The package piton*

F. Pantigny
fpantigny@wanadoo.fr

November 25, 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 LPEG[1] for parsing computer 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`. 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:int=10):
    """Compute the mathematical value of arctan(x)

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

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

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

*This document corresponds to the version 4.9b of piton, at the date of 2025/11/25.

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

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

2 Installation

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

Note that the character *quote* U+0027 : ' is never transformed by piton in an apostrophe U+2019. There is no point loading upquote.

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}`.

The package piton uses and *loads* the package xcolor. It does not use 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 two special languages called `minimal` and `verbatim`;
- the languages defined by the end user by using the built-in command `\NewPitonLanguage` described p. 10 (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 computer 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 computer listings: 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` or its friends: cf. 4.3 p. 9.

- The command `\PitonInputFile` is used to insert and typeset an external file: cf. 6.1 p. 12.

3.4 The double 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 LaTeX command `\verb`, that is to say with the argument delimited by two identical characters (e.g.: `\piton|...|` or `\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 also the character of end of 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 provided for individual braces;
  - the LaTeX commands[3] of the argument are fully expanded (in the TeX meaning) and not executed,
    so, it's possible to use `\\` to insert a backslash.

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

  Examples :

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

  It's possible to use the command `\piton` with that syntax in the arguments of a LaTeX command.[4]

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

- Syntax `\piton|...|`

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

  Examples :

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

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

[4] For example, it's possible to use the command `\piton` in a footnote. Example : `s = 123`.

4 Customization

4.1 The keys of the command \PitonOptions

The command `\PitonOptions` takes in as argument a comma-separated list of *key=value* pairs. The scope of the settings done by that command is the current TeX group.[5]
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. 10).

  The initial value is `Python`.

- 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 monospace 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 `gobble` is used without value, it is equivalent to the key `auto-gobble`, that we describe now.

- 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 `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`).[6]
  - 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`.[7]
  - 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. 12). 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.

[5] We remind that a LaTeX environment is, in particular, a TeX group.
[6] For the language Python, the empty lines in the docstrings are taken into account (by design).
[7] When the key `split-on-empty-lines` is in force, the labels of the empty lines are never printed.

– 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}
      }
  }
```

Be careful : the previous code is not enough to print the numbers of lines. For that, one also has to use the key `line-numbers` is a absolute way, that is to say without value.

- 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 special 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 9.2 on page 33.

- 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` or the key `max-width` described below).

  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}`.

  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.

  In that list, the special color `none` may be used to specify no color at all.

  Example : `\PitonOptions{background-color = {gray!15,none}}`

- It's possible to use the key `rounded-corners` to require rounded corners for the colored panels drawn by the key `background-color` The initial value of that is 0 pt, which means that the corners are not rounded. If the key `rounded-corners` is used, the extension tikz must be loaded because those rounded corners are drawn by using tikz. If tikz is not loaded, an error will be raised at the first use of the key `rounded-corners`.

  The default value of the key `rounded-corners` is 4 pt.[8]

- 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 initial value is: `gray!15`

[8] This value is the initial value of the *rounded corners* of TikZ.

- The key `width` fixes the width of the listing in the PDF. The initial value of that parameter is the current value of `\linewidth` (LaTeX parameter which corresponds to the width of the lines of text).

  That parameter is used for:

  - the breaking the lines which are too long (except, of course, when the key `break-lines` is set to false: cf. p. 20);
  - the width of the backgrounds specified by the keys `background-color` and `prompt-background-color` described below;
  - the width of the colored backgrounds added by `\rowcolor` (cf. p. 8);
  - the width of the LaTeX box created by the key `box` (cf. p. 15);
  - the width of the graphical box created by the key `tcolorbox` (cf. p. 16).

- The key `max-width` is similar to the key `width` but it fixes the *maximal* width of the lines. If all the lines of the listing are shorter than the value provided to `max-width`, the parameter `width` will be equal to the maximal length of the lines of the listing, that is to say the natural width of the listing.

  For legibility of the code, `width=min` is a shortcut for `max-width=\linewidth`.

- When the key `show-spaces-in-strings` is activated, the spaces in the strings of characters[9] are replaced by the character ␣ (U+2423 : OPEN BOX). Of course, that character U+2423 must be present in the monospace font which is used.[10]

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

  With the key `show-spaces`, all the spaces are replaced by U+2423 (and no line break can occur on those "visible spaces", even when the key `break-lines`[11] is in force). By the way, one should remark that all the trailing spaces (at the end of a line) are deleted by piton — and, therefore, won't be represented by ␣. Moreover, when the key `show-spaces` is in force, the tabulations at the beginning of the lines are represented by arrows.

