

The package **piton**^{*}

F. Pantigny
fpantigny@wanadoo.fr

March 4, 2025

Abstract

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

1 Presentation

The package **piton** uses the Lua library LPEG¹ 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) =  $\frac{\pi}{2}$  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 4.2c of **piton**, at the date of 2025/03/04.

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

²This 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.

³That language `minimal` may be used to format pseudo-codes: cf. p. 32

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 commands⁴ 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 :

<code>\piton{MyString = '\\n'}</code>	<code>MyString = '\n'</code>
<code>\piton{def even(n): return n%2==0}</code>	<code>def even(n): return n%2==0</code>
<code>\piton{c="#" # an affection }</code>	<code>c="#" # an affection</code>
<code>\piton{c="#" \\ \\ # an affection }</code>	<code>c="#" # an affection</code>
<code>\piton{MyDict = {'a': 3, 'b': 4 }}</code>	<code>MyDict = {'a': 3, 'b': 4 }</code>

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 :

<code>\piton MyString = '\n' </code>	<code>MyString = '\n'</code>
<code>\piton!def even(n): return n%2==0!</code>	<code>def even(n): return n%2==0</code>
<code>\piton+c="#" # an affection +</code>	<code>c="#" # an affection</code>
<code>\piton?MyDict = {'a': 3, 'b': 4}?</code>	<code>MyDict = {'a': 3, 'b': 4}</code>

⁴That concerns the commands beginning with a backslash but also the active characters (with catcode equal to 13).

⁵For 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 content⁷ 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

⁶We remind that a LaTeX environment is, in particular, a TeX group.

⁷In 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).

⁸For the language Python, the empty lines in the docstrings are taken into account (by design).

⁹When 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 LuaLaTeX¹⁰.

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 characters¹¹ are replaced by the character `\u2423` (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\u2423good\u2423answer'`

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}

1 void bubbleSort(int arr[], int n) {
2     int temp;
3     int swapped;
4     for (int i = 0; i < n-1; i++) {
5         swapped = 0;
6         for (int j = 0; j < n - i - 1; j++) {
7             if (arr[j] > arr[j + 1]) {
8                 temp = arr[j];
9                 arr[j] = arr[j + 1];
10                arr[j + 1] = temp;
11                swapped = 1;
12            }
13        }
14        if (!swapped) break;
15    }
16 }
```

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).

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

¹¹With 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*.

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

¹³cf. 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 `luatex` (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

¹⁴We remind that a LaTeX environment is, in particular, a TeX group.

¹⁵We recall, that, in the package `piton`, the names of the informatic languages are case-insensitive.

¹⁶As 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 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 others do not.)

Of course, the list of the names of Python functions previously defined is kept in the memory of LuaTeX (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}{\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).

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

¹⁸However, 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 `1stlang1.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=[1]//,%  
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.

¹⁹We 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.

²⁰Remark 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.

²¹With 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}
```

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

1 def cube(x):
2     """Calcule the cube of x"""
3     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 empty 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`.

²²We 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 are 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 choose 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 an 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  $x^2$ 
```

²³That 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: `variorref`, `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:

²⁴The 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}
```

```

1 def arctan(x,n=10):
2     if x < 0 :
3         return - arctan(-x)
4     elif x > 1 :
5         return π/2 - arctan(1/x)
6     else:
7         s = 0
8         for k in range(n): s += (-1)k / (2k+1) * x2k+1
9         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`.

²⁵Remind 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).

²⁶The extension `piton` detects the class `beamer` and the package `beameralternative` 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}`

²⁷One 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 strings²⁸ 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):
    """Compute 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:

²⁸The 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 `luatex` (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) and 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 preferably. 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 that 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` and 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}
\emph{\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) )

```

^aFirst recursive call.

^bSecond recursive call.

6.8 Tabulations

Even though it's probably recommended to indent the informatics listings with spaces and not tabulations³¹, 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).

²⁹First recursive call.

³⁰Second recursive call.

³¹For 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}

1 def arctan(x,n=10):
2     if x < 0:
3         return -arctan(-x)      (recursive call)
4     elif x > 1:
5         return pi/2 - arctan(1/x) (other recursive call)
6     else:
7         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 `luatex` (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 `pylumatex` 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}{!0{}}
{\PitonOptions{#1}}
{\begin{center}
\directlua{pylumatex.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
<code>Number</code>	the numbers
<code>String.Short</code>	the short strings (entre ' ou ")
<code>String.Long</code>	the long strings (entre ''' ou """) excepted the doc-strings (governed by <code>String.Doc</code>)
<code>String</code>	that key fixes both <code>String.Short</code> et <code>String.Long</code>
<code>String.Doc</code>	the doc-strings (only with """ following PEP 257)
<code>String.Interpol</code>	the syntactic elements of the fields of the f-strings (that is to say the characters { et }) ; that style inherits for the styles <code>String.Short</code> and <code>String.Long</code> (according the kind of string where the interpolation appears)
<code>Interpol.Inside</code>	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 <code>piton</code> as done for any Python code.
<code>Operator</code>	the following operators: != == << >> - ~ + / * % = < > & . @
<code>Operator.Word</code>	the following operators: <code>in</code> , <code>is</code> , <code>and</code> , <code>or</code> et <code>not</code>
<code>Name.Builtin</code>	almost all the functions predefined by Python
<code>Name.Decorator</code>	the decorators (instructions beginning by @)
<code>Name.Namespace</code>	the name of the modules
<code>Name.Class</code>	the name of the Python classes defined by the user <i>at their point of definition</i> (with the keyword <code>class</code>)
<code>Name.Function</code>	the name of the Python functions defined by the user <i>at their point of definition</i> (with the keyword <code>def</code>)
<code>UserFunction</code>	the name of the Python functions previously defined by the user (the initial value of that parameter is <code>\PitonStyle{Identifier}</code> and, therefore, the names of that functions are formatted like the identifiers).
<code>Exception</code>	les exceptions pré définies (ex.: <code>SyntaxError</code>)
<code>InitialValues</code>	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 <code>piton</code> as done for any Python code.
<code>Comment</code>	the comments beginning with #
<code>Comment.LaTeX</code>	the comments beginning with #>, which are composed by <code>piton</code> as LaTeX code (merely named "LaTeX comments" in this document)
<code>Keyword.Constant</code>	<code>True</code> , <code>False</code> et <code>None</code>
<code>Keyword</code>	the following keywords: <code>assert</code> , <code>break</code> , <code>case</code> , <code>continue</code> , <code>del</code> , <code>elif</code> , <code>else</code> , <code>except</code> , <code>exec</code> , <code>finally</code> , <code>for</code> , <code>from</code> , <code>global</code> , <code>if</code> , <code>import</code> , <code>in</code> , <code>lambda</code> , <code>non local</code> , <code>pass</code> , <code>raise</code> , <code>return</code> , <code>try</code> , <code>while</code> , <code>with</code> , <code>yield</code> et <code>yield from</code> .
<code>Identifier</code>	the identifiers.

³²See: <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 <i>quoted-strings</i>
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 <i>at their point of definition</i> (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 = C`.

Style	Use
<code>Number</code>	the numbers
<code>String.Long</code>	the strings (between ")
<code>String.Interpol</code>	the elements %d, %i, %f, %c, etc. in the strings; that style inherits from the style <code>String.Long</code>
<code>Operator</code>	the following operators : != == << >> - ~ + / * % = < > & . @
<code>Name.Type</code>	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
<code>Name.Builtin</code>	the following predefined functions: printf, scanf, malloc, sizeof and alignof
<code>Name.Class</code>	le nom des classes au moment de leur définition, c'est-à-dire après le mot-clé <code>class</code>
<code>Name.Function</code>	the name of the Python functions defined by the user <i>at their point of definition</i> (with the keyword <code>let</code>)
<code>UserFunction</code>	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).
<code>Preproc</code>	the instructions of the preprocessor (beginning par #)
<code>Comment</code>	the comments (beginning by // or between /* and */)
<code>Comment.LaTeX</code>	the comments beginning by //> which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
<code>Keyword.Constant</code>	default, false, NULL, nullptr and true
<code>Keyword</code>	the following keywords: alignas, asm, auto, break, case, catch, class, constexpr, const, continue, decltype, do, else, enum, extern, for, goto, if, noexcept, private, public, register, restricted, try, return, static, static_assert, struct, switch, thread_local, throw, typeid, union, using, virtual, volatile and while
<code>Identifier</code>	the identifiers.

9.4 The language SQL

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

Style	Use
<code>Number</code>	the numbers
<code>String.Long</code>	the strings (between ' and not " because the elements between " are names of fields and formatted with <code>Name.Field</code>)
<code>Operator</code>	the following operators : = != <> >= > < <= * + /
<code>Name.Table</code>	the names of the tables
<code>Name.Field</code>	the names of the fields of the tables
<code>Name.Builtin</code>	the following built-in functions (their names are <i>not</i> 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.
<code>Comment</code>	the comments (beginning by -- or between /* and */)
<code>Comment.LaTeX</code>	the comments beginning by --> which are composed by piton as LaTeX code (merely named “LaTeX comments” in this document)
<code>Keyword</code>	the following keywords (their names are <i>not</i> 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
<code>Number</code>	the numbers
<code>String.Long</code>	the strings defined in <code>\NewPitonLanguage</code> by the key <code>morestring</code>
<code>Comment</code>	the comments defined in <code>\NewPitonLanguage</code> by the key <code>morecomment</code>
<code>Comment.LaTeX</code>	the comments which are composed by piton as LaTeX code (merely named “LaTeX comments” in this document)
<code>Keyword</code>	the keywords defined in <code>\NewPitonLanguage</code> by the keys <code>morekeywords</code> and <code>moretexcs</code> (and also the key <code>sensitive</code> which specifies whether the keywords are case-sensitive or not)
<code>Directive</code>	the directives defined in <code>\NewPitonLanguage</code> by the key <code>moredirectives</code>
<code>Tag</code>	the “tags” defined by the key <code>tag</code> (the lexical units detected within the tag will also be formatted with their own style)
<code>Identifier</code>	the identifiers.

Here is for example a definition for the language HTML, obtained with a slight adaptation of the definition done by `listings` (file `lstlang1.sty`).

```
\NewPitonLanguage{HTML}%
{morekeywords={A,ABBR,ACRONYM,ADDRESS,APPLET,AREA,B,BASE,BASEFONT,%
BDO,BIG,BLOCKQUOTE,BODY,BR,BUTTON,CAPTION,CENTER,CITE,CODE,COL,%
COLGROUP,DD,DEL,DFN,DIR,DIV,DL,DOCTYPE,DT,EM,FIELDSET,FONT,FORM,%
FRAME,FRAMESET,HEAD,HR,H1,H2,H3,H4,H5,H6,HTML,I,IFRAME,IMG,INPUT,%
INS,ISINDEX,KBD,LABEL,LEGEND,LH,LI,LINK,LISTING,MAP,META,MENU,%
NOFRAMES,NOSCRIPT,OBJECT,OPTGROUP,OPTION,P,PARAM,PLAINTEXT,PRE,%
OL,Q,S,SAMP,SCRIPT,SELECT,SMALL,SPAN,STRIKE,STRING,STRONG,STYLE,%
SUB,SUP,TABLE,TBODY,TD,TEXTAREA,TFOOT,TH,THEAD,TITLE,TR,TT,U,UL,%
VAR,XMP,%
accesskey,action,align,alink,alt,archive,axis,background,bgcolor,%
border,cellpadding,cellspacing,charset,checked,cite,class,classid,%
code,codebase,codetype,color,cols,colspan,content,coords,data,%
datetime,defer,disabled,dir,event,error,for,frameborder,headers,%
height[href],hreflang[lang],hspace[width],http-equiv[id],ismap[ismap],label[label],lang[lang],link[link],%
longdesc[longdesc],marginheight[marginheight],maxlength[maxlength],media[media],method[method],multiple[multiple],%
name[name],nohref[nohref],noresize[noresize],nowrap[nowrap],onblur[onblur],onchange[onchange],onclick[onclick],%
ondblclick[ondblclick],onfocus[onfocus],onkeydown[onkeydown],onkeypress[onkeypress],onkeyup[onkeyup],onload[onload],onmousedown[onmousedown],%
profile[profile],readonly[readonly],onmousemove[onmousemove],onmouseout[onmouseout],onmouseover[onmouseover],onmouseup[onmouseup],%
onselect[onselect],onunload[onunload],rel[rel],rev[rev],rows[rows],rowspan[rowspan],scheme[scheme],scope[scope],scrolling[scrolling],%
selected[selected],shape[shape],size[size],src[src],standby[standby],style[style],tabindex[tabindex],text[text],title[title],type[type],%
units[units],usemap[usemap],valign[valign],value[value],valuetype[valuetype],vlink[vlink],vspace[vspace],width[width],xmlns[xmlns]},%
tag=<>,%
alsoletter = - ,%
sensitive=f,%
morestring=[d] ",%
}
```

9.6 The language “minimal”

It's possible to switch to the language “minimal” with the key `language = minimal`.

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

Style	Usage
<code>None...</code>	

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.CatcodeTableOtherc, "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:" }
```

^aEach 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`).

^bThe 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).

³³Recall 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}}
\_\_piton\_end\_line:{\PitonStyle{Name.Function}{parity}}(x):\_\_piton\_newline:
\_\_piton\_begin\_line:\_\_piton\_newline{\PitonStyle{Keyword}{return}}
\_\_piton\_end\_line:{\PitonStyle{Operator}{%}}{\PitonStyle{Number}{2}}\_\_piton\_newline:
```

10.2 The L3 part of the implementation

10.2.1 Declaration of the package

```
1 (*STY)
2 \NeedsTeXFormat{LaTeX2e}
3 \RequirePackage{l3keys2e}
4 \ProvidesExplPackage
5   {piton}
6   {\PitonFileVersion}
7   {\PitonFileDate}
8   {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.

```
9 \RequirePackage { amstext }

10 \cs_new_protected:Npn \@@_error:n { \msg_error:nn { piton } }
11 \cs_new_protected:Npn \@@_warning:n { \msg_warning:nn { piton } }
12 \cs_new_protected:Npn \@@_error:nn { \msg_error:nnn { piton } }
13 \cs_new_protected:Npn \@@_error:nnn { \msg_error:nnnn { piton } }
14 \cs_new_protected:Npn \@@_fatal:n { \msg_fatal:nn { piton } }
15 \cs_new_protected:Npn \@@_fatal:nn { \msg_fatal:nnn { piton } }
16 \cs_new_protected:Npn \@@_msg_new:nn { \msg_new:nnn { piton } }
17 \cs_new_protected:Npn \@@_gredirect_none:n #1
18 {
19   \group_begin:
20   \globaldefs = 1
21   \msg_redirect_name:nnn { piton } { #1 } { none }
22   \group_end:
23 }
```

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).

```
24 \cs_new_protected:Npn \@@_msg_new:nnn #1 #2 #3
25 {
26   \bool_if:NTF \g_@@_messages_for_Overleaf_bool
27     { \msg_new:nnn { piton } { #1 } { #2 \\ #3 } }
28     { \msg_new:nnnn { piton } { #1 } { #2 } { #3 } }
29 }
```

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

```
30 \cs_new_protected:Npn \@@_error_or_warning:n
31   { \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”.

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

```

34   {
35     \str_if_eq_p:on \c_sys_jobname_str { _region_ } % for Emacs
36     || \str_if_eq_p:on \c_sys_jobname_str { output } % for Overleaf
37   }

38 \@@_msg_new:nn { LuaLaTeX-mandatory }
39 {
40   LuaLaTeX-is-mandatory.\\
41   The-package-'piton'~requires~the~engine~LuaLaTeX.\\
42   \str_if_eq:onT \c_sys_jobname_str { output }
43   { If~you~use~Overleaf,~you~can~switch~to~LuaLaTeX~in~the~"Menu". \\}
44   If~you~go~on,~the~package~'piton'~won't~be~loaded.
45 }
46 \sys_if_engine_luatex:F { \msg_critical:nn { piton } { LuaLaTeX-mandatory } }

47 \RequirePackage { luatexbase }
48 \RequirePackage { luacode }

49 \@@_msg_new:nnn { piton.lua-not-found }
50 {
51   The~file~'piton.lua'~can't~be~found.\\
52   This~error~is~fatal.\\
53   If~you~want~to~know~how~to~retrieve~the~file~'piton.lua',~type~H~<return>.
54 }
55 {
56   On~the~site~CTAN,~go~to~the~page~of~'piton':~https://ctan.org/pkg/piton.~
57   The~file~'README.md'~explains~how~to~retrieve~the~files~'piton.sty'~and~
58   'piton.lua'.
59 }

60 \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.

```
61 \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.

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

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

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

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

```
65 \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.

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

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

```

67 \keys_define:nn { piton / package }
68 {
69   footnote .bool_gset:N = \g_@@_footnote_bool ,
70   footnotehyper .bool_gset:N = \g_@@_footnotehyper_bool ,
71   footnote .usage:n = load ,
72   footnotehyper .usage:n = load ,
73
74   beamer .bool_gset:N = \g_@@_beamer_bool ,
75   beamer .default:n = true ,

```

```
76     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.

```
77     old-PitonInputFile .bool_set:N = \l_@@_old_PitonInputFile_bool ,
78     old-PitonInputFile .default:n = true ,
79     old-PitonInputFile .usage:n = load ,
80
81     unknown .code:n = \@@_error:n { Unknown-key-for-package }
82 }
83 \@@_msg_new:nn { Unknown-key-for-package }
84 {
85     Unknown-key.\
86     You~have~used~the~key~'\l_keys_key_str'~but~the~only~keys~available~here~
87     are~'beamer',~'footnote',~'footnotehyper'~and~'old-PitonInputFile'.~
88     Other~keys~are~available~in~\token_to_str:N \PitonOptions.\
89     That~key~will~be~ignored.
90 }
```

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

```
91 \ProcessKeysOptions { piton / package }

92 \msg_new:nnn { piton } { old-PitonInputFile }
93 {
94     Be~careful:~The~key~'old-PitonInputFile'~will~be~deleted~
95     in~a~future~version~of~'piton'.
96 }
97 \bool_if:NT \l_@@_old_PitonInputFile_bool
98     { \msg_warning:nn { piton } { old-PitonInputFile } }

99 \IfClassLoadedTF { beamer } { \bool_gset_true:N \g_@@_beamer_bool } { }
100 \IfPackageLoadedTF { beamerarticle } { \bool_gset_true:N \g_@@_beamer_bool } { }
101 \lua_now:n { piton = piton-or-{ } }
102 \bool_if:NT \g_@@_beamer_bool { \lua_now:n { piton.beamer = true } }

103 \hook_gput_code:nnn { begindocument / before } { . }
104     { \IfPackageLoadedTF { xcolor } { } { \usepackage { xcolor } } }

105 \@@_msg_new:nn { footnote-with-footnotehyper-package }
106 {
107     Footnote-forbidden.\
108     You~can't~use~the~option~'footnote'~because~the~package~
109     footnotehyper~has~already~been~loaded.~
110     If~you~want,~you~can~use~the~option~'footnotehyper'~and~the~footnotes~
111     within~the~environments~of~piton~will~be~extracted~with~the~tools~
112     of~the~package~footnotehyper.\
113     If~you~go~on,~the~package~footnote~won't~be~loaded.
114 }

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

124 \bool_if:NT \g_@@_footnote_bool
125 {
```

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

```

126  \IfClassLoadedTF { beamer }
127    { \bool_gset_false:N \g_@@_footnote_bool }
128    {
129      \IfPackageLoadedTF { footnotehyper }
130      { \@@_error:n { footnote~with~footnotehyper~package } }
131      { \usepackage { footnote } }
132    }
133  }
134 \bool_if:NT \g_@@_footnotehyper_bool
135 {

```

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

```

136  \IfClassLoadedTF { beamer }
137    { \bool_gset_false:N \g_@@_footnote_bool }
138    {
139      \IfPackageLoadedTF { footnote }
140      { \@@_error:n { footnotehyper~with~footnote~package } }
141      { \usepackage { footnotehyper } }
142      \bool_gset_true:N \g_@@_footnote_bool
143    }
144 }

```

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}`.

```

145 \lua_now:n
146 {
147   piton.BeamerCommands = lpeg.P ([[uncover]]
148     + [[\only]] + [[\visible]] + [[\invisible]] + [[\alert]] + [[\action]]
149   piton.beamer_environments = { "uncoverenv" , "onlyenv" , "visibleenv" ,
150     "invisibleref" , "alertenv" , "actionenv" }
151   piton.DetectedCommands = lpeg.P ( false )
152   piton.last_code = ''
153   piton.last_language = ''
154 }

```

10.2.2 Parameters and technical definitions

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

```

155 \str_new:N \l_piton_language_str
156 \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.

```
157 \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`).

```
158 \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`).

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

In order to have a better control over the keys.

```

160 \bool_new:N \l_@@_in_PitonOptions_bool
161 \bool_new:N \l_@@_in_PitonInputFile_bool

```

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

```
162 \tl_new:N \l_@@_font_command_tl  
163 \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.

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

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

```
165 \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).

```
166 \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`.

```
167 \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).

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

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

```
169 \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).

```
170 \tl_new:N \l_@@_split_separation_tl  
171 \tl_set:Nn \l_@@_split_separation_tl  
172 { \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`.

```
173 \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`).

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

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

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

The argument of `\PitonInputFile`.

```
177 \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`).

```
178 \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).

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

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

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

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

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

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

```
184 \tl_new:N \l_@@_continuation_symbol_tl
185 \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`.

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

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

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

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

```
190 \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`.

```
191 \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`).

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

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

```
193 \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.

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

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

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

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

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

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

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

```
198 \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.

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

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

202 \cs_new_protected:Npn \@@_tab:
203 {
204     \bool_if:NTF \l_@@_show_spaces_bool
205     {
206         \hbox_set:Nn \l_tmpa_box
207         { \prg_replicate:nn \l_@@_tab_size_int { ~ } }
208         \dim_set:Nn \l_tmpa_dim { \box_wd:N \l_tmpa_box }
209         \color{gray}
210         { \hbox_to_wd:nn \l_tmpa_dim { \rightarrowfill } } \)
211     }
212     { \hbox:n { \prg_replicate:nn \l_@@_tab_size_int { ~ } } }
213     \int_gadd:Nn \g_@@_indentation_int \l_@@_tab_size_int
214 }
```

The following integer corresponds to the key `gobble`.

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

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

```
216 \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.

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

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

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

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

```
220 \cs_new_protected:Npn \@@_label:n #1
221 {
222     \bool_if:NTF \l_@@_line_numbers_bool
223     {
224         \@bsphack
225         \protected@write \auxout { }
226         {
227             \string \newlabel { #1 }
228         }
```

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

```

229         { \int_eval:n { \g_@@_visual_line_int + 1 } }
230         { \thepage }
231     }
232   }
233   \esphack
234 }
235 { \@@_error:n { label~with~lines~numbers } }
236 }
```

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).

