@highLight[alerted text.bg]}}{}}
\makeatother
```

That code redefines locally the environment `{alertenv}` within the environments `{Piton}` (we recall that the command `\alert` relies upon that environment `{alertenv}`).

6.7 Footnotes in the environments of piton

If you want to put footnotes in an environment `{Piton}` or (or, more unlikely, in a listing produced by `\PitonInputFile`), you can use a pair `\footnotemark`–`\footnotetext`.

However, it's also possible to extract the footnotes with the help of the package footnote or the package footnotehyper.

If piton is loaded with the option `footnote` (with `\usepackage[footnote]{piton}` or with `\PassOptionsToPackage`), the package footnote is loaded (if it is not yet loaded) and it is used to extract the footnotes.

If piton is loaded with the option `footnotehyper`, the package footnotehyper is loaded (if it is not yet loaded) ant it is used to extract footnotes.

Caution: The packages footnote and footnotehyper are incompatible. The package footnotehyper is the successor of the package footnote and should be used preferently. The package footnote has some drawbacks, in particular: it must be loaded after the package xcolor and it is not perfectly compatible with hyperref.

Important remark : If you use Beamer, you should know taht Beamer has its own system to extract the footnotes. Therefore, piton must be loaded in that class without the option `footnote` nor the option `footnotehyper`.

By default, in an environment `{Piton}`, a command `\footnote` may appear only within a "LaTeX comment". But it's also possible to add the command `\footnote` to the list of the "*detected-commands*" (cf. part 6.5.3, p. 18).

In this document, the package piton has been loaded with the option `footnotehyper` dans we added the command `\footnote` to the list of the "*detected-commands*" with the following instruction in the preamble of the LaTeX document.

```
\PitonOptions{detected-commands = footnote}
```

```
\PitonOptions{background-color=gray!15}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)\footnote{First recursive call.}]
    elif x > 1:
        return pi/2 - arctan(1/x)\footnote{Second recursive call.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)[29]
    elif x > 1:
        return pi/2 - arctan(1/x)[30]
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

If an environment `{Piton}` is used in an environment `{minipage}` of LaTeX, the notes are composed, of course, at the foot of the environment `{minipage}`. Recall that such `{minipage}` can't be broken by a page break.

```
\PitonOptions{background-color=gray!15}
\emphase\begin{minipage}{\linewidth}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)\footnote{First recursive call.}
    elif x > 1:
        return pi/2 - arctan(1/x)\footnote{Second recursive call.}
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
\end{minipage}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)[a]
    elif x > 1:
        return pi/2 - arctan(1/x)[b]
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

[a] First recursive call.
[b] Second recursive call.

6.8 Tabulations

Even though it's probably recommended to indent the informatics listings with spaces and not tabulations[31], piton accepts the characters of tabulation (that is to say the characters U+0009) at the beginning of the lines. Each character U+0009 is replaced by n spaces. The initial value of n is 4 but it's possible to change it with the key `tab-size` of `\PitonOptions`.

There exists also a key `tabs-auto-gobble` which computes the minimal value n of the number of consecutive characters U+0009 beginning each (non empty) line of the environment `{Piton}` and applies `gobble` with that value of n (before replacement of the tabulations by spaces, of course). Hence, that key is similar to the key `auto-gobble` but acts on U+0009 instead of U+0020 (spaces). The key `env-gobble` is not compatible with the tabulations.

7 API for the developpers

The L3 variable `\l_piton_language_str` contains the name of the current language of piton (in lower case).

[29] First recursive call.
[30] Second recursive call.
[31] For the language Python, see the note PEP 8

The extension piton provides a Lua function `piton.get_last_code` without argument which returns the code in the latest environment of piton.

- The carriage returns (which are present in the initial environment) appears as characters `\r` (i.e. U+000D).
- The code returned by `piton.get_last_code()` takes into account the potential application of a key `gobble`, `auto-gobble` or `env-gobble` (cf. p. 4).
- The extra formatting elements added in the code are deleted in the code returned by `piton.get_last_code()`. That concerns the LaTeX commands declared by the key `detected-commands` (cf. part 6.5.3) and the elements inserted by the mechanism "`escape`" (cf. part 6.5.4).
- `piton.get_last_code` is a Lua function and not a Lua string: the treatments outlined above are executed when the function is called. Therefore, it might be judicious to store the value returned by `piton.get_last_code()` in a variable of Lua if it will be used several times.

For an example of use, see the part concerning `pyluatex`, part 8.4, p. 26.

8 Examples

8.1 Line numbering

We remind that it's possible to have an automatic numbering of the lines in the informatic listings by using the key `line-numbers` (used without value).
By default, the numbers of the lines are composed by piton in an overlapping position on the left (by using internally the command `\llap` of LaTeX).
In order to avoid that overlapping, it's possible to use the option `left-margin=auto` which will insert automatically a margin adapted to the numbers of lines that will be written (that margin is larger when the numbers are greater than 10).

```
\PitonOptions{background-color=gray!15, left-margin = auto, line-numbers}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)        #> (recursive call)
    elif x > 1:
        return pi/2 - arctan(1/x) #> (other recursive call)
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)        (recursive call)
    elif x > 1:
        return pi/2 - arctan(1/x) (other recursive call)
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

8.2 Formatting of the LaTeX comments

It's possible to modify the style `Comment.LaTeX` (with `\SetPitonStyle`) in order to display the LaTeX comments (which begin with `#>`) aligned on the right margin.

```
\PitonOptions{background-color=gray!15}
\SetPitonStyle{Comment.LaTeX = \hfill \normalfont\color{gray}}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
```

```
        return -arctan(-x)        #> recursive call
    elif x > 1:
        return pi/2 - arctan(1/x) #> other recursive call
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)                                    recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)                     another recursive call
    else:
        return sum( (-1)**k/(2*k+1)*x**(2*k+1) for k in range(n) )
```

It's also possible to display these LaTeX comments in a kind of second column by limiting the width of the Python code with the key `width`. In the following example, we use the key `width` with the special value `min`. Several compilations are required.

```
\PitonOptions{background-color=gray!15, width=min}
\NewDocumentCommand{\MyLaTeXCommand}{m}{\hfill \normalfont\itshape\rlap{\quad #1}}
\SetPitonStyle{Comment.LaTeX = \MyLaTeXCommand}
\begin{Piton}
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x) #> recursive call
    elif x > 1:
        return pi/2 - arctan(1/x) #> another recursive call
    else:
        s = 0
        for k in range(n):
             s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
\end{Piton}
```

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)                  recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)           another recursive call
    else:
        s = 0
        for k in range(n):
             s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

8.3 An example of tuning of the styles

The graphical styles have been presented in the section 4.2, p. 7.

We present now an example of tuning of these styles adapted to the documents in black and white. That tuning uses the command `\highLight` of lua-ul (that package requires itself the package luacolor).

```
\SetPitonStyle
  {
    Number = ,
    String = \itshape ,
    String.Doc = \color{gray} \slshape ,
    Operator = ,
```

```
    Operator.Word = \bfseries ,
    Name.Builtin = ,
    Name.Function = \bfseries \highLight[gray!20] ,
    Comment = \color{gray} ,
    Comment.LaTeX = \normalfont \color{gray},
    Keyword = \bfseries ,
    Name.Namespace = ,
    Name.Class = ,
    Name.Type = ,
    InitialValues = \color{gray}
  }
```

In that tuning, many values given to the keys are empty: that means that the corresponding style won't insert any formatting instruction (the element will be composed in the standard color, usually in black, etc.). Nevertheless, those entries are mandatory because the initial value of those keys in piton is *not* empty.

```
from math import pi

def arctan(x,n=10):
    """Compute the mathematical value of arctan(x)

    n is the number of terms in the sum
    """
    if x < 0:
        return -arctan(-x) # recursive call
    elif x > 1:
        return pi/2 - arctan(1/x)
        (we have used that arctan(x) + arctan(1/x) = π/2 for x > 0)
    else:
        s = 0
        for k in range(n):
            s += (-1)**k/(2*k+1)*x**(2*k+1)
        return s
```

8.4 Use with pyluatex

The package pyluatex is an extension which allows the execution of some Python code from `lualatex` (provided that Python is installed on the machine and that the compilation is done with `lualatex` and `--shell-escape`).
Here is, for example, an environment `{PitonExecute}` which formats a Python listing (with piton) but also displays the output of the execution of the code with Python.

```
\NewPitonEnvironment{PitonExecute}{!O{}}
  {\PitonOptions{#1}}
  {\begin{center}
   \directlua{pyluatex.execute(piton.get_last_code(), false, true, false, true)}%
   \end{center}
   \ignorespacesafterend}
```

We have used the Lua function `piton.get_last_code` provided in the API of piton : cf. part 7, p. 23.

This environment `{PitonExecute}` takes in as optional argument (between square brackets) the options of the command `\PitonOptions`.

9 The styles for the different computer languages

9.1 The language Python

In piton, the default language is Python. If necessary, it's possible to come back to the language Python with `\PitonOptions{language=Python}`.

The initial settings done by piton in `piton.sty` are inspired by the style manni de Pygments, as applied by Pygments to the language Python.[32]

Style	Use
`Number`	the numbers
`String.Short`	the short strings (entre `'` ou `"`)
`String.Long`	the long strings (entre `'''` ou `"""`) excepted the doc-strings (governed by `String.Doc`)
`String`	that key fixes both `String.Short` et `String.Long`
`String.Doc`	the doc-strings (only with `"""` following PEP 257)
`String.Interpol`	the syntactic elements of the fields of the f-strings (that is to say the characters `{` et `}`); that style inherits for the styles `String.Short` and `String.Long` (according the kind of string where the interpolation appears)
`Interpol.Inside`	the content of the interpolations in the f-strings (that is to say the elements between `{` and `}`); if the final user has not set that key, those elements will be formatted by piton as done for any Python code.
`Operator`	the following operators: `!= == << >> - ~ + / * % = < > & . | @`
`Operator.Word`	the following operators: `in`, `is`, `and`, `or` et `not`
`Name.Builtin`	almost all the functions predefined by Python
`Name.Decorator`	the decorators (instructions beginning by `@`)
`Name.Namespace`	the name of the modules
`Name.Class`	the name of the Python classes defined by the user *at their point of definition* (with the keyword `class`)
`Name.Function`	the name of the Python functions defined by the user *at their point of definition* (with the keyword `def`)
`UserFunction`	the name of the Python functions previously defined by the user (the initial value of that parameter is `\PitonStyle{Identifier}` and, therefore, the names of that functions are formatted like the identifiers).
`Exception`	les exceptions prédéfinies (ex.: `SyntaxError`)
`InitialValues`	the initial values (and the preceding symbol `=`) of the optional arguments in the definitions of functions; if the final user has not set that key, those elements will be formatted by piton as done for any Python code.
`Comment`	the comments beginning with `#`
`Comment.LaTeX`	the comments beginning with `#>`, which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
`Keyword.Constant`	`True`, `False` et `None`
`Keyword`	the following keywords: `assert, break, case, continue, del, elif, else, except, exec, finally, for, from, global, if, import, in, lambda, non local, pass, raise, return, try, while, with, yield et yield from.`
`Identifier`	the identifiers.

[32] See: https://pygments.org/styles/. Remark that, by default, Pygments provides for its style manni a colored background whose color is the HTML color `#F0F3F3`. It's possible to have the same color in `{Piton}` with the instruction `\PitonOptions{background-color = [HTML]{F0F3F3}}`.

9.2 The language OCaml

It's possible to switch to the language `OCaml` with the key `language`: `language = OCaml`.

Style	Use
`Number`	the numbers
`String.Short`	the characters (between `'`)
`String.Long`	the strings, between `"` but also the *quoted-strings*
`String`	that key fixes both `String.Short` and `String.Long`
`Operator`	les opérateurs, en particulier `+`, `-`, `/`, `*`, `@`, `!=`, `==`, `&&`
`Operator.Word`	les opérateurs suivants : `asr`, `land`, `lor`, `lsl`, `lxor`, `mod` et `or`
`Name.Builtin`	les fonctions `not`, `incr`, `decr`, `fst` et `snd`
`Name.Type`	the name of a type of OCaml
`Name.Field`	the name of a field of a module
`Name.Constructor`	the name of the constructors of types (which begins by a capital)
`Name.Module`	the name of the modules
`Name.Function`	the name of the Python functions defined by the user *at their point of definition* (with the keyword `let`)
`UserFunction`	the name of the Python functions previously defined by the user (the initial value of that parameter is `\PitonStyle{Identifier}` and, therefore, the names of that functions are formatted like the identifiers).
`Exception`	the predefined exceptions (eg : `End_of_File`)
`TypeParameter`	the parameters of the types
`Comment`	the comments, between `(*` et `*)`; these comments may be nested
`Keyword.Constant`	`true` et `false`
`Keyword`	the following keywords: `assert`, `as`, `done`, `downto`, `do`, `else`, `exception`, `for`, `function` , `fun`, `if`, `lazy`, `match`, `mutable`, `new`, `of`, `private`, `raise`, `then`, `to`, `try` , `virtual`, `when`, `while` and `with`
`Keyword.Governing`	the following keywords: `and`, `begin`, `class`, `constraint`, `end`, `external`, `functor`, `include`, `inherit`, `initializer`, `in`, `let`, `method`, `module`, `object`, `open`, `rec`, `sig`, `struct`, `type` and `val`.
`Identifier`	the identifiers.

9.3 The language C (and C++)

It's possible to switch to the language `C` with the key `language`: `language = C`.

Style	Use
`Number`	the numbers
`String.Long`	the strings (between `"`)
`String.Interpol`	the elements `%d`, `%i`, `%f`, `%c`, etc. in the strings; that style inherits from the style `String.Long`
`Operator`	the following operators : `!= == << >> - ~ + / * % = < > & . | @`
`Name.Type`	the following predefined types: `bool`, `char`, `char16_t`, `char32_t`, `double`, `float`, `int`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `long`, `short`, `signed`, `unsigned`, `void` et `wchar_t`
`Name.Builtin`	the following predefined functions: `printf`, `scanf`, `malloc`, `sizeof` and `alignof`
`Name.Class`	le nom des classes au moment de leur définition, c'est-à-dire après le mot-clé `class`
`Name.Function`	the name of the Python functions defined by the user *at their point of definition* (with the keyword `let`)
`UserFunction`	the name of the Python functions previously defined by the user (the initial value of that parameter is `\PitonStyle{Identifier}` and, therefore, the names of that functions are formatted like the identifiers).
`Preproc`	the instructions of the preprocessor (beginning par `#`)
`Comment`	the comments (beginning by `//` or between `/*` and `*/`)
`Comment.LaTeX`	the comments beginning by `//>` which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
`Keyword.Constant`	`default`, `false`, `NULL`, `nullptr` and `true`
`Keyword`	the following keywords: `alignas`, `asm`, `auto`, `break`, `case`, `catch`, `class`, `constexpr`, `const`, `continue`, `decltype`, `do`, `else`, `enum`, `extern`, `for`, `goto`, `if`, `nexcept`, `private`, `public`, `register`, `restricted`, `try`, `return`, `static`, `static_assert`, `struct`, `switch`, `thread_local`, `throw`, `typedef`, `union`, `using`, `virtual`, `volatile` and `while`
`Identifier`	the identifiers.

9.4 The language SQL

It's possible to switch to the language SQL with the key language: language = SQL.

Style	Use
Number	the numbers
String.Long	the strings (between ' and not " because the elements between " are names of fields and formatted with Name.Field)
Operator	the following operators : = != <> >= > < <= * + /
Name.Table	the names of the tables
Name.Field	the names of the fields of the tables
Name.Builtin	the following built-in functions (their names are *not* case-sensitive): avg, count, char_length, concat, curdate, current_date, date_format, day, lower, ltrim, max, min, month, now, rank, round, rtrim, substring, sum, upper and year.
Comment	the comments (beginning by -- or between /* and */)
Comment.LaTeX	the comments beginning by --> which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
Keyword	the following keywords (their names are *not* case-sensitive): abort, action, add, after, all, alter, always, analyze, and, as, asc, attach, autoincrement, before, begin, between, by, cascade, case, cast, check, collate, column, commit, conflict, constraint, create, cross, current, current_date, current_time, current_timestamp, database, default, deferrable, deferred, delete, desc, detach, distinct, do, drop, each, else, end, escape, except, exclude, exclusive, exists, explain, fail, filter, first, following, for, foreign, from, full, generated, glob, group, groups, having, if, ignore, immediate, in, index, indexed, initially, inner, insert, instead, intersect, into, is, isnull, join, key, last, left, like, limit, match, materialized, natural, no, not, nothing, notnull, null, nulls, of, offset, on, or, order, others, outer, over, partition, plan, pragma, preceding, primary, query, raise, range, recursive, references, regexp, reindex, release, rename, replace, restrict, returning, right, rollback, row, rows, savepoint, select, set, table, temp, temporary, then, ties, to, transaction, trigger, unbounded, union, unique, update, using, vacuum, values, view, virtual, when, where, window, with, without

It's possible to automatically capitalize the keywords by modifying locally for the language SQL the style Keywords.

```
\SetPitonStyle[SQL]{Keywords = \bfseries \MakeUppercase}
```

9.5 The languages defined by \NewPitonLanguage

The command `\NewPitonLanguage`, which defines new informatic languages with the syntax of the extension listings, has been described p. 9.
All the languages defined by the command `\NewPitonLanguage` use the same styles.

Style	Use
`Number`	the numbers
`String.Long`	the strings defined in `\NewPitonLanguage` by the key `morestring`
`Comment`	the comments defined in `\NewPitonLanguage` by the key `morecomment`
`Comment.LaTeX`	the comments which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
`Keyword`	the keywords defined in `\NewPitonLanguage` by the keys `morekeywords` and `moretexcs` (and also the key `sensitive` which specifies whether the keywords are case-sensitive or not)
`Directive`	the directives defined in `\NewPitonLanguage` by the key `moredirectives`
`Tag`	the "tags" defined by the key `tag` (the lexical units detected within the tag will also be formatted with their own style)
`Identifier`	the identifiers.

9.6 The language "minimal"

It's possible to switch to the language "`minimal`" with the key `language`: `language = minimal`.

Style	Usage
`Number`	the numbers
`String`	the strings (between `"`)
`Comment`	the comments (which begin with `#`)
`Comment.LaTeX`	the comments beginning with `#>`, which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
`Identifier`	the identifiers.

That language is provided for the final user who might wish to add keywords in that language (with the command `\SetPitonIdentifier`: cf. 6.4, p. 15) in order to create, for example, a language for pseudo-code.

9.7 The language "verbatim"

New 4.1
It's possible to switch to the language "`verbatim`" with the key `language`: `language = verbatim`.

Style	Usage
`None...`	

The language `verbatim` doesn't provide any style and, thus, does not do any syntactic formating. However, it's possible to use the mechanism `detected-commands` (cf. part 6.5.3, p. 18) and the detection of the commands and environments of Beamer.