```
\begin{Piton}[language=C,line-numbers,gobble,background-color=gray!15
              rounded-corners,width=min,splittable=4]
    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++) {
```

[9] With the language Python that feature applies only to the short strings (delimited by ' or ") and, in particular, it does not apply for the *doc strings*. In OCaml, that feature does not apply to the *quoted strings*.

[10] The initial value of `font-command` is `\ttfamily` and, thus, by default, piton merely uses the current monospace font.

[11] cf. 7.3.1 p. 20

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

The command `\PitonOptions` provides in fact several other keys which will be described further (see in particular the "Pages breaks and line breaks" p. 20).

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 computer listings. The customizations done by that command are limited to the current TeX group.[12]

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

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

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

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

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

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

The different styles, and their use by piton in the different languages which it supports (Python, OCaml, C, SQL, "`minimal`" and "`verbatim`"), are described in the part 10, starting at the page 43.

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. That command is *fully expandable* (in the TeX sens).

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*.

[12] We remind that a LaTeX environment is, in particular, a TeX group.

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 computer 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 computer 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 computer language by providing the name of that language as optional argument (between square brackets) to the command `\SetPitonStyle`.[13]

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 a computer 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).[14]

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

4.2.3 The command \rowcolor

New 4.8

The extension piton provides the command `\rowcolor` which adds a colored background to the current line (the *whole* line and not only the part with text) which may be used in the styles.

The command `\rowcolor` has a syntax similar to the classical command `\color`. For example, it's possible to write `\rowcolor[rgb]{0.9,1,0.9}`.

The command `\rowcolor` is protected against the TeX expansions.

Here is an example for the language Python where we modify the style `String.Doc` of the "documentation strings" in order to have a colored background.

```
\SetPitonStyle{String.Doc = \rowcolor{gray!15}\color{black!80}}
\begin{Piton}[width=min]
def square(x):
    """Computes the square of x
       Second line of the documentation"""
    return x*x
\end{Piton}
```

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

If the command `\rowcolor` appears (through a style of piton) inside a command `\piton`, it is no-op (as expected).

[13] We recall, that, in the package piton, the names of the computer languages are case-insensitive.
[14] As regards the TeX groups, the definitions done by `\SetPitonStyle` are always local.

4.2.4 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 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 definition of the function.

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

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

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

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

Of course, the list of the names of Python functions previously défined is kept in the memory of LuaLaTeX (in a global way, that is to say independently of the TeX groups). The extension piton provides a command to clear that list: it's the command `\PitonClearUserFunctions`. When it is used without argument, that command is applied to all the computer 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 computer languages to which the command will be applied.[15]

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).
It's possible to use `\NewEnvironmentCopy` on the environment `{Piton}` but it's not very powerful.
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`.[16]

There also exist three other commands `\RenewPitonEnvironment`, `\DeclarePitonEnvironment` and `\ProvidePitonEnvironment`, similar to the corresponding commands of L3.

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

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

If one wishes to format Python code in a box of mdframed, it's possible to define an environment `{Python}` with the following code.

```
\usepackage[framemethod=tikz]{mdframed} % in the preamble
```

[15] We remind that, in piton, the name of the computer languages are case-insensitive.
[16] However, the specifier of argument `b` (used to catch the body of the environment as a LaTeX argument) is not allowed (of course)

```
\NewPitonEnvironment{Python}{}
  {\begin{mdframed}[roundcorner=3mm]}
  {\end{mdframed}}
```

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

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

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

It's possible to a similar construction with an environment of tcolorbox. However, for a better cooperation between piton and tcolorbox, the extension piton provides a key tcolorbox: cf. p. 16.

5 Definition of new languages with the syntax of listings

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

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

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

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

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

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

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

It's possible to use the language Java like any other language defined by piton.
Here is an example of code formatted in an environment `{Piton}` with the key `language=Java`.[17]

```
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 = \ , alsoother = _ }
```

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

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

6 Import and export

6.1 Importation of a listing

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`).

The syntax for the paths (absolute or relative) 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 one sense, `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 programming 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 (here `#[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`.[18]

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}
```

[18] In regard to LaTeX, both functions must be *fully expandable*.