```

237 \cs_new_protected:Npn \@@_marker_beginning:n #1 { }
238 \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.

```
239 \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.

```

240 \cs_new_protected:Npn \@@_prompt:
241 {
242   \tl_gset:Nn \g_@@_begin_line_hook_tl
243   {
244     \tl_if_empty:NF \l_@@_prompt_bg_color_tl
245     { \clist_set:No \l_@@_bg_color_clist \l_@@_prompt_bg_color_tl }
246   }
247 }
```

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:`.

```
248 \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

```

249 \cs_generate_variant:Nn \@@_replace_spaces:n { o }
250 \cs_new_protected:Npn \@@_replace_spaces:n #1
251 {
252   \tl_set:Nn \l_tmpa_tl { #1 }
253   \bool_if:NTF \l_@@_show_spaces_bool
254   {
255     \tl_set:Nn \l_@@_space_in_string_tl { \u2028 } % U+2423
256     \regex_replace_all:nnN { \x20 } { \u2028 } \l_tmpa_tl
257   }
258 }
```

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.

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

```

260     {
261         \regex_replace_all:nnN
262             { \x20 }
263             { \c { @@_breakable_space: } }
264             \l_tmpa_tl
265         \regex_replace_all:nnN
266             { \c { l_@@_space_in_string_t1 } }
267             { \c { @@_breakable_space: } }
268             \l_tmpa_tl
269     }
270 }
271 \l_tmpa_tl
272 }
```

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:`).

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

```

274 \cs_set_protected:Npn \@@_begin_line: #1 \@@_end_line:
275 {
276     \group_begin:
277     \g_@@_begin_line_hook_t1
278     \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.

```

279 \bool_if:NTF \l_@@_width_min_bool
280     \@@_put_in_coffin_i:n
281     \@@_put_in_coffin_i:n
282     {
283         \language = -1
284         \raggedright
285         \strut
286         \@@_replace_spaces:n { #1 }
287         \strut \hfil
288     }
```

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

```

289 \hbox_set:Nn \l_tmpa_box
290 {
291     \skip_horizontal:N \l_@@_left_margin_dim
292     \bool_if:NT \l_@@_line_numbers_bool
293     {
```

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

```

294     \int_set:Nn \l_tmpa_int
295     {
296         \lua_now:e
297         {
298             tex.print
299             (
300                 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`.

```

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

```

304
305
306
307
308     \bool_lazy_or:nnT
309     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
310     { ! \l_@@_skip_empty_lines_bool }
311     { \int_gincr:N \g_@@_visual_line_int }
312 \bool_lazy_or:nnT
313     { \int_compare_p:nNn \l_tmpa_int = \c_one_int }
314     { ! \l_@@_skip_empty_lines_bool && \l_@@_label_empty_lines_bool }
315     \@@_print_number:
316 }

```

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

```

317     \clist_if_empty:NF \l_@@_bg_color_clist
318     {
319         ... but if only if the key left-margin is not used !
320         \dim_compare:nNnT \l_@@_left_margin_dim = \c_zero_dim
321         { \skip_horizontal:n { 0.5 em } }
322     }
323     \coffin_typeset:Nnnnn \l_tmpa_coffin T l \c_zero_dim \c_zero_dim
324     \box_set_dp:Nn \l_tmpa_box { \box_dp:N \l_tmpa_box + 1.25 pt }
325     \box_set_ht:Nn \l_tmpa_box { \box_ht:N \l_tmpa_box + 1.25 pt }

```

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

```

326     \mode_leave_vertical:
327     \clist_if_empty:NTF \l_@@_bg_color_clist
328     { \box_use_drop:N \l_tmpa_box }
329     {
330         \vtop
331         {
332             \hbox:n
333             {
334                 \@@_color:N \l_@@_bg_color_clist
335                 \vrule height \box_ht:N \l_tmpa_box
336                     depth \box_dp:N \l_tmpa_box
337                     width \l_@@_width_dim
338             }
339             \skip_vertical:n { - \box_ht_plus_dp:N \l_tmpa_box }
340             \box_use_drop:N \l_tmpa_box
341         }
342     }
343     \group_end:
344     \tl_gclear:N \g_@@_begin_line_hook_tl
345 }

```

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 command takes in its argument by curryfication.

```

346 \cs_set_protected:Npn \@@_put_in_coffin_i:n
347     { \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`.

```

348 \cs_set_protected:Npn \@@_put_in_coffin_i:i:n #1
349     {

```

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`).

```

350     \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).

```

351 \dim_compare:nNnT { \box_wd:N \l_tmpa_box } > \g_@@_tmp_width_dim
352   { \dim_gset:Nn \g_@@_tmp_width_dim { \box_wd:N \l_tmpa_box } }
353 \hcoffin_set:Nn \l_tmpa_coffin
354   {
355     \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).

```

356   { \hbox_unpack:N \l_tmpa_box \hfil }
357 }
358 }
```

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).

```

359 \cs_set_protected:Npn \@@_color:N #1
360   {
361     \int_set:Nn \l_tmpa_int { \clist_count:N #1 }
362     \int_set:Nn \l_tmpb_int { \int_mod:nn \g_@@_line_int \l_tmpa_int + 1 }
363     \tl_set:Ne \l_tmpa_tl { \clist_item:Nn #1 \l_tmpb_int }
364     \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).

```

365   { \dim_zero:N \l_@@_width_dim }
366   { \@@_color_i:o \l_tmpa_tl }
367 }
```

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} }`.

```

368 \cs_generate_variant:Nn \@@_color_i:n { o }
369 \cs_set_protected:Npn \@@_color_i:n #1
370   {
371     \tl_if_head_eq_meaning:nNTF { #1 } [
372       {
373         \tl_set:Nn \l_tmpa_tl { #1 }
374         \tl_set_rescan:Nno \l_tmpa_tl { } \l_tmpa_tl
375         \exp_last_unbraced:No \color \l_tmpa_tl
376       }
377     { \color { #1 } }
378 }
```

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.

```

379 \cs_new_protected:Npn \@@_newline:
380   {
381     \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)...

```

382   \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.

```

383   \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.

```
384 \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.

```
385 \int_case:nn
386 {
387   \lua_now:e
388   {
389     tex.sprint
390     (
391       luatexbase.catcodetables.expl ,
392       tostring ( piton.lines_status [ \int_use:N \g_@@_line_int ] )
393     )
394   }
395 }
396 { 1 { \penalty 100 } 2 \nobreak }
397 \bool_if:NT \g_@@_footnote_bool \savenotes
398 }
```

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

```
399 \cs_set_protected:Npn \@@_breakable_space:
400 {
401   \discretionary
402   { \hbox:n { \color { gray } \l_@@_end_of_broken_line_tl } }
403   {
404     \hbox_overlap_left:n
405     {
406       {
407         \normalfont \footnotesize \color { gray }
408         \l_@@_continuation_symbol_tl
409       }
410       \skip_horizontal:n { 0.3 em }
411       \clist_if_empty:NF \l_@@_bg_color_clist
412       { \skip_horizontal:n { 0.5 em } }
413     }
414   \bool_if:NT \l_@@_indent_broken_lines_bool
415   {
416     \hbox:n
417     {
418       \prg_replicate:nn { \g_@@_indentation_int } { ~ }
419       { \color { gray } \l_@@_csoi_tl }
420     }
421   }
422 }
423 { \hbox { ~ } }
```

10.2.4 PitonOptions

```
425 \bool_new:N \l_@@_line_numbers_bool
426 \bool_new:N \l_@@_skip_empty_lines_bool
427 \bool_set_true:N \l_@@_skip_empty_lines_bool
428 \bool_new:N \l_@@_line_numbers_absolute_bool
429 \tl_new:N \l_@@_line_numbers_format_bool
430 \tl_set:Nn \l_@@_line_numbers_format_tl { \footnotesize \color { gray } }
431 \bool_new:N \l_@@_label_empty_lines_bool
```

```

432 \bool_set_true:N \l_@@_label_empty_lines_bool
433 \int_new:N \l_@@_number_lines_start_int
434 \bool_new:N \l_@@_resume_bool
435 \bool_new:N \l_@@_split_on_empty_lines_bool
436 \bool_new:N \l_@@_splittable_on_empty_lines_bool

437 \keys_define:nn { PitonOptions / marker }
438 {
439   beginning .code:n = \cs_set:Nn \@@_marker_beginning:n { #1 } ,
440   beginning .value_required:n = true ,
441   end .code:n = \cs_set:Nn \@@_marker_end:n { #1 } ,
442   end .value_required:n = true ,
443   include-lines .bool_set:N = \l_@@_marker_include_lines_bool ,
444   include-lines .default:n = true ,
445   unknown .code:n = \@@_error:n { Unknown-key-for-marker }
446 }

447 \keys_define:nn { PitonOptions / line-numbers }
448 {
449   true .code:n = \bool_set_true:N \l_@@_line_numbers_bool ,
450   false .code:n = \bool_set_false:N \l_@@_line_numbers_bool ,
451
452   start .code:n =
453     \bool_set_true:N \l_@@_line_numbers_bool
454     \int_set:Nn \l_@@_number_lines_start_int { #1 } ,
455   start .value_required:n = true ,
456
457   skip-empty-lines .code:n =
458     \bool_if:NF \l_@@_in_PitonOptions_bool
459       { \bool_set_true:N \l_@@_line_numbers_bool }
460     \str_if_eq:nnTF { #1 } { false }
461       { \bool_set_false:N \l_@@_skip_empty_lines_bool }
462       { \bool_set_true:N \l_@@_skip_empty_lines_bool } ,
463   skip-empty-lines .default:n = true ,
464
465   label-empty-lines .code:n =
466     \bool_if:NF \l_@@_in_PitonOptions_bool
467       { \bool_set_true:N \l_@@_line_numbers_bool }
468     \str_if_eq:nnTF { #1 } { false }
469       { \bool_set_false:N \l_@@_label_empty_lines_bool }
470       { \bool_set_true:N \l_@@_label_empty_lines_bool } ,
471   label-empty-lines .default:n = true ,
472
473   absolute .code:n =
474     \bool_if:NTF \l_@@_in_PitonOptions_bool
475       { \bool_set_true:N \l_@@_line_numbers_absolute_bool }
476       { \bool_set_true:N \l_@@_line_numbers_bool }
477     \bool_if:NT \l_@@_in_PitonInputFile_bool
478       {
479         \bool_set_true:N \l_@@_line_numbers_absolute_bool
480         \bool_set_false:N \l_@@_skip_empty_lines_bool
481       } ,
482   absolute .value_forbidden:n = true ,
483
484   resume .code:n =
485     \bool_set_true:N \l_@@_resume_bool
486     \bool_if:NF \l_@@_in_PitonOptions_bool
487       { \bool_set_true:N \l_@@_line_numbers_bool } ,
488   resume .value_forbidden:n = true ,
489
490   sep .dim_set:N = \l_@@_numbers_sep_dim ,
491   sep .value_required:n = true ,
492

```

```

493   format .tl_set:N = \l_@@_line_numbers_format_tl ,
494   format .value_required:n = true ,
495
496   unknown .code:n = \@@_error:n { Unknown~key~for~line~numbers }
497 }

```

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

```

498 \keys_define:nn { PitonOptions }
499 {
500   break-strings-anywhere .bool_set:N = \l_@@_break_strings_anywhere_bool ,
501   break-strings-anywhere .default:n = true ,
502   break-numbers-anywhere .bool_set:N = \l_@@_break_numbers_anywhere_bool ,
503   break-numbers-anywhere .default:n = true ,

```

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

```

504   detected-commands .code:n =
505     \lua_now:n { piton.addDetectedCommands('#1') } ,
506   detected-commands .value_required:n = true ,
507   detected-commands .usage:n = preamble ,
508   detected-beamer-commands .code:n =
509     \lua_now:n { piton.addBeamerCommands('#1') } ,
510   detected-beamer-commands .value_required:n = true ,
511   detected-beamer-commands .usage:n = preamble ,
512   detected-beamer-environments .code:n =
513     \lua_now:n { piton.addBeamerEnvironments('#1') } ,
514   detected-beamer-environments .value_required:n = true ,
515   detected-beamer-environments .usage:n = preamble ,

```

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

```

516   begin-escape .code:n =
517     \lua_now:e { piton.begin_escape = "\lua_escape:n{#1}" } ,
518   begin-escape .value_required:n = true ,
519   begin-escape .usage:n = preamble ,
520
521   end-escape .code:n =
522     \lua_now:e { piton.end_escape = "\lua_escape:n{#1}" } ,
523   end-escape .value_required:n = true ,
524   end-escape .usage:n = preamble ,
525
526   begin-escape-math .code:n =
527     \lua_now:e { piton.begin_escape_math = "\lua_escape:n{#1}" } ,
528   begin-escape-math .value_required:n = true ,
529   begin-escape-math .usage:n = preamble ,
530
531   end-escape-math .code:n =
532     \lua_now:e { piton.end_escape_math = "\lua_escape:n{#1}" } ,
533   end-escape-math .value_required:n = true ,
534   end-escape-math .usage:n = preamble ,
535
536   comment-latex .code:n = \lua_now:n { comment_latex = "#1" } ,
537   comment-latex .value_required:n = true ,
538   comment-latex .usage:n = preamble ,
539
540   math-comments .bool_gset:N = \g_@@_math_comments_bool ,
541   math-comments .default:n = true ,
542   math-comments .usage:n = preamble ,

```

Now, general keys.

```

543   language .code:n =
544     \str_set:Nc \l_piton_language_str { \str_lowercase:n { #1 } } ,
545   language .value_required:n = true ,
546   path .code:n =
547     \seq_clear:N \l_@@_path_seq

```

```

548     \clist_map_inline:nn { #1 }
549     {
550         \str_set:Nn \l_tmpa_str { ##1 }
551         \seq_put_right:No \l_@@_path_seq \l_tmpa_str
552     } ,
553     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.

```

554     path .initial:n = . ,
555     path-write .str_set:N = \l_@@_path_write_str ,
556     path-write .value_required:n = true ,
557     font-command .tl_set:N = \l_@@_font_command_tl ,
558     font-command .value_required:n = true ,
559     gobble .int_set:N = \l_@@_gobble_int ,
560     gobble .value_required:n = true ,
561     auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -1 } ,
562     auto-gobble .value_forbidden:n = true ,
563     env-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -2 } ,
564     env-gobble .value_forbidden:n = true ,
565     tabs-auto-gobble .code:n = \int_set:Nn \l_@@_gobble_int { -3 } ,
566     tabs-auto-gobble .value_forbidden:n = true ,
567
568     splittable-on-empty-lines .bool_set:N = \l_@@_splittable_on_empty_lines_bool ,
569     splittable-on-empty-lines .default:n = true ,
570
571     split-on-empty-lines .bool_set:N = \l_@@_split_on_empty_lines_bool ,
572     split-on-empty-lines .default:n = true ,
573
574     split-separation .tl_set:N = \l_@@_split_separation_tl ,
575     split-separation .value_required:n = true ,
576
577     marker .code:n =
578     \bool_lazy_or:nnTF
579         \l_@@_in_PitonInputFile_bool
580         \l_@@_in_PitonOptions_bool
581         { \keys_set:nn { PitonOptions / marker } { #1 } }
582         { \@@_error:n { Invalid~key } } ,
583     marker .value_required:n = true ,
584
585     line-numbers .code:n =
586         \keys_set:nn { PitonOptions / line-numbers } { #1 } ,
587     line-numbers .default:n = true ,
588
589     splittable .int_set:N = \l_@@_splittable_int ,
590     splittable .default:n = 1 ,
591     background-color .clist_set:N = \l_@@_bg_color_clist ,
592     background-color .value_required:n = true ,
593     prompt-background-color .tl_set:N = \l_@@_prompt_bg_color_tl ,
594     prompt-background-color .value_required:n = true ,
595
596     width .code:n =
597         \str_if_eq:nnTF { #1 } { min }
598         {
599             \bool_set_true:N \l_@@_width_min_bool
600             \dim_zero:N \l_@@_width_dim
601         }
602         {
603             \bool_set_false:N \l_@@_width_min_bool
604             \dim_set:Nn \l_@@_width_dim { #1 }
605         } ,
606     width .value_required:n = true ,
607
608     write .str_set:N = \l_@@_write_str ,

```

```

609 write .value_required:n = true ,
610
611 left-margin .code:n =
612 \str_if_eq:nnTF { #1 } { auto }
613 {
614     \dim_zero:N \l_@@_left_margin_dim
615     \bool_set_true:N \l_@@_left_margin_auto_bool
616 }
617 {
618     \dim_set:Nn \l_@@_left_margin_dim { #1 }
619     \bool_set_false:N \l_@@_left_margin_auto_bool
620 },
621 left-margin .value_required:n = true ,
622
623 tab-size .int_set:N = \l_@@_tab_size_int ,
624 tab-size .value_required:n = true ,
625 show-spaces .bool_set:N = \l_@@_show_spaces_bool ,
626 show-spaces .value_forbidden:n = true ,
627 show-spaces-in-strings .code:n =
628     \tl_set:Nn \l_@@_space_in_string_tl { \u } , % U+2423
629 show-spaces-in-strings .value_forbidden:n = true ,
630 break-lines-in-Piton .bool_set:N = \l_@@_break_lines_in_Piton_bool ,
631 break-lines-in-Piton .default:n = true ,
632 break-lines-in-piton .bool_set:N = \l_@@_break_lines_in_piton_bool ,
633 break-lines-in-piton .default:n = true ,
634 break-lines .meta:n = { break-lines-in-piton , break-lines-in-Piton } ,
635 break-lines .value_forbidden:n = true ,
636 indent-broken-lines .bool_set:N = \l_@@_indent_broken_lines_bool ,
637 indent-broken-lines .default:n = true ,
638 end-of-broken-line .tl_set:N = \l_@@_end_of_broken_line_tl ,
639 end-of-broken-line .value_required:n = true ,
640 continuation-symbol .tl_set:N = \l_@@_continuation_symbol_tl ,
641 continuation-symbol .value_required:n = true ,
642 continuation-symbol-on-indentation .tl_set:N = \l_@@_csoi_tl ,
643 continuation-symbol-on-indentation .value_required:n = true ,
644
645 first-line .code:n = \@@_in_PitonInputFile:n
646     { \int_set:Nn \l_@@_first_line_int { #1 } } ,
647 first-line .value_required:n = true ,
648
649 last-line .code:n = \@@_in_PitonInputFile:n
650     { \int_set:Nn \l_@@_last_line_int { #1 } } ,
651 last-line .value_required:n = true ,
652
653 begin-range .code:n = \@@_in_PitonInputFile:n
654     { \str_set:Nn \l_@@_begin_range_str { #1 } } ,
655 begin-range .value_required:n = true ,
656
657 end-range .code:n = \@@_in_PitonInputFile:n
658     { \str_set:Nn \l_@@_end_range_str { #1 } } ,
659 end-range .value_required:n = true ,
660
661 range .code:n = \@@_in_PitonInputFile:n
662     {
663         \str_set:Nn \l_@@_begin_range_str { #1 }
664         \str_set:Nn \l_@@_end_range_str { #1 }
665     },
666 range .value_required:n = true ,
667
668 env-used-by-split .code:n =
669     \lua_now:n { piton.env_used_by_split = '#1' } ,
670 env-used-by-split .initial:n = Piton ,
671

```

```

672 resume .meta:n = line-numbers/resume ,
673
674 unknown .code:n = \@@_error:n { Unknown-key-for-PitonOptions } ,
675
676 % deprecated
677 all-line-numbers .code:n =
678   \bool_set_true:N \l_@@_line_numbers_bool
679   \bool_set_false:N \l_@@_skip_empty_lines_bool ,
680 all-line-numbers .value_forbidden:n = true
681 }

682 \cs_new_protected:Npn \@@_in_PitonInputFile:n #1
683 {
684   \bool_if:NF \l_@@_in_PitonInputFile_bool
685   { #1 }
686   { \@@_error:n { Invalid-key } }
687 }

688 \NewDocumentCommand \PitonOptions { m }
689 {
690   \bool_set_true:N \l_@@_in_PitonOptions_bool
691   \keys_set:nn { PitonOptions } { #1 }
692   \bool_set_false:N \l_@@_in_PitonOptions_bool
693 }

```

When using `\NewPitonEnvironment` a user may use `\PitonOptions` inside. However, the set of keys available should be different than 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.

```

694 \NewDocumentCommand \@@_fake_PitonOptions { }
695   { \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.

```

696 \int_new:N \g_@@_visual_line_int
697 \cs_new_protected:Npn \@@_incr_visual_line:
698 {
699   \bool_if:NF \l_@@_skip_empty_lines_bool
700   { \int_gincr:N \g_@@_visual_line_int }
701 }
702 \cs_new_protected:Npn \@@_print_number:
703 {
704   \hbox_overlap_left:n
705   {
706     \l_@@_line_numbers_format_tl
707   }

```

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

```

708   { \int_to_arabic:n \g_@@_visual_line_int }
709   }
710   \skip_horizontal:N \l_@@_numbers_sep_dim
711 }
712 }

```

10.2.6 The command to write on the aux file

```

713 \cs_new_protected:Npn \@@_write_aux:
714 {
715     \tl_if_empty:NF \g_@@_aux_tl
716     {
717         \iow_now:Nn \mainaux { \ExplSyntaxOn }
718         \iow_now:Ne \mainaux
719         {
720             \tl_gset:cn { c_@@_int_use:N \g_@@_env_int _ tl }
721             { \exp_not:o \g_@@_aux_tl }
722         }
723         \iow_now:Nn \mainaux { \ExplSyntaxOff }
724     }
725     \tl_gclear:N \g_@@_aux_tl
726 }
```

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

```

727 \cs_new_protected:Npn \@@_width_to_aux:
728 {
729     \tl_gput_right:Ne \g_@@_aux_tl
730     {
731         \dim_set:Nn \l_@@_line_width_dim
732         { \dim_eval:n { \g_@@_tmp_width_dim } }
733     }
734 }
```

10.2.7 The main commands and environments for the final user

```

735 \NewDocumentCommand { \NewPitonLanguage } { O { } m ! o }
736 {
737     \tl_if_novalue:nTF { #3 }
```

The last argument is provided by currying.

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

The two last arguments are provided by currying.

```
739     { \@@_NewPitonLanguage:nnnn { #1 } { #2 } { #3 } }
740 }
```

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.

```

741 \prop_new:N \g_@@_languages_prop

742 \keys_define:nn { NewPitonLanguage }
743 {
744     morekeywords .code:n = ,
745     otherkeywords .code:n = ,
746     sensitive .code:n = ,
747     keywordsprefix .code:n = ,
748     moretexcs .code:n = ,
749     morestring .code:n = ,
750     morecomment .code:n = ,
751     moredelim .code:n = ,
752     moredirectives .code:n = ,
753     tag .code:n = ,
754     alsodigit .code:n = ,
755     alsoletter .code:n = ,
756     alsoother .code:n = ,
757     unknown .code:n = \@@_error:n { Unknown-key~NewPitonLanguage }
758 }
```

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

```
759 \cs_new_protected:Npn \@@_NewPitonLanguage:nnn #1 #2 #3
760 {
```

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}{...}`.