10 Implementation

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

10.1 Introduction

The main job of the package `piton` is to take in as input a Python 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 `python`. That LPEG, when matched against the string of a Python 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.[33]

Consider, for example, the following Python code:

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

The capture returned by the `lpeg` `python` against that code is the Lua table containing the following elements :

```
{ "\\__piton_begin_line:" }[a]
{ "{\PitonStyle{Keyword}{" }[b]
{ luatexbase.catcodetables.CatcodeTableOther[c], "def" }
{ "}}" }
{ luatexbase.catcodetables.CatcodeTableOther, " " }
{ "{\PitonStyle{Name.Function}{" }
{ luatexbase.catcodetables.CatcodeTableOther, "parity" }
{ "}}" }
{ luatexbase.catcodetables.CatcodeTableOther, "(" }
{ luatexbase.catcodetables.CatcodeTableOther, "x" }
{ luatexbase.catcodetables.CatcodeTableOther, ")" }
{ luatexbase.catcodetables.CatcodeTableOther, ":" }
{ "\\__piton_end_line: \\__piton_newline: \\__piton_begin_line:" }
{ luatexbase.catcodetables.CatcodeTableOther, "    " }
{ "{\PitonStyle{Keyword}{" }
{ luatexbase.catcodetables.CatcodeTableOther, "return" }
{ "}}" }
{ luatexbase.catcodetables.CatcodeTableOther, " " }
{ luatexbase.catcodetables.CatcodeTableOther, "x" }
{ "{\PitonStyle{Operator}{" }
{ luatexbase.catcodetables.CatcodeTableOther, "&" }
{ "}}" }
{ "{\PitonStyle{Number}{" }
{ luatexbase.catcodetables.CatcodeTableOther, "2" }
{ "}}" }
{ "\\__piton_end_line:" }
```

[a]Each line of the Python 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`).

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

[c]`luatexbase.catcodetables.CatcodeTableOther` 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).

[33]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.

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_newline:
\__piton_begin_line:␣␣␣␣{\PitonStyle{Keyword}{return}}
␣x{\PitonStyle{Operator}{%}}{\PitonStyle{Number}{2}}\__piton_end_line:
```

10.2 The L3 part of the implementation

10.2.1 Declaration of the package

```
⟨*STY⟩
\NeedsTeXFormat{LaTeX2e}
\RequirePackage{l3keys2e}
\ProvidesExplPackage
  {piton}
  {\PitonFileDate}
  {\PitonFileVersion}
  {Highlight informatic listings with LPEG on LuaLaTeX}
```

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

```
\RequirePackage { amstext }
```

```
\cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
\cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
\cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
\cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
\cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
\cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
\cs_new_protected:Npn \@@_msg_new:nn { \msg_new:nnn { piton } }
\cs_new_protected:Npn \@@_gredirect_none:n #1
  {
    \group_begin:
    \globaldefs = 1
    \msg_redirect_name:nnn { piton } { #1 } { none }
    \group_end:
  }
```

With Overleaf, 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).

```
\cs_new_protected:Npn \@@_msg_new:nnn #1 #2 #3
  {
    \bool_if:NTF \g_@@_messages_for_Overleaf_bool
      { \msg_new:nnn { piton } { #1 } { #2 \\ #3 } }
      { \msg_new:nnnn { piton } { #1 } { #2 } { #3 } }
  }
```

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

```
\cs_new_protected:Npn \@@_error_or_warning:n
  { \bool_if:NTF \g_@@_messages_for_Overleaf_bool \@@_warning:n \@@_error:n }
```

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`".

```
\bool_new:N \g_@@_messages_for_Overleaf_bool
\bool_gset:Nn \g_@@_messages_for_Overleaf_bool
```

```
  {
        \str_if_eq_p:on \c_sys_jobname_str { _region_ }  % for Emacs
    || \str_if_eq_p:on \c_sys_jobname_str { output }   % for Overleaf
  }
```

```
\@@_msg_new:nn { LuaLaTeX~mandatory }
  {
    LuaLaTeX~is~mandatory.\\
    The~package~'piton'~requires~the~engine~LuaLaTeX.\\
    \str_if_eq:onT \c_sys_jobname_str { output }
      { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~the~"Menu". \\}
    If~you~go~on,~the~package~'piton'~won't~be~loaded.
  }
\sys_if_engine_luatex:F { \msg_critical:nn { piton } { LuaLaTeX~mandatory } }
```

```
\RequirePackage { luatexbase }
\RequirePackage { luacode }
```

```
\@@_msg_new:nnn { piton.lua~not~found }
  {
    The~file~'piton.lua'~can't~be~found.\\
    This~error~is~fatal.\\
    If~you~want~to~know~how~to~retrieve~the~file~'piton.lua',~type~H~<return>.
  }
  {
    On~the~site~CTAN,~go~to~the~page~of~'piton':~https://ctan.org/pkg/piton.~
    The~file~'README.md'~explains~how~to~retrieve~the~files~'piton.sty'~and~
    'piton.lua'.
  }
```

```
\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.

```
\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.

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

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

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

```
\bool_new:N \g_@@_beamer_bool
\tl_new:N \g_@@_escape_inside_tl
```

In version 4.0 of piton, we changed the mechanism used by piton to search the file to load with `\PitonInputFile`. With the key `old-PitonInputFile`, it's possible to keep the old behaviour but it's only for backward compatibility and it will be deleted in a future version.

```
\bool_new:N \l_@@_old_PitonInputFile_bool
```

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

```
\keys_define:nn { piton / package }
  {
    footnote .bool_gset:N = \g_@@_footnote_bool ,
    footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
    footnote .usage:n = load ,
    footnotehyper .usage:n = load ,

    beamer .bool_gset:N = \g_@@_beamer_bool ,
    beamer .default:n = true ,
```

```
    beamer .usage:n = load ,
```

In version 4.0 of piton, we changed the mechanism used by piton to search the file to load with `\PitonInputFile`. With the key `old-PitonInputFile`, it's possible to keep the old behaviour but it's only for backward compatibility and it will be deleted in a future version.

```
    old-PitonInputFile .bool_set:N = \l_@@_old_PitonInputFile_bool ,
    old-PitonInputFile .default:n = true ,
    old-PitonInputFile .usage:n = load ,

    unknown .code:n = \@@_error:n { Unknown~key~for~package }
  }
\@@_msg_new:nn { Unknown~key~for~package }
  {
    Unknown~key.\\
    You~have~used~the~key~'\l_keys_key_str'~but~the~only~keys~available~here~
    are~'beamer',~'footnote',~'footnotehyper'~and~'old-PitonInputFile'.~
    Other~keys~are~available~in~\token_to_str:N \PitonOptions.\\
    That~key~will~be~ignored.
  }
```

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

```
\ProcessKeysOptions { piton / package }
```

```
\msg_new:nnn { piton } { old-PitonInputFile }
  {
    Be~careful:~The~key~'old-PitonInputFile'~will~be~deleted~
    in~a~future~version~of~'piton'.
  }
\bool_if:NT \l_@@_old_PitonInputFile_bool
  { \msg_warning:nn { piton } { old-PitonInputFile } }
```

```
\IfClassLoadedTF { beamer } { \bool_gset_true:N \g_@@_beamer_bool } { }
\IfPackageLoadedTF { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool } { }
\lua_now:n { piton = piton~or~{ } }
\bool_if:NT \g_@@_beamer_bool { \lua_now:n { piton.beamer = true } }
```

```
\hook_gput_code:nnn { begindocument / before } { . }
  { \IfPackageLoadedTF { xcolor } { } { \usepackage { xcolor } } }
\@@_msg_new:nn { footnote~with~footnotehyper~package }
  {
    Footnote~forbidden.\\
    You~can't~use~the~option~'footnote'~because~the~package~
    footnotehyper~has~already~been~loaded.~
    If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
    within~the~environments~of~piton~will~be~extracted~with~the~tools~
    of~the~package~footnotehyper.\\
    If~you~go~on,~the~package~footnote~won't~be~loaded.
  }
\@@_msg_new:nn { footnotehyper~with~footnote~package }
  {
    You~can't~use~the~option~'footnotehyper'~because~the~package~
    footnote~has~already~been~loaded.~
    If~you~want,~you~can~use~the~option~'footnote'~and~the~footnotes~
    within~the~environments~of~piton~will~be~extracted~with~the~tools~
    of~the~package~footnote.\\
    If~you~go~on,~the~package~footnotehyper~won't~be~loaded.
  }

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

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

```
    \IfClassLoadedTF { beamer }
      { \bool_gset_false:N \g_@@_footnote_bool }
      {
        \IfPackageLoadedTF { footnotehyper }
          { \@@_error:n { footnote~with~footnotehyper~package } }
          { \usepackage { footnote } }
      }
  }
\bool_if:NT \g_@@_footnotehyper_bool
  {
```

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

```
    \IfClassLoadedTF { beamer }
      { \bool_gset_false:N \g_@@_footnote_bool }
      {
        \IfPackageLoadedTF { footnote }
          { \@@_error:n { footnotehyper~with~footnote~package } }
          { \usepackage { footnotehyper } }
        \bool_gset_true:N \g_@@_footnote_bool
      }
  }
```

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}`.

```
\lua_now:n
  {
    piton.BeamerCommands = lpeg.P [[\uncover]]
       + [[\only]] + [[\visible]] + [[\invisible]] + [[\alert]] + [[\action]]
    piton.beamer_environments = { "uncoverenv" , "onlyenv" , "visibleenv" ,
                "invisibleenv" ,  "alertenv" ,  "actionenv" }
    piton.DetectedCommands = lpeg.P ( false )
    piton.last_code = ''
    piton.last_language = ''
  }
```

10.2.2 Parameters and technical definitions

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

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

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

```
\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`).

```
\seq_new:N \l_@@_path_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`).

```
\str_new:N \l_@@_path_write_str
```

In order to have a better control over the keys.

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

The following parameter corresponds to the key font-command.

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

```
\int_new:N \l_@@_nb_lines_int
```

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

```
\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).

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

The following token list will contain the (potential) information to write on the aux (to be used in the next compilation). The technic of the auxiliary file will be used when the key width is used with the value min.

```
\tl_new:N \g_@@_aux_tl
```

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).

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

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

```
\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 informatic code provided by the final user is split in chunks on the empty lines in the code).

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

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

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).

```
\tl_new:N \l_@@_prompt_bg_color_tl
```

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

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

The argument of \PitonInputFile.

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

We will count the environments {Piton} (and, in fact, also the commands \PitonInputFile, despite the name \g_@@_env_int).

```
\int_new:N \g_@@_env_int
```

The parameter `\l_@@_writer_str` corresponds to the key `write`. We will store the list of the files already used in `\g_@@_write_seq` (we must not erase a file which has been still been used).

```
\str_new:N \l_@@_write_str
\seq_new:N \g_@@_write_seq
```

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

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

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

```
\bool_new:N \l_@@_break_lines_in_Piton_bool
\bool_new:N \l_@@_indent_broken_lines_bool
```

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

```
\tl_new:N \l_@@_continuation_symbol_tl
\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`.

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

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

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

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

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

However, the key `break-lines_in_piton` raises that boolean but also executes the following instruction:

```
\tl_set_eq:NN \l_@@_space_in_string_tl \space
```

The initial value of `\l_@@_space_in_string_tl` is `\nobreakspace`.

The following dimension will be the width of the listing constructed by `{Piton}` or `\PitonInputFile`.

- If the user uses the key `width` of `\PitonOptions` with a numerical value, that value will be stored in `\l_@@_width_dim`.
- If the user uses the key `width` with the special value `min`, the dimension `\l_@@_width_dim` will, *in the second run*, be computed from the value of `\l_@@_line_width_dim` stored in the `aux` file (computed during the first run the maximal width of the lines of the listing). During the first run, `\l_@@_width_line_dim` will be set equal to `\linewidth`.
- Elsewhere, `\l_@@_width_dim` will be set at the beginning of the listing (in `\@@_pre_env:`) equal to the current value of `\linewidth`.

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

We will also use another dimension called `\l_@@_line_width_dim`. That will the width of the actual lines of code. That dimension may be lower than the whole `\l_@@_width_dim` because we have to take into account the value of `\l_@@_left_margin_dim` (for the numbers of lines when `line-numbers` is in force) and another small margin when a background color is used (with the key `background-color`).

```
\dim_new:N \l_@@_line_width_dim
```

The following flag will be raised with the key `width` is used with the special value `min`.

```
\bool_new:N \l_@@_width_min_bool
```

If the key `width` is used with the special value `min`, we will compute the maximal width of the lines of an environment `{Piton}` in `\g_@@_tmp_width_dim` because we need it for the case of the key `width` is used with the special value `min`. We need a global variable because, when the key `footnote` is in force, each line when be composed in an environment `{savenotes}` and we need to exit our `\g_@@_tmp_width_dim` from that environment.

```
\dim_new:N \g_@@_tmp_width_dim
```

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

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

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

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

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

```
\dim_new:N \l_@@_numbers_sep_dim
\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 all the list of languages for which at least a user function has been defined.

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

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

```
\cs_new_protected:Npn \@@_tab:
  {
    \bool_if:NTF \l_@@_show_spaces_bool
      {
        \hbox_set:Nn \l_tmpa_box
          { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
        \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
        \( \mathcolor { gray }
             { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \)
      }
      { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
    \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
  }
```

The following integer corresponds to the key `gobble`.

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

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

```
\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.

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

Be careful: when executed, the following command does *not* create a space (only an incrementation of the counter).

```
\cs_new_protected:Npn \@@_leading_space:
  { \int_gincr:N \g_@@_indentation_int }
```

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

```
\cs_new_protected:Npn \@@_label:n #1
  {
    \bool_if:NTF \l_@@_line_numbers_bool
      {
        \@bsphack
        \protected@write \@auxout { }
          {
            \string \newlabel { #1 }
            {
```

Remember that the content of a line is typeset in a box *before* the composition of the potential number of line.

```
                { \int_eval:n { \g_@@_visual_line_int + 1 } }
                { \thepage }
              }
            }
          \@esphack
        }
        { \@@_error:n { label~with~lines~numbers } }
    }
```

The following commands corresponds 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 final 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).

```
\cs_new_protected:Npn \@@_marker_beginning:n #1 { }
\cs_new_protected:Npn \@@_marker_end:n #1 { }
```

The following token list will be evaluated at the beginning 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 at the beginning of the line.

```
\tl_new:N \g_@@_begin_line_hook_tl
```

For example, the LPEG `Prompt` will trigger the following command which will insert an instruction in the hook `\g_@@_begin_line_hook` to specify that a background must be inserted to the current line of code.

```
\cs_new_protected:Npn \@@_prompt:
  {
    \tl_gset:Nn \g_@@_begin_line_hook_tl
      {
        \tl_if_empty:NF \l_@@_prompt_bg_color_tl
          { \clist_set:No \l_@@_bg_color_clist \l_@@_prompt_bg_color_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:`.

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

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

10.2.3 Treatment of a line of code

```
\cs_generate_variant:Nn \@@_replace_spaces:n { o }
\cs_new_protected:Npn \@@_replace_spaces:n #1
  {
    \tl_set:Nn \l_tmpa_tl { #1 }
    \bool_if:NTF \l_@@_show_spaces_bool
      {
        \tl_set:Nn \l_@@_space_in_string_tl { ␣ } % U+2423
        \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl
      }
      {
```

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.

```
        \bool_if:NT \l_@@_break_lines_in_Piton_bool
```

```
          {
            \regex_replace_all:nnN
              { \x20 }
              { \c { @@_breakable_space: } }
              \l_tmpa_tl
            \regex_replace_all:nnN
              { \c { l_@@_space_in_string_tl } }
              { \c { @@_breakable_space: } }
              \l_tmpa_tl
          }
      }
    \l_tmpa_tl
  }
```

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:`).

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

```
\cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
  {
    \group_begin:
    \g_@@_begin_line_hook_tl
    \int_gzero:N \g_@@_indentation_int
```

First, we will put in the coffin `\l_tmpa_coffin` the actual content of a line of the code (without the potential number of line).
Be careful: There is curryfication in the following code.

```
    \bool_if:NTF \l_@@_width_min_bool
      \@@_put_in_coffin_ii:n
      \@@_put_in_coffin_i:n
      {
        \language = -1
        \raggedright
        \strut
        \@@_replace_spaces:n { #1 }
        \strut \hfil
      }
```

Now, we add the potential number of line, the potential left margin and the potential background.

```
    \hbox_set:Nn \l_tmpa_box
      {
        \skip_horizontal:N \l_@@_left_margin_dim
        \bool_if:NT \l_@@_line_numbers_bool
          {
```

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

```
            \int_set:Nn \l_tmpa_int
              {
                \lua_now:e
                  {
                    tex.sprint
                      (
                        luatexbase.catcodetables.expl ,
```

Since the argument of `tostring` will be 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`.

```
                        tostring
                          ( piton.empty_lines
                              [ \int_eval:n { \g_@@_line_int + 1 } ]
```

```
                          )
                      )
                  }
              }
            \bool_lazy_or:nnT
              { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
              { ! \l_@@_skip_empty_lines_bool }
              { \int_gincr:N \g_@@_visual_line_int }
            \bool_lazy_or:nnT
              { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
              { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
              \@@_print_number:
          }
```

If there is a background, we must remind that there is a left margin of 0.5 em for the background...

```
        \clist_if_empty:NF \l_@@_bg_color_clist
          {
```

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

```
            \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
              { \skip_horizontal:n { 0.5 em } }
          }
        \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
      }
    \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
    \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }
```

We have to explicitely begin a paragraph because we will insert a TeX box (and we don't want that box to be inserted in the vertical list).

```
    \mode_leave_vertical:
    \clist_if_empty:NTF \l_@@_bg_color_clist
      { \box_use_drop:N \l_tmpa_box }
      {
        \vtop
          {
            \hbox:n
              {
                \@@_color:N \l_@@_bg_color_clist
                \vrule height \box_ht:N \l_tmpa_box
                       depth \box_dp:N \l_tmpa_box
                       width \l_@@_width_dim
              }
            \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
            \box_use_drop:N \l_tmpa_box
          }
      }
    \group_end:
    \tl_gclear:N \g_@@_begin_line_hook_tl
  }
```

In the general case (which is also the simpler), the key `width` is not used, or (if used) it is not used with the special value `min`. In that case, the content of a line of code is composed in a vertical coffin with a width equal to `\l_@@_line_width_dim`. That coffin may, eventually, contains several lines when the key `break-lines-in-Piton` (or `break-lines`) is used.
That commands takes in its argument by curryfication.

```
\cs_set_protected:Npn \@@_put_in_coffin_i:n
  { \vcoffin_set:Nnn \l_tmpa_coffin \l_@@_line_width_dim }
```

The second case is the case when the key `width` is used with the special value `min`.

```
\cs_set_protected:Npn \@@_put_in_coffin_ii:n #1
  {
```

First, we compute the natural width of the line of code because we have to compute the natural width of the whole listing (and it will be written on the `aux` file in the variable `\l_@@_width_dim`).

```
    \hbox_set:Nn \l_tmpa_box { #1 }
```

Now, you can actualize the value of `\g_@@_tmp_width_dim` (it will be used to write on the `aux` file the natural width of the environment).

```
    \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_tmp_width_dim
      { \dim_gset:Nn \g_@@_tmp_width_dim { \box_wd:N \l_tmpa_box } }
    \hcoffin_set:Nn \l_tmpa_coffin
      {
        \hbox_to_wd:nn \l_@@_line_width_dim
```

We unpack the block in order to free the potential `\hfill` springs present in the LaTeX comments (cf. section 8.2, p. 24).

```
          { \hbox_unpack:N \l_tmpa_box \hfil }
      }
  }
```

The command `\@@_color:N` will take in as argument a reference to a comma-separated list of colors. A color will be picked by using the value of `\g_@@_line_int` (modulo the number of colors in the list).

```
\cs_set_protected:Npn \@@_color:N #1
  {
    \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
    \int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
    \tl_set:Ne \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
    \tl_if_eq:NnTF \l_tmpa_tl { none }
```

By setting `\l_@@_width_dim` to zero, the colored rectangle will be drawn with zero width and, thus, it will be a mere strut (and we need that strut).

```
      { \dim_zero:N \l_@@_width_dim }
      { \@@_color_i:o \l_tmpa_tl }
  }
```

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} }`.

```
\cs_generate_variant:Nn \@@_color_i:n { o }
\cs_set_protected:Npn \@@_color_i:n #1
  {
    \tl_if_head_eq_meaning:nNTF { #1 } [
      {
        \tl_set:Nn \l_tmpa_tl { #1 }
        \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
        \exp_last_unbraced:No \color \l_tmpa_tl
      }
      { \color { #1 } }
  }
```

The command `\@@_newline:` will be inserted by Lua between two lines of the informatic 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 informatic code (the *input*) may result in several lines in the PDF (the *output*).
- Remind that `\@@_newline:` has a rather complex behaviour because it will finish and start paragraphs.

```
\cs_new_protected:Npn \@@_newline:
  {
    \bool_if:NT \g_@@_footnote_bool \endsavenotes
```

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

```
    \int_gincr:N \g_@@_line_int
```

... it will be used to allow or disallow page breaks.
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.

```
    \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.