6.2 Exportation of a listing

Alongside the command `\PitonInputFile` which allows the final user to import a listing from an external file, piton provides tools to export some listings included in the PDF file to an external file or as joined files embedded in PDF generated by LuaLaTeX.

- The key `write` takes in as argument a name of file (with its extension) and write the content[19] of the current environment in that file. At the first use of a file by piton (during a given compilation done by LuaLaTeX), it is erased. In fact, the file is written once at the end of the compilation of the file by LuaLaTeX.

  For legibility, piton provides the key `no-write` (without value) as alias for `write={}`.

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

- The key `join` is similar to the key `write` but the files which are created are joined (as *joined files*) in the PDF. Be careful: Some PDF readers don't provide any tool to access to these joined files.

  For legibility, piton provides the key `no-join` (without value) as alias for `join={}`.

- The key `print` controls whether the content of the environment is actually printed (with the syntactic formating) in the PDF. Of course, the initial value of `print` is `true`. However, it may be useful to use `print=false` in some circumstancies (for example, when the key `write` or the key `join` is used).

- **New 4.9**

  The key `paperclip` will, for each environment `{Piton}`, add in the right margin a PDF annotation linked with a file joined in the PDF corresponding to the listing of the environment.

  The value provided to the key `paperclip` is the name that will be given to the embedded file. If no value is provided, the file will have the name `listing_i.txt` where *i* is a counter incremented by piton each time that a new file is created by use of `paperclip` without value.

```
\begin{Piton}[paperclip,background-color=gray!15]
def square(x):
    """Computes the square of x"""
    return x*x
\end{Piton}
```

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

- **New 4.9**

  The key `annotation` will, for each environment `{Piton}`, add in the right margin a PDF annotation whose content is directly the body of the environment `{Piton}`.

```
\begin{Piton}[annotation,background-color=gray!15]
def square(x):
    """Computes the square of x"""
    return x*x
\end{Piton}
```

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

[19] In fact, it's not exactly the body of the environment but the value of `piton.get_last_code()` which is the body without the overwritten LaTeX formatting instructions (cf. the part 8, p. 32).

7 Advanced features

7.1 The key “box”

If one wishes to compose a listing in a box of LaTeX, he should use the key `box`. That key takes in as value `c`, `t` or `b` corresponding to the parameter of vertical position (as for the envionment `{minipage}` of LaTeX which creates also a LaTeX box). The default value is `c` (as for `{minipage}`).
When the key `box` is used, `width=min` is activated (except, of course, when the key `width` or the key `max-width` is explicitly used). For the keys `width` and `max-width`, cf. p. 6.

```
\begin{center}
\PitonOptions{box,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def cube(x):
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    return x*x*x
```

It’s possible to use the key `box` with a numerical value for the key `width`.

```
\begin{center}
\PitonOptions{box,width=5cm,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def cube(x):
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    return x*x*x
```

Here is an exemple with the key `max-width`, equal to 7 cm for both listings.

```
\begin{center}
\PitonOptions{box=t,max-width=7cm,background-color=gray!15}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def P(x):
    return 24*x**8 - 7*x**7 + 12*x**6 -4*x**5 + 4*x**3 + x**2 - 5*x + 2
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def P(x):
    return 24*x**8 - 7*x**7 + \
12*x**6 -4*x**5 + 4*x**3 + x**2 - \
5*x + 2
```

7.2 The key “tcolorbox”

The extension piton provides a key `tcolorbox` in order to ease the use of the extension tcolorbox in conjunction with the extension piton. However, the extension piton does not load tcolorbox and the end user should have loaded it. Moreover, he must load the library `breakable` of tcolorbox with `\tcbuselibrary{breakable}` in the preamble of the LaTeX document. If this is not the case, an error will be raised at the first use of the key `tcolorbox`.