```
761 \tl_set:Ne \l_tmpa_tl
762 {
763     \tl_if_blank:nF { #1 } { [ \str_lowercase:n { #1 } ] }
764     \str_lowercase:n { #2 }
765 }
```

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

```
766 \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.

```
767 \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).

```
768 \@@_NewPitonLanguage:on \l_tmpa_tl { #3 }
769 }
770 \cs_generate_variant:Nn \@@_NewPitonLanguage:nn { o }
771 \cs_new_protected:Npn \@@_NewPitonLanguage:nn #1 #2
772 {
773     \hook_gput_code:nnn { begindocument } { . }
774     { \lua_now:e { piton.new_language("#1", "\lua_escape:n{#2}") } }
775 }
```

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

```
776 \cs_new_protected:Npn \@@_NewPitonLanguage:nnnn #1 #2 #3 #4 #5
777 {
```

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}{...}`

```
778 \tl_set:Ne \l_tmpa_tl
779 {
780     \tl_if_blank:nF { #3 } { [ \str_lowercase:n { #3 } ] }
781     \str_lowercase:n { #4 }
782 }
```

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`.

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

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

```
784 { \@@_NewPitonLanguage:nnno { #1 } { #2 } { #5 } \l_tmpb_tl }
785 { \@@_error:n { Language-not-defined } }
786 }
```

```
787 \cs_generate_variant:Nn \@@_NewPitonLanguage:nnnn { n n n o }
788 \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.

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

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

```

793   {
794     \group_begin:
795     \bool_lazy_or:nnT
796     \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 difficulties during the insertion of all the commands `\-` (to allow breaks anywhere in the string).

```

797     \l_@@_break_strings_anywhere_bool
798     { \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.

```

799     \automatichyphenmode = 1

```

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

```

800     \cs_set_eq:NN \\ \c_backslash_str
801     \cs_set_eq:NN \% \c_percent_str
802     \cs_set_eq:NN \{ \c_left_brace_str
803     \cs_set_eq:NN \} \c_right_brace_str
804     \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.

```

805     \cs_set_eq:cN { ~ } \space
806     \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
807     \tl_set:N \l_tmpa_tl
808     {
809       \lua_now:e
810       { \piton.ParseBis('l_piton_language_str',token.scan_string()) }
811       { #1 }
812     }
813     \bool_if:NTF \l_@@_show_spaces_bool
814     { \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.

```

815     {
816       \bool_if:NT \l_@@_break_lines_in_piton_bool
817       { \regex_replace_all:nnN { \x20 } { \x20 } \l_tmpa_tl }
818     }

```

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

```

819     \if_mode_math:
820       \text { \l_@@_font_command_tl \l_tmpa_tl }
821     \else:
822       \l_@@_font_command_tl \l_tmpa_tl
823     \fi:
824     \group_end:
825   }
826 \NewDocumentCommand { \@@_piton_verbatim } { v }
827   {
828     \group_begin:
829     \automatichyphenmode = 1
830     \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
831     \tl_set:N \l_tmpa_tl
832     {
833       \lua_now:e
834       { \piton.Parse('l_piton_language_str',token.scan_string()) }
835       { #1 }
836     }
837     \bool_if:NT \l_@@_show_spaces_bool
838     { \regex_replace_all:nnN { \x20 } { \_ } \l_tmpa_tl } % U+2423
839     \if_mode_math:
840       \text { \l_@@_font_command_tl \l_tmpa_tl }

```

```

841     \else:
842         \l_@@_font_command_tl \l_tmpa_tl
843     \fi:
844     \group_end:
845 }

```

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).

```

846 \cs_new_protected:Npn \@@_piton:n #1
847   { \tl_if_blank:nF { #1 } { \@@_piton_i:n { #1 } } }
848
849 \cs_new_protected:Npn \@@_piton_i:n #1
850   {
851     \group_begin:
852     \cs_set_eq:NN \@@_begin_line: \prg_do_nothing:
853     \cs_set:cpn { pitonStyle _ \l_piton_language_str _ Prompt } { }
854     \cs_set:cpn { pitonStyle _ Prompt } { }
855     \cs_set_eq:NN \@@_trailing_space: \space
856     \tl_set:Ne \l_tmpa_tl
857     {
858       \lua_now:e
859       { piton.ParseTer('l_piton_language_str',token.scan_string()) }
860       { #1 }
861     }
862     \bool_if:NT \l_@@_show_spaces_bool
863       { \regex_replace_all:nnN { \x20 } { \l_@@_space } \l_tmpa_tl } % U+2423
864     \@@_replace_spaces:o \l_tmpa_tl
865   \group_end:
866 }

```

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

```

867 \cs_new:Npn \@@_pre_env:
868   {
869     \automatichyphenmode = 1
870     \int_gincr:N \g_@@_env_int
871     \tl_gclear:N \g_@@_aux_tl
872     \dim_compare:nNnT \l_@@_width_dim = \c_zero_dim
873       { \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`).

```

874   \cs_if_exist_use:c { c_@@_int_use:N \g_@@_env_int _ tl }
875   \bool_if:NF \l_@@_resume_bool { \int_gzero:N \g_@@_visual_line_int }
876   \dim_gzero:N \g_@@_tmp_width_dim
877   \int_gzero:N \g_@@_line_int
878   \dim_zero:N \parindent
879   \dim_zero:N \lineskip
880   \cs_set_eq:NN \label \@@_label:n
881 }

```

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.

```

882 \cs_generate_variant:Nn \@@_compute_left_margin:nn { n o }
883 \cs_new_protected:Npn \@@_compute_left_margin:nn #1 #2
884   {
885     \bool_lazy_and:nnT \l_@@_left_margin_auto_bool \l_@@_line_numbers_bool

```

```

886   {
887     \hbox_set:Nn \l_tmpa_box
888     {
889       \l_@@_line_numbers_format_tl
890       \bool_if:NTF \l_@@_skip_empty_lines_bool
891       {
892         \lua_now:n
893         { piton.#1(token.scan_argument()) }
894         { #2 }
895         \int_to_arabic:n
896         { \g_@@_visual_line_int + \l_@@_nb_non_empty_lines_int }
897       }
898       {
899         \int_to_arabic:n
900         { \g_@@_visual_line_int + \l_@@_nb_lines_int }
901       }
902     }
903     \dim_set:Nn \l_@@_left_margin_dim
904     { \box_wd:N \l_tmpa_box + \l_@@_numbers_sep_dim + 0.1 em }
905   }
906 }

```

Whereas $\l_@@_width_dim$ 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.

```

907 \cs_new_protected:Npn \@@_compute_width:
908   {
909     \dim_compare:nNnTF \l_@@_line_width_dim = \c_zero_dim
910     {
911       \dim_set_eq:NN \l_@@_line_width_dim \l_@@_width_dim
912       \clist_if_empty:NTF \l_@@_bg_color_clist

```

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

```
913       { \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.

```

914   {
915     \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 value³⁴ and we use that value. Elsewhere, we use a value of 0.5 em.

```

916   \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
917   { \dim_sub:Nn \l_@@_line_width_dim { 0.5 em } }
918   { \dim_sub:Nn \l_@@_line_width_dim \l_@@_left_margin_dim }
919 }
920 }

```

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.

```

921   {
922     \dim_set_eq:NN \l_@@_width_dim \l_@@_line_width_dim
923     \clist_if_empty:NTF \l_@@_bg_color_clist
924     { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }
925     {
926       \dim_add:Nn \l_@@_width_dim { 0.5 em }
927       \dim_compare:nNnTF \l_@@_left_margin_dim = \c_zero_dim
928       { \dim_add:Nn \l_@@_width_dim { 0.5 em } }
929       { \dim_add:Nn \l_@@_width_dim \l_@@_left_margin_dim }

```

³⁴If 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.

```

930         }
931     }
932 }

933 \NewDocumentCommand { \NewPitonEnvironment } { m m m m }
934 {

```

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.

```

935 \use:x
936 {
937     \cs_set_protected:Npn
938         \use:c { _@@_collect_ #1 :w }
939         #####1
940         \c_backslash_str end \c_left_brace_str #1 \c_right_brace_str
941     }
942     {
943         \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...

```
944     \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.

```
945     \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.

```

946     @@_compute_left_margin:nn { CountNonEmptyLines } { ##1 }
947     @@_compute_width:
948     \l_@@_font_command_tl
949     \dim_zero:N \parskip
950     \noindent

```

Now, the key `write`.

```

951     \str_if_empty:NTF \l_@@_path_write_str
952         { \lua_now:e { piton.write = "\l_@@_write_str" } }
953         {
954             \lua_now:e
955                 { piton.write = "\l_@@_path_write_str / \l_@@_write_str" }
956         }
957     \str_if_empty:NTF \l_@@_write_str
958         { \lua_now:n { piton.write = '' } }
959         {
960             \seq_if_in:NoTF \g_@@_write_seq \l_@@_write_str
961                 { \lua_now:n { piton.write_mode = "a" } }
962                 {
963                     \lua_now:n { piton.write_mode = "w" }
964                     \seq_gput_left:No \g_@@_write_seq \l_@@_write_str
965                 }
966         }

```

Now, the main job.

```

967     \bool_if:NTF \l_@@_split_on_empty_lines_bool
968         \@@_retrieve_gobble_split_parse:n
969         \@@_retrieve_gobble_parse:n
970         { ##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).

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

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

```

972     \end { #1 }
973     \@@_write_aux:
974 }
```

We can now define the new environment.

We are still in the definition of the command `\NewPitonEnvironment`...

```

975   \NewDocumentEnvironment { #1 } { #2 }
976   {
977     \cs_set_eq:Nn \PitonOptions \@@_fake_PitonOptions
978     #3
979     \@@_pre_env:
980     \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int
981       { \int_gset:Nn \g_@@_visual_line_int { \l_@@_number_lines_start_int - 1 } }
982     \group_begin:
983     \tl_map_function:nN
984       { \ \\ \{ \} \$ \& \^ \_ \% \~ \^\I }
985     \char_set_catcode_other:N
986     \use:c { _@@_collect_ #1 :w }
987   }
988   { #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).

```

989   \AddToHook { env / #1 / begin } { \char_set_catcode_other:N \^\M }
990 }
```

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.

```

991 \cs_new_protected:Npn \@@_retrieve_gobble_parse:n
992 {
993   \lua_now:e
994   {
995     piton.RetrieveGobbleParse
996     (
997       '\l_piton_language_str' ,
998       \int_use:N \l_@@_gobble_int ,
999       \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1000         { \int_eval:n { - \l_@@_splittable_int } }
1001         { \int_use:N \l_@@_splittable_int } ,
1002       token.scan_argument ( )
1003     )
1004   }
1005 }
```

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.

```

1006 \cs_new_protected:Npn \@@_retrieve_gobble_split_parse:n
1007 {
1008   \lua_now:e
1009   {
1010     piton.RetrieveGobbleSplitParse
1011     (
1012       '\l_piton_language_str' ,
1013       \int_use:N \l_@@_gobble_int ,
1014       \int_use:N \l_@@_splittable_int ,
```

```

1015         token.scan_argument ( )
1016     )
1017 }
1018 }
```

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

```

1019 \bool_if:NTF \g_@@_beamer_bool
1020 {
1021   \NewPitonEnvironment { Piton } { d < > 0 { } }
1022   {
1023     \keys_set:nn { PitonOptions } { #2 }
1024     \tl_if_novalue:nTF { #1 }
1025       { \begin { uncoverenv } }
1026       { \begin { uncoverenv } < #1 > }
1027   }
1028   { \end { uncoverenv } }
1029 }
1030 {
1031   \NewPitonEnvironment { Piton } { 0 { } }
1032   { \keys_set:nn { PitonOptions } { #1 } }
1033   { }
1034 }
```

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.

```

1035 \NewDocumentCommand { \PitonInputFileTF } { d < > 0 { } m m m }
1036 {
1037   \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.

```

1038 \bool_if:NTF \l_@@_old_PitonInputFile_bool
1039 {
1040   \bool_set_false:N \l_tmpa_bool
1041   \seq_map_inline:Nn \l_@@_path_seq
1042   {
1043     \str_set:Nn \l_@@_file_name_str { ##1 / #3 }
1044     \file_if_exist:nT { \l_@@_file_name_str }
1045     {
1046       \@@_input_file:nn { #1 } { #2 }
1047       \bool_set_true:N \l_tmpa_bool
1048       \seq_map_break:
1049     }
1050   }
1051   \bool_if:NTF \l_tmpa_bool { #4 } { #5 }
1052 }
1053 {
1054   \seq_concat:NNN
1055   \l_file_search_path_seq
1056   \l_@@_path_seq
1057   \l_file_search_path_seq
1058   \file_get_full_name:nNTF { #3 } \l_@@_file_name_str
1059   {
1060     \@@_input_file:nn { #1 } { #2 }
1061     #4
1062   }
1063   { #5 }
1064 }
1065 \group_end:
1066 }
```

```

1067 \cs_new_protected:Npn \@@_unknown_file:n #1
1068   { \msg_error:nnn { piton } { Unknown~file } { #1 } }
1069 \NewDocumentCommand { \PitonInputFile } { d < > O { } m }
1070   { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { \@@_unknown_file:n { #3 } } }
1071 \NewDocumentCommand { \PitonInputFileT } { d < > O { } m m }
1072   { \PitonInputFileTF < #1 > [ #2 ] { #3 } { #4 } { \@@_unknown_file:n { #3 } } }
1073 \NewDocumentCommand { \PitonInputFileF } { d < > O { } m m }
1074   { \PitonInputFileTF < #1 > [ #2 ] { #3 } { } { #4 } }

```

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

```

1075 \cs_new_protected:Npn \@@_input_file:nn #1 #2
1076   {

```

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 >).

```

1077 \tl_if_no_value:nF { #1 }
1078   {
1079     \bool_if:NTF \g_@@_beamer_bool
1080       { \begin{uncoverenv} < #1 > }
1081       { \@@_error_or_warning:n { overlay~without~beamer } }
1082   }
1083 \group_begin:
1084 % The following line is to allow programs such as |latexmk| to be aware that the
1085 % file (read by |\PitonInputFile|) is loaded during the compilation of the LaTeX
1086 % document.
1087 % \begin{macrocode}
1088   \iow_log:e {(\l_@@_file_name_str)}
1089   \int_zero_new:N \l_@@_first_line_int
1090   \int_zero_new:N \l_@@_last_line_int
1091   \int_set_eq:NN \l_@@_last_line_int \c_max_int
1092   \bool_set_true:N \l_@@_in_PitonInputFile_bool
1093   \keys_set:nn { PitonOptions } { #2 }
1094   \bool_if:NT \l_@@_line_numbers_absolute_bool
1095     { \bool_set_false:N \l_@@_skip_empty_lines_bool }
1096   \bool_if:nTF
1097     {
1098       (
1099         \int_compare_p:nNn \l_@@_first_line_int > \c_zero_int
1100         || \int_compare_p:nNn \l_@@_last_line_int < \c_max_int
1101       )
1102       && ! \str_if_empty_p:N \l_@@_begin_range_str
1103     }
1104   {
1105     \@@_error_or_warning:n { bad~range~specification }
1106     \int_zero:N \l_@@_first_line_int
1107     \int_set_eq:NN \l_@@_last_line_int \c_max_int
1108   }
1109   {
1110     \str_if_empty:NF \l_@@_begin_range_str
1111     {
1112       \@@_compute_range:
1113       \bool_lazy_or:nnT
1114         \l_@@_marker_include_lines_bool
1115         { ! \str_if_eq_p:NN \l_@@_begin_range_str \l_@@_end_range_str }
1116       {
1117         \int_decr:N \l_@@_first_line_int
1118         \int_incr:N \l_@@_last_line_int
1119       }
1120     }
1121   }
1122 \@@_pre_env:
1123 \bool_if:NT \l_@@_line_numbers_absolute_bool
1124   { \int_gset:Nn \g_@@_visual_line_int { \l_@@_first_line_int - 1 } }
1125   \int_compare:nNnT \l_@@_number_lines_start_int > \c_zero_int

```

```

1126      {
1127          \int_gset:Nn \g_@@_visual_line_int
1128              { \l_@@_number_lines_start_int - 1 }
1129      }

```

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

```

1130      \int_compare:nNnT \g_@@_visual_line_int < \c_zero_int
1131          { \int_gzero:N \g_@@_visual_line_int }
1132      \mode_if_vertical:TF \mode_leave_vertical: \newline
1133      \dim_zero:N \parskip % added 2025/03/03

```

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

```
1134      \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.

```

1135      \@@_compute_left_margin:no { CountNonEmptyLinesFile } \l_@@_file_name_str
1136      \@@_compute_width:
1137      \l_@@_font_command_tl
1138      \lua_now:e
1139      {
1140          piton.ParseFile(
1141              '\l_piton_language_str' ,
1142              '\l_@@_file_name_str' ,
1143              \int_use:N \l_@@_first_line_int ,
1144              \int_use:N \l_@@_last_line_int ,
1145              \bool_if:NTF \l_@@_splittable_on_empty_lines_bool
1146                  { \int_eval:n { - \l_@@_splittable_int } }
1147                  { \int_use:N \l_@@_splittable_int } ,
1148                  \bool_if:NTF \l_@@_split_on_empty_lines_bool { 1 } { 0 } )
1149      }
1150      \bool_if:NT \l_@@_width_min_bool \@@_width_to_aux:
1151  \group_end:

```

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

```

1152      \tl_if_novalue:nF { #1 }
1153          { \bool_if:NT \g_@@_beamer_bool { \end { uncoverenv } } }
1154      \@@_write_aux:
1155  }

```

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

```

1156  \cs_new_protected:Npn \@@_compute_range:
1157  {

```

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

```

1158      \str_set:Ne \l_tmpa_str { \@@_marker_beginning:n \l_@@_begin_range_str }
1159      \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`

```

1160      \regex_replace_all:nVN { \\# } \c_hash_str \l_tmpa_str
1161      \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).

```

1162      \lua_now:e
1163      {
1164          piton.ComputeRange
1165              ( '\l_tmpa_str' , '\l_tmpb_str' , '\l_@@_file_name_str' )
1166      }
1167  }

```

10.2.8 The styles

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

```
1168 \NewDocumentCommand { \PitonStyle } { m }
1169   {
1170     \cs_if_exist_use:cF { pitonStyle _ \l_piton_language_str _ #1 }
1171     { \use:c { pitonStyle _ #1 } }
1172   }

1173 \NewDocumentCommand { \SetPitonStyle } { O {} m }
1174   {
1175     \str_clear_new:N \l_@@_SetPitonStyle_option_str
1176     \str_set:Ne \l_@@_SetPitonStyle_option_str { \str_lowercase:n { #1 } }
1177     \str_if_eq:onT \l_@@_SetPitonStyle_option_str { current-language }
1178     { \str_set_eq:NN \l_@@_SetPitonStyle_option_str \l_piton_language_str }
1179     \keys_set:nn { piton / Styles } { #2 }
1180   }

1181 \cs_new_protected:Npn \@@_math_scantokens:n #1
1182   { \normalfont \scantextokens { \begin{math} #1 \end{math} } }

1183 \clist_new:N \g_@@_styles_clist
1184 \clist_gset:Nn \g_@@_styles_clist
1185   {
1186     Comment ,
1187     Comment.LaTeX ,
1188     Discard ,
1189     Exception ,
1190     FormattingType ,
1191     Identifier.Internal ,
1192     Identifier ,
1193     InitialValues ,
1194     Interpol.Inside ,
1195     Keyword ,
1196     Keyword.Governing ,
1197     Keyword.Constant ,
1198     Keyword2 ,
1199     Keyword3 ,
1200     Keyword4 ,
1201     Keyword5 ,
1202     Keyword6 ,
1203     Keyword7 ,
1204     Keyword8 ,
1205     Keyword9 ,
1206     Name.Builtin ,
1207     Name.Class ,
1208     Name.Constructor ,
1209     Name.Decorator ,
1210     Name.Field ,
1211     Name.Function ,
1212     Name.Module ,
1213     Name.Namespace ,
1214     Name.Table ,
1215     Name.Type ,
1216     Number ,
1217     Number.Internal ,
1218     Operator ,
1219     Operator.Word ,
1220     Preproc ,
1221     Prompt ,
1222     String.Doc ,
1223     String.Interpol ,
```

```

1224 String.Long ,
1225 String.Long.Internal ,
1226 String.Short ,
1227 String.Short.Internal ,
1228 Tag ,
1229 TypeParameter ,
1230 UserFunction ,

```

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

```
1231 TypeExpression ,
```

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

```

1232 Directive
1233 }

1234
1235 \clist_map_inline:Nn \g_@@_styles_clist
1236 {
1237 \keys_define:nn { piton / Styles }
1238 {
1239 #1 .value_required:n = true ,
1240 #1 .code:n =
1241 \tl_set:cn
1242 {
1243 pitonStyle _
1244 \str_if_empty:NF \l_@@_SetPitonStyle_option_str
1245 { \l_@@_SetPitonStyle_option_str _ }
1246 #1
1247 }
1248 { ##1 }
1249 }
1250 }
1251
1252 \keys_define:nn { piton / Styles }
1253 {
1254 String .meta:n = { String.Long = #1 , String.Short = #1 } ,
1255 Comment.Math .tl_set:c = pitonStyle _ Comment.Math ,
1256 unknown .code:n =
1257 \@@_error:n { Unknown~key~for~SetPitonStyle }
1258 }

1259 \SetPitonStyle[OCaml]
1260 {
1261 TypeExpression =
1262 \SetPitonStyle { Identifier = \PitonStyle { Name.Type } }
1263 \@@_piton:n ,
1264 }
```

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.

```
1265 \clist_gput_left:Nn \g_@@_styles_clist { String }
```

Of course, we sort that clist.

```

1266 \clist_gsort:Nn \g_@@_styles_clist
1267 {
1268 \str_compare:nNnTF { #1 } < { #2 }
1269 \sort_return_same:
1270 \sort_return_swapped:
1271 }
```

```

1272 % \bool_new:N \l_@@_break_strings_anywhere_bool
1273 \cs_set_eq:NN \@@_break_strings_anywhere:n \prg_do_nothing:
1274
1275 \cs_set_eq:NN \@@_break_numbers_anywhere:n \prg_do_nothing:
1276
1277 \cs_new_protected:Npn \@@_actually_break_anywhere:n #1
1278 {
1279     \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`.