```
    \kern -2.5 pt
```

Now, we control page breaks after the paragraph. 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.

```
    \int_case:nn
      {
        \lua_now:e
          {
            tex.sprint
              (
                luatexbase.catcodetables.expl ,
                tostring ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
              )
          }
      }
      { 1 { \penalty 100 } 2 \nobreak }
   \bool_if:NT \g_@@_footnote_bool \savenotes
  }
```

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

```
\cs_set_protected:Npn \@@_breakable_space:
  {
    \discretionary
      { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
      {
        \hbox_overlap_left:n
          {
            {
              \normalfont \footnotesize \color { gray }
              \l_@@_continuation_symbol_tl
            }
            \skip_horizontal:n { 0.3 em }
            \clist_if_empty:NF \l_@@_bg_color_clist
              { \skip_horizontal:n { 0.5 em } }
          }
        \bool_if:NT \l_@@_indent_broken_lines_bool
          {
            \hbox:n
              {
                \prg_replicate:nn { \g_@@_indentation_int } { ~ }
                { \color { gray } \l_@@_csoi_tl }
              }
          }
      }
      { \hbox { ~ } }
  }
```

10.2.4 PitonOptions

```
\bool_new:N \l_@@_line_numbers_bool
\bool_new:N \l_@@_skip_empty_lines_bool
\bool_set_true:N \l_@@_skip_empty_lines_bool
\bool_new:N \l_@@_line_numbers_absolute_bool
\tl_new:N \l_@@_line_numbers_format_bool
\tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
\bool_new:N \l_@@_label_empty_lines_bool
```

```
\bool_set_true:N \l_@@_label_empty_lines_bool
\int_new:N \l_@@_number_lines_start_int
\bool_new:N \l_@@_resume_bool
\bool_new:N \l_@@_split_on_empty_lines_bool
\bool_new:N \l_@@_splittable_on_empty_lines_bool
```

```
\keys_define:nn { PitonOptions / marker }
  {
    beginning .code:n = \cs_set:Nn \@@_marker_beginning:n { #1 } ,
    beginning .value_required:n = true ,
    end .code:n = \cs_set:Nn \@@_marker_end:n { #1 } ,
    end .value_required:n = true ,
    include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
    include-lines .default:n = true ,
    unknown .code:n = \@@_error:n { Unknown~key~for~marker }
  }
```

```
\keys_define:nn { PitonOptions / line-numbers }
  {
    true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
    false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,

    start .code:n =
      \bool_set_true:N \l_@@_line_numbers_bool
      \int_set:Nn \l_@@_number_lines_start_int { #1 }  ,
    start .value_required:n = true ,

    skip-empty-lines .code:n =
      \bool_if:NF \l_@@_in_PitonOptions_bool
        { \bool_set_true:N \l_@@_line_numbers_bool }
      \str_if_eq:nnTF { #1 } { false }
        { \bool_set_false:N \l_@@_skip_empty_lines_bool }
        { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
    skip-empty-lines .default:n = true ,

    label-empty-lines .code:n =
      \bool_if:NF \l_@@_in_PitonOptions_bool
        { \bool_set_true:N \l_@@_line_numbers_bool }
      \str_if_eq:nnTF { #1 } { false }
        { \bool_set_false:N \l_@@_label_empty_lines_bool }
        { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
    label-empty-lines .default:n = true ,

    absolute .code:n =
      \bool_if:NTF \l_@@_in_PitonOptions_bool
        { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
        { \bool_set_true:N \l_@@_line_numbers_bool }
      \bool_if:NT \l_@@_in_PitonInputFile_bool
        {
          \bool_set_true:N \l_@@_line_numbers_absolute_bool
          \bool_set_false:N \l_@@_skip_empty_lines_bool
        } ,
    absolute .value_forbidden:n = true ,

    resume .code:n =
      \bool_set_true:N \l_@@_resume_bool
      \bool_if:NF \l_@@_in_PitonOptions_bool
        { \bool_set_true:N \l_@@_line_numbers_bool } ,
    resume .value_forbidden:n = true ,

    sep .dim_set:N = \l_@@_numbers_sep_dim ,
    sep .value_required:n = true ,

```

```
    format .tl_set:N = \l_@@_line_numbers_format_tl ,
    format .value_required:n = true ,

    unknown .code:n = \@@_error:n { Unknown~key~for~line-numbers }
  }
```

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

```
\keys_define:nn { PitonOptions }
  {
    break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
    break-strings-anywhere .default:n = true ,
    break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,
    break-numbers-anywhere .default:n = true  ,
```

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

```
    detected-commands .code:n =
      \lua_now:n { piton.addDetectedCommands('#1') } ,
    detected-commands .value_required:n = true ,
    detected-commands .usage:n = preamble ,
    detected-beamer-commands .code:n =
      \lua_now:n { piton.addBeamerCommands('#1') } ,
    detected-beamer-commands .value_required:n = true ,
    detected-beamer-commands .usage:n = preamble ,
    detected-beamer-environments .code:n =
      \lua_now:n { piton.addBeamerEnvironments('#1') } ,
    detected-beamer-environments .value_required:n = true ,
    detected-beamer-environments .usage:n = preamble ,
```

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

```
    begin-escape .code:n =
      \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
    begin-escape .value_required:n = true ,
    begin-escape .usage:n = preamble ,

    end-escape   .code:n =
      \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
    end-escape   .value_required:n = true ,
    end-escape .usage:n = preamble ,

    begin-escape-math .code:n =
      \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
    begin-escape-math .value_required:n = true ,
    begin-escape-math .usage:n = preamble ,

    end-escape-math .code:n =
      \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
    end-escape-math .value_required:n = true ,
    end-escape-math .usage:n = preamble ,

    comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
    comment-latex .value_required:n = true ,
    comment-latex .usage:n = preamble ,

    math-comments .bool_gset:N = \g_@@_math_comments_bool ,
    math-comments .default:n  = true ,
    math-comments .usage:n = preamble ,
```

Now, general keys.

```
    language      .code:n =
      \str_set:Ne \l_piton_language_str { \str_lowercase:n { #1 } } ,
    language      .value_required:n  = true ,
    path          .code:n =
      \seq_clear:N \l_@@_path_seq
```

```
    \clist_map_inline:nn { #1 }
      {
        \str_set:Nn \l_tmpa_str { ##1 }
        \seq_put_right:No \l_@@_path_seq \l_tmpa_str
      } ,
  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.

```
  path             .initial:n         = . ,
  path-write       .str_set:N         = \l_@@_path_write_str ,
  path-write       .value_required:n  = true ,
  font-command     .tl_set:N          = \l_@@_font_command_tl ,
  font-command     .value_required:n  = true ,
  gobble           .int_set:N         = \l_@@_gobble_int ,
  gobble           .value_required:n  = true ,
  auto-gobble      .code:n            = \int_set:Nn \l_@@_gobble_int { -1 } ,
  auto-gobble      .value_forbidden:n = true ,
  env-gobble       .code:n            = \int_set:Nn \l_@@_gobble_int { -2 } ,
  env-gobble       .value_forbidden:n = true ,
  tabs-auto-gobble .code:n            = \int_set:Nn \l_@@_gobble_int { -3 } ,
  tabs-auto-gobble .value_forbidden:n = true ,

  splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
  splittable-on-empty-lines .default:n  = true ,

  split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,
  split-on-empty-lines .default:n  = true ,

  split-separation .tl_set:N        = \l_@@_split_separation_tl ,
  split-separation .value_required:n = true ,

  marker .code:n =
    \bool_lazy_or:nnTF
      \l_@@_in_PitonInputFile_bool
      \l_@@_in_PitonOptions_bool
      { \keys_set:nn { PitonOptions / marker } { #1 } }
      { \@@_error:n { Invalid~key } } ,
  marker .value_required:n = true ,

  line-numbers .code:n =
    \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
  line-numbers .default:n = true ,

  splittable       .int_set:N         = \l_@@_splittable_int ,
  splittable       .default:n         = 1 ,
  background-color .clist_set:N       = \l_@@_bg_color_clist ,
  background-color .value_required:n  = true ,
  prompt-background-color .tl_set:N         = \l_@@_prompt_bg_color_tl ,
  prompt-background-color .value_required:n = true ,

  width .code:n =
    \str_if_eq:nnTF  { #1 } { min }
      {
        \bool_set_true:N \l_@@_width_min_bool
        \dim_zero:N \l_@@_width_dim
      }
      {
        \bool_set_false:N \l_@@_width_min_bool
        \dim_set:Nn \l_@@_width_dim { #1 }
      } ,
  width .value_required:n  = true ,

  write .str_set:N = \l_@@_write_str ,
```

```
    write .value_required:n = true ,

    left-margin      .code:n =
      \str_if_eq:nnTF { #1 } { auto }
        {
          \dim_zero:N \l_@@_left_margin_dim
          \bool_set_true:N \l_@@_left_margin_auto_bool
        }
        {
          \dim_set:Nn \l_@@_left_margin_dim { #1 }
          \bool_set_false:N \l_@@_left_margin_auto_bool
        } ,
    left-margin      .value_required:n  = true ,

    tab-size         .int_set:N         = \l_@@_tab_size_int ,
    tab-size         .value_required:n  = true ,
    show-spaces      .bool_set:N        = \l_@@_show_spaces_bool ,
    show-spaces      .value_forbidden:n = true ,
    show-spaces-in-strings .code:n      =
        \tl_set:Nn \l_@@_space_in_string_tl { ␣ } , % U+2423
    show-spaces-in-strings .value_forbidden:n = true ,
    break-lines-in-Piton .bool_set:N    = \l_@@_break_lines_in_Piton_bool ,
    break-lines-in-Piton .default:n     = true ,
    break-lines-in-piton .bool_set:N    = \l_@@_break_lines_in_piton_bool ,
    break-lines-in-piton .default:n     = true ,
    break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
    break-lines .value_forbidden:n      = true ,
    indent-broken-lines .bool_set:N     = \l_@@_indent_broken_lines_bool ,
    indent-broken-lines .default:n      = true ,
    end-of-broken-line  .tl_set:N       = \l_@@_end_of_broken_line_tl ,
    end-of-broken-line  .value_required:n = true ,
    continuation-symbol .tl_set:N       = \l_@@_continuation_symbol_tl ,
    continuation-symbol .value_required:n = true ,
    continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
    continuation-symbol-on-indentation .value_required:n = true ,

    first-line .code:n = \@@_in_PitonInputFile:n
      { \int_set:Nn \l_@@_first_line_int { #1 } } ,
    first-line .value_required:n = true ,

    last-line .code:n = \@@_in_PitonInputFile:n
      { \int_set:Nn \l_@@_last_line_int { #1 } } ,
    last-line .value_required:n = true ,

    begin-range .code:n = \@@_in_PitonInputFile:n
      { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
    begin-range .value_required:n = true ,

    end-range .code:n = \@@_in_PitonInputFile:n
      { \str_set:Nn \l_@@_end_range_str { #1 } } ,
    end-range .value_required:n = true ,

    range .code:n = \@@_in_PitonInputFile:n
      {
        \str_set:Nn \l_@@_begin_range_str { #1 }
        \str_set:Nn \l_@@_end_range_str { #1 }
      } ,
    range .value_required:n = true ,

    env-used-by-split .code:n =
      \lua_now:n { piton.env_used_by_split = '#1' } ,
    env-used-by-split .initial:n = Piton ,
```

```
    resume .meta:n = line-numbers/resume ,

    unknown .code:n = \@@_error:n { Unknown~key~for~PitonOptions } ,

    % deprecated
    all-line-numbers .code:n =
      \bool_set_true:N \l_@@_line_numbers_bool
      \bool_set_false:N \l_@@_skip_empty_lines_bool ,
    all-line-numbers .value_forbidden:n = true
  }
```

```
\cs_new_protected:Npn \@@_in_PitonInputFile:n #1
  {
    \bool_if:NTF \l_@@_in_PitonInputFile_bool
      { #1 }
      { \@@_error:n { Invalid~key } }
  }
```

```
\NewDocumentCommand \PitonOptions { m }
  {
    \bool_set_true:N \l_@@_in_PitonOptions_bool
    \keys_set:nn { PitonOptions } { #1 }
    \bool_set_false:N \l_@@_in_PitonOptions_bool
  }
```

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.

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

10.2.5 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.

```
\int_new:N \g_@@_visual_line_int
```

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

```
\cs_new_protected:Npn \@@_print_number:
  {
    \hbox_overlap_left:n
      {
        {
          \l_@@_line_numbers_format_tl
```

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

```
            { \int_to_arabic:n \g_@@_visual_line_int }
          }
          \skip_horizontal:N \l_@@_numbers_sep_dim
        }
  }
```

10.2.6 The command to write on the aux file

```
\cs_new_protected:Npn \@@_write_aux:
  {
    \tl_if_empty:NF \g_@@_aux_tl
      {
        \iow_now:Nn \@mainaux { \ExplSyntaxOn }
        \iow_now:Ne \@mainaux
          {
            \tl_gset:cn { c_@@_ \int_use:N \g_@@_env_int _ tl }
              { \exp_not:o \g_@@_aux_tl }
          }
        \iow_now:Nn \@mainaux { \ExplSyntaxOff }
      }
    \tl_gclear:N \g_@@_aux_tl
  }
```

The following macro with be used only when the key `width` is used with the special value `min`.

```
\cs_new_protected:Npn \@@_width_to_aux:
  {
    \tl_gput_right:Ne \g_@@_aux_tl
      {
        \dim_set:Nn \l_@@_line_width_dim
          { \dim_eval:n { \g_@@_tmp_width_dim } }
      }
  }
```

10.2.7 The main commands and environments for the final user

```
\NewDocumentCommand { \NewPitonLanguage } { O { } m ! o }
  {
    \tl_if_novalue:nTF { #3 }
```

The last argument is provided by curryfication.

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

The two last arguments are provided by curryfication.

```
      { \@@_NewPitonLanguage:nnnnn { #1 } { #2 } { #3 } }
  }
```

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

```
\prop_new:N \g_@@_languages_prop
```

```
\keys_define:nn { NewPitonLanguage }
  {
    morekeywords .code:n = ,
    otherkeywords .code:n = ,
    sensitive .code:n = ,
    keywordsprefix .code:n = ,
    moretexcs .code:n = ,
    morestring .code:n = ,
    morecomment .code:n = ,
    moredelim .code:n = ,
    moredirectives .code:n = ,
    tag .code:n = ,
    alsodigit .code:n = ,
    alsoletter .code:n = ,
    alsoother .code:n = ,
    unknown .code:n = \@@_error:n { Unknown~key~NewPitonLanguage }
  }
```

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

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

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 final user may have written `\NewPitonLanguage[ ]{Java}{...}`.

```
    \tl_set:Ne \l_tmpa_tl
      {
        \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
        \str_lowercase:n { #2 }
      }
```

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

```
    \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.

```
    \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 utilisation of the Lua function `piton.new_language` (which does the main job).

```
    \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
  }
```

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

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

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

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 final user may have used `\NewPitonLanguage[Handel]{C}[ ]{C}{...}`

```
    \tl_set:Ne \l_tmpa_tl
      {
        \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
        \str_lowercase:n { #4 }
      }
```

We retrieve in `\l_tmpb_tl` the definition (as provided by the final 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`.

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

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

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

```
\cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
\cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4
```

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

```
  { \@@_NewPitonLanguage:nnn { #1 } { #2 } { #4 , #3 } }
```

```
\NewDocumentCommand { \piton } { }
  { \peek_meaning:NTF \bgroup \@@_piton_standard \@@_piton_verbatim }
\NewDocumentCommand { \@@_piton_standard } { m }
```

```
  {
    \group_begin:
    \bool_lazy_or:nnT
    \l_@@_break_lines_in_piton_bool
```

We have to deal with the case of `break-strings-anywhere` because, otherwise, the `\nobreakspace` would result in a sequence of TeX instructions and we would have difficulities during the insertion of all the commands `\-` (to allow breaks anywhere in the string).

```
    \l_@@_break_strings_anywhere_bool
      { \tl_set_eq:NN \l_@@_space_in_string_tl \space }
```

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

```
    \automatichyphenmode = 1
```

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

```
    \cs_set_eq:NN \\ \c_backslash_str
    \cs_set_eq:NN \% \c_percent_str
    \cs_set_eq:NN \{ \c_left_brace_str
    \cs_set_eq:NN \} \c_right_brace_str
    \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.