When the key `tcolorbox` is used, the listing formated by piton is included in an environment `{tcolorbox}`. That applies both to the command `\PitonInputFile` and the environment `{Piton}` (or, more generally, an environment created by the dedicated command `\NewPitonEnvironment`: cf. p. 9). If the key `splittable` of piton is used (cf. p. 21), the graphical box created by tcolorbox will be splittable by a change of page.

In the present document, we have loaded, besides tcolorbox and its library `breakable`, the library `skins` of tcolorbox and we have activated the “*skin*” `enhanced`, in order to have a better appearance at the page break.

```
\tcbuselibrary{skins,breakable} % in the preamble
\tcbset{enhanced}               % in the preamble

\begin{Piton}[tcolorbox,splittable=3]
def square(x):
    """Computes the square of x"""
    return x*x
...
def square(x):
    """Computes the square of x"""
    return x*x
\end{Piton}
```

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

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

Of course, if we want to change the color of the background, we won't use the key `background-color` of piton but the tools provided by tcolorbox (the key `colback` for the color of the background).

If we want to adjust the width of the graphical box to its content, we only have to use the key

`width=min` provided by piton (cf. p. 6). It's also possible to use `width` or `max-width` with a numerical value. The environment is splittable if the key `splittable` is used (cf. p. 21).

```
\begin{Piton}[tcolorbox,width=min,splittable=3]
def square(x):
    """Computes the square of x"""
    return x*x
...
def square(x):
    """Computes the square of x"""
    return x*x
\end{Piton}
```

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

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

If we want an output composed in a LaTeX box (despites its name, an environment of tcolorbox does not always create a LaTeX box), we only have to use, in conjunction with the key `tcolorbox`, the key `box` provided by piton (cf. p. 15). Of course, such LaTeX box, as all the LaTeX boxes, can't be broken by a change of page, even if the key `splittable` (cf. p. 21) is in force.
We recall that, when the key `box` is used, `width=min` is activated (except, when the key `width` or the key `max-width` is explicitly used).

```
\begin{center}
\PitonOptions{tcolorbox,box=t}
\begin{Piton}
def square(x):
    return x*x
\end{Piton}
\hspace{1cm}
\begin{Piton}
def cube(x):
    """The cube of x"""
    return x*x*x
\end{Piton}
\end{center}
```

```
def square(x):
    return x*x
```

```
def cube(x):
    """The cube of x"""
    return x*x*x
```

For a more sophisticated example of use of the key `tcolorbox`, see the example given at the page 36.

7.3 Line breaks and page breaks

7.3.1 Line breaks

There are keys to control the line breaks (the possible breaking points are the spaces, even the spaces which appear in the strings of the computer 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 initial value of that parameter is `true` (and not `false`).**
- 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 monospace 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,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
```

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

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

7.3.2 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.

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

We illustrate that point with the following code (the current environment `{tcolorbox}` uses the key `breakable`).

```
\begin{Piton}[background-color=gray!30,rounded-corners,width=min,splittable=4]
def square(x):
    """Computes the square of x"""
    return x*x
...
def square(x):
    """Computes the square of x"""
    return x*x
\end{Piton}
```

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

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

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

7.4 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 computer listing which contains several definitions of computer functions. Usually, in the computer 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.
- In fact, the extension piton provides also the key `add-to-split-separation` to add elements on the right of the parameter `split-separation`.

Each chunk of the computer 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. 9).

Each chunk of the computer 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
```

If we wish to have a continuity of the line numbers between the sublistings it's possible to add `\PitonOptions{resume}` to the parameter `split-separation`.

```
\begin{Piton}[
split-on-empty-lines, add-to-split-separation = \PitonOptions{resume} , background-color=gray!15, l
def square(x):
    """Computes the square of x"""
    return x*x

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

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

```
4 def cube(x):
5     """Computes the square of x"""
6     return x*x*x
```

Caution: Since each chunk is treated independently of the others, the commands specified by `detected-commands` or `raw-detected-commands` (cf. p. 26) and the commands and environments of Beamer automatically detected by piton must not cross the empty lines of the original listing.

7.5 Highlighting some identifiers

The command `\SetPitonIdentifier` allows to automatically change the formatting of some identifiers. That change is only based on the name of those indentifiers.

That command takes in three arguments:

- The optional argument (within square brackets) specifies the computer language. If this argument is not present, the tunings done by `\SetPitonIdentifier` will apply to all the computer languages of piton.[21]
- The first mandatory argument is a comma-separated list of names of identifiers.

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

- 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 `\SetPitonIdentifier`.