```

1280     \regex_replace_all:nnN { \x20 } { \c{space} } \l_tmpa_tl
1281     \seq_clear:N \l_tmpa_seq % added 2025/03/03
1282     \tl_map_inline:Nn \l_tmpa_tl
1283         { \seq_put_right:Nn \l_tmpa_seq { ##1 } }
1284     \seq_use:Nn \l_tmpa_seq { \ - }
1285 }

```

```

1286 \cs_new_protected:Npn \@@_string_long:n #1
1287 {
1288     \PitonStyle { String.Long }
1289     {
1290         \bool_if:NT \l_@@_break_strings_anywhere_bool
1291             { \@@_actually_break_anywhere:n }
1292             { #1 }
1293     }
1294 }
1295 \cs_new_protected:Npn \@@_string_short:n #1
1296 {
1297     \PitonStyle { String.Short }
1298     {
1299         \bool_if:NT \l_@@_break_strings_anywhere_bool
1300             { \@@_actually_break_anywhere:n }
1301             { #1 }
1302     }
1303 }
1304 \cs_new_protected:Npn \@@_number:n #1
1305 {
1306     \PitonStyle { Number }
1307     {
1308         \bool_if:NT \l_@@_break_numbers_anywhere_bool
1309             { \@@_actually_break_anywhere:n }
1310             { #1 }
1311     }
1312 }

```

10.2.9 The initial styles

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

```

1313 \SetPitonStyle
1314 {
1315     Comment          = \color[HTML]{0099FF} \itshape ,
1316     Exception        = \color[HTML]{CC0000} ,
1317     Keyword          = \color[HTML]{006699} \bfseries ,
1318     Keyword.Governing = \color[HTML]{006699} \bfseries ,
1319     Keyword.Constant = \color[HTML]{006699} \bfseries ,
1320     Name.Builtin      = \color[HTML]{336666} ,
1321     Name.Decorator    = \color[HTML]{9999FF},
1322     Name.Class        = \color[HTML]{00AA88} \bfseries ,
1323     Name.Function     = \color[HTML]{CC00FF} ,

```

```

1324 Name.Namespace          = \color[HTML]{00CCFF} ,
1325 Name.Constructor        = \color[HTML]{006000} \bfseries ,
1326 Name.Field              = \color[HTML]{AA6600} ,
1327 Name.Module             = \color[HTML]{0060A0} \bfseries ,
1328 Name.Table              = \color[HTML]{309030} ,
1329 Number                  = \color[HTML]{FF6600} ,
1330 Number.Internal         = \texttt{@@_number:n} ,
1331 Operator                = \color[HTML]{555555} ,
1332 Operator.Word           = \bfseries ,
1333 String                  = \color[HTML]{CC3300} ,
1334 String.Long.Internal   = \texttt{@@_string_long:n} ,
1335 String.Short.Internal  = \texttt{@@_string_short:n} ,
1336 String.Doc              = \color[HTML]{CC3300} \itshape ,
1337 String.Interpol          = \color[HTML]{AA0000} ,
1338 Comment.LaTeX            = \normalfont \color[rgb]{.468,.532,.6} ,
1339 Name.Type               = \color[HTML]{336666} ,
1340 InitialValues           = \texttt{@@_piton:n} ,
1341 Interpol.Inside         = \l_@@_font_command_tl \texttt{@@_piton:n} ,
1342 TypeParameter            = \color[HTML]{336666} \itshape ,
1343 Preproc                 = \color[HTML]{AA6600} \slshape ,

```

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

```

1344 Identifier.Internal     = \texttt{@@_identifier:n} ,
1345 Identifier              = ,
1346 Directive               = \color[HTML]{AA6600} ,
1347 Tag                     = \colorbox{gray!10},
1348 UserFunction            = \PitonStyle{Identifier} ,
1349 Prompt                  = ,
1350 Discard                 = \use_none:n
1351 }

```

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].

```

1352 \hook_gput_code:nnn { begindocument } { . }
1353 {
1354   \bool_if:NT \g_@@_math_comments_bool
1355     { \SetPitonStyle { Comment.Math = \texttt{@@_math_scantokens:n} } }
1356 }

```

10.2.10 Highlighting some identifiers

```

1357 \NewDocumentCommand { \SetPitonIdentifier } { o m m }
1358 {
1359   \clist_set:Nn \l_tmpa_clist { #2 }
1360   \tl_if_no_value:nTF { #1 }
1361   {
1362     \clist_map_inline:Nn \l_tmpa_clist
1363       { \cs_set:cpn { PitonIdentifier _ ##1 } { #3 } }
1364   }
1365   {
1366     \str_set:Ne \l_tmpa_str { \str_lowercase:n { #1 } }
1367     \str_if_eq:onT \l_tmpa_str { current-language }
1368       { \str_set_eq:NN \l_tmpa_str \l_piton_language_str }
1369     \clist_map_inline:Nn \l_tmpa_clist
1370       { \cs_set:cpn { PitonIdentifier _ \l_tmpa_str _ ##1 } { #3 } }
1371   }
1372 }

```

```

1373 \cs_new_protected:Npn \@@_identifier:n #1
1374 {
1375     \cs_if_exist_use:cF { PitonIdentifier _ \l_piton_language_str _ #1 }
1376     {
1377         \cs_if_exist_use:cF { PitonIdentifier _ #1 }
1378         { \PitonStyle { Identifier } }
1379     }
1380     { #1 }
1381 }

```

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`).

```

1382 \cs_new_protected:cpn { pitonStyle _ Name.Function.Internal } #1
1383 {

```

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

```

1384 { \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}`.

```

1385 \cs_gset_protected:cpn { PitonIdentifier _ \l_piton_language_str _ #1 }
1386 { \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`**.

```

1387 \seq_if_exist:cF { g_@@_functions _ \l_piton_language_str _ seq }
1388 { \seq_new:c { g_@@_functions _ \l_piton_language_str _ seq } }
1389 \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.

```

1390 \seq_if_in:NoF \g_@@_languages_seq \l_piton_language_str
1391 { \seq_gput_left:No \g_@@_languages_seq \l_piton_language_str }
1392 }

```

```

1393 \NewDocumentCommand \PitonClearUserFunctions { ! o }
1394 {
1395     \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.

```

1396 { \@@_clear_all_functions: }
1397 { \@@_clear_list_functions:n { #1 } }
1398 }

```

```

1399 \cs_new_protected:Npn \@@_clear_list_functions:n #1
1400 {
1401     \clist_set:Nn \l_tmpa_clist { #1 }
1402     \clist_map_function:NN \l_tmpa_clist \@@_clear_functions_i:n
1403     \clist_map_inline:nn { #1 }
1404     { \seq_gremove_all:Nn \g_@@_languages_seq { ##1 } }
1405 }

```

```

1406 \cs_new_protected:Npn \@@_clear_functions_i:n #1
1407 { \@@_clear_functions_i: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).

```

1408 \cs_generate_variant:Nn \@@_clear_functions_i:n { e }
1409 \cs_new_protected:Npn \@@_clear_functions_i:n #1
1410 {

```

```

1411 \seq_if_exist:cT { g_@@_functions _ #1 _ seq }
1412 {
1413     \seq_map_inline:cn { g_@@_functions _ #1 _ seq }
1414     { \cs_undefine:c { PitonIdentifier _ #1 _ ##1} }
1415     \seq_gclear:c { g_@@_functions _ #1 _ seq }
1416 }
1417 }

1418 \cs_new_protected:Npn \@@_clear_functions:n #1
1419 {
1420     \@@_clear_functions_i:n { #1 }
1421     \seq_gremove_all:Nn \g_@@_languages_seq { #1 }
1422 }

```

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

```

1423 \cs_new_protected:Npn \@@_clear_all_functions:
1424 {
1425     \seq_map_function:NN \g_@@_languages_seq \@@_clear_functions_i:n
1426     \seq_gclear:N \g_@@_languages_seq
1427 }

```

10.2.11 Security

```

1428 \AddToHook { env / piton / begin }
1429 { \@@_fatal:n { No~environment~piton } }
1430
1431 \msg_new:nnn { piton } { No~environment~piton }
1432 {
1433     There~is~no~environment~piton!\\
1434     There~is~an~environment~{Piton}~and~a~command~
1435     \token_to_str:N \piton\ but~there~is~no~environment~
1436     {piton}.~This~error~is~fatal.
1437 }

```

10.2.12 The error messages of the package

```

1438 \@@_msg_new:nn { Language~not~defined }
1439 {
1440     Language~not~defined \\
1441     The~language~'\l_tmpa_tl'~has~not~been~defined~previously.\\
1442     If~you~go~on,~your~command~\token_to_str:N \NewPitonLanguage\\
1443     will~be~ignored.
1444 }

1445 \@@_msg_new:nn { bad~version~of~piton.lua }
1446 {
1447     Bad~number~version~of~'piton.lua'\\
1448     The~file~'piton.lua'~loaded~has~not~the~same~number~of~
1449     version~as~the~file~'piton.sty'.~You~can~go~on~but~you~should~
1450     address~that~issue.
1451 }

1452 \@@_msg_new:nn { Unknown~key~NewPitonLanguage }
1453 {
1454     Unknown~key~for~\token_to_str:N \NewPitonLanguage.\\
1455     The~key~'\l_keys_key_str'~is~unknown.\\
1456     This~key~will~be~ignored.\\
1457 }

1458 \@@_msg_new:nn { Unknown~key~for~SetPitonStyle }
1459 {
1460     The~style~'\l_keys_key_str'~is~unknown.\\
1461     This~key~will~be~ignored.\\
1462     The~available~styles~are~(in~alphabetic~order):~

```

```

1463   \clist_use:Nnnn \g_@@_styles_clist { ~and~ } { ,~ } { ~and~ }.
1464 }
1465 \@@_msg_new:nn { Invalid-key }
1466 {
1467   Wrong-use-of-key.\\
1468   You-can't-use-the-key~'\l_keys_key_str'~here.\\
1469   That-key-will-be-ignored.
1470 }
1471 \@@_msg_new:nn { Unknown-key-for-line-numbers }
1472 {
1473   Unknown-key. \\
1474   The-key-'line-numbers' / \l_keys_key_str'~is~unknown.\\
1475   The-available-keys-of-the-family-'line-numbers'~are~(in-
1476   alphabetic-order):~\\
1477   absolute,~false,~label-empty-lines,~resume,~skip-empty-lines,~
1478   sep,~start-and-true.\\
1479   That-key-will-be-ignored.
1480 }
1481 \@@_msg_new:nn { Unknown-key-for-marker }
1482 {
1483   Unknown-key. \\
1484   The-key-'marker' / \l_keys_key_str'~is~unknown.\\
1485   The-available-keys-of-the-family-'marker'~are~(in-
1486   alphabetic-order):~ beginning,~end~and~include-lines.\\
1487   That-key-will-be-ignored.
1488 }
1489 \@@_msg_new:nn { bad-range-specification }
1490 {
1491   Incompatible-keys.\\
1492   You-can't-specify-the-range-of-lines-to-include-by-using-both-
1493   markers-and-explicit-number-of-lines.\\
1494   Your-whole-file~'\l_@@_file_name_str'~will~be~included.
1495 }
1496 \cs_new_nopar:Nn \@@_thepage:
1497 {
1498   \thepage
1499   \cs_if_exist:NT \insertframenumber
1500   {
1501     ~(frame~\insertframenumber
1502     \cs_if_exist:NT \beamer@slidenumber { ,~slide-\insertslidenumber }
1503   )
1504 }
1505 }
```

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`.

```

1506 \@@_msg_new:nn { SyntaxError }
1507 {
1508   Syntax-Error-on-page~\@@_thepage:.\\
1509   Your-code-of-the-language~'\l_piton_language_str'~is~not~
1510   syntactically-correct.\\
1511   It~won't~be~printed~in~the~PDF~file.
1512 }
1513 \@@_msg_new:nn { FileError }
1514 {
1515   File-Error.\\
1516   It's-not-possible~to~write~on~the~file~'\l_@@_write_str'.\\
1517   \sys_if_shell_unrestricted:F { Be~sure~to~compile~with~'-shell-escape'.\\ }
1518   If~you~go~on,~nothing~will~be~written~on~the~file.
1519 }
```

```

1520 \@@_msg_new:nn { begin-marker-not-found }
1521 {
1522     Marker-not-found.\\
1523     The~range~'\l_@@_begin_range_str'~provided~to~the~
1524     command~\token_to_str:N \PitonInputFile\ has~not~been~found.~
1525     The~whole~file~'\l_@@_file_name_str'~will~be~inserted.
1526 }
1527 \@@_msg_new:nn { end-marker-not-found }
1528 {
1529     Marker-not-found.\\
1530     The~marker~of~end~of~the~range~'\l_@@_end_range_str'~
1531     provided~to~the~command~\token_to_str:N \PitonInputFile\
1532     has~not~been~found.~The~file~'\l_@@_file_name_str'~will~
1533     be~inserted~till~the~end.
1534 }
1535 \@@_msg_new:nn { Unknown-file }
1536 {
1537     Unknown-file. \\
1538     The~file~'#1'~is~unknown.\\
1539     Your~command~\token_to_str:N \PitonInputFile\ will~be~discarded.
1540 }
1541 \@@_msg_new:nnn { Unknown-key-for-PitonOptions }
1542 {
1543     Unknown-key. \\
1544     The~key~'\l_keys_key_str'~is~unknown~for~\token_to_str:N \PitonOptions.~
1545     It~will~be~ignored.\\
1546     For~a~list~of~the~available~keys,~type~H~<return>.
1547 }
1548 {
1549     The~available~keys~are~(in~alphabetic~order):~
1550     auto-gobble,~
1551     background-color,~
1552     begin-range,~
1553     break-lines,~
1554     break-lines-in-piton,~
1555     break-lines-in-Piton,~
1556     break-numbers-anywhere,~
1557     break-strings-anywhere,~
1558     continuation-symbol,~
1559     continuation-symbol-on-indentation,~
1560     detected-beamer-commands,~
1561     detected-beamer-environments,~
1562     detected-commands,~
1563     end-of-broken-line,~
1564     end-range,~
1565     env-gobble,~
1566     env-used-by-split,~
1567     font-command,~
1568     gobble,~
1569     indent-broken-lines,~
1570     language,~
1571     left-margin,~
1572     line-numbers/,~
1573     marker/,~
1574     math-comments,~
1575     path,~
1576     path-write,~
1577     prompt-background-color,~
1578     resume,~
1579     show-spaces,~
1580     show-spaces-in-strings,~
1581     splittable,~
1582     splittable-on-empty-lines,~

```

```

1583 split-on-empty-lines,~
1584 split-separation,~
1585 tabs-auto-gobble,~
1586 tab-size,~
1587 width-and-write.
1588 }

1589 \@@_msg_new:nn { label-with-lines-numbers }
1590 {
1591   You~can't~use~the~command~\token_to_str:N \label\
1592   because~the~key~'line-numbers'~is~not~active.\\
1593   If~you~go~on,~that~command~will~be~ignored.
1594 }

1595 \@@_msg_new:nn { overlay~without~beamer }
1596 {
1597   You~can't~use~an~argument~<...>~for~your~command~\\
1598   \token_to_str:N \PitonInputFile\ because~you~are~not~\\
1599   in~Beamer.\\
1600   If~you~go~on,~that~argument~will~be~ignored.
1601 }

```

10.2.13 We load piton.lua

```

1602 \cs_new_protected:Npn \@@_test_version:n #1
1603 {
1604   \str_if_eq:onF \PitonFileVersion { #1 }
1605   { \@@_error:n { bad~version~of~piton.lua } }
1606 }

1607 \hook_gput_code:nnn { begindocument } { . }
1608 {
1609   \lua_now:n
1610   {
1611     require ( "piton" )
1612     tex.sprint ( luatexbase.catcodetables.CatcodeTableExpl ,
1613                 "\\\@_test_version:n {" .. piton_version .. "}" )
1614   }
1615 }

```

10.2.14 Detected commands

```

1616 \ExplSyntaxOff
1617 \begin{luacode*}
1618 lpeg.locale(lpeg)
1619 local P , alpha , C , space , S , V
1620   = lpeg.P , lpeg.alpha , lpeg.C , lpeg.space , lpeg.S , lpeg.V
1621 local add
1622 function add(...)
1623   local s = P ( false )
1624   for _ , x in ipairs({...}) do s = s + x end
1625   return s
1626 end
1627 local my_lpeg =
1628   P { "E" ,
1629       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.

```

1630   F = space ^ 0 * ( ( alpha ^ 1 ) / "\\\%0" ) * space ^ 0
1631 }
```

```

1632     function piton.addDetectedCommands ( key_value )
1633         piton.DetectedCommands = piton.DetectedCommands + my_lpeg : match ( key_value )
1634     end
1635     function piton.addBeamerCommands( key_value )
1636         piton.BeamerCommands
1637             = piton.BeamerCommands + my_lpeg : match ( key_value )
1638     end
1639     for _ , v in ipairs ( { 'uncover', 'only',
1640                         'visible', 'invisible', 'alert', 'action' } ) do
1641         piton.addBeamerCommands(v)
1642     end
1643     local insert
1644     function insert(x)
1645         local s = piton.beamer_environments
1646         table.insert(s,x)
1647         return s
1648     end
1649     local my_lpeg_bis =
1650         P { "E" ,
1651             E = ( V "F" * ( "," * V "F" ) ^ 0 ) / insert ,
1652             F = space ^ 0 * ( alpha ^ 1 ) * space ^ 0
1653         }
1654     function piton.addBeamerEnvironments( key_value )
1655         piton.beamer_environments = my_lpeg_bis : match ( key_value )
1656     end
1657 \end{luacode*}
1658 \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`.

```

1659 (*LUA)
1660 piton.comment_latex = piton.comment_latex or ">"
1661 piton.comment_latex = "#" .. piton.comment_latex
1662 local sprintL3
1663 function sprintL3 ( s )
1664     tex.print ( luatexbase.catcodetables.expl , s )
1665 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.

```

1666 local P, S, V, C, Ct, Cc = lpeg.P, lpeg.S, lpeg.V, lpeg.C, lpeg.Ct, lpeg.Cc
1667 local Cs, Cg, Cmt, Cb = lpeg.Cs, lpeg.Cg, lpeg.Cmt, lpeg.Cb
1668 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”).

```

1669 local Q
1670 function Q ( pattern )
1671     return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
1672 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.

```
1673 local L
1674 function L ( pattern ) return
1675   Ct ( C ( pattern ) )
1676 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.

```
1677 local Lc
1678 function Lc ( string ) return
1679   Cc ( { luatexbase.catcodetables.expl , string } )
1680 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)

```
1681 e
1682 local K
1683 function K ( style , pattern ) return
1684   Lc ( [[ {\PitonStyle{}} .. style .. "}{"] )
1685   * Q ( pattern )
1686   * Lc "}}"
1687 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.

```
1688 local WithStyle
1689 function WithStyle ( style , pattern ) return
1690   Ct ( Cc "Open" * Cc ( [[{\PitonStyle{}} .. style .. "}{"] ) * Cc "}}"
1691   * pattern
1692   * Ct ( Cc "Close" )
1693 end
```

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

```
1694 Escape = P ( false )
1695 EscapeClean = P ( false )
1696 if piton.begin_escape then
1697   Escape =
1698     P ( piton.begin_escape )
1699     * L ( ( 1 - P ( piton.end_escape ) ) ^ 1 )
1700     * 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).

```
1701 EscapeClean =
1702   P ( piton.begin_escape )
```

```

1703     * ( 1 - P ( piton.end_escape ) ) ^ 1
1704     * P ( piton.end_escape )
1705 end

1706 EscapeMath = P ( false )
1707 if piton.begin_escape_math then
1708   EscapeMath =
1709   P ( piton.begin_escape_math )
1710   * Lc "$"
1711   * L ( ( 1 - P(piton.end_escape_math) ) ^ 1 )
1712   * Lc "$"
1713   * P ( piton.end_escape_math )
1714 end

```

The following line is mandatory.

```
1715 lpeg.locale(lpeg)
```

The basic syntactic LPEG

```

1716 local alpha , digit = lpeg.alpha , lpeg.digit
1717 local space = P " "

```

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

```

1718 local letter = alpha + "_" + "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "í" + "î"
1719           + "ô" + "û" + "ü" + "Ã" + "À" + "Ç" + "É" + "È" + "Ê" + "Ë"
1720           + "Ĵ" + "Î" + "Ô" + "Û" + "Ü"
1721
1722 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).

```
1723 local identifier = letter * alphanum ^ 0
```

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

```
1724 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).

```

1725 local Number =
1726   K ( 'Number.Internal' ,
1727     ( digit ^ 1 * P "." * # ( 1 - P "." ) * digit ^ 0
1728       + digit ^ 0 * P "." * digit ^ 1
1729       + digit ^ 1 )
1730     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
1731     + digit ^ 1
1732   )

```

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?).

```
1733 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`.

```

1734 if piton.begin_escape then
1735   lpeg_central = lpeg_central - piton.begin_escape
1736 end
1737 if piton.begin_escape_math then
1738   lpeg_central = lpeg_central - piton.begin_escape_math
1739 end
1740 local Word = Q ( lpeg_central ^ 1 )

1741 local Space = Q " " ^ 1
1742
1743 local SkipSpace = Q " " ^ 0
1744
1745 local Punct = Q ( S ",;:;" )
1746
1747 local Tab = "\t" * Lc [[ \@@_tab: ]]
```

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

```

1748 local SpaceIndentation = Lc [[ \@@_leading_space: ]] * Q " "
1749 local Delim = Q ( S "[({})]" )
```

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

```
1750 local SpaceInString = space * Lc [[ \l@@_space_in_string_t1 ]]
```

Several tools for the construction of the main LPEG

```

1751 local LPEG0 = { }
1752 local LPEG1 = { }
1753 local LPEG2 = { }
1754 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*.