```
    \cs_set_eq:cN { ~ } \space
    \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
    \tl_set:Ne \l_tmpa_tl
      {
        \lua_now:e
          { piton.ParseBis('\l_piton_language_str',token.scan_string()) }
          { #1 }
      }
    \bool_if:NTF \l_@@_show_spaces_bool
      { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
```

The following code replaces the characters U+0020 (spaces) by characters U+0020 of catcode 10: thus, they become breakable by an end of line. Maybe, this programmation is not very efficient but the key `break-lines-in-piton` will be rarely used.

```
      {
        \bool_if:NT \l_@@_break_lines_in_piton_bool
          { \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl }
      }
```

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

```
    \if_mode_math:
       \text { \l_@@_font_command_tl \l_tmpa_tl }
    \else:
       \l_@@_font_command_tl \l_tmpa_tl
    \fi:
    \group_end:
  }
\NewDocumentCommand { \@@_piton_verbatim } { v }
  {
    \group_begin:
    \automatichyphenmode = 1
    \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
    \tl_set:Ne \l_tmpa_tl
      {
        \lua_now:e
          { piton.Parse('\l_piton_language_str',token.scan_string()) }
          { #1 }
      }
    \bool_if:NT \l_@@_show_spaces_bool
      { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
    \if_mode_math:
       \text { \l_@@_font_command_tl \l_tmpa_tl }
```

```
    \else:
       \l_@@_font_command_tl \l_tmpa_tl
    \fi:
    \group_end:
  }
```

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

```
\cs_new_protected:Npn \@@_piton:n #1
  { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }

\cs_new_protected:Npn \@@_piton_i:n #1
  {
    \group_begin:
    \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
    \cs_set:cpn { pitonStyle _ \l_piton_language_str  _ Prompt } { }
    \cs_set:cpn { pitonStyle _ Prompt } { }
    \cs_set_eq:NN \@@_trailing_space: \space
    \tl_set:Ne \l_tmpa_tl
      {
        \lua_now:e
          { piton.ParseTer('\l_piton_language_str',token.scan_string()) }
          { #1 }
      }
    \bool_if:NT \l_@@_show_spaces_bool
      { \regex_replace_all:nnN { \x20 } { ␣ } \l_tmpa_tl } % U+2423
    \@@_replace_spaces:o \l_tmpa_tl
    \group_end:
  }
```

Despite its name, `\@@_pre_env:` will be used both in `\PitonInputFile` and in the environments such as `{Piton}`.

```
\cs_new:Npn \@@_pre_env:
  {
    \automatichyphenmode = 1
    \int_gincr:N \g_@@_env_int
    \tl_gclear:N \g_@@_aux_tl
    \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
      { \dim_set_eq:NN \l_@@_width_dim \linewidth }
```

We read the information written on the `aux` file by a previous run (when the key `width` is used with the special value `min`). At this time, the only potential information written on the `aux` file is the value of `\l_@@_line_width_dim` when the key `width` has been used with the special value `min`).

```
    \cs_if_exist_use:c { c_@@ _ \int_use:N \g_@@_env_int _ tl }
    \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
    \dim_gzero:N \g_@@_tmp_width_dim
    \int_gzero:N \g_@@_line_int
    \dim_zero:N \parindent
    \dim_zero:N \lineskip
    \cs_set_eq:NN \label \@@_label:n
  }
```

If the final 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`. The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

```
\cs_generate_variant:Nn \@@_compute_left_margin:nn { n o }
\cs_new_protected:Npn \@@_compute_left_margin:nn #1 #2
  {
    \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool
```

```
      {
       \hbox_set:Nn \l_tmpa_box
         {
           \l_@@_line_numbers_format_tl
           \bool_if:NTF \l_@@_skip_empty_lines_bool
             {
               \lua_now:n
                 { piton.#1(token.scan_argument()) }
                 { #2 }
               \int_to_arabic:n
                 { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
             }
             {
               \int_to_arabic:n
                 { \g_@@_visual_line_int + \l_@@_nb_lines_int }
             }
         }
       \dim_set:Nn \l_@@_left_margin_dim
         { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
      }
  }
```

Whereas `\l_@@_with_dim` is the width of the environment, `\l_@@_line_width_dim` is the width of the lines of code without the potential margins for the numbers of lines and the background. Depending on the case, you have to compute `\l_@@_line_width_dim` from `\l_@@_width_dim` or we have to do the opposite.

```
\cs_new_protected:Npn \@@_compute_width:
  {
    \dim_compare:nNnTF \l_@@_line_width_dim = \c_zero_dim
      {
        \dim_set_eq:NN \l_@@_line_width_dim \l_@@_width_dim
        \clist_if_empty:NTF \l_@@_bg_color_clist
```

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

```
          { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim }
```

If there is a background, we subtract 0.5 em for the margin on the right.

```
          {
            \dim_sub:Nn \l_@@_line_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[34] and we use that value. Elsewhere, we use a value of 0.5 em.

```
            \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
              { \dim_sub:Nn \l_@@_line_width_dim { 0.5 em } }
              { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim }
          }
      }
```

If `\l_@@_line_width_dim` has yet a non-zero value, that means that it has been read in the `aux` file: it has been written by a previous run because the key `width` is used with the special value `min`). We compute now the width of the environment by computations opposite to the preceding ones.

```
      {
        \dim_set_eq:NN \l_@@_width_dim \l_@@_line_width_dim
        \clist_if_empty:NTF \l_@@_bg_color_clist
          { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }
          {
            \dim_add:Nn \l_@@_width_dim { 0.5 em }
            \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
              { \dim_add:Nn \l_@@_width_dim { 0.5 em } }
              { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }
```

[34] 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.

```
        }
    }
  }
```

```
\NewDocumentCommand { \NewPitonEnvironment } { m m m m }
  {
```

We construct a TeX macro which will catch as argument all the tokens until `\end{name_env}` with, in that `\end{name_env}`, the catcodes of `\`, `{` and `}` equal to 12 ("`other`"). The latter explains why the definition of that function is a bit complicated.

```
    \use:x
      {
        \cs_set_protected:Npn
          \use:c { _@@_collect_ #1 :w }
          ####1
          \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
      }
          {
            \group_end:
```

Maybe, we should deactivate all the "shorthands" of babel (when babel is loaded) with the following instruction:

`\IfPackageLoadedT { babel } { \languageshorthands { none } }`

But we should be sure that there is no consequence in the LaTeX comments...

```
            \mode_if_vertical:TF \noindent \newline
```

The following line is only to compute `\l_@@_lines_int` which will be used only when both `left-margin=auto` and `skip-empty-lines = false` are in force. We should change that.

```
            \lua_now:e { piton.CountLines ( '\lua_escape:n{##1}' ) }
```

The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

```
            \@@_compute_left_margin:nn { CountNonEmptyLines } { ##1 }
            \@@_compute_width:
            \l_@@_font_command_tl
            \dim_zero:N \parskip
            \noindent
```

Now, the key `write`.

```
            \str_if_empty:NTF \l_@@_path_write_str
              { \lua_now:e { piton.write = "\l_@@_write_str" } }
              {
                \lua_now:e
                  { piton.write = "\l_@@_path_write_str / \l_@@_write_str" }
              }
            \str_if_empty:NTF \l_@@_write_str
              { \lua_now:n { piton.write = '' } }
              {
                \seq_if_in:NoTF \g_@@_write_seq \l_@@_write_str
                  { \lua_now:n { piton.write_mode = "a" } }
                  {
                    \lua_now:n { piton.write_mode = "w" }
                    \seq_gput_left:No \g_@@_write_seq \l_@@_write_str
                  }
              }
```

Now, the main job.

```
            \bool_if:NTF \l_@@_split_on_empty_lines_bool
              \@@_retrieve_gobble_split_parse:n
              \@@_retrieve_gobble_parse:n
              { ##1 }
```

If the user has used the key `width` with the special value `min`, we write on the `aux` file the value of `\l_@@_line_width_dim` (largest width of the lines of code of the environment).

```
            \bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:
```

The following `\end{#1}` is only for the stack of environments of LaTeX.

```
            \end { #1 }
            \@@_write_aux:
          }
```

We can now define the new environment.
We are still in the definition of the command `\NewPitonEnvironment`...

```
    \NewDocumentEnvironment { #1 } { #2 }
      {
        \cs_set_eq:NN \PitonOptions \@@_fake_PitonOptions
        #3
        \@@_pre_env:
        \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
          { \int_gset:Nn \g_@@_visual_line_int { \l_@@_number_lines_start_int - 1 } }
        \group_begin:
        \tl_map_function:nN
          { \ \\ \{ \} \$ \& \# \^ \_ \% \~ \^^I }
          \char_set_catcode_other:N
        \use:c { _@@_collect_ #1 :w }
      }
      { #4 }
```

The following code is for technical reasons. We want to change the catcode of `^^M` before catching the arguments of the new environment we are defining. Indeed, if not, we will have problems if there is a final optional argument in our environment (if that final argument is not used by the user in an instance of the environment, a spurious space is inserted, probably because the `^^M` is converted to space).

```
    \AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^^M }
  }
```

This is the end of the definition of the command `\NewPitonEnvironment`.

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.

```
\cs_new_protected:Npn \@@_retrieve_gobble_parse:n
  {
    \lua_now:e
      {
        piton.RetrieveGobbleParse
          (
            '\l_piton_language_str' ,
            \int_use:N \l_@@_gobble_int ,
            \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
              { \int_eval:n { - \l_@@_splittable_int } }
              { \int_use:N \l_@@_splittable_int } ,
            token.scan_argument ( )
          )
      }
  }
```

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.

```
\cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
  {
    \lua_now:e
      {
        piton.RetrieveGobbleSplitParse
          (
            '\l_piton_language_str' ,
            \int_use:N \l_@@_gobble_int ,
            \int_use:N \l_@@_splittable_int ,
```

```
            token.scan_argument ( )
          )
      }
  }
```

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

```
\bool_if:NTF \g_@@_beamer_bool
  {
    \NewPitonEnvironment { Piton } { d < > O { } }
      {
        \keys_set:nn { PitonOptions } { #2 }
        \tl_if_novalue:nTF { #1 }
          { \begin { uncoverenv } }
          { \begin { uncoverenv } < #1 > }
      }
      { \end { uncoverenv } }
  }
  {
    \NewPitonEnvironment { Piton } { O { } }
      { \keys_set:nn { PitonOptions } { #1 } }
      { }
  }
```

The code of the command `\PitonInputFile` is somewhat similar to the code of the environment `{Piton}`. In fact, it's simpler because there isn't the problem of catching the content of the environment in a verbatim mode.

```
\NewDocumentCommand { \PitonInputFileTF } { d < > O { } m m m }
  {
    \group_begin:
```

In version 4.0 of piton, we changed the mechanism used by piton to search the file to load with `\PitonInputFile`. With the key `old-PitonInputFile`, it's possible to keep the old behaviour but it's only for backward compatibility and it will be deleted in a future version.

```
    \bool_if:NTF \l_@@_old_PitonInputFile_bool
      {
        \bool_set_false:N \l_tmpa_bool
        \seq_map_inline:Nn \l__piton_path_seq
          {
            \str_set:Nn \l__piton_file_name_str { ##1 / #3 }
            \file_if_exist:nT { \l__piton_file_name_str }
              {
                \__piton_input_file:nn { #1 } { #2 }
                \bool_set_true:N \l_tmpa_bool
                \seq_map_break:
              }
          }
        \bool_if:NTF \l_tmpa_bool { #4 } { #5 }
      }
      {
        \seq_concat:NNN
          \l_file_search_path_seq
          \l_@@_path_seq
          \l_file_search_path_seq
        \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
          {
            \@@_input_file:nn { #1 } { #2 }
            #4
          }
          { #5 }
      }
    \group_end:
  }
```

```
\cs_new_protected:Npn \@@_unknown_file:n #1
  { \msg_error:nnn { piton } { Unknown~file } { #1 } }
\NewDocumentCommand { \PitonInputFile } { d < > O { } m }
  { \PitonInputFileTF < #1 >  [ #2 ] { #3 } { } { \@@_unknown_file:n { #3 } } }
\NewDocumentCommand { \PitonInputFileT } { d < > O { } m m }
  { \PitonInputFileTF < #1 >  [ #2 ] { #3 } { #4 } { \@@_unknown_file:n { #3 } } }
\NewDocumentCommand { \PitonInputFileF } { d < > O { } m m }
  { \PitonInputFileTF < #1 >  [ #2 ] { #3 } { } { #4 } }
```

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

```
\cs_new_protected:Npn \@@_input_file:nn #1 #2
  {
```

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 `>`).

```
    \tl_if_novalue:nF { #1 }
      {
        \bool_if:NTF \g_@@_beamer_bool
          { \begin { uncoverenv } < #1 > }
          { \@@_error_or_warning:n { overlay~without~beamer } }
      }
    \group_begin:
      \int_zero_new:N \l_@@_first_line_int
      \int_zero_new:N \l_@@_last_line_int
      \int_set_eq:NN \l_@@_last_line_int \c_max_int
      \bool_set_true:N \l_@@_in_PitonInputFile_bool
      \keys_set:nn { PitonOptions } { #2 }
      \bool_if:NT \l_@@_line_numbers_absolute_bool
        { \bool_set_false:N \l_@@_skip_empty_lines_bool }
      \bool_if:nTF
        {
          (
            \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
            || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
          )
          && ! \str_if_empty_p:N \l_@@_begin_range_str
        }
        {
          \@@_error_or_warning:n { bad~range~specification }
          \int_zero:N \l_@@_first_line_int
          \int_set_eq:NN \l_@@_last_line_int \c_max_int
        }
        {
          \str_if_empty:NF \l_@@_begin_range_str
            {
              \@@_compute_range:
              \bool_lazy_or:nnT
                \l_@@_marker_include_lines_bool
                { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
                {
                  \int_decr:N \l_@@_first_line_int
                  \int_incr:N \l_@@_last_line_int
                }
            }
        }
      \@@_pre_env:
      \bool_if:NT \l_@@_line_numbers_absolute_bool
        { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
      \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
        {
          \int_gset:Nn \g_@@_visual_line_int
            { \l_@@_number_lines_start_int - 1 }
        }
```

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

```
      \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
        { \int_gzero:N \g_@@_visual_line_int }
      \mode_if_vertical:TF \mode_leave_vertical: \newline
```

We count with Lua the number of lines of the argument. The result will be stored by Lua in `\l_@@_nb_lines_int`.

```
      \lua_now:e { piton.CountLinesFile ( '\l_@@_file_name_str' ) }
```

The first argument of the following function is the name of the Lua function that will be applied to the second argument in order to count the number of lines.

```
      \@@_compute_left_margin:no { CountNonEmptyLinesFile } \l_@@_file_name_str
      \@@_compute_width:
      \l_@@_font_command_tl
      \lua_now:e
        {
          piton.ParseFile(
           '\l_piton_language_str' ,
           '\l_@@_file_name_str' ,
           \int_use:N \l_@@_first_line_int ,
           \int_use:N \l_@@_last_line_int ,
           \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
             { \int_eval:n { - \l_@@_splittable_int } }
             { \int_use:N \l_@@_splittable_int } ,
           \bool_if:NTF \l_@@_split_on_empty_lines_bool { 1 } { 0 } )
        }
      \bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:
    \group_end:
```

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

```
    \iow_log:e {(\l_@@_file_name_str)}
```

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.

```
    \tl_if_novalue:nF { #1 }
      { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
    \@@_write_aux:
  }
```

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

```
\cs_new_protected:Npn \@@_compute_range:
  {
```

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

```
    \str_set:Ne \l_tmpa_str { \@@_marker_beginning:n \l_@@_begin_range_str }
    \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, more unlikely, suffixes) added to the markers by the functions `\@@_marker_beginning:n` and `\@@_marker_end:n`

```
    \regex_replace_all:nVN { \\\# } \c_hash_str \l_tmpa_str
    \regex_replace_all:nVN { \\\# } \c_hash_str \l_tmpb_str
```

However, it seems that our programmation is not good programmation because our `\l_tmpa_str` is not a valid `str` value (maybe we should correct that).

```
    \lua_now:e
      {
        piton.ComputeRange
          ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
      }
  }
```

10.2.8 The styles

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

```
\NewDocumentCommand { \PitonStyle } { m }
  {
    \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str  _ #1 }
      { \use:c { pitonStyle _ #1 } }
  }

\NewDocumentCommand { \SetPitonStyle } { O { } m }
  {
    \str_clear_new:N \l_@@_SetPitonStyle_option_str
    \str_set:Ne \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
    \str_if_eq:onT \l_@@_SetPitonStyle_option_str { current-language }
      { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
    \keys_set:nn { piton / Styles } { #2 }
  }

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

\clist_new:N \g_@@_styles_clist
\clist_gset:Nn \g_@@_styles_clist
  {
    Comment ,
    Comment.LaTeX ,
    Discard ,
    Exception ,
    FormattingType ,
    Identifier.Internal ,
    Identifier ,
    InitialValues ,
    Interpol.Inside ,
    Keyword ,
    Keyword.Governing ,
    Keyword.Constant ,
    Keyword2 ,
    Keyword3 ,
    Keyword4 ,
    Keyword5 ,
    Keyword6 ,
    Keyword7 ,
    Keyword8 ,
    Keyword9 ,
    Name.Builtin ,
    Name.Class ,
    Name.Constructor ,
    Name.Decorator ,
    Name.Field ,
    Name.Function ,
    Name.Module ,
    Name.Namespace ,
    Name.Table ,
    Name.Type ,
    Number ,
    Number.Internal ,
    Operator ,
    Operator.Word ,
    Preproc ,
    Prompt ,
    String.Doc ,
    String.Interpol ,
    String.Long ,
    String.Long.Internal ,
    String.Short ,
    String.Short.Internal ,
```

```
    Tag ,
    TypeParameter ,
    UserFunction ,
```

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

```
    TypeExpression ,
```

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

```
    Directive
  }

\clist_map_inline:Nn \g_@@_styles_clist
  {
    \keys_define:nn { piton / Styles }
      {
        #1 .value_required:n = true ,
        #1 .code:n =
          \tl_set:cn
            {
              pitonStyle _
              \str_if_empty:NF \l_@@_SetPitonStyle_option_str
                { \l_@@_SetPitonStyle_option_str _ }
              #1
            }
            { ##1 }
      }
  }

\keys_define:nn { piton / Styles }
  {
    String       .meta:n = { String.Long = #1 , String.Short = #1 } ,
    Comment.Math .tl_set:c = pitonStyle _ Comment.Math  ,
    unknown         .code:n =
      \@@_error:n { Unknown~key~for~SetPitonStyle }
  }

\SetPitonStyle[OCaml]
  {
    TypeExpression =
      \SetPitonStyle { Identifier = \PitonStyle { Name.Type } }
      \@@_piton:n ,
  }
```

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`.

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

Of course, we sort that clist.