```
\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)
```

7.6 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 keys `detected-commands`, `raw-detected-commands` and `vertical-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. 7.7 p. 28.

7.6.1 The "LaTeX comments"

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

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

  For example, if the preamble contains the following instruction:

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

  the LaTeX comments will begin by `#LaTeX`.

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

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

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

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

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

  For other examples of customization of the LaTeX comments, see the part 9.3 p. 34

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.[22] The same goes for the `\zlabel` command from the zref package.[23]

7.6.2 The key "label-as-zlabel"

The key `label-as-zlabel` will be used to indicate if the user wants `\label` inside `Piton` environments to be replaced by a `\zlabel`-compatible command (which is the default behavior of zref outside of such environments).
That feature is activated by the key `label-as-zlabel`, *which is available only in the preamble of the document.*

7.6.3 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.*

```
\PitonOptions{math-comment} % in the preamble
```

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

[23] Using the command `\zcref` command from zref-clever is also supported.

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

```
def square(x):
    return x*x # compute x^2
```

7.6.4 The key "detected-commands" and its variants

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 computer listing).
- These commands must be **protected**[24] against expansion in the TeX sens (because the command `\piton` expands its arguments before throwing it to Lua for syntactic analysis).

In the following example, which is a recursive programming in C of the factorial function, we decide to highlight the recursive call. The command `\highLight` of lua-ul[25] directly does the job.

```
\PitonOptions{detected-commands = highLight} % in the preamble

\begin{Piton}[language=C]
int factorielle(int n)
  {
    if (n > 0) \highLight{return n * factorielle(n - 1)} ;
    else return 1;
  }
\end{Piton}
```

```
int factorielle(int n)
  {
    if (n > 0) return n * factorielle(n - 1) ;
    else return 1;
  }
```

The key `raw-detected-commands` is similar to the key `detected-commands` but piton won't do any syntactic analysis of the arguments of the LaTeX commands which are detected.
If there is a line break within the argument of a command detected by the mean of `raw-detected-commands`, that line break is replaced by a space (as does LaTeX by default).

Imagine, for example, that we wish, in the main text of a document about databases, introduce some specifications of tables of the language SQL by the the name of the table, followed, between brackets, by the names of its fields (ex. : `client(name,town)`).
If we insert that element in a command `\piton`, the word *client* won't be recognized as a name of table but as a name of field. It's possible to define a personal command `\NomTable` which we will apply by hand to the names of the tables. In that aim, we declare that command with `raw-detected-commands` and, thus, its argument won't be re-analyzed by piton (that second analysis would format it as a name of field).

In the preamble of the LaTeX document, we insert the following lines:

[24] We recall that the command `\NewDocumentCommand` creates protected commands, unlike the historical LaTeX command `\newcommand` (and unlike the command `\def` of TeX).
[25] The package lua-ul requires itself the package luacolor.

```
\NewDocumentCommand{\NameTable}{m}{{\PitonStyle{Name.Table}{#1}}}
\PitonOptions{language=SQL, raw-detected-commands = NameTable}
```

In the main document, the instruction:

```
Exemple : \piton{\NameTable{client} (name, town)}
```

produces the following output :

Exemple : client (nom, prénom)

The key `vertical-detected-commands` is similar to the key `raw-detected-commands` but the commands which are detected by this key must be LaTeX commands (with one argument) which are executed in *vertical* mode between the lines of the code.

For example, it's possible to detect the command `\newpage` by

```
\PitonOptions{vertical-detected-commands = newpage}
```

and ask in a listing a mandatory break of page with `\newpage{}` (the pair of braces `{}` is mandatory because the commands detected by piton are meant to be LaTeX commands with one mandatory argument).