```

1755 local Compute_braces
1756 function Compute_braces ( lpeg_string ) return
1757   P { "E" ,
1758     E =
1759     (
1760       "{" * V "E" * "}"
1761       +
1762       lpeg_string
1763       +
1764       ( 1 - S "{}" )
1765     ) ^ 0
1766   }
1767 end
```

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

```

1768 local Compute_DetectedCommands
1769 function Compute_DetectedCommands ( lang , braces ) return
1770   Ct (
1771     Cc "Open"
1772       * C ( piton.DetectedCommands * space ^ 0 * P "{" )
1773       * Cc "}"
1774   )
1775   * ( braces
1776     / ( function ( s )
1777       if s ~= '' then return
1778         LPEG1[lang] : match ( s )
1779       end
1780     end )
1781   )
1782   * P "}"
1783   * Ct ( Cc "Close" )
1784 end

1785 local Compute_LPEG_cleaner
1786 function Compute_LPEG_cleaner ( lang , braces ) return
1787   Ct ( ( piton.DetectedCommands * "{"
1788     * ( braces
1789       / ( function ( s )
1790         if s ~= '' then return
1791           LPEG_cleaner[lang] : match ( s )
1792         end
1793       end )
1794     )
1795     * "}"
1796   + EscapeClean
1797   + C ( P ( 1 ) )
1798 ) ^ 0 ) / table.concat
1799 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`.

```

1800 local ParseAgain
1801 function ParseAgain ( code )
1802   if code ~= '' then return

```

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

```

1803   LPEG1[piton.language] : match ( code )
1804 end
1805 end

```

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

```

1806 local Beamer = P ( false )
1807 local BeamerBeginEnvironments = P ( true )
1808 local BeamerEndEnvironments = P ( true )
1809 piton.BeamerEnvironments = P ( false )
1810 for _ , x in ipairs ( piton.beamer_environments ) do
1811   piton.BeamerEnvironments = piton.BeamerEnvironments + x
1812 end

```

```

1813 BeamerBeginEnvironments =
1814   ( space ^ 0 *
1815     L
1816     (
1817       P [[\begin{}]] * piton.BeamerEnvironments * "}"
1818       * ("<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
1819     )
1820   * "\r"
1821 ) ^ 0

1822 BeamerEndEnvironments =
1823   ( space ^ 0 *
1824     L ( P [[\end{}]] * piton.BeamerEnvironments * "}" )
1825     * "\r"
1826 ) ^ 0

```

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

```

1827 local Compute_Beamer
1828 function Compute_Beamer ( lang , braces )

```

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

```

1829   local lpeg = L ( P [[\pause]] * ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1 )
1830   lpeg = lpeg +
1831     Ct ( Cc "Open"
1832       * C ( piton.BeamerCommands
1833         * ("<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
1834         * P "{"
1835       )
1836     * Cc "}"
1837   )
1838   * ( braces /
1839     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1840   * "}"
1841   * Ct ( Cc "Close" )

```

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

```

1842   lpeg = lpeg +
1843     L ( P [[\alt]] * "<" * ( 1 - P ">" ) ^ 0 * ">{"
1844   * ( braces /
1845     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1846   * L ( P "}{" )
1847   * ( braces /
1848     ( function ( s ) if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1849   * L ( P "}" )

```

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

```

1850   lpeg = lpeg +
1851     L ( P [[\temporal]] * "<" * ( 1 - P ">" ) ^ 0 * ">{"
1852   * ( braces
1853     / ( function ( s )
1854       if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1855   * L ( P "}{" )
1856   * ( braces
1857     / ( function ( s )
1858       if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1859   * L ( P "}{" )
1860   * ( braces
1861     / ( function ( s )
1862       if s ~= '' then return LPEG1[lang] : match ( s ) end end )
1863   * L ( P "}" )

```

Now, the environments of Beamer.

```
1864 for _, x in ipairs ( piton.beamer_environments ) do
1865   lpeg = lpeg +
1866   Ct ( Cc "Open"
1867     * C (
1868       P ( [ [\begin{[]} .. x .. \}] )
1869       * ( < * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
1870     )
1871     * Cc ( [ [\end{[]} ] .. x .. \} ] )
1872   )
1873   * (
1874     ( ( 1 - P ( [ [\end{[]} ] .. x .. \} ] ) ^ 0 )
1875     / ( function ( s )
1876       if s ~= '' then return
1877         LPEG1[lang] : match ( s )
1878       end
1879     end )
1880   )
1881   * P ( [ [\end{[]} ] .. x .. \} ] )
1882   * Ct ( Cc "Close" )
1883 end
```

Now, you can return the value we have computed.

```
1884   return lpeg
1885 end
```

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

```
1886 local CommentMath =
1887   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).

```
1888 local PromptHastyDetection =
1889   ( # ( 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).

```
1890 local Prompt =
1891   K ( 'Prompt' , ( ( P "">>>" + "..." ) * P " " ^ -1 + P ( true ) ) ) ^ -1
```

The `P (true)` at the end is mandatory because we want the style to be *always* applied, even with an empty argument, in order, for example to add a “false” prompt marker with the tuning:

```
\SetPitonStyle{ Prompt = >>>\space }
```

The following LPEG EOL is for the end of lines.

```
1892 local EOL =
1893   P "\r"
1894   *
1895   (
1896     space ^ 0 * -1
1897     +
```

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.

```

1898 Ct (
1899   Cc "EOL"
1900   *
1901   Ct ( Lc [[ @_end_line: ]]
1902     * BeamerEndEnvironments
1903     *
1904   (

```

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:`).

```

1905           -1
1906           +
1907           BeamerBeginEnvironments
1908           * PromptHastyDetection
1909           * Lc [[ @_newline:@_begin_line: ]]
1910           * Prompt
1911         )
1912       )
1913     )
1914   )
1915   * ( 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`).

```

1916 local CommentLaTeX =
1917   P ( piton.comment_latex )
1918   * Lc [[\PitonStyle{Comment.LaTeX}{\ignorespaces}]]
1919   * L ( ( 1 - P "\r" ) ^ 0 )
1920   * Lc "}"
1921   * ( EOL + -1 )

```

10.3.2 The language Python

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

```
1922 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.

```

1923 local Operator =
1924   K ( 'Operator' ,
1925     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":" + "//" + "**"
1926     + S "-~+/*%=<>&@"
1927
1928 local OperatorWord =
1929   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`.

```

1930 local For = K ( 'Keyword' , P "for" )
1931   * Space
1932   * Identifier

```

³⁵Remember 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:`

```

1933     * Space
1934     * K ( 'Keyword' , P "in" )
1935
1936 local Keyword =
1937     K ( 'Keyword' ,
1938         P "assert" + "as" + "break" + "case" + "class" + "continue" + "def" +
1939         "del" + "elif" + "else" + "except" + "exec" + "finally" + "for" + "from" +
1940         "global" + "if" + "import" + "lambda" + "non local" + "pass" + "return" +
1941         "try" + "while" + "with" + "yield" + "yield from" )
1942     + K ( 'Keyword.Constant' , P "True" + "False" + "None" )
1943
1944 local Builtin =
1945     K ( 'Name.Builtin' ,
1946         P "__import__" + "abs" + "all" + "any" + "bin" + "bool" + "bytearray" +
1947         "bytes" + "chr" + "classmethod" + "compile" + "complex" + "delattr" +
1948         "dict" + "dir" + "divmod" + "enumerate" + "eval" + "filter" + "float" +
1949         "format" + "frozenset" + "getattr" + "globals" + "hasattr" + "hash" +
1950         "hex" + "id" + "input" + "int" + "isinstance" + "issubclass" + "iter" +
1951         "len" + "list" + "locals" + "map" + "max" + "memoryview" + "min" + "next"
1952         + "object" + "oct" + "open" + "ord" + "pow" + "print" + "property" +
1953         "range" + "repr" + "reversed" + "round" + "set" + "setattr" + "slice" +
1954         "sorted" + "staticmethod" + "str" + "sum" + "super" + "tuple" + "type" +
1955         "vars" + "zip" )
1956
1957 local Exception =
1958     K ( 'Exception' ,
1959         P "ArithError" + "AssertionError" + "AttributeError" +
1960         "BaseException" + "BufferError" + "BytesWarning" + "DeprecationWarning" +
1961         "EOFError" + "EnvironmentError" + "Exception" + "FloatingPointError" +
1962         "FutureWarning" + "GeneratorExit" + "IOError" + "ImportError" +
1963         "ImportWarning" + "IndentationError" + "IndexError" + "KeyError" +
1964         "KeyboardInterrupt" + "LookupError" + "MemoryError" + "NameError" +
1965         "NotImplementedError" + "OSError" + "OverflowError" +
1966         "PendingDeprecationWarning" + "ReferenceError" + "ResourceWarning" +
1967         "RuntimeError" + "RuntimeWarning" + "StopIteration" + "SyntaxError" +
1968         "SyntaxWarning" + "SystemError" + "SystemExit" + "TabError" + "TypeError"
1969         + "UnboundLocalError" + "UnicodeDecodeError" + "UnicodeEncodeError" +
1970         "UnicodeError" + "UnicodeTranslateError" + "UnicodeWarning" +
1971         "UserWarning" + "ValueError" + "VMSError" + "Warning" + "WindowsError" +
1972         "ZeroDivisionError" + "BlockingIOError" + "ChildProcessError" +
1973         "ConnectionError" + "BrokenPipeError" + "ConnectionAbortedError" +
1974         "ConnectionRefusedError" + "ConnectionResetError" + "FileExistsError" +
1975         "FileNotFoundException" + "InterruptedError" + "IsADirectoryError" +
1976         "NotADirectoryError" + "PermissionError" + "ProcessLookupError" +
1977         "TimeoutError" + "StopAsyncIteration" + "ModuleNotFoundError" +
1978         "RecursionError" )
1979
1980 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.

```
1981     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:`

```
1982     local DefClass =
1983         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`

```

1984 local ImportAs =
1985   K ( 'Keyword' , "import" )
1986   * Space
1987   * K ( 'Name.Namespace' , identifier * ( "." * identifier ) ^ 0 )
1988   *
1989   ( Space * K ( 'Keyword' , "as" ) * Space
1990     * K ( 'Name.Namespace' , identifier ) )
1991   +
1992   ( SkipSpace * Q "," * SkipSpace
1993     * K ( 'Name.Namespace' , identifier ) ) ^ 0
1994 )

```

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`

```

1995 local FromImport =
1996   K ( 'Keyword' , "from" )
1997   * Space * K ( 'Name.Namespace' , identifier )
1998   * 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 instruction³⁶ in that interpolation:

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

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

```

1999 local PercentInterpol =
2000   K ( 'String.Interpol' ,
2001     P "%"
2002     * ( "(" * alphanum ^ 1 * ")" ) ^ -1
2003     * ( S "-#0 +" ) ^ 0
2004     * ( digit ^ 1 + "*" ) ^ -1
2005     * ( "." * ( digit ^ 1 + "*" ) ) ^ -1
2006     * ( S "HLL" ) ^ -1
2007     * S "sdfFeExXorgiGauc%"
2008   )

```

³⁶There 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`.

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

```

2009 local SingleShortString =
210   WithStyle ( 'String.Short.Internal' ,
211
212     Q ( P "f'" + "F'" )
213     *
214       (
215         K ( 'String.Interpol' , "{}" )
216         *
217           K ( 'Interpol.Inside' , ( 1 - S "}:;" ) ^ 0 )
218         *
219           Q ( P ":" * ( 1 - S "}:;" ) ^ 0 ) ^ -1
220         *
221           K ( 'String.Interpol' , "}" )
222         +
223
224         SpaceInString
225         +
226
227         Q ( ( P "\\" + "\\\\" + "{{" + "}}}" + 1 - S " {}}'" ) ^ 1 )
228       ) ^ 0
229     *
230     Q """
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1179
1180
1181
1182
1183
1184
1184
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1207
1208
1209
1210
1211
1212
1213
1214
1215
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1240
1241
1242
1243
1244
1245
1245
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1293
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1303
1304
1305
1306
1307
1307
1308
1309
1310
1311
1312
1313
1313
1314
1315
1316
1316
1317
1318
1319
1319
1320
1321
1322
1323
1323
1324
1325
1326
1326
1327
1328
1329
1329
1330
1331
1332
1333
1333
1334
1335
1336
1336
1337
1338
1339
1339
1340
1341
1342
1343
1343
1344
1345
1346
1346
1347
1348
1349
1349
1350
1351
1352
1353
1353
1354
1355
1356
1356
1357
1358
1359
1359
1360
1361
1362
1363
1363
1364
1365
1366
1366
1367
1368
1369
1369
1370
1371
1372
1372
1373
1374
1375
1375
1376
1377
1378
1378
1379
1380
1381
1381
1382
1383
1384
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1392
1393
1394
1394
1395
1396
1396
1397
1398
1398
1399
1399
1400
1401
1401
1402
1403
1403
1404
1405
1405
1406
1407
1407
1408
1409
1409
1410
1411
1411
1412
1413
1413
1414
1415
1415
1416
1417
1417
1418
1419
1419
1420
1421
1421
1422
1423
1423
1424
1425
1425
1426
1427
1427
1428
1429
1429
1430
1431
1431
1432
1433
1433
1434
1435
1435
1436
1437
1437
1438
1439
1439
1440
1441
1441
1442
1443
1443
1444
1445
1445
1446
1447
1447
1448
1449
1449
1450
1451
1451
1452
1453
1453
1454
1455
1455
1456
1457
1457
1458
1459
1459
1460
1461
1461
1462
1463
1463
1464
1465
1465
1466
1467
1467
1468
1469
1469
1470
1471
1471
1472
1473
1473
1474
1475
1475
1476
1477
1477
1478
1479
1479
1480
1481
1481
1482
1483
1483
1484
1485
1485
1486
1487
1487
1488
1489
1489
1490
1491
1491
1492
1493
1493
1494
1495
1495
1496
1497
1497
1498
1499
1499
1500
1501
1501
1502
1503
1503
1504
1505
1505
1506
1507
1507
1508
1509
1509
1510
1511
1511
1512
1513
1513
1514
1515
1515
1516
1517
1517
1518
1519
1519
1520
1521
1521
1522
1523
1523
1524
1525
1525
1526
1527
1527
1528
1529
1529
1530
1531
1531
1532
1533
1533
1534
1535
1535
1536
1537
1537
1538
1539
1539
1540
1541
1541
1542
1543
1543
1544
1545
1545
1546
1547
1547
1548
1549
1549
1550
1551
1551
1552
1553
1553
1554
1555
1555
1556
1557
1557
1558
1559
1559
1560
1561
1561
1562
1563
1563
1564
1565
1565
1566
1567
1567
1568
1569
1569
1570
1571
1571
1572
1573
1573
1574
1575
1575
1576
1577
1577
1578
1579
1579
1580
1581
1581
1582
1583
1583
1584
1585
1585
1586
1587
1587
1588
1589
1589
1590
1591
1591
1592
1593
1593
1594
1595
1595
1596
1597
1597
1598
1599
1599
1600
1601
1601
1602
1603
1603
1604
1605
1605
1606
1607
1607
1608
1609
1609
1610
1611
1611
1612
1613
1613
1614
1615
1615
1616
1617
1617
1618
1619
1619
1620
1621
1621
1622
1623
1623
1624
1625
1625
1626
1627
1627
1628
1629
1629
1630
1631
1631
1632
1633
1633
1634
1635
1635
1636
1637
1637
1638
1639
1639
1640
1641
1641
1642
1643
1643
1644
1645
1645
1646
1647
1647
1648
1649
1649
1650
1651
1651
1652
1653
1653
1654
1655
1655
1656
1657
1657
1658
1659
1659
1660
1661
1661
1662
1663
1663
1664
1665
1665
1666
1667
1667
1668
1669
1669
1670
1671
1671
1672
1673
1673
1674
1675
1675
1676
1677
1677
1678
1679
1679
1680
1681
1681
1682
1683
1683
1684
1685
1685
1686
1687
1687
1688
1689
1689
1690
1691
1691
1692
1693
1693
1694
1695
1695
1696
1697
1697
1698
1699
1699
1700
1701
1701
1702
1703
1703
1704
1705
1705
1706
1707
1707
1708
1709
1709
1710
1711
1711
1712
1713
1713
1714
1715
1715
1716
1717
1717
1718
1719
1719
1720
1721
1721
1722
1723
1723
1724
1725
1725
1726
1727
1727
1728
1729
1729
1730
1731
1731
1732
1733
1733
1734
1735
1735
1736
1737
1737
1738
1739
1739
1740
1741
1741
1742
1743
1743
1744
1745
1745
1746
1747
1747
1748
1749
1749
1750
1751
1751
1752
1753
1753
1754
1755
1755
1756
1757
1757
1758
1759
1759
1760
1761
1761
1762
1763
1763
1764
1765
1765
1766
1767
1767
1768
1769
1769
1770
1771
1771
1772
1773
1773
1774
1775
1775
1776
1777
1777
1778
1779
1779
1780
1781
1781
1782
1783
1783
1784
1785
1785
1786
1787
1787
1788
1789
1789
1790
1791
1791
1792
1793
1793
1794
1795
1795
1796
1797
1797
1798
1799
1799
1800
1801
1801
1802
1803
1803
1804
1805
1805
1806
1807
1807
1808
1809
1809
1810
1811
1811
1812
1813
1813
1814
1815
1815
1816
1817
1817
1818
1819
1819
1820
1821
1821
1822
1823
1823
1824
1825
1825
1826
1827
1827
1828
1829
1829
1830
1831
1831
1832
1833
1833
1834
1835
1835
1836
1837
1837
1838
1839
1839
1840
1841
1841
1842
1843
1843
1844
1845
1845
1846
1847
1847
1848
1849
1849
1850
1851
1851
1852
1853
1853
1854
1855
1855
1856
1857
1857
1858
1859
1859
1860
1861
1861
1862
1863
1863
1864
1865
1865
1866
1867
1867
1868
1869
1869
1870
1871
1871
1872
1873
1873
1874
1875
1875
1876
1877
1877
1878
1879
1879
1880
1881
1881
1882
1883
1883
1884
1885
1885
1886
1887
1887
1888
1889
1889
1890
1891
1891
1892
1893
1893
1894
1895
1895
1896
1897
1897
1898
1899
1899
1900
1901
1901
1902
1903
1903
1904
1905
1905
1906
1907
1907
1908
1909
1909
1910
1911
1911
1912
1913
1913
1914
1915
1915
1916
1917
1917
1918
1919
1919
1920
1921
1921
1922
1923
1923
1924
1925
1925
1926
1927
1927
1928
1929
1929
1930
1931
1931
1932
1933
1933
1934
1935
1935
1936
1937
1937
1938
1939
1939
1940
1941
1941
1942
1943
1943
1944
1945
1945
1946
1947
1947
1948
1949
1949
1950
1951
1951
1952
1953
1953
1954
1955
1955
1956
1957
1957
1958
1959
1959
1960
1961
1961
1962
1963
1963
1964
1965
1965
1966
1967
1967
1968
1969
1969
1970
1971
1971
1972
1973
1973
1974
1975
1975
1976
1977
1977
1978
1979
1979
1980
1981
1981
1982
1983
1983
1984
1985
1985
1986
1987
1987
1988
1989
1989
1990
1991
1991
1992
1993
1993
1994
1995
1995
1996
1997
1997
1998
1999
1999
2000
2001
2001
2002
2003
2003
2004
2005
2005
2006
2007
2007
2008
2009
2009
2010
2011
2011
2012
2013
2013
2014
2015
2015
2016
2017
2017
2018
2019
2019
2020
2021
2021
2022
2023
2023
2024
2025
2025
2026
2027
2027
2028
2029
2029
2030
2031
2031
2032
2033
2033
2034
2035
2035
2036
2037
2037
2038
2039
2039
2040
2041
2041
2042
2043
2043
2044
2045
2045
2046
2047
2047
2048
2049
2049
2050
2051
2051
2052
2053
2053
2054
2055
2055
2056
2057
2057
2058
2059
2059
2060
2061
2061
2062
2063
2063
2064
2065
2065
2066
2067
2067
2068
2069
2069
2070
2071
2071
2072
2073
2073
2074
2075
2075
2076
2077
2077
2078
2079
2079
2080
2081
2081
2082
2083
2083
2084
2085
2085
2086
2087
2087
2088
2089
2089
2090
2091
2091
2092
2093
2093
2094
2095
209
```

```

2056 Compute_braces
2057 (
2058     ( P "\\" + "r\\"" + "R\\"" + "f\\"" + "F\\"" )
2059     * ( P "\\\\" + 1 - S "\\" ) ^ 0 * "\\""
2060 +
2061     ( P '\' + 'r\' + 'R\' + 'f\' + 'F\' )
2062     * ( P '\\\\' + 1 - S '\') ^ 0 * '\'
2063 )
2064 if piton.beamer then Beamer = Compute_Beamer ( 'python' , braces ) end

```

Detected commands

```
2065 DetectedCommands = Compute_DetectedCommands ( 'python' , braces )
```

LPEG_cleaner

```
2066 LPEG_cleaner.python = Compute_LPEG_cleaner ( 'python' , braces )
```

The long strings

```

2067 local SingleLongString =
2068   WithStyle ( 'String.Long.Internal' ,
2069     ( Q ( S "fF" * P "****" )
2070       *
2071       K ( 'String.Interpol' , "{}" )
2072       * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "****" ) ^ 0 )
2073       * Q ( P ":" * (1 - S "}):\\r" - "****" ) ^ 0 ) ^ -1
2074       * K ( 'String.Interpol' , "}" )
2075     +
2076     Q ( ( 1 - P "****" - S "{}\\r" ) ^ 1 )
2077     +
2078     EOL
2079   ) ^ 0
2080 +
2081   Q ( ( S "rR" ) ^ -1 * "****" )
2082   *
2083     Q ( ( 1 - P "****" - S "\\r%" ) ^ 1 )
2084     +
2085     PercentInterpol
2086     +
2087     P "%"
2088     +
2089     EOL
2090   ) ^ 0
2091 )
2092 * Q "****" )

2093 local DoubleLongString =
2094   WithStyle ( 'String.Long.Internal' ,
2095   (
2096     Q ( S "fF" * "\\\"\\\"\\\"\\\"")
2097     *
2098       K ( 'String.Interpol' , "{}" )
2099       * K ( 'Interpol.Inside' , ( 1 - S "}:\\r" - "\\\"\\\"\\\"\\\"") ^ 0 )
2100       * Q ( ":" * (1 - S "}):\\r" - "\\\"\\\"\\\"\\\"") ^ 0 ) ^ -1
2101       * K ( 'String.Interpol' , "}" )
2102     +
2103     Q ( ( 1 - S "{}\\r" - "\\\"\\\"\\\"\\\"") ^ 1 )
2104     +
2105     EOL
2106   ) ^ 0

```

```

2107     +
2108     Q ( S "rR" ^ -1 * "\\" \\
2109     * (
2110         Q ( ( 1 - P "\\" \\
2111             + \\
2112             PercentInterpol \\
2113             + \\
2114             P "%" \\
2115             + \\
2116             EOL \\
2117         ) ^ 0
2118     )
2119     * Q "\\" \\
2120   )
2121 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`).

```

2122 local StringDoc =
2123   K ( 'String.Doc' , P "r" ^ -1 * "\\" \\
2124     * ( K ( 'String.Doc' , ( 1 - P "\\" \\
2125         * Tab ^ 0
2126     ) ^ 0
2127     * K ( 'String.Doc' , ( 1 - P "\\" \\
2128       * "r" ) ^ 0 * "\\" )

```

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

```

2128 local Comment =
2129   WithStyle
2130   ( 'Comment' ,
2131     Q "#" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
2132   )
2133   * ( 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).