```
\clist_gsort:Nn \g_@@_styles_clist
  {
    \str_compare:nNnTF { #1 } < { #2 }
      \sort_return_same:
      \sort_return_swapped:
  }

% \bool_new:N \l_@@_break_strings_anywhere_bool
\cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:

\cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:

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

```
    \regex_replace_all:nnN { \x20 } { \c { space } } \l_tmpa_tl
    \tl_map_inline:Nn \l_tmpa_tl
      { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
    \seq_use:Nn \l_tmpa_seq { \- }
  }
```

```
\cs_new_protected:Npn \@@_string_long:n #1
  {
    \PitonStyle { String.Long }
      {
        \bool_if:NT \l_@@_break_strings_anywhere_bool
          { \@@_actually_break_anywhere:n }
        { #1 }
      }
  }
\cs_new_protected:Npn \@@_string_short:n #1
  {
    \PitonStyle { String.Short }
      {
        \bool_if:NT \l_@@_break_strings_anywhere_bool
          { \@@_actually_break_anywhere:n }
        { #1 }
      }
  }
\cs_new_protected:Npn \@@_number:n #1
  {
    \PitonStyle { Number }
      {
        \bool_if:NT \l_@@_break_numbers_anywhere_bool
          { \@@_actually_break_anywhere:n }
        { #1 }
      }
  }
```

10.2.9 The initial styles

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

```
\SetPitonStyle
  {
    Comment               = \color[HTML]{0099FF} \itshape ,
    Exception             = \color[HTML]{CC0000} ,
    Keyword               = \color[HTML]{006699} \bfseries ,
    Keyword.Governing     = \color[HTML]{006699} \bfseries ,
    Keyword.Constant      = \color[HTML]{006699} \bfseries ,
    Name.Builtin          = \color[HTML]{336666} ,
    Name.Decorator        = \color[HTML]{9999FF},
    Name.Class            = \color[HTML]{00AA88} \bfseries ,
    Name.Function         = \color[HTML]{CC00FF} ,
    Name.Namespace        = \color[HTML]{00CCFF} ,
    Name.Constructor      = \color[HTML]{006000} \bfseries ,
    Name.Field            = \color[HTML]{AA6600} ,
    Name.Module           = \color[HTML]{0060A0} \bfseries ,
    Name.Table            = \color[HTML]{309030} ,
    Number                = \color[HTML]{FF6600} ,
    Number.Internal       = \@@_number:n ,
    Operator              = \color[HTML]{555555} ,
    Operator.Word         = \bfseries ,
    String                = \color[HTML]{CC3300} ,
```

```
    String.Long.Internal  = \@@_string_long:n ,
    String.Short.Internal = \@@_string_short:n ,
    String.Doc            = \color[HTML]{CC3300} \itshape ,
    String.Interpol       = \color[HTML]{AA0000} ,
    Comment.LaTeX         = \normalfont \color[rgb]{.468,.532,.6} ,
    Name.Type             = \color[HTML]{336666} ,
    InitialValues         = \@@_piton:n ,
    Interpol.Inside       = \l_@@_font_command_tl \@@_piton:n ,
    TypeParameter         = \color[HTML]{336666} \itshape ,
    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).

```
    Identifier.Internal   = \@@_identifier:n ,
    Identifier            = ,
    Directive             = \color[HTML]{AA6600} ,
    Tag                   = \colorbox{gray!10},
    UserFunction          = \PitonStyle{Identifier} ,
    Prompt                = ,
    Discard               = \use_none:n
  }
```

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)].

```
\hook_gput_code:nnn { begindocument } { . }
  {
    \bool_if:NT \g_@@_math_comments_bool
      { \SetPitonStyle { Comment.Math = \@@_math_scantokens:n } }
  }
```

10.2.10 Highlighting some identifiers

```
\NewDocumentCommand { \SetPitonIdentifier } { o m m }
  {
    \clist_set:Nn \l_tmpa_clist { #2 }
    \tl_if_novalue:nTF { #1 }
      {
        \clist_map_inline:Nn \l_tmpa_clist
          { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
      }
      {
        \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
        \str_if_eq:onT \l_tmpa_str { current-language }
          { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
        \clist_map_inline:Nn \l_tmpa_clist
          { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
      }
  }
\cs_new_protected:Npn \@@_identifier:n #1
  {
    \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }
      {
        \cs_if_exist_use:cF { PitonIdentifier _ #1 }
          { \PitonStyle { Identifier } }
      }
    { #1 }
  }
```

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`).

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

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

```
    { \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}`).

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

```
    \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
      { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
    \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.

```
    \seq_if_in:NoF \g_@@_languages_seq \l_piton_language_str
      { \seq_gput_left:No \g_@@_languages_seq \l_piton_language_str }
  }

\NewDocumentCommand \PitonClearUserFunctions { ! o }
  {
    \tl_if_novalue:nTF { #1 }
```

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

```
      { \@@_clear_all_functions: }
      { \@@_clear_list_functions:n { #1 } }
  }

\cs_new_protected:Npn \@@_clear_list_functions:n #1
  {
    \clist_set:Nn \l_tmpa_clist { #1 }
    \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
    \clist_map_inline:nn { #1 }
      { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
  }

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

```
\cs_generate_variant:Nn \@@_clear_functions_ii:n { e }
\cs_new_protected:Npn \@@_clear_functions_ii:n #1
  {
    \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
      {
        \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
          { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} }
        \seq_gclear:c { g_@@_functions _ #1 _ seq }
      }
  }

\cs_new_protected:Npn \@@_clear_functions:n #1
  {
```

```
    \@@_clear_functions_i:n { #1 }
    \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
  }
```

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

```
\cs_new_protected:Npn \@@_clear_all_functions:
  {
    \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
    \seq_gclear:N \g_@@_languages_seq
  }
```

10.2.11 Security

```
\AddToHook { env / piton / begin }
  { \@@_fatal:n { No~environment~piton } }

\msg_new:nnn { piton } { No~environment~piton }
  {
    There~is~no~environment~piton!\\
    There~is~an~environment~{Piton}~and~a~command~
    \token_to_str:N \piton\ but~there~is~no~environment~
    {piton}.~This~error~is~fatal.
  }
```

10.2.12 The error messages of the package

```
\@@_msg_new:nn { Language~not~defined }
  {
    Language~not~defined \\
    The~language~'\l_tmpa_tl'~has~not~been~defined~previously.\\
    If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\
    will~be~ignored.
  }
\@@_msg_new:nn { bad~version~of~piton.lua }
  {
    Bad~number~version~of~'piton.lua'\\
    The~file~'piton.lua'~loaded~has~not~the~same~number~of~
    version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
    address~that~issue.
  }
\@@_msg_new:nn { Unknown~key~NewPitonLanguage }
  {
    Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
    The~key~'\l_keys_key_str'~is~unknown.\\
    This~key~will~be~ignored.\\
  }
\@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
  {
    The~style~'\l_keys_key_str'~is~unknown.\\
    This~key~will~be~ignored.\\
    The~available~styles~are~(in~alphabetic~order):~
    \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
  }
\@@_msg_new:nn { Invalid~key }
  {
    Wrong~use~of~key.\\
    You~can't~use~the~key~'\l_keys_key_str'~here.\\
    That~key~will~be~ignored.
  }
\@@_msg_new:nn { Unknown~key~for~line-numbers }
  {
```

```
    Unknown~key. \\
    The~key~'line-numbers / \l_keys_key_str'~is~unknown.\\
    The~available~keys~of~the~family~'line-numbers'~are~(in~
    alphabetic~order):~
    absolute,~false,~label-empty-lines,~resume,~skip-empty-lines,~
    sep,~start~and~true.\\
    That~key~will~be~ignored.
  }
\@@_msg_new:nn { Unknown~key~for~marker }
  {
    Unknown~key. \\
    The~key~'marker / \l_keys_key_str'~is~unknown.\\
    The~available~keys~of~the~family~'marker'~are~(in~
    alphabetic~order):~ beginning,~end~and~include-lines.\\
    That~key~will~be~ignored.
  }
\@@_msg_new:nn { bad~range~specification }
  {
    Incompatible~keys.\\
    You~can't~specify~the~range~of~lines~to~include~by~using~both~
    markers~and~explicit~number~of~lines.\\
    Your~whole~file~'\l_@@_file_name_str'~will~be~included.
  }
```

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`.

```
\@@_msg_new:nn { SyntaxError }
  {
    Syntax~Error.\\
    Your~code~of~the~language~'\l_piton_language_str'~is~not~
    syntactically~correct.\\
    It~won't~be~printed~in~the~PDF~file.
  }
\@@_msg_new:nn { FileError }
  {
    File~Error.\\
    It's~not~possible~to~write~on~the~file~'\l_@@_write_str'.\\
    \sys_if_shell_unrestricted:F { Be~sure~to~compile~with~'-shell-escape'.\\ }
    If~you~go~on,~nothing~will~be~written~on~the~file.
  }
\@@_msg_new:nn { begin~marker~not~found }
  {
    Marker~not~found.\\
    The~range~'\l_@@_begin_range_str'~provided~to~the~
    command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
    The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
  }
\@@_msg_new:nn { end~marker~not~found }
  {
    Marker~not~found.\\
    The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
    provided~to~the~command~\token_to_str:N \PitonInputFile\
    has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
    be~inserted~till~the~end.
  }
\@@_msg_new:nn { Unknown~file }
  {
    Unknown~file. \\
    The~file~'#1'~is~unknown.\\
    Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
  }
```

```
\@@_msg_new:nnn { Unknown~key~for~PitonOptions }
  {
    Unknown~key. \\
    The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
    It~will~be~ignored.\\
    For~a~list~of~the~available~keys,~type~H~<return>.
  }
  {
    The~available~keys~are~(in~alphabetic~order):~
    auto-gobble,~
    background-color,~
    begin-range,~
    break-lines,~
    break-lines-in-piton,~
    break-lines-in-Piton,~
    break-numbers-anywhere,~
    break-strings-anywhere,~
    continuation-symbol,~
    continuation-symbol-on-indentation,~
    detected-beamer-commands,~
    detected-beamer-environments,~
    detected-commands,~
    end-of-broken-line,~
    end-range,~
    env-gobble,~
    env-used-by-split,~
    font-command,~
    gobble,~
    indent-broken-lines,~
    language,~
    left-margin,~
    line-numbers/,~
    marker/,~
    math-comments,~
    path,~
    path-write,~
    prompt-background-color,~
    resume,~
    show-spaces,~
    show-spaces-in-strings,~
    splittable,~
    splittable-on-empty-lines,~
    split-on-empty-lines,~
    split-separation,~
    tabs-auto-gobble,~
    tab-size,~
    width~and~write.
  }

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

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

10.2.13 We load piton.lua

```
\cs_new_protected:Npn \@@_test_version:n #1
  {
    \str_if_eq:onF \PitonFileVersion { #1 }
      { \@@_error:n { bad~version~of~piton.lua } }
  }
```

```
\hook_gput_code:nnn { begindocument } { . }
  {
    \lua_now:n
      {
        require ( "piton" )
        tex.sprint ( luatexbase.catcodetables.CatcodeTableExpl ,
                     "\\@@_test_version:n {" .. piton_version ..  "}" )
      }
  }
```

10.2.14 Detected commands

```
\ExplSyntaxOff
\begin{luacode*}
    lpeg.locale(lpeg)
    local P , alpha , C , space , S , V
      = lpeg.P , lpeg.alpha , lpeg.C , lpeg.space , lpeg.S , lpeg.V
    local add
    function add(...)
      local s = P ( false )
      for _ , x in ipairs({...}) do s = s + x end
      return s
    end
    local my_lpeg =
      P {  "E" ,
           E = ( V "F" * ( "," * V "F" ) ^ 0 ) / add ,
```

Be careful: in Lua, / has no priority over *. Of course, we want a behaviour for this comma-separated list equal to the behaviour of a `clist` of L3.

```
           F = space ^ 0 * ( ( alpha ^ 1 ) / "\\%0" ) * space ^ 0
        }
    function piton.addDetectedCommands ( key_value )
      piton.DetectedCommands = piton.DetectedCommands + my_lpeg : match ( key_value )
    end
    function piton.addBeamerCommands( key_value )
      piton.BeamerCommands
       = piton.BeamerCommands + my_lpeg : match ( key_value )
    end
    local insert
    function insert(...)
      local s = piton.beamer_environments
      for _ , x in ipairs({...}) do table.insert(s,x) end
      return s
    end
    local my_lpeg_bis =
      P {  "E" ,
           E = ( V "F" * ( "," * V "F" ) ^ 0 ) / insert ,
           F = space ^ 0 * ( alpha ^ 1 ) * space ^ 0
        }
    function piton.addBeamerEnvironments( key_value )
      piton.beamer_environments = my_lpeg_bis : match ( key_value )
    end
\end{luacode*}
⟨/STY⟩
```

10.3 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 piton.

```
⟨*LUA⟩
piton.comment_latex = piton.comment_latex or ">"
piton.comment_latex = "#" .. piton.comment_latex

local sprintL3
function sprintL3 ( s )
  tex.sprint ( luatexbase.catcodetables.expl , s )
end
```

10.3.1 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.

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

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 informatic listings that piton will typeset verbatim (thanks to the catcode "other").

```
local Q
function Q ( pattern )
  return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
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.

```
local L
function L ( pattern ) return
  Ct ( C ( pattern ) )
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.

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

```
e
local K
function K ( style , pattern ) return
```

```
  Lc ( [[ {\PitonStyle{ ]] .. style .. "}{" )
  * Q ( pattern )
  * Lc "}}"
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.

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

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

```
Escape = P ( false )
EscapeClean = P ( false )
if piton.begin_escape then
  Escape =
    P ( piton.begin_escape )
    * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
    * 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).

```
  EscapeClean =
    P ( piton.begin_escape )
    * ( 1 - P ( piton.end_escape ) ) ^ 1
    * P ( piton.end_escape )
end
```

```
EscapeMath = P ( false )
if piton.begin_escape_math then
  EscapeMath =
    P ( piton.begin_escape_math )
    * Lc "$"
    * L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
    * Lc "$"
    * P ( piton.end_escape_math )
end
```

The following line is mandatory.

```
lpeg.locale(lpeg)
```

The basic syntactic LPEG

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

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

```
local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
                    + "ô" + "û" + "ü" + "Â" + "À" + "Ç" + "É" + "È" + "Ê" + "Ë"
                    + "Ï" + "Î" + "Ô" + "Û" + "Ü"

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).

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

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

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

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

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).

```
local Number =
  K ( 'Number.Internal' ,
      ( digit ^ 1 * P "." * # ( 1 - P "." ) * digit ^ 0
        + digit ^ 0 * P "." * digit ^ 1
        + digit ^ 1 )
      * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
      + digit ^ 1
    )
```

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?).

```
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`.

```
if piton.begin_escape then
  lpeg_central = lpeg_central - piton.begin_escape
end
if piton.begin_escape_math then
  lpeg_central = lpeg_central - piton.begin_escape_math
end
local Word = Q ( lpeg_central ^ 1 )
```

```
local Space = Q " " ^ 1

local SkipSpace = Q " " ^ 0

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

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

Remember that `\@@_leading_space:` does *not* create a space, only an incrementation of the counter `\g_@@_indentation_int`.

```
local SpaceIndentation = Lc [[ \@@_leading_space: ]] * Q " "
```

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

The following LPEG catches a space (U+0020) and replace 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 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.

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

Several tools for the construction of the main LPEG

```
local LPEG0 = { }
local LPEG1 = { }
local LPEG2 = { }
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 catching*.

```
local Compute_braces
function Compute_braces ( lpeg_string ) return
  P { "E" ,
       E =
           (
             "{" * V "E" * "}"
             +
             lpeg_string
             +
             ( 1 - S "{}" )
           ) ^ 0
    }
end
```

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

```
local Compute_DetectedCommands
function Compute_DetectedCommands ( lang , braces ) return
  Ct (
       Cc "Open"
        * C ( piton.DetectedCommands * space ^ 0 * P "{" )
        * Cc "}"
     )
   * ( braces
       / ( function ( s )
             if s ~= '' then return
               LPEG1[lang] : match ( s )
             end
           end )
     )
   * P "}"
   * Ct ( Cc "Close" )
end
```

```
local Compute_LPEG_cleaner
function Compute_LPEG_cleaner ( lang , braces ) return
  Ct ( ( piton.DetectedCommands * "{"
          * ( braces
              / ( function ( s )
                    if s ~= '' then return
                      LPEG_cleaner[lang] : match ( s )
                    end
                  end )
            )
          * "}"
         + EscapeClean
         +  C ( P ( 1 ) )
        ) ^ 0 ) / table.concat
end
```

The following function `ParseAgain` will be used in the definitions of the LPEG of the different informatic 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 final user (if he wish to format that element with a uniform font instead of an analyze by `ParseAgain`), we have to use `\@@_piton:n`.

```
local ParseAgain
function ParseAgain ( code )
  if code ~= '' then return
```

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

```
    LPEG1[piton.language] : match ( code )
  end
end
```

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

```
local Beamer = P ( false )
local BeamerBeginEnvironments = P ( true )
local BeamerEndEnvironments = P ( true )

piton.BeamerEnvironments = P ( false )
for _ , x  in ipairs ( piton.beamer_environments )  do
  piton.BeamerEnvironments = piton.BeamerEnvironments + x
end


BeamerBeginEnvironments =
    ( space ^ 0 *
      L
        (
          P [[\begin{]] * piton.BeamerEnvironments * "}"
          * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
        )
      * "\r"
    ) ^ 0


BeamerEndEnvironments =
    ( space ^ 0 *
      L ( P [[\end{]] * piton.BeamerEnvironments * "}" )
      * "\r"
    ) ^ 0
```

The following Lua function will be used to compute the LPEG `Beamer` for each informatic language.

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

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

```
  local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
  lpeg = lpeg +
      Ct ( Cc "Open"
            * C ( piton.BeamerCommands
                  * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
                  * P "{"
                )
            * Cc "}"
         )
       * ( braces /
           ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
       * "}"
       * Ct ( Cc "Close" )
```

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

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

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

```
  lpeg = lpeg +
      L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{" )
      * ( braces
          / ( function ( s )
              if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
      * L ( P "}{" )
      * ( braces
          / ( function ( s )
              if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
      * L ( P "}{" )
      * ( braces
          / ( function ( s )
              if s ~= '' then return LPEG1[lang] : match ( s ) end end ) )
      * L ( P "}" )
```

Now, the environments of Beamer.

```
  for _ , x in ipairs ( piton.beamer_environments ) do
    lpeg = lpeg +
          Ct ( Cc "Open"
               * C (
                      P ( [[\begin{]] .. x .. "}" )
                      * ( "<" * ( 1 - P ">") ^ 0 * ">" ) ^ -1
                    )
               * Cc ( [[\end{]] .. x ..  "}" )
              )
          * (
              ( ( 1 - P ( [[\end{]] .. x .. "}" ) ) ^ 0 )
                  / ( function ( s )
                        if s ~= '' then return
                          LPEG1[lang] : match ( s )
                        end
                      end )
            )
          * P ( [[\end{]] .. x .. "}" )
          * Ct ( Cc "Close" )
  end
```

Now, you can return the value we have computed.