```
\begin{Piton}
def square(x):
    return x*x  \newpage{}
def cube(x):
    return x*x*x
\end{Piton}
```

It would also be possible to require the detection of the command `\vspace`.

7.6.5 The mechanism "escape"

It's also possible to overwrite the computer 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 programming 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:

```
\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 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).

7.6.6 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=$,end-escape-math=$}
```

Note: 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 use.

```
\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) x^(2k+1)
9         return s
```

7.7 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]`.[26]
Note that, if the frame contains only one slide, it's recommended to write `\begin{frame}[fragile=singleslide]`.

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

When the package piton is used within the class beamer[27], the behaviour of piton is slightly modified, as described now. This is done via an environment `{actionenv}` of Beamer.

7.7.1 {Piton} and \PitonInputFile are "overlay-aware"

When piton is used in the class beamer, the command `\PitonInputFile` and the environment `{Piton}` (but not the environments created by `\NewPitonEnvironment`) 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}
```

7.7.2 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`[28]. ;
- one mandatory argument : `\action`, `\alert`, `\invisible`, `\only`, `\uncover` and `\visible` ; `detected-beamer-commands` (the names of the commands must *not* be preceded by a backslash).
- two mandatory arguments : `\alt` ;
- three mandatory arguments : `\temporal`.

These commands must be used preceded and following by a space. In the mandatory arguments of these commands, the braces must be balanced. However, the braces included in short strings[29] 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).

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

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

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

7.7.3 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 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:

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

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

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

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

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

7.8 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}`), the package footnote is loaded (if it is not yet loaded) and it is used to extract the footnotes.

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

Caution: The packages footnote and footnotehyper are incompatible. The package footnotehyper is the successor of the package footnote and should be used 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 7.6.4, p. 26).

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

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

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

```
def arctan(x,n=10):
    if x < 0:
        return -arctan(-x)[30]
    elif x > 1:
        return pi/2 - arctan(1/x)[31]
    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}
\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}
```

[30] First recursive call.
[31] Second recursive call.

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

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

7.9 Tabulations

Even though it's probably recommended to indent the computers listings with spaces and not tabulations[32], 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.

8 API for the developpers

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

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 by `piton.get_last_code()`. That concerns the LaTeX commands declared by the key `detected-commands` and its variants (cf. part 7.6.4) and the elements inserted by the mechanism "`escape`" (cf. part 7.6.5).
- `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 9.6.1, p. 39.

9 Examples

9.1 An example of tuning of the styles

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

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

[32] For the language Python, see the note PEP 8.

```
\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, except those in the value of the parameter `font-command`, whose initial value is `\ttfamily` (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
```

9.2 Line numbering

We remind that it's possible to have an automatic numbering of the lines in the computer 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).

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

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

In order to avoid that overlapping, it's possible to use the option `left-margin=auto` which will insert automatically a margin adapted to the numbers of lines that will be written (that margin is larger when the numbers are greater than 10).

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

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

9.3 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 listing with the key `width`.

```
\PitonOptions{background-color=gray!15, width=9cm}
\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
```

9.4 The command \rowcolor

The command `\rowcolor` has been presented in the part 4.2.3, at the page 8. We recall that this command adds a colored background to the current line (the *whole* line, and not only the part with text).
It's possible to use that command in a style of piton, as shown in p. 8, but maybe we wish to use it directly in a listing. In that aim, it's mandatory to use one of the mechanisms to escape to LaTeX provided by piton. In the following example, we use the key `raw-detected-commands` (cf. p. 26). Since the "detected commands" are commands with only one argument, it won't be possible to write (for example) `\rowcolor[rgb]{0.9,1,0.9}` but the syntax `\rowcolor{[rgb]{0.9,1,0.9}}` will be allowed.

```
\PitonOptions{raw-detected-commands = rowcolor} % in the preamble

\begin{Piton}[width=min]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

Here is now the same example with the join use of the key `background-color` (cf. p. 5).