```

2134 local expression =
2135   P { "E" ,
2136     E = ( "" * ( P "\\" + 1 - S "'\r" ) ^ 0 * ""
2137       + "\\" * ( P "\\\\" + 1 - S "\\" \r" ) ^ 0 * "\\" \\
2138       + "{} * V "F" * "}" \\
2139       + "(" * V "F" * ")" \\
2140       + "[" * V "F" * "]"
2141       + ( 1 - S "{}()[]\r," ) ^ 0 ,
2142     F = ( "{} * V "F" * "}" \\
2143       + "(" * V "F" * ")" \\
2144       + "[" * V "F" * "]"
2145       + ( 1 - S "{}()[]\r'" ) ^ 0
2146   }

```

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.`

```

2147 local Params =
2148   P { "E" ,
2149     E = ( V "F" * ( Q "," * V "F" ) ^ 0 ) ^ -1 ,
2150     F = SkipSpace * ( Identifier + Q "*args" + Q "**kwargs" ) * SkipSpace
2151     *
2152       K ( 'InitialValues' , "=" * expression )
2153       + Q ":" * SkipSpace * K ( 'Name.Type' , identifier )
2154     ) ^ -1
2155 }
```

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`...

```

2156 local DefFunction =
2157   K ( 'Keyword' , "def" )
2158   * Space
2159   * K ( 'Name.Function.Internal' , identifier )
2160   * SkipSpace
2161   * Q "(" * Params * Q ")"
2162   * SkipSpace
2163   * ( Q "->" * SkipSpace * K ( 'Name.Type' , identifier ) ) ^ -1
2164   * ( C ( ( 1 - S ":\\r" ) ^ 0 ) / ParseAgain )
2165   * Q ":" 
2166   * ( SkipSpace
2167     * ( EOL + CommentLaTeX + Comment ) -- in all cases, that contains an EOL
2168     * Tab ^ 0
2169     * SkipSpace
2170     * StringDoc ^ 0 -- there may be additional docstrings
2171   ) ^ -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

```
2172 local ExceptionInConsole = Exception * Q ( ( 1 - P "\r" ) ^ 0 ) * EOL
```

The main LPEG for the language Python

```

2173 local EndKeyword
2174   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
2175   EscapeMath + -1
```

First, the main loop :

```

2176 local Main =
2177   space ^ 0 * EOL -- faut-il le mettre en commentaire ?
2178   +
2179   + Space
2180   + Tab
2181   + Escape + EscapeMath
2182   + CommentLaTeX
2183   + Beamer
2184   + DetectedCommands
2185   + LongString
2186   + Comment
2187   + ExceptionInConsole
2188   + Delim
2189   + Operator
```

```

2189     + OperatorWord * EndKeyword
2190     + ShortString
2191     + Punct
2192     + FromImport
2193     + RaiseException
2194     + DefFunction
2195     + DefClass
2196     + For
2197     + Keyword * EndKeyword
2198     + Decorator
2199     + Builtin * EndKeyword
2200     + Identifier
2201     + Number
2202     + Word

```

Here, we must not put local, of course.

```
2203 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.

```

2204 LPEG2.python =
2205   Ct (
2206     ( space ^ 0 * "\r" ) ^ -1
2207     * BeamerBeginEnvironments
2208     * PromptHastyDetection
2209     * Lc [[ \@@_begin_line: ]]
2210     * Prompt
2211     * SpaceIndentation ^ 0
2212     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2213     * -1
2214     * Lc [[ \@@_end_line: ]]
2215   )

```

End of the Lua scope for the language Python.

```
2216 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).

```

2217 do
2218
2219   local SkipSpace = ( Q " " + EOL ) ^ 0
   local Space = ( Q " " + EOL ) ^ 1
2220
2221   local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
2222
2223   if piton.beamer then
2224     Beamer = Compute_Beamer ( 'ocaml' , braces )
2225   end
2226   DetectedCommands = Compute_DetectedCommands ( 'ocaml' , braces )
2227   local Q

```

³⁸Remember 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:`

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

```

2226   function Q ( pattern, strict )
2227     if strict ~= nil then
2228       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2229     else
2230       return Ct ( Cc ( luatexbase.catcodetables.CatcodeTableOther ) * C ( pattern ) )
2231         + Beamer + DetectedCommands + EscapeMath + Escape
2232     end
2233   end

2234   local K
2235   function K ( style , pattern, strict ) return
2236     Lc ( [[ {\PitonStyle[ ]} .. style .. "}{"
2237       * Q ( pattern, strict )
2238       * Lc "}" ]]
2239   end

2240   local WithStyle
2241   function WithStyle ( style , pattern ) return
2242     Ct ( Cc "Open" * Cc ( [[{\PitonStyle[ ]} .. style .. "}{"] * Cc "}" ] )
2243       * (pattern + Beamer + DetectedCommands + EscapeMath + Escape)
2244       * Ct ( Cc "Close" )
2245   end

```

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

```

2246   local balanced_parens =
2247     P { "E" , E = ( "(" * V "E" * ")" + ( 1 - S "()" ) ) ^ 0 }

```

The strings of OCaml

```

2248   local ocaml_string =
2249     P "\""
2250   *
2251     P " "
2252     +
2253     P ( ( 1 - S " \r" ) ^ 1 )
2254     +
2255     EOL -- ?
2256   ) ^ 0
2257   * P "\""

2258   local String =
2259     WithStyle
2260     ( 'String.Long.Internal' ,
2261       Q "\""
2262     *
2263       SpaceInString
2264       +
2265       Q ( ( 1 - S " \r" ) ^ 1 )
2266       +
2267       EOL
2268   ) ^ 0
2269   * Q "\""
2270 )

```

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.

```

2271 local ext = ( R "az" + "_" ) ^ 0
2272 local open = "{" * Cg ( ext , 'init' ) * "|"
2273 local close = "|" * C ( ext ) * "}"
2274 local closeeq =
2275   Cmt ( close * Cb ( 'init' ) ,
2276         function ( s , i , a , b ) return a == b end )

```

The LPEG `QuotedStringBis` will do the second analysis.

```

2277 local QuotedStringBis =
2278   WithStyle ( 'String.Long.Internal' ,
2279   (
2280     Space
2281     +
2282     Q ( ( 1 - S "\r" ) ^ 1 )
2283     +
2284     EOL
2285   ) ^ 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.

```

2286 local QuotedString =
2287   C ( open * ( 1 - closeeq ) ^ 0 * close ) /
2288   ( 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).

```

2289 local comment =
2290   P {
2291     "A" ,
2292     A = Q "(*"
2293       * ( V "A"
2294         + Q ( ( 1 - S "\r$\" - "( * - *)" ) ^ 1 ) -- $
2295         + ocaml_string
2296         + $" * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) * $" -- $
2297         + EOL
2298       ) ^ 0
2299     * Q "*)"
2300   }
2301 local Comment = WithStyle ( 'Comment' , comment )

```

Some standard LPEG

```

2302 local Delim = Q ( P "[|]" + "[|]" + S "[()]" )
2303 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.

```

2304 local cap_identifier = R "AZ" * ( R "az" + R "AZ" + S "_" + digit ) ^ 0

2305 local Constructor =
2306   K ( 'Name.Constructor' ,
2307     Q "`" ^ -1 * cap_identifier

```

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

```

2308   + Q "::"
2309   + Q ( "[" , true ) * SkipSpace * Q ( "]" , true )

```

```

2310 local ModuleType = K ( 'Name.Type' , cap_identifier )

2311 local OperatorWord =
2312   K ( 'Operator.Word' ,
2313       P "asr" + "land" + "lor" + "lsl" + "lxor" + "mod" + "or" + "not" )

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

2314 local governing_keyword = P "and" + "begin" + "class" + "constraint" +
2315   "end" + "external" + "functor" + "include" + "inherit" + "initializer" +
2316   "in" + "let" + "method" + "module" + "object" + "open" + "rec" + "sig" +
2317   "struct" + "type" + "val"

2318 local Keyword =
2319   K ( 'Keyword' ,
2320       P "assert" + "as" + "done" + "downto" + "do" + "else" + "exception"
2321       + "for" + "function" + "fun" + "if" + "lazy" + "match" + "mutable"
2322       + "new" + "of" + "private" + "raise" + "then" + "to" + "try"
2323       + "virtual" + "when" + "while" + "with" )
2324   + K ( 'Keyword.Constant' , P "true" + "false" )
2325   + K ( 'Keyword.Governing' , governing_keyword )

2326 local EndKeyword
2327   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape
2328   + EscapeMath + -1

```

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

```

2329 local identifier = ( R "az" + "_" ) * ( R "az" + R "AZ" + S "_" + digit ) ^ 0
2330   - ( 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`.

```

2331 local Identifier = K ( 'Identifier.Internal' , identifier )

```

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

```

2332 local ocaml_char =
2333   P "!" *
2334   (
2335     ( 1 - S "'\\\" )
2336     + "\\\" "
2337     * ( S "\\\'ntbr \""
2338       + digit * digit * digit
2339       + P "x" * ( digit + R "af" + R "AF" )
2340       * ( digit + R "af" + R "AF" )
2341       * ( digit + R "af" + R "AF" )
2342       + P "o" * R "03" * R "07" * R "07" )
2343   )
2344   * "!"
2345 local Char =
2346   K ( 'String.Short.Internal' , ocaml_char )

```

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

```

2347 local TypeParameter =
2348   K ( 'TypeParameter' ,
2349       "!" * Q"_" ^ -1 * alpha ^ 1 * ( # ( 1 - P "!" ) + -1 ) )

```

The records

```

2350 local expression_for_fields_type =
2351   P { "E" ,
2352     E = ( "{ " * V "F" * "}" *
2353           + "(" * V "F" * ")"
2354           + TypeParameter
2355           + ( 1 - S "{()}[]\r;" ) ) ^ 0 ,
2356     F = ( "{ " * V "F" * "}" *
2357           + "(" * V "F" * ")"
2358           + ( 1 - S "{()}[]\r\"'" ) + TypeParameter ) ^ 0
2359   }
2360
2360 local expression_for_fields_value =
2361   P { "E" ,
2362     E = ( "{ " * V "F" * "}" *
2363           + "(" * V "F" * ")"
2364           + "[" * V "F" * "]"
2365           + ocaml_string + ocaml_char
2366           + ( 1 - S "{()}[];" ) ) ^ 0 ,
2367     F = ( "{ " * V "F" * "}" *
2368           + "(" * V "F" * ")"
2369           + "[" * V "F" * "]"
2370           + ocaml_string + ocaml_char
2371           + ( 1 - S "{()}[]\"'" ) ) ^ 0
2372   }
2373
2373 local OneFieldDefinition =
2374   ( K ( 'Keyword' , "mutable" ) * SkipSpace ) ^ -1
2375   * K ( 'Name.Field' , identifier ) * SkipSpace
2376   * Q ":" * SkipSpace
2377   * K ( 'TypeExpression' , expression_for_fields_type )
2378   * SkipSpace
2379
2379 local OneField =
2380   K ( 'Name.Field' , identifier ) * SkipSpace
2381   * Q "=" * SkipSpace

```

Don't forget the parentheses!

```

2382   * ( C ( expression_for_fields_value ) / ParseAgain )
2383   * SkipSpace

```

The *records*.

```

2384 local RecordVal =
2385   Q "{" * SkipSpace
2386   *
2387   (
2388     OneField * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneField ) ^ 0
2389   )
2390   * SkipSpace
2391   * Q ";" ^ -1
2392   * SkipSpace
2393   * Comment ^ -1
2394   * SkipSpace
2395   * Q "}"
2396 local RecordType =
2397   Q "{" * SkipSpace
2398   *
2399   (
2400     OneFieldDefinition
2401     * ( Q ";" * SkipSpace * ( Comment * SkipSpace ) ^ 0 * OneFieldDefinition ) ^ 0

```

```

2402     )
2403     * SkipSpace
2404     * Q ";" ^ -1
2405     * SkipSpace
2406     * Comment ^ -1
2407     * SkipSpace
2408     * Q "}"
2409 local Record = RecordType + RecordVal

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

2410 local DotNotation =
2411 (
2412   K ( 'Name.Module' , cap_identifier )
2413   * Q "."
2414   * ( Identifier + Constructor + Q "(" + Q "[" + Q "{" ) ^ -1
2415   +
2416   Identifier
2417   * Q "."
2418   * K ( 'Name.Field' , identifier )
2419 )
2420 * ( Q "." * K ( 'Name.Field' , identifier ) ) ^ 0

2421 local Operator =
2422   K ( 'Operator' ,
2423     P "!=" + "<>" + "==" + "<<" + ">>" + "<=" + ">=" + ":" + "||" + "&&" +
2424     "://" + "*" + ";" + "->" + "+." + "-." + "*." + "./"
2425     + S "--+/*%=<>&%|")

2426 local Builtin =
2427   K ( 'Name.Builtin' , P "incr" + "decr" + "fst" + "snd" + "ref" )

2428 local Exception =
2429   K ( 'Exception' ,
2430     P "Division_by_zero" + "End_of_File" + "Failure" + "Invalid_argument" +
2431     "Match_failure" + "Not_found" + "Out_of_memory" + "Stack_overflow" +
2432     "Sys_blocked_io" + "Sys_error" + "Undefined_recursive_module" )

2433 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).

```

2434 local pattern_part =
2435   ( 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).

```

2436 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.

```

2437   ( Q "~" * Identifier * Q ":" * SkipSpace ) ^ -1
2438   *

```

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

```

2439      (
2440          K ( 'Identifier.Internal' , identifier )
2441      +
2442          Q "(" * SkipSpace
2443          * ( C ( pattern_part ) / ParseAgain )
2444          * SkipSpace

```

Of course, the specification of type is optional.

```

2445      * ( Q ":" * K ( 'TypeExpression' , balanced_parens ) * SkipSpace ) ^ -1
2446      * Q ")"
2447  )

```

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

```

2448 local DefFunction =
2449     K ( 'Keyword.Governing' , "let open" )
2450     * Space
2451     * K ( 'Name.Module' , cap_identifier )
2452     +
2453     K ( 'Keyword.Governing' , P "let rec" + "let" + "and" )
2454     * Space
2455     * K ( 'Name.Function.Internal' , identifier )
2456     * Space
2457     *

```

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

```

2458     Q "=" * SkipSpace * K ( 'Keyword' , "function" , true )
2459     +
2460     Argument * ( SkipSpace * Argument ) ^ 0
2461     *
2462         SkipSpace
2463         * Q ":" 
2464         * K ( 'TypeExpression' , ( 1 - P "=" ) ^ 0 )
2465     ) ^ -1
2466 )

```

DefModule

```

2467 local DefModule =
2468     K ( 'Keyword.Governing' , "module" ) * Space
2469     *
2470     (
2471         K ( 'Keyword.Governing' , "type" ) * Space
2472         * K ( 'Name.Type' , cap_identifier )
2473     +
2474         K ( 'Name.Module' , cap_identifier ) * SkipSpace
2475         *
2476         (
2477             Q "(" * SkipSpace
2478             * K ( 'Name.Module' , cap_identifier ) * SkipSpace
2479             * Q ":" * SkipSpace
2480             * K ( 'Name.Type' , cap_identifier ) * SkipSpace
2481             *
2482             (
2483                 Q "," * SkipSpace
2484                 * K ( 'Name.Module' , cap_identifier ) * SkipSpace
2485                 * Q ":" * SkipSpace
2486                 * K ( 'Name.Type' , cap_identifier ) * SkipSpace
2487             ) ^ 0
2488             * Q ")"
2489         ) ^ -1
2490         *
2491         (

```

```

2492     Q "==" * SkipSpace
2493     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
2494     * Q "("
2495     * K ( 'Name.Module' , cap_identifier ) * SkipSpace
2496     *
2497     (
2498         Q ","
2499         *
2500         K ( 'Name.Module' , cap_identifier ) * SkipSpace
2501         ) ^ 0
2502         * Q ")"
2503     ) ^ -1
2504 )
2505 +
2506 K ( 'Keyword.Governing' , P "include" + "open" )
2507 * Space
2508 * K ( 'Name.Module' , cap_identifier )

```

DefType

```

2509 local DefType =
2510     K ( 'Keyword.Governing' , "type" )
2511     * Space
2512     * K ( 'TypeExpression' , Q ( 1 - P "==" - P "+=" ) ^ 1 )
2513     * SkipSpace
2514     * ( Q "+=" + Q "==" )
2515     * SkipSpace
2516     *
2517     RecordType
2518 +

```

The following lines are a suggestion of Y. Salmon.

```

2519     WithStyle
2520     (
2521         'TypeExpression' ,
2522         (
2523             (
2524                 EOL
2525                 + comment
2526                 + Q ( 1
2527                     - P ";;"
2528                     - ( ( Space + EOL ) * governing_keyword * EndKeyword )
2529                 )
2530             ) ^ 0
2531             *
2532             (
2533                 # ( ( Space + EOL ) * governing_keyword * EndKeyword )
2534                 + Q ";;"
2535                 + -1
2536             )
2537         )
2538     )
2539 )

```

The main LPEG for the language OCaml

```

2540 local Main =
2541     space ^ 0 * EOL
2542     + Space
2543     + Tab
2544     + Escape + EscapeMath
2545     + Beamer
2546     + DetectedCommands

```

```

2547      + TypeParameter
2548      + String + QuotedString + Char
2549      + Comment
2550      + Operator

```

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

```

2551      + Q "~" * Identifier * ( Q ":" ) ^ -1
2552      + Q ":" * # ( 1 - P ":" ) * SkipSpace
2553          * K ( 'TypeExpression' , balanced_parens ) * SkipSpace * Q ")"
2554      + Exception
2555      + DefType
2556      + DefFunction
2557      + DefModule
2558      + Record
2559      + Keyword * EndKeyword
2560      + OperatorWord * EndKeyword
2561      + Builtin * EndKeyword
2562      + DotNotation
2563      + Constructor
2564      + Identifier
2565      + Punct
2566      + Delim
2567      + Number
2568      + Word

```

Here, we must not put local, of course.

```
2569  LPEG1.ocaml = Main ^ 0
```

```

2570  LPEG2.ocaml =
2571  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).

```

2572      ( P ":" + Identifier * SkipSpace * Q ":" ) * # ( 1 - P ":" )
2573          * SkipSpace
2574          * K ( 'TypeExpression' , ( 1 - P "\r" ) ^ 0 )
2575      +
2576      ( space ^ 0 * "\r" ) ^ -1
2577      * BeamerBeginEnvironments
2578      * Lc [[ \@@_begin_line: ]]
2579      * SpaceIndentation ^ 0
2580      * ( ( space * Lc [[ \@@_trailing_space: ]] ) ^ 1 * -1
2581          + space ^ 0 * EOL
2582          + Main
2583      ) ^ 0
2584      * -1
2585      * Lc [[ \@@_end_line: ]]
2586  )

```

End of the Lua scope for the language OCaml.

```
2587 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).

```
2588 do
```

```
2589  local Delim = Q ( S "{[()]}")
```

```
2590 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.

```
2591 local identifier = letter * alphanum ^ 0
2592
2593 local Operator =
2594   K ( 'Operator' ,
2595     P "!=" + "==" + "<<" + ">>" + "<=" + ">=" + "||" + "&&"
2596     + S "-~+/*%=<>&.@|!" )
2597
2598 local Keyword =
2599   K ( 'Keyword' ,
2600     P "alignas" + "asm" + "auto" + "break" + "case" + "catch" + "class" +
2601     "const" + "constexpr" + "continue" + "decltype" + "do" + "else" + "enum" +
2602     "extern" + "for" + "goto" + "if" + "nexcept" + "private" + "public" +
2603     "register" + "restricted" + "return" + "static" + "static_assert" +
2604     "struct" + "switch" + "thread_local" + "throw" + "try" + "typedef" +
2605     "union" + "using" + "virtual" + "volatile" + "while"
2606   )
2607   + K ( 'Keyword.Constant' , P "default" + "false" + "NULL" + "nullptr" + "true" )
2608
2609 local Builtin =
2610   K ( 'Name.Builtin' ,
2611     P "alignof" + "malloc" + "printf" + "scanf" + "sizeof" )
2612
2613 local Type =
2614   K ( 'Name.Type' ,
2615     P "bool" + "char" + "char16_t" + "char32_t" + "double" + "float" + "int" +
2616     "int8_t" + "int16_t" + "int32_t" + "int64_t" + "long" + "short" + "signed"
2617     + "unsigned" + "void" + "wchar_t" ) * Q "*" ^ 0
2618
2619 local DefFunction =
2620   Type
2621   * Space
2622   * Q "*" ^ -1
2623   * K ( 'Name.Function.Internal' , identifier )
2624   * SkipSpace
2625   * # 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:`

```
2626 local DefClass =
2627   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

```
2628 String =
2629   WithStyle ( 'String.Long.Internal' ,
2630     Q "\""
2631     * ( SpaceInString
2632       + K ( 'String.Interpol' ,
2633         "%" * ( S "difcspxXou" + "ld" + "li" + "hd" + "hi" )
2634       )
2635       + Q ( ( P "\\\\" + 1 - S " \" " ) ^ 1 )
```

```

2636      ) ^ 0
2637      * Q "\""
2638 )

```

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

```

2639 local braces = Compute_braces ( "\"" * ( 1 - S "\"" ) ^ 0 * "\"" )
2640 if piton.beamer then Beamer = Compute_Beamer ( 'c' , braces ) end
2641 DetectedCommands = Compute_DetectedCommands ( 'c' , braces )
2642 LPEG_cleaner.c = Compute_LPEG_cleaner ( 'c' , braces )

```

The directives of the preprocessor

```

2643 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.

```

2644 local Comment =
2645   WithStyle ( 'Comment' ,
2646     Q("//" * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
2647     * ( EOL + -1 )
2648
2649 local LongComment =
2650   WithStyle ( 'Comment' ,
2651     Q("/*"
2652       * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
2653       * Q "*/"
2654     ) -- $

```

The main LPEG for the language C

```

2655 local EndKeyword
2656   = Space + Punct + Delim + Beamer + DetectedCommands + Escape +
2657     EscapeMath + -1

```

First, the main loop :

```

2658 local Main =
2659   space ^ 0 * EOL
2660   + Space
2661   + Tab
2662   + Escape + EscapeMath
2663   + CommentLaTeX
2664   + Beamer
2665   + DetectedCommands
2666   + Preproc
2667   + Comment + LongComment
2668   + Delim
2669   + Operator
2670   + String
2671   + Punct
2672   + DefFunction
2673   + DefClass
2674   + Type * ( Q "*" ^ -1 + EndKeyword )
2675   + Keyword * EndKeyword
2676   + Builtin * EndKeyword
2677   + Identifier
2678   + Number
2679   + Word

```

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

```
2680 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.

```
2681 LPEG2.c =
2682 Ct (
2683     ( space ^ 0 * P "\r" ) ^ -1
2684     * BeamerBeginEnvironments
2685     * Lc [[ \@@_begin_line: ]]
2686     * SpaceIndentation ^ 0
2687     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2688     * -1
2689     * Lc [[ \@@_end_line: ]]
2690 )
```

End of the Lua scope for the language C.

```
2691 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).