```
  return lpeg
end
```

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

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

EOL The following LPEG will detect the Python prompts when the user is typesetting an interactive session of Python (directly or through `{pyconsole}` of pyluatex). We have to detect that prompt twice. The first detection (called *hasty detection*) will be before the `\@@_begin_line:` because you want to trigger a special background color for that row (and, after the `\@@_begin_line:`, it's too late to change de background).

```
local PromptHastyDetection =
  ( # ( P ">>>" + "..." ) * Lc [[ \@@_prompt: ]] ) ^ -1
```

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

With the following LPEG, a style will actually be applied to the prompt (for instance, it's possible to decide to discard these prompts).

```
local Prompt = K ( 'Prompt' , ( ( P ">>>" + "..." ) * P " " ^ -1 ) ) ^ -1
```

The following LPEG `EOL` is for the end of lines.

```
local EOL =
  P "\r"
  *
  (
    space ^ 0 * -1
    +
```

We recall that each line of the informatic code we have to parse will be sent back to LaTeX between a pair `\@@_begin_line:` – `\@@_end_line:`[35].

```
    Ct (
         Cc "EOL"
         *
         Ct ( Lc [[ \@@_end_line: ]]
              * BeamerEndEnvironments
              *
                (
```

If the last line of the listing is the end of an environment of Beamer (eg. `\end{uncoverenv}`), then, we don't open a new line. A token `\@@_end_line:` will be added at the end of the environment but it will be no-op since we have defined the macro `\@@_end_line:` to be no-op (even though it is also used as a marker for the TeX delimited macro `\@@_begin_line:`).

```
                  -1
                +
                  BeamerBeginEnvironments
                * PromptHastyDetection
                * Lc [[ \@@_newline:\@@_begin_line: ]]
                * Prompt
              )
           )
       )
  )
  * ( SpaceIndentation ^ 0 * # ( 1 - S " \r" ) ) ^ -1
```

The following LPEG `CommentLaTeX` is for what is called in that document the "LaTeX comments". Since the elements that will be caught must be sent to LaTeX with standard LaTeX catcodes, we put the capture (done by the function `C`) in a table (by using `Ct`, which is an alias for `lpeg.Ct`).

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

[35]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:`

10.3.2 The language Python

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

```
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.

```
  local Operator =
    K ( 'Operator' ,
        P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":=" + "//" + "**"
        + S "-~+/*%=<>&.@|" )

  local OperatorWord =
    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`.

```
  local For = K ( 'Keyword' , P "for" )
              * Space
              * Identifier
              * Space
              * K ( 'Keyword' , P "in" )

  local Keyword =
    K ( 'Keyword' ,
        P "as" + "assert" + "break" + "case" + "class" + "continue" + "def" +
        "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
        "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
        "try" + "while" + "with" + "yield" + "yield from" )
    + K ( 'Keyword.Constant' , P "True" + "False" + "None" )

  local Builtin =
    K ( 'Name.Builtin' ,
        P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
        "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
        "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
        "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
        "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
        "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
        + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
        "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
        "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
        "vars" + "zip" )

  local Exception =
    K ( 'Exception' ,
        P "ArithmeticError" + "AssertionError" + "AttributeError" +
        "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
        "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
        "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
        "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
        "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
        "NotImplementedError" + "OSError" + "OverflowError" +
        "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
        "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
        "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
        + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
        "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
        "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
        "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
        "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
        "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
        "FileNotFoundError" + "InterruptedError" + "IsADirectoryError" +
```

```
        "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
        "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
        "RecursionError" )

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.

```
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:`

```
local DefClass =
  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`

```
local ImportAs =
  K ( 'Keyword' , "import" )
   * Space
   * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
   * (
       ( Space * K ( 'Keyword' , "as" ) * Space
          * K ( 'Name.Namespace' , identifier ) )
       +
       ( SkipSpace * Q "," * SkipSpace
          * K ( 'Name.Namespace' , identifier ) ) ^ 0
     )
```

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`

```
local FromImport =
  K ( 'Keyword' , "from" )
    * Space * K ( 'Name.Namespace' , identifier )
    * 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[36] in that interpolation:
`\piton{f'Total price: {total+1:.2f} €'}`

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

```
local PercentInterpol =
  K ( 'String.Interpol' ,
      P "%"
      * ( "(" * alphanum ^ 1 * ")" ) ^ -1
      * ( S "-#0 +" ) ^ 0
      * ( digit ^ 1 + "*" ) ^ -1
      * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
      * ( S "HlL" ) ^ -1
      * S "sdfFeExXorgiGauc%"
    )
```

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.[37]

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

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

```
         Q ( P "f'" + "F'" )
         * (
             K ( 'String.Interpol' , "{" )
              * K ( 'Interpol.Inside' , ( 1 - S "}':" ) ^ 0  )
              * Q ( P ":" * ( 1 - S "}:'" ) ^ 0 ) ^ -1
              * K ( 'String.Interpol' , "}" )
             +
             SpaceInString
             +
             Q ( ( P "\\'" + "\\\\" + "{{" + "}}" + 1 - S " {}'" ) ^ 1 )
           ) ^ 0
         * Q "'"
       +
```

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

```
         Q ( P "'" + "r'" + "R'" )
         * ( Q ( ( P "\\'" + "\\\\" + 1 - S " '\r%" ) ^ 1 )
             + SpaceInString
             + PercentInterpol
             + Q "%"
           ) ^ 0
         * Q "'" )
```

[36]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`.
[37]The interpolations are formatted with the piton style `Interpol.Inside`. The initial value of that style is `\@@_piton:n` which means that the interpolations are parsed once again by piton.

```
local DoubleShortString =
  WithStyle ( 'String.Short.Internal' ,
        Q ( P "f\"" + "F\"" )
        * (
            K ( 'String.Interpol' , "{" )
              * K ( 'Interpol.Inside' , ( 1 - S "}\":" ) ^ 0 )
              * ( K ( 'String.Interpol' , ":" ) * Q ( (1 - S "}:\"") ^ 0 ) ) ^ -1
              * K ( 'String.Interpol' , "}" )
            +
            SpaceInString
            +
            Q ( ( P "\\\"" + "\\\\" + "{{" + "}}" + 1 - S " {}\"" ) ^ 1 )
           ) ^ 0
        * Q "\""
      +
        Q ( P "\"" + "r\"" + "R\"" )
        * ( Q ( ( P "\\\"" + "\\\\" + 1 - S " \"\r%" ) ^ 1 )
            + SpaceInString
            + PercentInterpol
            + Q "%"
          ) ^ 0
        * Q "\""  )

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.

```
local braces =
  Compute_braces
   (
       ( P "\"" + "r\"" + "R\"" + "f\"" + "F\"" )
           * ( P "\\\"" + 1 - S "\"" ) ^ 0 * "\""
     +
       ( P '\'' + 'r\'' + 'R\'' + 'f\'' + 'F\'' )
           * ( P '\\\'' + 1 - S '\'' ) ^ 0 * '\''
   )
if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end
```

Detected commands

```
DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
```

LPEG_cleaner

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

The long strings

```
local SingleLongString =
  WithStyle ( 'String.Long.Internal' ,
     ( Q ( S "fF" * P "'''" )
        * (
            K ( 'String.Interpol' , "{" )
              * K ( 'Interpol.Inside' , ( 1 - S "}:\r" - "'''" ) ^ 0  )
              * Q ( P ":" * (1 - S "}:\r" - "'''" ) ^ 0 ) ^ -1
              * K ( 'String.Interpol' , "}"  )
            +
            Q ( ( 1 - P "'''" - S "{}'\r" ) ^ 1 )
```

```
             +
             EOL
           ) ^ 0
       +
         Q ( ( S "rR" ) ^ -1  * "'''" )
         * (
             Q ( ( 1 - P "'''" - S "\r%" ) ^ 1 )
             +
             PercentInterpol
             +
             P "%"
             +
             EOL
           ) ^ 0
      )
      * Q "'''"  )
local DoubleLongString =
  WithStyle ( 'String.Long.Internal' ,
     (
        Q ( S "fF" * "\"\"\"" )
        * (
            K ( 'String.Interpol', "{"  )
              * K ( 'Interpol.Inside' , ( 1 - S "}:\r" - "\"\"\"" ) ^ 0 )
              * Q ( ":" * (1 - S "}:\r" - "\"\"\"" ) ^ 0 ) ^ -1
              * K ( 'String.Interpol' , "}"  )
            +
            Q ( ( 1 - S "{}\"\r" - "\"\"\"" ) ^ 1 )
            +
            EOL
          ) ^ 0
      +
        Q ( S "rR" ^ -1  * "\"\"\"" )
        * (
            Q ( ( 1 - P "\"\"\"" - S "%\r" ) ^ 1 )
            +
            PercentInterpol
            +
            P "%"
            +
            EOL
          ) ^ 0
     )
     * Q "\"\"\""
  )
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`).

```
local StringDoc =
    K ( 'String.Doc' , P "r" ^ -1 * "\"\"\"" )
      * ( K ( 'String.Doc' , (1 - P "\"\"\"" - "\r" ) ^ 0  ) * EOL
          * Tab ^ 0
        ) ^ 0
      * K ( 'String.Doc' , ( 1 - P "\"\"\"" - "\r" ) ^ 0 * "\"\"\"" )
```

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

```
local Comment =
  WithStyle
   ( 'Comment' ,
```

```
      Q "#" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0  -- $
    )
  * ( 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).

```
local expression =
  P { "E" ,
      E = ( "'" * ( P "\\'" + 1 - S "'\r" ) ^ 0 * "'"
            + "\"" * ( P "\\\"" + 1 - S "\"\r" ) ^ 0 * "\""
            + "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + "[" * V "F" * "]"
            + ( 1 - S "{}()[]\r," ) ) ^ 0 ,
      F = (   "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + "[" * V "F" * "]"
            + ( 1 - S "{}()[]\r\"'" ) ) ^ 0
    }
```

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`.

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

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`...

```
local DefFunction =
  K ( 'Keyword' , "def" )
  * Space
  * K ( 'Name.Function.Internal' , identifier )
  * SkipSpace
  * Q "("  * Params * Q ")"
  * SkipSpace
  * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
  * ( C ( ( 1 - S ":\r" ) ^ 0 ) / ParseAgain )
  * Q ":"
  * ( SkipSpace
      * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
      * Tab ^ 0
      * SkipSpace
      * StringDoc ^ 0 -- there may be additional docstrings
    ) ^ -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 final user wants to speak of the keyword `def`).

Miscellaneous

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

The main LPEG for the language Python

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

First, the main loop :

```
local Main =
     space ^ 0 * EOL -- faut-il le mettre en commentaire ?
     + Space
     + Tab
     + Escape + EscapeMath
     + CommentLaTeX
     + Beamer
     + DetectedCommands
     + LongString
     + Comment
     + ExceptionInConsole
     + Delim
     + Operator
     + OperatorWord * EndKeyword
     + ShortString
     + Punct
     + FromImport
     + RaiseException
     + DefFunction
     + DefClass
     + For
     + Keyword * EndKeyword
     + Decorator
     + Builtin * EndKeyword
     + Identifier
     + Number
     + Word
```

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

```
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:`[38].

```
LPEG2.python =
  Ct (
       ( space ^ 0 * "\r" ) ^ -1
       * BeamerBeginEnvironments
       * PromptHastyDetection
       * Lc [[ \@@_begin_line: ]]
       * Prompt
       * SpaceIndentation ^ 0
       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
```

[38] 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:`

```
          * -1
          * Lc [[ \@@_end_line: ]]
        )
```

End of the Lua scope for the language Python.

```
end
```

10.3.3 The language Ocaml

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

```
do
```

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

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

```
  if piton.beamer then
    Beamer = Compute_Beamer ( 'ocaml' , braces )
  end
  DetectedCommands = Compute_DetectedCommands ( 'ocaml' , braces )
  local Q
  function Q ( pattern ) return
    Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
    + Beamer + DetectedCommands + EscapeMath + Escape
  end
```

```
  local K
  function K ( style , pattern ) return
    Lc ( [[ {\PitonStyle{ ]] .. style  .. "}{" )
    * Q ( pattern )
    * Lc "}}"
  end
```

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

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

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

The strings of OCaml

```
local ocaml_string =
  P "\""
* (
    P " "
    +
    P ( ( 1 - S " \"\r" ) ^ 1 )
    +
    EOL -- ?
  ) ^ 0
* P "\""
```

```
local String =
  WithStyle
    ( 'String.Long.Internal' ,
        Q "\""
      * (
          SpaceInString
          +
          Q ( ( 1 - S " \"\r" ) ^ 1 )
          +
          EOL
        ) ^ 0
      * Q "\""
    )
```

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`.

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

The LPEG `QuotedStringBis` will do the second analysis.

```
local QuotedStringBis =
  WithStyle ( 'String.Long.Internal' ,
      (
        Space
        +
        Q ( ( 1 - S " \r" ) ^ 1 )
        +
        EOL
      ) ^ 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.

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

```
local Comment =
  WithStyle ( 'Comment' ,
```

```
      P {
          "A" ,
          A = Q "(*"
              * ( V "A"
                  + Q ( ( 1 - S "\r$\"" - "(*" - "*)" ) ^ 1 ) -- $
                  + ocaml_string
                  + "$" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * "$" -- $
                  + EOL
                ) ^ 0
              * Q "*)"
        }   )
```

Some standard LPEG

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

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


local Constructor =
  K ( 'Name.Constructor' ,
      Q "`" ^ -1 * cap_identifier
```

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

```
      + Q "::"
      + Q "[" * SkipSpace * Q "]" )


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


local OperatorWord =
  K ( 'Operator.Word' ,
      P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" )
```

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

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


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


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

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

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

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

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

```
local ocaml_char =
    P "'" *
    (
      ( 1 - S "'\\" )
      + "\\"
        * ( S "\\'ntbr \""
            + digit * digit * digit
            + P "x" * ( digit + R "af" + R "AF" )
                    * ( digit + R "af" + R "AF" )
                    * ( digit + R "af" + R "AF" )
            + P "o" * R "03" * R "07" * R "07" )
    )
    * "'"
local Char =
  K ( 'String.Short.Internal', ocaml_char )
```

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

```
local TypeParameter =
  K ( 'TypeParameter' ,
      "'" * Q"_" ^ -1 * alpha ^ 1 * ( # ( 1 - P "'" ) + -1 ) )
```

The records

```
local expression_for_fields_type =
  P { "E" ,
      E =  (  "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + TypeParameter
            + ( 1 - S "{}()[]\r;" ) ) ^ 0 ,
      F = (    "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + ( 1 - S "{}()[]\r\"'" ) + TypeParameter ) ^ 0
    }


local expression_for_fields_value =
  P { "E" ,
      E =  (    "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + "[" * V "F" * "]"
            + ocaml_string + ocaml_char
            + ( 1 - S "{}()[];" ) ) ^ 0 ,
      F = (    "{" * V "F" * "}"
            + "(" * V "F" * ")"
            + "[" * V "F" * "]"
            + ocaml_string + ocaml_char
            + ( 1 - S "{}()[]\"'" )) ^ 0
    }


local OneFieldDefinition =
      ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
    * K ( 'Name.Field' , identifier ) * SkipSpace
    * Q ":" * SkipSpace
    * K ( 'TypeExpression' , expression_for_fields_type )
    * SkipSpace
```

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

Don't forget the parentheses!

```
  * ( C ( expression_for_fields_value ) / ParseAgain )
  * SkipSpace
```

The *records* may occur in the definitions of type (beginning by `type`) but also when used as values.

```
local Record =
  Q "{" * SkipSpace
  *
    (
      OneFieldDefinition
      * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0
      +
      OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
    )
  * SkipSpace
  * Q ";" ^ -1
  * SkipSpace
  * Comment ^ -1
  * SkipSpace
  * Q "}"
```

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.

```
local DotNotation =
  (
      K ( 'Name.Module' , cap_identifier )
        * Q "."
        * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
      +
       Identifier
        * Q "."
        * K ( 'Name.Field' , identifier )
  )
  * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0


local Operator =
  K ( 'Operator' ,
      P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":=" + "||" + "&&" +
      "//" + "**" + ";;" + "->" + "+." + "-." + "*." + "/."
      + S "-~+/*%=<>&@|" )


local Builtin =
  K ( 'Name.Builtin' , P "not" + "incr" + "decr" + "fst" + "snd" + "ref" )


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


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).

```
local pattern_part =
  ( 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).

```
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.

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

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

```
  (
      K ( 'Identifier.Internal' , identifier )
    +
      Q "(" * SkipSpace
      * ( C ( pattern_part ) / ParseAgain )
      * SkipSpace
```

Of course, the specification of type is optional.

```
      * ( Q ":" * K ( 'TypeExpression' , balanced_parens ) * SkipSpace ) ^ -1
      * Q ")"
  )
```

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

```
local DefFunction =
  K ( 'Keyword.Governing' , "let open" )
  * Space
  * K ( 'Name.Module' , cap_identifier )
  +
  K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
    * Space
    * K ( 'Name.Function.Internal' , identifier )
    * Space
    * (
        Q "=" * SkipSpace * K ( 'Keyword' , "function" )
        +
        Argument * ( SkipSpace * Argument ) ^ 0
        * (
            SkipSpace
            * Q ":"
            * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
          ) ^ -1
      )
```

DefModule

```
local DefModule =
  K ( 'Keyword.Governing' , "module" ) * Space
  *
    (
          K ( 'Keyword.Governing' , "type" ) * Space
        * K ( 'Name.Type' , cap_identifier )
      +
        K ( 'Name.Module' , cap_identifier ) * SkipSpace
        *
          (
```

```
      Q "(" * SkipSpace
       * K ( 'Name.Module' , cap_identifier ) * SkipSpace
       * Q ":" * SkipSpace
       * K ( 'Name.Type' , cap_identifier ) * SkipSpace
       *
        (
         Q "," * SkipSpace
          * K ( 'Name.Module' , cap_identifier ) * SkipSpace
          * Q ":" * SkipSpace
          * K ( 'Name.Type' , cap_identifier ) * SkipSpace
        ) ^ 0
       * Q ")"
     ) ^ -1
    *
     (
      Q "=" * SkipSpace
      * K ( 'Name.Module' , cap_identifier )  * SkipSpace
      * Q "("
      * K ( 'Name.Module' , cap_identifier ) * SkipSpace
       *
       (
        Q ","
        *
        K ( 'Name.Module' , cap_identifier ) * SkipSpace
       ) ^ 0
      * Q ")"
     ) ^ -1
  )
 +
 K ( 'Keyword.Governing' , P "include" + "open" )
 * Space
 * K ( 'Name.Module' , cap_identifier )
```

DefType

```
local DefType =
 K ( 'Keyword.Governing' , "type" )
 * Space
 * K ( 'TypeExpression' , Q ( 1 - P "=" ) ^ 1 )
 * SkipSpace
 * ( Q "+=" + Q "=" )
 * SkipSpace
 * (
   Record
   +
   WithStyle
    (
     'TypeExpression' ,
     (
      ( EOL + Q ( 1 - P ";;" - governing_keyword ) ) ^ 0
      * ( # ( governing_keyword ) + Q ";;" )
     )
    )
  )
```

The main LPEG for the language OCaml

```
local Main =
  space ^ 0 * EOL
  + Space
  + Tab
  + Escape + EscapeMath
  + Beamer
```

```
      + DetectedCommands
      + TypeParameter
      + String + QuotedString + Char
      + Comment
      + Operator
```

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

```
      + Q "~" * Identifier * ( Q ":" ) ^ -1
      + Q ":" * # (1 - P ":") * SkipSpace
          * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
      + Exception
      + DefType
      + DefFunction
      + DefModule
      + Record
      + Keyword * EndKeyword
      + OperatorWord * EndKeyword
      + Builtin * EndKeyword
      + DotNotation
      + Constructor
      + Identifier
      + Punct
      + Delim
      + Number
      + Word
```

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

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

```
  LPEG2.ocaml =
    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).

```
        ( P ":" + Identifier * SkipSpace * Q ":" ) * # ( 1 - P ":" )
          * SkipSpace
          * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 )
        +
        ( space ^ 0 * "\r" ) ^ -1
        * BeamerBeginEnvironments
        * Lc [[ \@@_begin_line: ]]
        * SpaceIndentation ^ 0
        * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
              + space ^ 0 * EOL
              + Main
          ) ^ 0
        * -1
        * Lc [[ \@@_end_line: ]]
      )
```

End of the Lua scope for the language OCaml.

```
end
```

10.3.4 The language C

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

```
do
```

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

```
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.