```
2692 do
```

```
2693     local LuaKeyword
2694     function LuaKeyword ( name ) return
2695         Lc [[ {\PitonStyle{Keyword}}{ }]
2696         * Q ( Cmt (
2697             C ( letter * alphanum ^ 0 ) ,
2698             function ( s , i , a ) return string.upper ( a ) == name end
2699         )
2700         )
2701     * Lc "}"}
2702 end
```

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

```
2703     local identifier =
2704         letter * ( alphanum + "-" ) ^ 0
2705         + P '!' * ( ( 1 - P '!' ) ^ 1 ) * '!'
2706     local Operator =
2707         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).

```
2708     local Set
2709     function Set ( list )
2710         local set = { }
2711         for _ , l in ipairs ( list ) do set[l] = true end
2712         return set
2713     end
```

³⁹Remember 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.

```

2714 local set_keywords = Set
2715 {
2716     "ABORT", "ACTION", "ADD", "AFTER", "ALL", "ALTER", "ALWAYS", "ANALYZE",
2717     "AND", "AS", "ASC", "ATTACH", "AUTOINCREMENT", "BEFORE", "BEGIN", "BETWEEN",
2718     "BY", "CASCADE", "CASE", "CAST", "CHECK", "COLLATE", "COLUMN", "COMMIT",
2719     "CONFLICT", "CONSTRAINT", "CREATE", "CROSS", "CURRENT", "CURRENT_DATE",
2720     "CURRENT_TIME", "CURRENT_TIMESTAMP", "DATABASE", "DEFAULT", "DEFERRABLE",
2721     "DEFERRED", "DELETE", "DESC", "DETACH", "DISTINCT", "DO", "DROP", "EACH",
2722     "ELSE", "END", "ESCAPE", "EXCEPT", "EXCLUDE", "EXCLUSIVE", "EXISTS",
2723     "EXPLAIN", "FAIL", "FILTER", "FIRST", "FOLLOWING", "FOR", "FOREIGN", "FROM",
2724     "FULL", "GENERATED", "GLOB", "GROUP", "GROUPS", "HAVING", "IF", "IGNORE",
2725     "IMMEDIATE", "IN", "INDEX", "INDEXED", "INITIALLY", "INNER", "INSERT",
2726     "INSTEAD", "INTERSECT", "INTO", "IS", "ISNULL", "JOIN", "KEY", "LAST",
2727     "LEFT", "LIKE", "LIMIT", "MATCH", "MATERIALIZED", "NATURAL", "NO", "NOT",
2728     "NOTHING", "NOTNULL", "NULL", "NULLS", "OF", "OFFSET", "ON", "OR", "ORDER",
2729     "OTHERS", "OUTER", "OVER", "PARTITION", "PLAN", "PRAGMA", "PRECEDING",
2730     "PRIMARY", "QUERY", "RAISE", "RANGE", "RECURSIVE", "REFERENCES", "REGEXP",
2731     "REINDEX", "RELEASE", "RENAME", "REPLACE", "RESTRICT", "RETURNING", "RIGHT",
2732     "ROLLBACK", "ROW", "ROWS", "SAVEPOINT", "SELECT", "SET", "TABLE", "TEMP",
2733     "TEMPORARY", "THEN", "TIES", "TO", "TRANSACTION", "TRIGGER", "UNBOUNDED",
2734     "UNION", "UNIQUE", "UPDATE", "USING", "VACUUM", "VALUES", "VIEW", "VIRTUAL",
2735     "WHEN", "WHERE", "WINDOW", "WITH", "WITHOUT"
2736 }
2737 local set_builtins = Set
2738 {
2739     "AVG" , "COUNT" , "CHAR_LENGTH" , "CONCAT" , "CURDATE" , "CURRENT_DATE" ,
2740     "DATE_FORMAT" , "DAY" , "LOWER" , "LTRIM" , "MAX" , "MIN" , "MONTH" , "NOW" ,
2741     "RANK" , "ROUND" , "RTRIM" , "SUBSTRING" , "SUM" , "UPPER" , "YEAR"
2742 }
```

The LPEG Identifier will catch the identifiers of the fields but also the keywords and the built-in functions of SQL. It will *not* catch the names of the SQL tables.

```

2743 local Identifier =
2744     C ( identifier ) /
2745     (
2746         function ( s )
2747             if set_keywords[string.upper(s)] then return
```

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

```

2748     { {[{\PitonStyle{Keyword}{}]}] } ,
2749     { luatexbase.catcodetables.other , s } ,
2750     { "}" } }
2751     else
2752         if set_builtins[string.upper(s)] then return
2753             { {[{\PitonStyle{Name.Builtin}{}]}] } ,
2754             { luatexbase.catcodetables.other , s } ,
2755             { "}" } }
2756         else return
2757             { {[{\PitonStyle{Name.Field}{}]}] } ,
2758             { luatexbase.catcodetables.other , s } ,
2759             { "}" } }
2760         end
2761     end
2762 end
2763 )
```

The strings of SQL

```

2764 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.

```

2765 local braces = Compute_braces ( '"" * ( 1 - P "" ) ^ 1 * "" )
2766 if piton.beamer then Beamer = Compute_Beamer ( 'sql' , braces ) end
2767 DetectedCommands = Compute_DetectedCommands ( 'sql' , braces )
2768 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.

```

2769 local Comment =
2770   WithStyle ( 'Comment' ,
2771     Q "--" -- syntax of SQL92
2772     * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 ) -- $
2773     * ( EOL + -1 )
2774
2775 local LongComment =
2776   WithStyle ( 'Comment' ,
2777     Q "/*"
2778     * ( CommentMath + Q ( ( 1 - P "*/" - S "$\r" ) ^ 1 ) + EOL ) ^ 0
2779     * Q "*/"
2780   ) -- $

```

The main LPEG for the language SQL

```

2781 local EndKeyword
2782   = Space + Punct + Delim + EOL + Beamer + DetectedCommands + Escape +
2783   EscapeMath + -1
2784
2785 local TableField =
2786   K ( 'Name.Table' , identifier )
2787   * Q "."
2788   * ( DetectedCommands + ( K ( 'Name.Field' , identifier ) ) ^ 0 )
2789
2790 local OneField =
2791   (
2792     Q ( "(" * ( 1 - P ")" ) ^ 0 * ")" )
2793     +
2794       K ( 'Name.Table' , identifier )
2795       * Q "."
2796       * K ( 'Name.Field' , identifier )
2797       +
2798       K ( 'Name.Field' , identifier )
2799   )
2800   * (
2801     Space * LuaKeyword "AS" * Space * K ( 'Name.Field' , identifier )
2802   ) ^ -1
2803   * ( Space * ( LuaKeyword "ASC" + LuaKeyword "DESC" ) ) ^ -1
2804
2805 local OneTable =
2806   K ( 'Name.Table' , identifier )
2807   *
2808     Space
2809     * LuaKeyword "AS"
2810     * Space
2811     * K ( 'Name.Table' , identifier )
2812   ) ^ -1
2813
2814 local WeCatchTableName =
2815   LuaKeyword "FROM"
2816   * ( Space + EOL )

```

```

2816 * OneTable * ( SkipSpace * Q "," * SkipSpace * OneTable ) ^ 0
2817 +
2818     LuaKeyword "JOIN" + LuaKeyword "INTO" + LuaKeyword "UPDATE"
2819     + LuaKeyword "TABLE"
2820 )
2821 * ( Space + EOL ) * OneTable

2822 local EndKeyword
2823 = Space + Punct + Delim + EOL + Beamer
2824     + DetectedCommands + Escape + EscapeMath + -1

```

First, the main loop :

```

2825 local Main =
2826     space ^ 0 * EOL
2827     + Space
2828     + Tab
2829     + Escape + EscapeMath
2830     + CommentLaTeX
2831     + Beamer
2832     + DetectedCommands
2833     + Comment + LongComment
2834     + Delim
2835     + Operator
2836     + String
2837     + Punct
2838     + WeCatchTableNames
2839     + ( TableField + Identifier ) * ( Space + Operator + Punct + Delim + EOL + -1 )
2840     + Number
2841     + Word

```

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

```
2842 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.

```

2843 LPEG2.sql =
2844 Ct (
2845     ( space ^ 0 * "\r" ) ^ -1
2846     * BeamerBeginEnvironments
2847     * Lc [[ @_begin_line: ]]
2848     * SpaceIndentation ^ 0
2849     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2850     * -1
2851     * Lc [[ @_end_line: ]]
2852 )

```

End of the Lua scope for the language SQL.

```
2853 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).

```

2854 do
2855     local Punct = Q ( S ",:;!\\\" )
2856
2857     local Comment =
2858         WithStyle ( 'Comment' ,
2859                     Q "#"

```

⁴⁰Remember 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:`

```

2860           * ( CommentMath + Q ( ( 1 - S "$\r" ) ^ 1 ) ) ^ 0 -- $
2861           )
2862       * ( EOL + -1 )
2863
2864   local String =
2865     WithStyle ( 'String.Short.Internal' ,
2866     Q "\\""
2867     * ( SpaceInString
2868       + Q ( ( P "\\\\" + 1 - S " \\" ) ^ 1 )
2869       ) ^ 0
2870     * Q "\\""
2871   )

```

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

```

2872   local braces = Compute_braces ( P "\\" * ( P "\\\\" + 1 - P "\\" ) ^ 1 * "\\" )
2873
2874   if piton.beamer then Beamer = Compute_Beamer ( 'minimal' , braces ) end
2875
2876   DetectedCommands = Compute_DetectedCommands ( 'minimal' , braces )
2877
2878   LPEG_cleaner.minimal = Compute_LPEG_cleaner ( 'minimal' , braces )
2879
2880   local identifier = letter * alphanum ^ 0
2881
2882   local Identifier = K ( 'Identifier.Internal' , identifier )
2883
2884   local Delim = Q ( S "[()]" )
2885
2886   local Main =
2887     space ^ 0 * EOL
2888     + Space
2889     + Tab
2890     + Escape + EscapeMath
2891     + CommentLaTeX
2892     + Beamer
2893     + DetectedCommands
2894     + Comment
2895     + Delim
2896     + String
2897     + Punct
2898     + Identifier
2899     + Number
2900     + Word

```

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

```

2901   LPEG1.minimal = Main ^ 0
2902
2903   LPEG2.minimal =
2904     Ct (
2905       ( space ^ 0 * "\r" ) ^ -1
2906       * BeamerBeginEnvironments
2907       * Lc [[ \@@_begin_line: ]]
2908       * SpaceIndentation ^ 0
2909       * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2910       * -1
2911       * Lc [[ \@@_end_line: ]]
2912     )

```

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

```
2913 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).

```
2914 do
```

Here, we don't use **braces** as done with the other languages because we don't have to take into account the strings (there is no string in the language “Verbatim”).

```
2915 local braces =
2916   P { "E" ,
2917     E = ( "{" * V "E" * "}" + ( 1 - S "{}" ) ) ^ 0
2918   }
2919
2920 if piton.beamer then Beamer = Compute_Beamer ( 'verbatim' , braces ) end
2921
2922 DetectedCommands = Compute_DetectedCommands ( 'verbatim' , braces )
2923
2924 LPEG_cleaner.verbatim = Compute_LPEG_cleaner ( 'verbatim' , braces )
```

Now, you will construct the LPEG Word.

```
2925 local lpeg_central = 1 - S " \\r"
2926 if piton.begin_escape then
2927   lpeg_central = lpeg_central - piton.begin_escape
2928 end
2929 if piton.begin_escape_math then
2930   lpeg_central = lpeg_central - piton.begin_escape_math
2931 end
2932 local Word = Q ( lpeg_central ^ 1 )
2933
2934 local Main =
2935   space ^ 0 * EOL
2936   + Space
2937   + Tab
2938   + Escape + EscapeMath
2939   + Beamer
2940   + DetectedCommands
2941   + Q [[ ]]
2942   + Word
```

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

```
2943 LPEG1.verbatim = Main ^ 0
2944
2945 LPEG2.verbatim =
2946   Ct (
2947     ( space ^ 0 * "\r" ) ^ -1
2948     * BeamerBeginEnvironments
2949     * Lc [[ \@@_begin_line: ]]
2950     * SpaceIndentation ^ 0
2951     * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
2952     * -1
2953     * Lc [[ \@@_end_line: ]]
2954   )
```

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

```
2955 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.

```
2956 function piton.Parse ( language , code )
```

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

```
2957     piton.language = language
2958     local t = LPEG2[language] : match ( code )
2959     if t == nil then
2960         sprintL3 [[ \@@_error_or_warning:n { SyntaxError } ]]
2961         return -- to exit in force the function
2962     end
2963     local left_stack = {}
2964     local right_stack = {}
2965     for _ , one_item in ipairs ( t ) do
2966         if one_item[1] == "EOL" then
2967             for _ , s in ipairs ( right_stack ) do
2968                 tex.sprint ( s )
2969             end
2970             for _ , s in ipairs ( one_item[2] ) do
2971                 tex.tprint ( s )
2972             end
2973             for _ , s in ipairs ( left_stack ) do
2974                 tex.sprint ( s )
2975             end
2976         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 (`\begin{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}`.

```
2977     if one_item[1] == "Open" then
2978         tex.sprint( one_item[2] )
2979         table.insert ( left_stack , one_item[2] )
2980         table.insert ( right_stack , one_item[3] )
2981     else
2982         if one_item[1] == "Close" then
2983             tex.sprint ( right_stack[#right_stack] )
2984             left_stack[#left_stack] = nil
2985             right_stack[#right_stack] = nil
2986         else
2987             tex.tprint ( one_item )
2988         end
2989     end
2990   end
2991 end
2992 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.

```
2993 function piton.ParseFile
2994   ( lang , name , first_line , last_line , splittable , split )
2995   local s = ''
2996   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.

```

2997   for line in io.lines ( name ) do
2998     i = i + 1
2999     if i >= first_line then
3000       s = s .. '\r' .. line
3001     end
3002     if i >= last_line then break end
3003   end

```

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

```

3004   if string.byte ( s , 1 ) == 13 then
3005     if string.byte ( s , 2 ) == 239 then
3006       if string.byte ( s , 3 ) == 187 then
3007         if string.byte ( s , 4 ) == 191 then
3008           s = string.sub ( s , 5 , -1 )
3009         end
3010       end
3011     end
3012   end
3013   if split == 1 then
3014     piton.RetrieveGobbleSplitParse ( lang , 0 , splittable , s )
3015   else
3016     piton.RetrieveGobbleParse ( lang , 0 , splittable , s )
3017   end
3018 end

3019 function piton.RetrieveGobbleParse ( lang , n , splittable , code )
3020   local s
3021   s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
3022   piton.GobbleParse ( lang , n , splittable , s )
3023 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 `#`.

```

3024 function piton.ParseBis ( lang , code )
3025   local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( code )
3026   return piton.Parse ( lang , s )
3027 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 syntactic element. In that case, you have to remove the potential `\@@_breakable_space:` that have been inserted when the key `break-lines` is in force.

```
3028 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:`.

```

3029   local s
3030   s = ( Cs ( ( P [[\@@_breakable_space:]] / ' ' + 1 ) ^ 0 ) )
3031   : 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...

```

3032   s = ( Cs ( ( P [[\@@_leading_space:]] / ' ' + 1 ) ^ 0 ) )
3033   : match ( s )
3034   return piton.Parse ( lang , s )
3035 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.

```
3036 local AutoGobbleLPEG =
3037   (
3038     P " " ^ 0 * "\r"
3039     +
3040     Ct ( C " " ^ 0 ) / table.getn
3041     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 * "\r"
3042   ) ^ 0
3043   * ( Ct ( C " " ^ 0 ) / table.getn
3044     * ( 1 - P " " ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3045 ) / math.min
```

The following LPEG is similar but works with the tabulations.

```
3046 local TabsAutoGobbleLPEG =
3047   (
3048     (
3049       P "\t" ^ 0 * "\r"
3050       +
3051       Ct ( C "\t" ^ 0 ) / table.getn
3052       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 * "\r"
3053     ) ^ 0
3054     * ( Ct ( C "\t" ^ 0 ) / table.getn
3055       * ( 1 - P "\t" ) * ( 1 - P "\r" ) ^ 0 ) ^ -1
3056   ) / 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).

```
3057 local EnvGobbleLPEG =
3058   ( ( 1 - P "\r" ) ^ 0 * "\r" ) ^ 0
3059   * Ct ( C " " ^ 0 * -1 ) / table.getn
3060 local remove_before_cr
3061 function remove_before_cr ( input_string )
3062   local match_result = ( P "\r" ) : match ( input_string )
3063   if match_result then return
3064     string.sub ( input_string , match_result )
3065   else return
3066     input_string
3067   end
3068 end
```

The function `gobble` gobbles n characters on the left of the code. The negative values of n have special significations.

```
3069 local gobble
3070 function gobble ( n , code )
3071   code = remove_before_cr ( code )
3072   if n == 0 then return
3073     code
3074   else
3075     if n == -1 then
3076       n = AutoGobbleLPEG : match ( code )
3077     else
3078       if n == -2 then
3079         n = EnvGobbleLPEG : match ( code )
```

```

3080     else
3081         if n == -3 then
3082             n = TabsAutoGobbleLPEG : match ( code )
3083         end
3084     end
3085 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...

```

3086     if n == 0 then return
3087         code
3088     else return

```

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

```

3089     ( Ct (
3090         ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3091         * ( C "\r" * ( 1 - P "\r" ) ^ (-n) * C ( ( 1 - P "\r" ) ^ 0 )
3092             ) ^ 0 )
3093         / table.concat
3094     ) : match ( code )
3095 end
3096 end
3097 end

```

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

```

3098 function piton.GobbleParse ( lang , n , splittable , code )
3099     piton.ComputeLinesStatus ( code , splittable )
3100     piton.last_code = gobble ( n , code )
3101     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`.

```

3102     piton.CountLines ( piton.last_code )
3103     sprintL3 [[ \bool_if:NT \g_@@_footnote_bool \savenotes ]]
3104     piton.Parse ( lang , piton.last_code )

3105     sprintL3 [[ \vspace{2.5pt} ]]
3106     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).

```
3107     sprintL3 [[ \par ]]
```

Now, if the final user has used the key `write` to write the code of the environment on an external file.

```

3108     if piton.write and piton.write ~= '' then
3109         local file = io.open ( piton.write , piton.write_mode )
3110         if file then
3111             file : write ( piton.get_last_code ( ) )
3112             file : close ( )
3113         else
3114             sprintL3 [[ \@@_error_or_warning:n { FileError } ]]
3115         end
3116     end
3117 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).

```

3118 function piton.GobbleSplitParse ( lang , n , splittable , code )
3119     local chunks

```

```

3120 chunks =
3121   (
3122     Ct (
3123       (
3124         P " " ^ 0 * "\r"
3125         +
3126         C ( ( ( 1 - P "\r" ) ^ 1 * "\r" - ( P " " ^ 0 * "\r" ) ) ^ 1 )
3127       ) ^ 0
3128     )
3129   ) : match ( gobble ( n , code ) )
3130   sprintL3 [[ \begingroup ]]
3131   sprintL3
3132   (
3133     [[ \PitonOptions { split-on-empty-lines = false, gobble = 0, }]
3134     .. "language = " .. lang .. ","
3135     .. "splittable = " .. splittable .. "]"
3136   )
3137   for k , v in pairs ( chunks ) do
3138     if k > 1 then
3139       sprintL3 [[ \l_@@_split_separation_t1 ]]
3140     end
3141     tex.sprint
3142     (
3143       [[\begin{}]] .. piton.env_used_by_split .. "}\r"
3144       .. v
3145       .. [[\end{}]] .. piton.env_used_by_split .. "]"
3146     )
3147   end
3148   sprintL3 [[ \endgroup ]]
3149 end

3150 function piton.RetrieveGobbleSplitParse ( lang , n , splittable , code )
3151   local s
3152   s = ( ( P " " ^ 0 * "\r" ) ^ -1 * C ( P ( 1 ) ^ 0 ) * -1 ) : match ( code )
3153   piton.GobbleSplitParse ( lang , n , splittable , s )
3154 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_t1` corresponds to the key `split-separation`. That token list must contain elements inserted in *vertical mode* of TeX.

```

3155 piton.string_between_chunks =
3156   [[ \par \l_@@_split_separation_t1 \mode_leave_vertical: ]]
3157   .. [[ \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.

```

3158 function piton.get_last_code ( )
3159   return LPEG_cleaner[piton.last_language] : match ( piton.last_code )
3160 end

```

10.3.11 To count the number of lines

```

3161 function piton.CountLines ( code )
3162   local count = 0
3163   count =
3164     ( Ct ( ( ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
3165           * ( ( 1 - P "\r" ) ^ 1 * Cc "\r" ) ^ -1
3166           * -1

```

```

3167         ) / table.getn
3168     ) : match ( code )
3169   sprintL3 ( string.format ( [[ \int_set:Nn \l_@@_nb_lines_int { %i } ]] , count ) )
3170 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).
3171 function piton.CountNonEmptyLines ( code )
3172   local count = 0
3173   count =
3174     ( Ct ( ( P " " ^ 0 * "\r"
3175           + ( 1 - P "\r" ) ^ 0 * C "\r" ) ^ 0
3176           * ( 1 - P "\r" ) ^ 0
3177           * -1
3178         ) / table.getn
3179     ) : match ( code )
3180   sprintL3
3181   ( string.format ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { %i } ]] , count ) )
3182 end

3183 function piton.CountLinesFile ( name )
3184   local count = 0
3185   for line in io.lines ( name ) do count = count + 1 end
3186   sprintL3
3187   ( string.format ( [[ \int_set:Nn \l_@@_nb_lines_int { %i } ]] , count ) )
3188 end

3189 function piton.CountNonEmptyLinesFile ( name )
3190   local count = 0
3191   for line in io.lines ( name ) do
3192     if not ( ( P " " ^ 0 * -1 ) : match ( line ) ) then
3193       count = count + 1
3194     end
3195   end
3196   sprintL3
3197   ( string.format ( [[ \int_set:Nn \l_@@_nb_non_empty_lines_int { %i } ]] , count ) )
3198 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.

```

3199 function piton.ComputeRange(marker_beginning,marker_end,file_name)
3200   local s = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker_beginning )
3201   local t = ( Cs ( ( P '##' / '#' + 1 ) ^ 0 ) ) : match ( marker_end )
3202   local first_line = -1
3203   local count = 0
3204   local last_found = false
3205   for line in io.lines ( file_name ) do
3206     if first_line == -1 then
3207       if string.sub ( line , 1 , #s ) == s then
3208         first_line = count
3209       end
3210     else
3211       if string.sub ( line , 1 , #t ) == t then
3212         last_found = true
3213         break
3214       end
3215     end
3216     count = count + 1
3217   end

```

```

3218 if first_line == -1 then
3219   sprintL3 [[ \@@_error_or_warning:n { begin~marker~not~found } ]]
3220 else
3221   if last_found == false then
3222     sprintL3 [[ \@@_error_or_warning:n { end~marker~not~found } ]]
3223   end
3224 end
3225 sprintL3 (
3226   [[ \int_set:Nn \l_@@_first_line_int { } ] .. first_line .. ' + 2 ']
3227   .. [[ \int_set:Nn \l_@@_last_line_int { } ] .. count .. ' }' )
3228 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`.