```
local identifier = letter * alphanum ^ 0

local Operator =
  K ( 'Operator' ,
      P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
        + S "-~+/*%=<>&.@|!" )

local Keyword =
  K ( 'Keyword' ,
      P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
      "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
      "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
      "register" + "restricted" + "return" + "static" + "static_assert" +
      "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
      "union" + "using" + "virtual" + "volatile" + "while"
    )
  + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )

local Builtin =
  K ( 'Name.Builtin' ,
      P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )

local Type =
  K ( 'Name.Type' ,
      P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" + "int" +
      "int8_t" + "int16_t" + "int32_t" + "int64_t" + "long" + "short" + "signed"
      + "unsigned" + "void" + "wchar_t" ) * Q "*" ^ 0

local DefFunction =
  Type
  * Space
  * Q "*" ^ -1
  * K ( 'Name.Function.Internal' , identifier )
  * SkipSpace
  * # 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:`

```
local DefClass =
  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 strings of C

```
String =
  WithStyle ( 'String.Long.Internal' ,
      Q "\""
      * ( SpaceInString
          + K ( 'String.Interpol' ,
                "%" * ( S "difcspxXou" + "ld" + "li" + "hd" + "hi" )
              )
          + Q ( ( P "\\\"" + 1 - S " \"" ) ^ 1 )
```

```
          ) ^ 0
        * Q "\""
      )
```

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

```
local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
DetectedCommands = Compute_DetectedCommands ( 'c' , braces )
LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )
```

The directives of the preprocessor

```
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.

```
local Comment =
  WithStyle ( 'Comment' ,
     Q "//" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
            * ( EOL + -1 )

local LongComment =
  WithStyle ( 'Comment' ,
               Q "/*"
               * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
               * Q "*/"
            ) -- $
```

The main LPEG for the language C

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

First, the main loop :

```
local Main =
     space ^ 0 * EOL
     + Space
     + Tab
     + Escape + EscapeMath
     + CommentLaTeX
     + Beamer
     + DetectedCommands
     + Preproc
     + Comment + LongComment
     + Delim
     + Operator
     + String
     + Punct
     + DefFunction
     + DefClass
     + Type * ( Q "*" ^ -1 + EndKeyword )
     + Keyword * EndKeyword
     + Builtin * EndKeyword
     + Identifier
     + Number
     + Word
```

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

```
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:`[39].

```
LPEG2.c =
  Ct (
       ( space ^ 0 * P "\r" ) ^ -1
       * BeamerBeginEnvironments
       * Lc [[ \@@_begin_line: ]]
       * SpaceIndentation ^ 0
       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
       * -1
       * Lc [[ \@@_end_line: ]]
     )
```

End of the Lua scope for the language C.

```
end
```

10.3.5 The language SQL

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

```
do
```

```
local LuaKeyword
function LuaKeyword ( name ) return
  Lc [[ {\PitonStyle{Keyword}{ ]]
  * Q ( Cmt (
              C ( letter * alphanum ^ 0 ) ,
              function ( s , i , a ) return string.upper ( a ) == name end
            )
      )
  * Lc "}}"
end
```

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

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

```
local Operator =
  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).

```
local Set
function Set ( list )
  local set = { }
  for _ , l in ipairs ( list ) do set[l] = true end
  return set
end
```

[39]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:`

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

```
local set_keywords = Set
 {
   "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
   "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
   "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
   "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
   "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
   "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
   "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
   "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
   "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
   "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
   "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
   "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
   "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
   "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
   "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
   "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
   "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
   "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
   "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
   "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
 }
```

```
local set_builtins = Set
 {
   "AVG" , "COUNT" , "CHAR_LENGTH" , "CONCAT" , "CURDATE" , "CURRENT_DATE" ,
   "DATE_FORMAT" , "DAY" , "LOWER" , "LTRIM" , "MAX" , "MIN" , "MONTH" , "NOW" ,
   "RANK" , "ROUND" , "RTRIM" , "SUBSTRING" , "SUM" , "UPPER" , "YEAR"
 }
```

The LPEG `Identifier` will catch the identifiers of the fields but also the keywords and the built-in functions of SQL. If will *not* catch the names of the SQL tables.

```
local Identifier =
  C ( identifier ) /
  (
    function ( s )
        if set_keywords[string.upper(s)] then return
```

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

```
          { [[{\PitonStyle{Keyword}{]] } ,
          { luatexbase.catcodetables.other , s } ,
          { "}}" }
        else
          if set_builtins[string.upper(s)] then return
            { [[{\PitonStyle{Name.Builtin}{]] } ,
            { luatexbase.catcodetables.other , s } ,
            { "}}" }
          else return
            { [[{\PitonStyle{Name.Field}{]] } ,
            { luatexbase.catcodetables.other , s } ,
            { "}}" }
          end
        end
    end
  )
```

The strings of SQL

```
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.

```
local braces = Compute_braces ( "'" * ( 1 - P "'" ) ^ 1 * "'" )
if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
DetectedCommands = Compute_DetectedCommands ( 'sql' , braces )
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.

```
local Comment =
  WithStyle ( 'Comment' ,
     Q "--"   -- syntax of SQL92
     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
  * ( EOL + -1 )

local LongComment =
  WithStyle ( 'Comment' ,
               Q "/*"
               * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
               * Q "*/"
            ) -- $
```

The main LPEG for the language SQL

```
local EndKeyword
  = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
    EscapeMath + -1
local TableField =
       K ( 'Name.Table' , identifier )
     * Q "."
     * K ( 'Name.Field' , identifier )

local OneField =
  (
    Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
    +
        K ( 'Name.Table' , identifier )
      * Q "."
      * K ( 'Name.Field' , identifier )
    +
    K ( 'Name.Field' , identifier )
  )
  * (
      Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
    ) ^ -1
  * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1

local OneTable =
     K ( 'Name.Table' , identifier )
   * (
       Space
       * LuaKeyword "AS"
       * Space
       * K ( 'Name.Table' , identifier )
     ) ^ -1

local WeCatchTableNames =
     LuaKeyword "FROM"
   * ( Space + EOL )
```

```
   * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
  + (
      LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
      + LuaKeyword "TABLE"
    )
    * ( Space + EOL ) * OneTable
```

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

First, the main loop :

```
local Main =
     space ^ 0 * EOL
     + Space
     + Tab
     + Escape + EscapeMath
     + CommentLaTeX
     + Beamer
     + DetectedCommands
     + Comment + LongComment
     + Delim
     + Operator
     + String
     + Punct
     + WeCatchTableNames
     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
     + Number
     + Word
```

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

```
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:`[40].

```
LPEG2.sql =
  Ct (
       ( space ^ 0 * "\r" ) ^ -1
       * BeamerBeginEnvironments
       * Lc [[ \@@_begin_line: ]]
       * SpaceIndentation ^ 0
       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
       * -1
       * Lc [[ \@@_end_line: ]]
     )
```

End of the Lua scope for the language SQL.

```
end
```

10.3.6 The language "Minimal"

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

```
do
```

```
  local Punct = Q ( S ",:;!\\" )

  local Comment =
    WithStyle ( 'Comment' ,
                Q "#"
```

[40]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:`

```
                  * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
                )
        * ( EOL + -1 )

  local String =
    WithStyle ( 'String.Short.Internal' ,
                Q "\""
                * ( SpaceInString
                    + Q ( ( P "\\\"" + 1 - S " \"" ) ^ 1 )
                  ) ^ 0
                * Q "\""
              )
```

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

```
  local braces = Compute_braces ( P "\"" * ( P "\\\"" + 1 - P "\"" ) ^ 1 * "\"" )

  if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end

  DetectedCommands = Compute_DetectedCommands ( 'minimal' , braces )

  LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )

  local identifier = letter * alphanum ^ 0

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

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

  local Main =
       space ^ 0 * EOL
       + Space
       + Tab
       + Escape + EscapeMath
       + CommentLaTeX
       + Beamer
       + DetectedCommands
       + Comment
       + Delim
       + String
       + Punct
       + Identifier
       + Number
       + Word
```

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

```
  LPEG1.minimal = Main ^ 0

  LPEG2.minimal =
    Ct (
         ( space ^ 0 * "\r" ) ^ -1
         * BeamerBeginEnvironments
         * Lc [[ \@@_begin_line: ]]
         * SpaceIndentation ^ 0
         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
         * -1
         * Lc [[ \@@_end_line: ]]
       )
```

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

```
end
```

10.3.7 The language “Verbatim”

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

```
do
```

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 langage “Verbatim”).

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

  if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end

  DetectedCommands = Compute_DetectedCommands ( 'verbatim' , braces )

  LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )
```

Now, you will construct the LPEG Word.

```
  local lpeg_central = 1 - S " \\\r"
  if piton.begin_escape then
    lpeg_central = lpeg_central - piton.begin_escape
  end
  if piton.begin_escape_math then
    lpeg_central = lpeg_central - piton.begin_escape_math
  end
  local Word = Q ( lpeg_central ^ 1 )

  local Main =
       space ^ 0 * EOL
       + Space
       + Tab
       + Escape + EscapeMath
       + Beamer
       + DetectedCommands
       + Q [[\]]
       + Word
```

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

```
  LPEG1.verbatim = Main ^ 0

  LPEG2.verbatim =
    Ct (
         ( space ^ 0 * "\r" ) ^ -1
         * BeamerBeginEnvironments
         * Lc [[ \@@_begin_line: ]]
         * SpaceIndentation ^ 0
         * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
         * -1
         * Lc [[ \@@_end_line: ]]
       )
```

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

```
end
```

10.3.8 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.

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

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

```
  piton.language = language
  local t = LPEG2[language] : match ( code )
  if t == nil then
    sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
    return -- to exit in force the function
  end
  local left_stack = {}
  local right_stack = {}
  for _ , one_item in ipairs ( t ) do
    if one_item[1] == "EOL" then
      for _ , s in ipairs ( right_stack ) do
        tex.sprint ( s )
      end
      for _ , s in ipairs ( one_item[2] ) do
        tex.tprint ( s )
      end
      for _ , s in ipairs ( left_stack ) do
        tex.sprint ( s )
      end
    else
```

Here is an example of an item beginning with `"Open"`.
`{ "Open" , "\begin{uncover}<2>" , "\end{uncover}" }`
In order to deal with the ends of lines, we have to close the environment (`{uncover}` 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{uncover}<2>` and `right_stack` will be for the elements like `\end{uncover}`.

```
      if one_item[1] == "Open" then
        tex.sprint( one_item[2] )
        table.insert ( left_stack , one_item[2] )
        table.insert ( right_stack , one_item[3] )
      else
        if one_item[1] == "Close" then
          tex.sprint ( right_stack[#right_stack] )
          left_stack[#left_stack] = nil
          right_stack[#right_stack] = nil
        else
          tex.tprint ( one_item )
        end
      end
    end
  end
end
```

The function `ParseFile` will be used by the LaTeX command `\PitonInputFile`. That function merely reads the file (between `first_line` and `last_line`) and then apply the function `Parse` to the resulting Lua string.

```
function piton.ParseFile
  ( lang , name , first_line , last_line , splittable , split )
  local s = ''
  local i = 0
```

At the date of septembre 2024, LuaLaTeX uses Lua 5.3 and not 5.4. In the version 5.4, `io.lines` returns four values (and not just one) but the following code should be correct.

```
  for line in io.lines ( name ) do
    i = i + 1
    if i >= first_line then
      s = s .. '\r' .. line
    end
    if i >= last_line then break end
  end
```

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

```
  if string.byte ( s , 1 ) == 13 then
    if string.byte ( s , 2 ) == 239 then
      if string.byte ( s , 3 ) == 187 then
        if string.byte ( s , 4 ) == 191 then
          s = string.sub ( s , 5 , -1 )
        end
      end
    end
  end
  if split == 1 then
    piton.RetrieveGobbleSplitParse ( lang , 0 , splittable , s )
  else
    piton.RetrieveGobbleParse ( lang , 0 , splittable , s )
  end
end
```

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

10.3.9 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 `#`.

```
function piton.ParseBis ( lang , code )
  local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
  return piton.Parse ( lang , s )
end
```

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 syntaxic element. In that case, you have to remove the potential `\@@_breakable_space:` that have been inserted when the key `break-lines` is in force.

```
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:`.

```
  local s
  s = ( Cs ( ( P [[\@@_breakable_space: ]] / ' ' + 1 ) ^ 0 ) )
      : match ( code )
```

Remember that `\@@_leading_space:` does not create a space, only an incrementation of the counter `\g_@@_indentation_int`. That's why we don't replace it by a space...

```
  s = ( Cs ( ( P [[\@@_leading_space: ]] / '' + 1 ) ^ 0 ) )
      : match ( s )
  return piton.Parse ( lang , s )
end
```

10.3.10 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.

```
local AutoGobbleLPEG =
      (  (
           P " " ^ 0 * "\r"
           +
           Ct ( C " " ^ 0 ) / table.getn
           * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
         ) ^ 0
         * ( Ct ( C " " ^ 0 ) / table.getn
              * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
       ) / math.min
```

The following LPEG is similar but works with the tabulations.

```
local TabsAutoGobbleLPEG =
       (
         (
           P "\t" ^ 0 * "\r"
           +
           Ct ( C "\t" ^ 0 ) / table.getn
           * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
         ) ^ 0
         * ( Ct ( C "\t" ^ 0 ) / table.getn
             * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
       ) / math.min
```

The following LPEG returns as capture the number of spaces at the last line, that is to say before the `\end{Piton}` (and usually it's also the number of spaces before the corresponding `\begin{Piton}` because that's the traditional way to indent in LaTeX).

```
local EnvGobbleLPEG =
      ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
    * Ct ( C " " ^ 0 * -1 ) / table.getn
```

```
local remove_before_cr
function remove_before_cr ( input_string )
  local match_result = ( P "\r" ) : match ( input_string )
  if match_result then return
    string.sub ( input_string , match_result )
  else return
    input_string
  end
end
```

The function `gobble` gobbles n characters on the left of the code. The negative values of n have special significations.

```
local gobble
function gobble ( n , code )
  code = remove_before_cr ( code )
  if n == 0 then return
    code
  else
    if n == -1 then
      n = AutoGobbleLPEG : match ( code )
    else
      if n == -2 then
        n = EnvGobbleLPEG : match ( code )
```

```
      else
        if n == -3 then
          n = TabsAutoGobbleLPEG : match ( code )
        end
      end
    end
```

We have a second test `if n == 0` because the, even if the key like `auto-gobble` is in force, it's possible that, in fact, there is no space to gobble...

```
    if n == 0 then return
      code
    else return
```

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

```
      ( Ct (
              ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
                * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
            ) ^ 0 )
        / table.concat
      ) : match ( code )
    end
  end
end
```

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

```
function piton.GobbleParse ( lang , n , splittable , code )
  piton.ComputeLinesStatus ( code , splittable )
  piton.last_code = gobble ( n , code )
  piton.last_language = lang
```

We count the number of lines of the informatic code. The result will be stored by Lua in `\l_@@_nb_lines_int`.

```
  piton.CountLines ( piton.last_code )
  sprintL3 [[ \bool_if:NT \g_@@_footnote_bool \savenotes ]]
  piton.Parse ( lang , piton.last_code )

  sprintL3 [[ \vspace{2.5pt} ]]
  sprintL3 [[ \bool_if:NT \g_@@_footnote_bool \endsavenotes ]]
```

We finish the paragraph (each line of the listing is composed in a TeX box — with potentially several lines when `break-lines-in-Piton` is in force — put alone in a paragraph.

```
  sprintL3 [[ \par ]]
```

Now, if the final user has used the key `write` to write the code of the environment on an external file.

```
  if piton.write and piton.write ~= '' then
    local file = io.open ( piton.write , piton.write_mode )
    if file then
      file : write ( piton.get_last_code ( ) )
      file : close ( )
    else
      sprintL3 [[ \@@_error_or_warning:n { FileError } ]]
    end
  end
end
```

The following function will be used when the key `split-on-empty-lines` is in force. With that key, the informatic code is split in chunks at the empty lines (usually between the informatic functions defined in the informatic 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).