```
3229 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).

```

3230 local lpeg_line_beamer
3231 if piton.beamer then
3232   lpeg_line_beamer =
3233   space ^ 0
3234   * P [[\begin{}]] * piton.BeamerEnvironments * "}"
3235   * ( "<" * ( 1 - P ">" ) ^ 0 * ">" ) ^ -1
3236   +
3237   space ^ 0
3238   * P [[\end{}]] * piton.BeamerEnvironments * "}"
3239 else
3240   lpeg_line_beamer = P ( false )
3241 end
3242 local lpeg_empty_lines =
3243 Ct (
3244   ( lpeg_line_beamer * "\r"
3245   +
3246   P " " ^ 0 * "\r" * Cc ( 0 )
3247   +
3248   ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
3249   ) ^ 0
3250   *
3251   ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
3252   )
3253   * -1
3254 local lpeg_all_lines =
3255 Ct (
3256   ( lpeg_line_beamer * "\r"
3257   +

```

```

3258      ( 1 - P "\r" ) ^ 0 * "\r" * Cc ( 1 )
3259      ) ^ 0
3260      *
3261      ( lpeg_line_beamer + ( 1 - P "\r" ) ^ 1 * Cc ( 1 ) ) ^ -1
3262    )
3263  * -1

```

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

```
3264  piton.empty_lines = lpeg_empty_lines : match ( code )
```

Now, we compute `piton.lines_status`. It will be used in conjunction with `splittable` and `splittable-on-empty-lines`.

Now, we will take into account the current value of `\l_@@_splittable_int` (provided by the *absolute value* of the argument `splittable`).

```

3265  local lines_status
3266  local s = splittable
3267  if splittable < 0 then s = - splittable end
3268  if splittable > 0 then
3269    lines_status = lpeg_all_lines : match ( code )
3270  else

```

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

```

3271  lines_status = lpeg_empty_lines : match ( code )
3272  for i , x in ipairs ( lines_status ) do
3273    if x == 0 then
3274      for j = 1 , s - 1 do
3275        if i + j > #lines_status then break end
3276        if lines_status[i+j] == 0 then break end
3277        lines_status[i+j] = 2
3278      end
3279      for j = 1 , s - 1 do
3280        if i - j == 1 then break end
3281        if lines_status[i-j-1] == 0 then break end
3282        lines_status[i-j-1] = 2
3283      end
3284    end
3285  end
3286 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.

```

3287  for j = 1 , s - 1 do
3288    if j > #lines_status then break end
3289    if lines_status[j] == 0 then break end
3290    lines_status[j] = 2
3291  end

```

Now, from the end of the code.

```

3292  for j = 1 , s - 1 do
3293    if #lines_status - j == 0 then break end
3294    if lines_status[#lines_status - j] == 0 then break end
3295    lines_status[#lines_status - j] = 2
3296  end

3297  piton.lines_status = lines_status
3298 end

```

10.3.13 To create new languages with the syntax of listings

```

3299 function piton.new_language ( lang , definition )
3300   lang = string.lower ( lang )

3301   local alpha , digit = lpeg.alpha , lpeg.digit
3302   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`.

```

3303   function add_to_letter ( c )
3304     if c ~= " " then table.insert ( extra_letters , c ) end
3305   end

```

For the digits, it's straightforward.

```

3306   function add_to_digit ( c )
3307     if c ~= " " then digit = digit + c end
3308   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 `}`).

```

3309   local other = S ":_@+-*/<>!?:.()~^=#&\\"\\\$" -- $
3310   local extra_others = { }
3311   function add_to_other ( c )
3312     if c ~= " " then

```

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

```

3313     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 `</....>`.

```

3314     other = other + P ( c )
3315   end
3316 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`.

```

3317   local def_table
3318   if ( S ", " ^ 0 * -1 ) : match ( definition ) then
3319     def_table = {}
3320   else
3321     local strict_braces =
3322       P { "E" ,
3323         E = ( "{" * V "F" * "}" + ( 1 - S ",{}}" ) ) ^ 0 ,
3324         F = ( "{" * V "F" * "}" + ( 1 - S "{}}" ) ) ^ 0
3325       }
3326     local cut_definition =
3327       P { "E" ,
3328         E = Ct ( V "F" * ( "," * V "F" ) ^ 0 ) ,
3329         F = Ct ( space ^ 0 * C ( alpha ^ 1 ) * space ^ 0
3330                   * ( "=" * space ^ 0 * C ( strict_braces ) ) ^ -1 )
3331       }
3332     def_table = cut_definition : match ( definition )
3333   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.).

```

3334 local tex_braced_arg = "{" * C ( ( 1 - P ".") ^ 0 ) * "}"
3335 local tex_arg = tex_braced_arg + C ( 1 )
3336 local tex_option_arg = "[" * C ( ( 1 - P ".") ^ 0 ) * "]"
3337 local args_for_tag
3338   = tex_option_arg
3339   * space ^ 0
3340   * tex_arg
3341   * space ^ 0
3342   * tex_arg
3343 local args_for_morekeywords
3344   = "[" * C ( ( 1 - P ".") ^ 0 ) * "]"
3345   * space ^ 0
3346   * tex_option_arg
3347   * space ^ 0
3348   * tex_arg
3349   * space ^ 0
3350   * ( tex_braced_arg + Cc ( nil ) )
3351 local args_for_moredelims
3352   = ( C ( P "*" ^ -2 ) + Cc ( nil ) ) * space ^ 0
3353   * args_for_morekeywords
3354 local args_for_morecomment
3355   = "[" * C ( ( 1 - P ".") ^ 0 ) * "]"
3356   * space ^ 0
3357   * tex_option_arg
3358   * space ^ 0
3359   * 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`.

```

3360 local sensitive = true
3361 local style_tag , left_tag , right_tag
3362 for _ , x in ipairs ( def_table ) do
3363   if x[1] == "sensitive" then
3364     if x[2] == nil or ( P "true" ) : match ( x[2] ) then
3365       sensitive = true
3366     else
3367       if ( P "false" + P "f" ) : match ( x[2] ) then sensitive = false end
3368     end
3369   end
3370   if x[1] == "alsodigit" then x[2] : gsub ( ".", add_to_digit ) end
3371   if x[1] == "alsoletter" then x[2] : gsub ( ".", add_to_letter ) end
3372   if x[1] == "alsoother" then x[2] : gsub ( ".", add_to_other ) end
3373   if x[1] == "tag" then
3374     style_tag , left_tag , right_tag = args_for_tag : match ( x[2] )
3375     style_tag = style_tag or [[\PitonStyle{Tag}]]
3376   end
3377 end

```

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

```

3378 local Number =
3379   K ( 'Number.Internal' ,
3380     ( digit ^ 1 * "." * # ( 1 - P ".") * digit ^ 0
3381       + digit ^ 0 * "." * digit ^ 1
3382       + digit ^ 1 )
3383     * ( S "eE" * S "+-" ^ -1 * digit ^ 1 ) ^ -1
3384     + digit ^ 1
3385   )
3386 local string_extra_letters = ""
3387 for _ , x in ipairs ( extra_letters ) do
3388   if not ( extra_others[x] ) then
3389     string_extra_letters = string_extra_letters .. x

```

```

3390     end
3391   end
3392   local letter = alpha + S ( string_extra_letters )
3393     + P "â" + "à" + "ç" + "é" + "è" + "ê" + "ë" + "ï" + "î"
3394     + "ô" + "û" + "ü" + "Ã" + "À" + "Ç" + "É" + "È" + "Ê" + "Ë"
3395     + "í" + "Î" + "Ô" + "Û" + "Ü"
3396   local alphanum = letter + digit
3397   local identifier = letter * alphanum ^ 0
3398   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.

```

3399   local split_clist =
3400     P { "E" ,
3401       E = ( "[" * ( 1 - P "]" ) ^ 0 * "]" ) ^ -1
3402         * ( P "{" ) ^ 1
3403         * Ct ( V "F" * ( "," * V "F" ) ^ 0 )
3404         * ( P "}" ) ^ 1 * space ^ 0 ,
3405       F = space ^ 0 * C ( letter * alphanum ^ 0 + other ^ 1 ) * space ^ 0
3406     }

```

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

```

3407   local keyword_to_lpeg
3408   function keyword_to_lpeg ( name ) return
3409     Q ( Cmt (
3410       C ( identifier ) ,
3411       function ( s , i , a ) return
3412         string.upper ( a ) == string.upper ( name )
3413       end
3414     )
3415   )
3416 end
3417 local Keyword = P ( false )
3418 local PrefixedKeyword = P ( false )

```

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

```

3419   for _ , x in ipairs ( def_table )
3420     do if x[1] == "morekeywords"
3421       or x[1] == "otherkeywords"
3422       or x[1] == "moredirectives"
3423       or x[1] == "moretexcs"
3424     then
3425       local keywords = P ( false )
3426       local style = {[PitonStyle{Keyword}]}
3427       if x[1] == "moredirectives" then style = {[PitonStyle{Directive}]} end
3428       style = tex_option_arg : match ( x[2] ) or style
3429       local n = tonumber ( style )
3430       if n then
3431         if n > 1 then style = {[PitonStyle{Keyword}]] .. style .. "}" end
3432       end
3433       for _ , word in ipairs ( split_clist : match ( x[2] ) ) do
3434         if x[1] == "moretexcs" then
3435           keywords = Q ( [[\]] .. word ) + keywords
3436         else
3437           if sensitive

```

The documentation of `lslistings` 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*.

```

3438       then keywords = Q ( word ) + keywords
3439       else keywords = keyword_to_lpeg ( word ) + keywords
3440     end
3441   end

```

```

3442     end
3443     Keyword = Keyword +
3444     Lc ( "{" .. style .. "(" .. keywords * Lc ")" )
3445 end

```

Of course, the feature with the key `keywordsprefix` is designed for the languages TeX, LaTeX, etc. In that case, there are 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.

```

3446     if x[1] == "keywordsprefix" then
3447         local prefix = ( ( C ( 1 - P " " ) ^ 1 ) * P " " ^ 0 ) : match ( x[2] )
3448         PrefixedKeyword = PrefixedKeyword
3449         + K ( 'Keyword' , P ( prefix ) * ( letter ^ 1 + other ) )
3450     end
3451 end

```

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

```

3452     local long_string = P ( false )
3453     local Long_string = P ( false )
3454     local LongString = P ( false )
3455     local central_pattern = P ( false )
3456     for _ , x in ipairs ( def_table ) do
3457         if x[1] == "morestring" then
3458             arg1 , arg2 , arg3 , arg4 = args_for_morekeywords : match ( x[2] )
3459             arg2 = arg2 or {[PitonStyle{String.Long}]}
3460             if arg1 ~= "s" then
3461                 arg4 = arg3
3462             end
3463             central_pattern = 1 - S ( " \r" .. arg4 )
3464             if arg1 : match "b" then
3465                 central_pattern = P ( {[}] .. arg3 ) + central_pattern
3466             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...

```

3467     if arg1 : match "d" or arg1 == "m" then
3468         central_pattern = P ( arg3 .. arg3 ) + central_pattern
3469     end
3470     if arg1 == "m"
3471     then prefix = B ( 1 - letter - ")" - "]" )
3472     else prefix = P ( true )
3473     end

```

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

```

3474     long_string = long_string +
3475         prefix
3476         * arg3
3477         * ( space + central_pattern ) ^ 0
3478         * arg4

```

Now a pattern *with captures*.

```

3479     local pattern =
3480         prefix
3481         * Q ( arg3 )
3482         * ( SpaceInString + Q ( central_pattern ^ 1 ) + EOL ) ^ 0
3483         * Q ( arg4 )

```

We will need `Long_string` in the nested comments.

```

3484     Long_string = Long_string + pattern
3485     LongString = LongString +
3486         Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
3487         * pattern
3488         * Ct ( Cc "Close" )
3489     end
3490 end

```

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

```

3491 local braces = Compute_braces ( long_string )
3492 if piton.beamer then Beamer = Compute_Beamer ( lang , braces ) end
3493
3494 DetectedCommands = Compute_DetectedCommands ( lang , braces )
3495
3496 LPEG_cleaner[lang] = Compute_LPEG_cleaner ( lang , braces )

```

Now, we deal with the comments and the delims.

```

3497 local CommentDelim = P ( false )
3498
3499 for _ , x in ipairs ( def_table ) do
3500     if x[1] == "morecomment" then
3501         local arg1 , arg2 , other_args = args_for_morecomment : match ( x[2] )
3502         arg2 = arg2 or {[\\PitonStyle{Comment}]}
3503
3504         if arg1 : match "i" then arg2 = {[\\PitonStyle{Discard}]} end
3505         if arg1 : match "l" then
3506             local arg3 = ( tex_braced_arg + C ( P ( 1 ) ^ 0 * -1 ) )
3507                 : match ( other_args )
3508             if arg3 == {[\\#]} then arg3 = "#" end -- mandatory
3509             if arg3 == {[\\%]} then arg3 = "%" end -- mandatory
3510             CommentDelim = CommentDelim +
3511                 Ct ( Cc "Open"
3512                     * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
3513                     * Q ( arg3 )
3514                     * ( CommentMath + Q ( ( 1 - S "$\\r" ) ^ 1 ) ) ^ 0 -- $
3515                     * Ct ( Cc "Close" )
3516                     * ( EOL + -1 )
3517
3518         else
3519             local arg3 , arg4 =
3520                 ( tex_arg * space ^ 0 * tex_arg ) : match ( other_args )
3521             if arg1 : match "s" then
3522                 CommentDelim = CommentDelim +
3523                     Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
3524                     * Q ( arg3 )
3525                     *
3526                         CommentMath
3527                         + Q ( ( 1 - P ( arg4 ) - S "$\\r" ) ^ 1 ) -- $
3528                         + EOL
3529                         ) ^ 0
3530                         * Q ( arg4 )
3531                         * Ct ( Cc "Close" )
3532
3533         end
3534         if arg1 : match "n" then
3535             CommentDelim = CommentDelim +
3536                 Ct ( Cc "Open" * Cc ( "{" .. arg2 .. "}" ) * Cc "}" )
3537                 * P { "A" ,
3538                     A = Q ( arg3 )
3539                     * ( V "A"
3540                         + Q ( ( 1 - P ( arg3 ) - P ( arg4 )

```

```

3538           - S "\r$\"" ) ^ 1 ) -- $
3539           + long_string
3540           + $" -- $
3541           * K ( 'Comment.Math' , ( 1 - S "$\r" ) ^ 1 ) --$
3542           * $" -- $
3543           + EOL
3544           ) ^ 0
3545           * Q ( arg4 )
3546       }
3547       * Ct ( Cc "Close" )
3548   end
3549 end
3550 end

```

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

```

3551 if x[1] == "moredelim" then
3552     local arg1 , arg2 , arg3 , arg4 , arg5
3553     = args_for_moredelims : match ( x[2] )
3554     local MyFun = Q
3555     if arg1 == "*" or arg1 == "**" then
3556         function MyFun ( x )
3557             if x ~= '' then return
3558             LPEG1[lang] : match ( x )
3559             end
3560         end
3561     end
3562     local left_delim
3563     if arg2 : match "i" then
3564         left_delim = P ( arg4 )
3565     else
3566         left_delim = Q ( arg4 )
3567     end
3568     if arg2 : match "l" then
3569         CommentDelim = CommentDelim +
3570             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" * Cc "}" ) )
3571             * left_delim
3572             * ( MyFun ( ( 1 - P "\r" ) ^ 1 ) ) ^ 0
3573             * Ct ( Cc "Close" )
3574             * ( EOL + -1 )
3575     end
3576     if arg2 : match "s" then
3577         local right_delim
3578         if arg2 : match "i" then
3579             right_delim = P ( arg5 )
3580         else
3581             right_delim = Q ( arg5 )
3582         end
3583         CommentDelim = CommentDelim +
3584             Ct ( Cc "Open" * Cc ( "{" .. arg3 .. "}" * Cc "}" ) )
3585             * left_delim
3586             * ( MyFun ( ( 1 - P ( arg5 ) - "\r" ) ^ 1 ) + EOL ) ^ 0
3587             * right_delim
3588             * Ct ( Cc "Close" )
3589     end
3590   end
3591 end
3592
3593 local Delim = Q ( S "{[()]}")
3594 local Punct = Q ( S "=,:;!\\"\'\"")
3595 local Main =
3596     space ^ 0 * EOL
3597     + Space
3598     + Tab
3599     + Escape + EscapeMath

```

```

3600      + CommentLaTeX
3601      + Beamer
3602      + DetectedCommands
3603      + CommentDelim

```

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

```

3604      + LongString
3605      + Delim
3606      + PrefixedKeyword
3607      + Keyword * ( -1 + # ( 1 - alphanum ) )
3608      + Punct
3609      + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
3610      + Number
3611      + 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.

```
3612  LPEG1[lang] = Main ^ 0
```

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

```

3613  LPEG2[lang] =
3614  Ct (
3615      ( space ^ 0 * P "\r" ) ^ -1
3616      * BeamerBeginEnvironments
3617      * Lc [[ \@_begin_line: ]]
3618      * SpaceIndentation ^ 0
3619      * ( space ^ 1 * -1 + space ^ 0 * EOL + Main ) ^ 0
3620      * -1
3621      * Lc [[ \@_end_line: ]]
3622  )

```

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

```

3623  if left_tag then
3624      local Tag = Ct ( Cc "Open" * Cc ( "{" .. style_tag .. "}" * Cc "}" ) )
3625          * Q ( left_tag * other ^ 0 ) -- $
3626          * ( ( ( 1 - P ( right_tag ) ) ^ 0 )
3627              / ( function ( x ) return LPEG0[lang] : match ( x ) end ) )
3628          * Q ( right_tag )
3629          * Ct ( Cc "Close" )
3630  MainWithoutTag
3631      = space ^ 1 * -1
3632      + space ^ 0 * EOL
3633      + Space
3634      + Tab
3635      + Escape + EscapeMath
3636      + CommentLaTeX
3637      + Beamer
3638      + DetectedCommands
3639      + CommentDelim
3640      + Delim
3641      + LongString
3642      + PrefixedKeyword
3643      + Keyword * ( -1 + # ( 1 - alphanum ) )
3644      + Punct
3645      + K ( 'Identifier.Internal' , letter * alphanum ^ 0 )
3646      + Number
3647      + Word
3648  LPEG0[lang] = MainWithoutTag ^ 0
3649  local LPEGaux = Tab + Escape + EscapeMath + CommentLaTeX
3650          + Beamer + DetectedCommands + CommentDelim + Tag
3651  MainWithTag
3652      = space ^ 1 * -1

```

```

3653     + space ^ 0 * EOL
3654     + Space
3655     + LPEGaux
3656     + Q ( ( 1 - EOL - LPEGaux ) ^ 1 )
3657 LPEG1[lang] = MainWithTag ^ 0
3658 LPEG2[lang] =
3659   Ct (
3660     ( space ^ 0 * P "\r" ) ^ -1
3661     * BeamerBeginEnvironments
3662     * Lc [[ \@@_begin_line: ]]
3663     * SpaceIndentation ^ 0
3664     * LPEG1[lang]
3665     * -1
3666     * Lc [[ \@@_end_line: ]]
3667   )
3668 end
3669 end
3670 
```

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. A 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_t1` 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`.

Acknowledgments

Acknowledgments to Yann Salmon for its numerous suggestions of improvements.

Contents

1	Presentation	1
2	Installation	2

3	Use of the package	2
3.1	Loading the package	2
3.2	Choice of the computer language	2
3.3	The tools provided to the user	2
3.4	The syntax of the command \piton	3
4	Customization	4
4.1	The keys of the command \PitonOptions	4
4.2	The styles	7
4.2.1	Notion of style	7
4.2.2	Global styles and local styles	7
4.2.3	The style UserFunction	8
4.3	Creation of new environments	8
5	Definition of new languages with the syntax of listings	9
6	Advanced features	11
6.1	Insertion of a file	11
6.1.1	The command \PitonInputFile	11
6.1.2	Insertion of a part of a file	11
6.2	Page breaks and line breaks	13
6.2.1	Line breaks	13
6.2.2	Page breaks	14
6.3	Splitting of a listing in sub-listings	14
6.4	Highlighting some identifiers	15
6.5	Mechanisms to escape to LaTeX	16
6.5.1	The “LaTeX comments”	17
6.5.2	The key “math-comments”	17
6.5.3	The key “detected-commands”	18
6.5.4	The mechanism “escape”	18
6.5.5	The mechanism “escape-math”	19
6.6	Behaviour in the class Beamer	20
6.6.1	{Piton} et \PitonInputFile are “overlay-aware”	20
6.6.2	Commands of Beamer allowed in {Piton} and \PitonInputFile	20
6.6.3	Environments of Beamer allowed in {Piton} and \PitonInputFile	21
6.7	Footnotes in the environments of piton	22
6.8	Tabulations	23
7	API for the developpers	23
8	Examples	24
8.1	Line numbering	24
8.2	Formatting of the LaTeX comments	24
8.3	An example of tuning of the styles	25
8.4	Use with pyluatex	26
9	The styles for the different computer languages	27
9.1	The language Python	27
9.2	The language OCaml	28
9.3	The language C (and C++)	29
9.4	The language SQL	30
9.5	The languages defined by \NewPitonLanguage	31
9.6	The language “minimal”	32
9.7	The language “verbatim”	32

10	Implementation	33
10.1	Introduction	33
10.2	The L3 part of the implementation	34
10.2.1	Declaration of the package	34
10.2.2	Parameters and technical definitions	37
10.2.3	Treatment of a line of code	41
10.2.4	PitonOptions	45
10.2.5	The numbers of the lines	50
10.2.6	The command to write on the aux file	51
10.2.7	The main commands and environments for the final user	51
10.2.8	The styles	61
10.2.9	The initial styles	63
10.2.10	Highlighting some identifiers	64
10.2.11	Security	66
10.2.12	The error messages of the package	66
10.2.13	We load piton.lua	69
10.2.14	Detected commands	69
10.3	The Lua part of the implementation	70
10.3.1	Special functions dealing with LPEG	70
10.3.2	The language Python	77
10.3.3	The language Ocaml	84
10.3.4	The language C	92
10.3.5	The language SQL	95
10.3.6	The language “Minimal”	98
10.3.7	The language “Verbatim”	100
10.3.8	The function Parse	101
10.3.9	Two variants of the function Parse with integrated preprocessors	102
10.3.10	Preprocessors of the function Parse for gobble	103
10.3.11	To count the number of lines	105
10.3.12	To determine the empty lines of the listings	107
10.3.13	To create new languages with the syntax of listings	109
11	History	116