```
function piton.GobbleSplitParse ( lang , n , splittable , code )
  local chunks
```

```
  chunks =
     (
       Ct (
            (
              P " " ^ 0 * "\r"
              +
              C ( ( ( 1 - P "\r" ) ^ 1 * "\r" - ( P " " ^ 0 * "\r" ) ) ^ 1 )
            ) ^ 0
          )
     ) : match ( gobble ( n , code ) )
  sprintL3 [[ \begingroup ]]
  sprintL3
    (
      [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, ]]
      .. "language = " .. lang .. ","
      .. "splittable = " .. splittable .. "}"
    )
  for k , v in pairs ( chunks ) do
    if k > 1 then
      sprintL3 [[ \l_@@_split_separation_tl ]]
    end
    tex.sprint
      (
        [[\begin{]] .. piton.env_used_by_split .. "}\r"
        .. v
        .. [[\end{]] .. piton.env_used_by_split .. "}"
      )
  end
  sprintL3 [[ \endgroup ]]
end
```

```
function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
  local s
  s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
  piton.GobbleSplitParse ( lang , n , splittable , s )
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 legibily. 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.

```
piton.string_between_chunks =
 [[ \par \l_@@_split_separation_tl \mode_leave_vertical: ]]
 .. [[ \int_gzero:N \g_@@_line_int ]]
```

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.

```
function piton.get_last_code ( )
  return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
end
```

10.3.11 To count the number of lines

```
function piton.CountLines ( code )
  local count = 0
  count =
     ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
            * ( ( 1 - P "\r" ) ^ 1 * Cc "\r" ) ^ -1
            * -1
```

```
      ) / table.getn
   ) : match ( code )
 sprintL3 ( string.format ( [[ \int_set:Nn  \l_@@_nb_lines_int { %i } ]] , count ) )
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).

```
function piton.CountNonEmptyLines ( code )
  local count = 0
  count =
     ( Ct ( ( P " " ^ 0 * "\r"
              + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
            * ( 1 - P "\r" ) ^ 0
            * -1
          ) / table.getn
     ) : match ( code )
  sprintL3
   ( string.format ( [[ \int_set:Nn  \l_@@_nb_non_empty_lines_int { %i } ]] , count ) )
end
```

```
function piton.CountLinesFile ( name )
  local count = 0
  for line in io.lines ( name ) do count = count + 1 end
  sprintL3
   ( string.format ( [[ \int_set:Nn \l_@@_nb_lines_int { %i } ]], count ) )
end
```

```
function piton.CountNonEmptyLinesFile ( name )
  local count = 0
  for line in io.lines ( name ) do
    if not ( ( P " " ^ 0 * -1 ) : match ( line ) ) then
       count = count + 1
    end
  end
  sprintL3
   ( string.format ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { % i } ]] , count ) )
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`.

```
function piton.ComputeRange(marker_beginning,marker_end,file_name)
  local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker_beginning )
  local t = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker_end )
  local first_line = -1
  local count = 0
  local last_found = false
  for line in io.lines ( file_name ) do
    if first_line == -1 then
      if string.sub ( line , 1 , #s ) == s then
        first_line = count
      end
    else
      if string.sub ( line , 1 , #t ) == t then
        last_found = true
        break
      end
    end
    count = count + 1
  end
```

```
  if first_line == -1 then
    sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
  else
    if last_found == false then
      sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
    end
  end
  sprintL3 (
      [[ \int_set:Nn \l_@@_first_line_int { ]] .. first_line .. ' + 2 }'
      .. [[ \int_set:Nn \l_@@_last_line_int { ]] .. count .. ' }' )
end
```

10.3.12 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`.

```
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).

```
  local lpeg_line_beamer
  if piton.beamer then
    lpeg_line_beamer =
       space ^ 0
        * P [[\begin{]] * piton.BeamerEnvironments * "}"
        * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
       +
       space ^ 0
        * P [[\end{]] * piton.BeamerEnvironments * "}"
  else
    lpeg_line_beamer = P ( false )
  end

  local lpeg_empty_lines =
    Ct (
         ( lpeg_line_beamer * "\r"
           +
           P " " ^ 0 * "\r" * Cc ( 0 )
           +
           ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
         ) ^ 0
         *
         ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
       )
    * -1

  local lpeg_all_lines =
    Ct (
         ( lpeg_line_beamer * "\r"
           +
```

```
          ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
        ) ^ 0
        *
        ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
      )
    * -1
```

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

```
  piton.empty_lines = lpeg_empty_lines : match ( code )
```

Now, we compute `piton.lines_status`. It will be used in conjonction with `splittable` and `splittable-on-empty-lines`.
Now, we will take into account the current value of `\l_@@_splittable_int` (provided by the *absolute value* of the argument `splittable`).

```
  local lines_status
  local s = splittable
  if splittable < 0 then s = - splittable end

  if splittable > 0 then
    lines_status = lpeg_all_lines : match ( code )
  else
```

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

```
    lines_status = lpeg_empty_lines : match ( code )
    for i , x in ipairs ( lines_status ) do
      if x == 0 then
        for j = 1 , s - 1 do
          if i + j > #lines_status then break end
          if lines_status[i+j] == 0 then break end
            lines_status[i+j] = 2
        end
        for j = 1 , s - 1 do
          if i - j == 1 then break end
          if lines_status[i-j-1] == 0 then break end
          lines_status[i-j-1] = 2
        end
      end
    end
  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.

```
  for j = 1 , s - 1 do
    if j > #lines_status then break end
    if lines_status[j] == 0 then break end
    lines_status[j] = 2
  end
```

Now, from the end of the code.

```
  for j = 1 , s - 1 do
    if #lines_status - j == 0 then break end
    if lines_status[#lines_status - j] == 0 then break end
    lines_status[#lines_status - j] = 2
  end


  piton.lines_status = lines_status
end
```

10.3.13 To create new languages with the syntax of listings

```
function piton.new_language ( lang , definition )
  lang = string.lower ( lang )

  local alpha , digit = lpeg.alpha , lpeg.digit
  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`.

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

For the digits, it's straitforward.

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

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

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

```
      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 `</....>`).

```
      other = other + P ( c )
    end
  end
```

Now, the first transformation of the definition of the language, as provided by the final user in the argument `definition` of `piton.new_language`.

```
  local def_table
  if ( S ", " ^ 0 * -1 ) : match ( definition ) then
    def_table = {}
  else
    local strict_braces  =
      P { "E" ,
          E = ( "{" * V "F" * "}" + ( 1 - S ",{}" ) ) ^ 0  ,
          F = ( "{" * V "F" * "}" + ( 1 - S "{}" ) ) ^ 0
        }
    local cut_definition =
      P { "E" ,
          E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
          F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
                  * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
        }
    def_table = cut_definition : match ( definition )
  end
```

The definition of the language, provided by the final 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.).

```
local tex_braced_arg = "{" * C ( ( 1 - P "}" ) ^ 0 ) * "}"
local tex_arg = tex_braced_arg + C ( 1 )
local tex_option_arg =  "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]" + Cc ( nil )
local args_for_tag
  =  tex_option_arg
     * space ^ 0
     * tex_arg
     * space ^ 0
     * tex_arg
local args_for_morekeywords
  = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
     * space ^ 0
     * tex_option_arg
     * space ^ 0
     * tex_arg
     * space ^ 0
     * ( tex_braced_arg + Cc ( nil ) )
local args_for_moredelims
  = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
    * args_for_morekeywords
local args_for_morecomment
  = "[" * C ( ( 1 - P "]" ) ^ 0 ) * "]"
     * space ^ 0
     * tex_option_arg
     * space ^ 0
     * 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`.

```
local sensitive = true
local style_tag , left_tag , right_tag
for _ , x in ipairs ( def_table ) do
  if x[1] == "sensitive" then
    if x[2] == nil or ( P "true" ) : match ( x[2] ) then
      sensitive = true
    else
      if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
    end
  end
  if x[1] == "alsodigit" then x[2] : gsub ( "." , add_to_digit ) end
  if x[1] == "alsoletter" then x[2] : gsub ( "." , add_to_letter ) end
  if x[1] == "alsoother" then x[2] : gsub ( "." , add_to_other ) end
  if x[1] == "tag" then
    style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
    style_tag = style_tag or [[\PitonStyle{Tag}]]
  end
end
```

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

```
local Number =
  K ( 'Number.Internal' ,
      ( digit ^ 1 * "." * # ( 1 - P "." ) * digit ^ 0
        + digit ^ 0 * "." * digit ^ 1
        + digit ^ 1 )
      * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
      + digit ^ 1
    )
local string_extra_letters = ""
for _ , x in ipairs ( extra_letters ) do
  if not ( extra_others[x] ) then
   string_extra_letters = string_extra_letters .. x
```

```
 end
end
local letter = alpha + S ( string_extra_letters )
                + P "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
                  + "ô" + "û" + "ü" + "Â" + "À" + "Ç" + "É" + "È" + "Ê" + "Ë"
                  + "Ï" + "Î" + "Ô" + "Û" + "Ü"
local alphanum = letter + digit
local identifier = letter * alphanum ^ 0
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.

```
local split_clist =
  P { "E" ,
       E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
           * ( P "{" ) ^ 1
           * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
           * ( P "}" ) ^ 1 * space ^ 0 ,
       F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
    }
```

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

```
local keyword_to_lpeg
function keyword_to_lpeg ( name ) return
  Q ( Cmt (
            C ( identifier ) ,
            function ( s , i , a ) return
              string.upper ( a ) == string.upper ( name )
            end
          )
    )
end
local Keyword = P ( false )
local PrefixedKeyword = P ( false )
```

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

```
for _ , x in ipairs ( def_table )
do if x[1] == "morekeywords"
      or x[1] == "otherkeywords"
      or x[1] == "moredirectives"
      or x[1] == "moretexcs"
   then
      local keywords = P ( false )
      local style = [[\PitonStyle{Keyword}]]
      if x[1] == "moredirectives" then style = [[\PitonStyle{Directive}]] end
      style =  tex_option_arg : match ( x[2] ) or style
      local n = tonumber ( style )
      if n then
        if n > 1 then style = [[\PitonStyle{Keyword]] .. style .. "}" end
      end
      for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
        if x[1] == "moretexcs" then
          keywords = Q ( [[\]] .. word ) + keywords
        else
          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*.

```
          then keywords = Q ( word  ) + keywords
          else keywords = keyword_to_lpeg ( word ) + keywords
          end
        end
```

```
        end
        Keyword = Keyword +
           Lc ( "{" .. style .. "{" ) * keywords * Lc "}}"
      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.

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

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

```
   local long_string  = P ( false )
   local Long_string = P ( false )
   local LongString = P (false )
   local central_pattern = P ( false )
   for _ , x in ipairs ( def_table ) do
     if x[1] == "morestring" then
       arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
       arg2 = arg2 or [[\PitonStyle{String.Long}]]
       if arg1 ~= "s" then
         arg4 = arg3
       end
       central_pattern = 1 - S ( " \r" .. arg4 )
       if arg1 : match "b" then
         central_pattern = P ( [[\]] .. arg3 ) + central_pattern
       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...

```
       if arg1 : match "d" or arg1 == "m" then
         central_pattern = P ( arg3 .. arg3 ) + central_pattern
       end
       if arg1 == "m"
       then prefix = B ( 1 - letter - ")" - "]" )
       else prefix = P ( true )
       end
```

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

```
      long_string = long_string +
          prefix
          * arg3
          * ( space + central_pattern ) ^ 0
          * arg4
```

Now a pattern *with captures.*

```
      local pattern =
          prefix
          * Q ( arg3 )
          * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
          * Q ( arg4 )
```

We will need `Long_string` in the nested comments.

```
    Long_string = Long_string + pattern
    LongString = LongString +
       Ct ( Cc "Open" * Cc ( "{" ..  arg2 .. "{" ) * Cc "}}" )
       * pattern
       * Ct ( Cc "Close" )
  end
end
```

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

```
local braces = Compute_braces ( long_string )
if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end

DetectedCommands = Compute_DetectedCommands ( lang , braces )

LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )
```

Now, we deal with the comments and the delims.

```
local CommentDelim = P ( false )

for _ , x in ipairs ( def_table ) do
  if x[1] == "morecomment" then
    local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
    arg2 = arg2 or [[\PitonStyle{Comment}]]
```

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

```
    if arg1 : match "i" then arg2 = [[\PitonStyle{Discard}]] end
    if arg1 : match "l" then
      local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
                   : match ( other_args )
      if arg3 == [[\#]] then arg3 = "#" end -- mandatory
      if arg3 == [[\%]] then arg3 = "%" end -- mandatory¨
      CommentDelim = CommentDelim +
          Ct ( Cc "Open"
               * Cc ( "{" .. arg2 .. "{" ) * Cc "}}" )
               * Q ( arg3 )
               * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
          * Ct ( Cc "Close" )
          * ( EOL + -1 )
    else
      local arg3 , arg4 =
        ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
      if arg1 : match "s" then
        CommentDelim = CommentDelim +
            Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}}" )
            * Q ( arg3 )
            * (
                CommentMath
                + Q ( ( 1 - P ( arg4 ) - S "$\r" ) ^ 1 ) -- $
                + EOL
              ) ^ 0
            * Q ( arg4 )
            * Ct ( Cc "Close" )
      end
      if arg1 : match "n" then
        CommentDelim = CommentDelim +
          Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "{" ) * Cc "}}" )
           * P { "A" ,
                A = Q ( arg3 )
                    * ( V "A"
                        + Q ( ( 1 - P ( arg3 ) - P ( arg4 )
```

```
                - S "\r$\"" ) ^ 1 ) -- $
            + long_string
            +   "$" -- $
              * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
              * "$" -- $
            + EOL
           ) ^ 0
          * Q ( arg4 )
       }
     * Ct ( Cc "Close" )
   end
  end
 end
```

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

```
 if x[1] == "moredelim" then
  local arg1 , arg2 , arg3 , arg4 , arg5
   = args_for_moredelims : match ( x[2] )
  local MyFun = Q
  if arg1 == "*" or arg1 == "**" then
   function MyFun ( x )
    if x ~= '' then return
     LPEG1[lang] : match ( x )
    end
   end
  end
  local left_delim
  if arg2 : match "i" then
   left_delim = P ( arg4 )
  else
   left_delim = Q ( arg4 )
  end
  if arg2 : match "l" then
   CommentDelim = CommentDelim +
     Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}}" )
     * left_delim
     * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
     * Ct ( Cc "Close" )
     * ( EOL + -1 )
  end
  if arg2 : match "s" then
   local right_delim
   if arg2 : match "i" then
    right_delim = P ( arg5 )
   else
    right_delim = Q ( arg5 )
   end
   CommentDelim = CommentDelim +
     Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "{" ) * Cc "}}" )
     * left_delim
     * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
     * right_delim
     * Ct ( Cc "Close" )
  end
 end
end

local Delim = Q ( S "{[()]}" )
local Punct = Q ( S "=,:;!\\'\"" )
local Main =
   space ^ 0 * EOL
   + Space
   + Tab
   + Escape + EscapeMath
```

```
        + CommentLaTeX
        + Beamer
        + DetectedCommands
        + CommentDelim
```

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

```
        + LongString
        + Delim
        + PrefixedKeyword
        + Keyword * ( -1 + # ( 1 - alphanum ) )
        + Punct
        + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
        + Number
        + 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.

```
   LPEG1[lang] = Main ^ 0
```

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

```
   LPEG2[lang] =
     Ct (
          ( space ^ 0 * P "\r" ) ^ -1
          * BeamerBeginEnvironments
          * Lc [[ \@@_begin_line: ]]
          * SpaceIndentation ^ 0
          * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
          * -1
          * Lc [[ \@@_end_line: ]]
        )
```

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

```
   if left_tag then
     local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "{" ) * Cc "}}" )
                 * Q ( left_tag * other ^ 0 ) -- $
                 * ( ( ( 1 - P ( right_tag ) ) ^ 0 )
                   / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
                 * Q ( right_tag )
                 * Ct ( Cc "Close" )
     MainWithoutTag
             = space ^ 1 * -1
             + space ^ 0 * EOL
             + Space
             + Tab
             + Escape + EscapeMath
             + CommentLaTeX
             + Beamer
             + DetectedCommands
             + CommentDelim
             + Delim
             + LongString
             + PrefixedKeyword
             + Keyword * ( -1 + # ( 1 - alphanum ) )
             + Punct
             + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
             + Number
             + Word
     LPEG0[lang] = MainWithoutTag ^ 0
     local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
                     + Beamer + DetectedCommands + CommentDelim + Tag
     MainWithTag
             = space ^ 1 * -1
```

```
            + space ^ 0 * EOL
            + Space
            + LPEGaux
            + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
    LPEG1[lang] = MainWithTag ^ 0
    LPEG2[lang] =
      Ct (
           ( space ^ 0 * P "\r" ) ^ -1
           * BeamerBeginEnvironments
           * Lc [[ \@@_begin_line: ]]
           * SpaceIndentation ^ 0
           * LPEG1[lang]
           * -1
           * Lc [[ \@@_end_line: ]]
         )
  end
end
```
⟨/LUA⟩

11 History

The successive versions of the file `piton.sty` provided by TeXLive are available on the SVN server of TeXLive:

`https://tug.org/svn/texlive/trunk/Master/texmf-dist/tex/lualatex/piton/piton.sty`

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

Changes between versions 4.1 and 4.2

New key `break-numbers-anywhere`.

Changes between versions 4.0 and 4.1

New language `verbatim`.
New key `break-strings-anywhere`.

Changes between versions 3.1 and 4.0

This version introduces an incompatibility: the syntax for the relative and absolute paths in `\PitonInputFile` and the key `path` has been changed to be conform to usual conventions. An temporary key `old-PitonInputFile`, available at load-time, has been added for backward compatibility.
New keys `font-command`, `splittable-on-empty-lines` and `env-used-by-split`.

Changes between versions 3.0 and 3.1

Keys `line-numbers/format`, `detected-beamer-commands` and `detected-beamer-environments`.

Changes between versions 2.8 and 3.0

New command `\NewPitonLanguage`. Thanks to that command, it's now possible to define new informatic languages with the syntax used by listings. Therefore, it's possible to say that virtually all the informatic languages are now supported by piton.

Changes between versions 2.7 and 2.8

The key `path` now accepts a *list* of paths where the files to include will be searched.
New commands `\PitonInputFileT`, `\PitonInputFileF` and `\PitonInputFileTF`.

Changes between versions 2.6 and 2.7

New keys `split-on-empty-lines` and `split-separation`

Changes between versions 2.5 and 2.6

API: `piton.last_code` and `\g_piton_last_code_tl` are provided.

Changes between versions 2.4 and 2.5

New key `path-write`

Changes between versions 2.3 and 2.4

The key `identifiers` of the command `\PitonOptions` is now deprecated and replaced by the new command `\SetPitonIdentifier`.
A new special language called "minimal" has been added.
New key `detected-commands`.

Changes between versions 2.2 and 2.3

New key `detected-commands`
The variable `\l_piton_language_str` is now public.
New key `write`.

Changes between versions 2.1 and 2.2

New key `path` for `\PitonOptions`.
New language SQL.
It's now possible to define styles locally to a given language (with the optional argument of `\SetPitonStyle`).

Changes between versions 2.0 and 2.1

The key `line-numbers` has now subkeys `line-numbers/skip-empty-lines`, `line-numbers/label-empty-lines`, etc.
The key `all-line-numbers` is deprecated: use `line-numbers/skip-empty-lines=false`.
New system to import, with `\PitonInputFile`, only a part (of the file) delimited by textual markers.
New keys `begin-escape`, `end-escape`, `begin-escape-math` and `end-escape-math`.
The key `escape-inside` is deprecated: use `begin-escape` and `end-escape`.

Contents