```
\begin{Piton}[width=min,background-color=gray!15]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

As you can see, a margin has been added on both sides of the code by the key `background-color`. If you wish those margins without general background, you should use `background-color` with the special value `none`.

```
\begin{Piton}[width=min,background-color=none]
def fact(n):
    if n==0:
        return 1 \rowcolor{yellow!50}
    else:
        return n*fact(n-1)
\end{Piton}
```

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

9.5 Use with tcolorbox

The key `tcolorbox` of piton has been presented at the page 16.
If, when that key is used, we wish to customize the graphical box created by tcolorbox (with the keys provided by tcolorbox), we should use the command `\tcbset` provided by tcolorbox. In order to limit the scope of the settings done by that command, the best way is to create a new environment with the dedicated command `\NewPitonEnvironment` (cf. p. 9). That environment with contain the settings done by piton (with `\PitonOptions`) and those done by tcolorbox (with `\tcbset`).

Here is an example of such environment `{Python}` with a colored column on the left for the numbers of lines.

```
\usepackage{tcolorbox}          % in the preamble
\tcbuselibrary{breakable,skins} % in the preamble


\NewPitonEnvironment{Python}{m}
  {%
    \PitonOptions
      {
        tcolorbox,
        splittable=3,
        width=min,
        line-numbers,        % activate the numbers of lines
        line-numbers =       % tuning for the numbers of lines
         {
           format = \footnotesize\color{white}\sffamily ,
           sep = 2.5mm
         }
      }%
    \tcbset
      {
        enhanced,
        title=#1,
        fonttitle=\sffamily,
        left = 6mm,
        top = 0mm,
        bottom = 0mm,
        overlay=
         {%
            \begin{tcbclipinterior}%
                \fill[gray!80]
                    (frame.south west) rectangle
                    ([xshift=6mm]frame.north west);
            \end{tcbclipinterior}%
         }
      }
  }
  { }
```

In the following example of use, we have illustrated the fact that it is possible to impose a break of page in such environment with `\newpage{}` if we have required the detection of the LaTeX command

`\newpage` with the key `vertical-detected-commands` (cf. p. 26) in the preamble of the LaTeX document.
Remark that we must use `\newpage{}` and not `\newpage` because the LaTeX commands detected by piton are meant to be commands with one argument (between curly braces).

```
\PitonOptions{vertical-detected-commands = newpage} % in the preamble


\begin{Python}{My example}
def square(x):
    """Computes the square of x"""
    return x*x
def square(x):
    """Computes the square of x"""
    return x*x
def square(x):
    """Computes the square of x"""
    return x*x
def square(x):
    """Computes the square of x"""
    return x*x \newpage{}
def square(x):
    """Computes the square of x"""
    return x*x
...
def square(x):
    """Computes the square of x"""
    return x*x
\end{Python}
```

My example

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

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

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

9.6 Use with pyluatex

9.6.1 Standard use of pyluatex

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

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

We have used the Lua function `piton.get_last_code` provided in the API of piton : cf. part 8, p. 32.

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

```
\begin{PitonExecute}[background-color=gray!15]
def square(x):
    """Computes the square of x"""
    return x*x
print(f'The square of x is {square(12)}.')
\end{PitonExecute}
```

```
def square(x):
    """Computes the square of x"""
    return x*x
print(f'The square of x is {square(12)}.')
```

```
The square of x is 144.
```

It's also possible to use, in that environment, the mechanims for escape to LaTeX as previously (cf. p. 24).

```
\usepackage{luacolor,lua-ul}                % in the preamble
\PitonOptions{detected-commands = highLight}  % in the preamble
```

```
\begin{PitonExecute}[background-color=gray!15]
def square(x):
    """Computes the square of x"""
    \highLight{return x*x}
print(f'The square of 12 is {square(12)}.')
\end{PitonExecute}
```

```
def square(x):
    """Computes the square of x"""
    return x*x
print(f'The square of 12 is {square(12)}.')
```

```
The square of 12 is 144.
```

9.6.2 Use of the environment {pythonrepl} of pyluatex

The environment `{pythonrepl}` of pyluatex submit its content to Python and return what we obtain when we submit that code to a REPL (*read-eval-print loop*) of Python. We obtain a succession of instructions preceded by the prompt `>>>` of Python and values returned by Python (and the outputs of potential commands `print` of Python).

It's possible to give that to an environment `{Piton}` which will do the usual formatting and put on a colored background the lignes corresponding to the instructions provided to the Python interpreter (the color of that background may be changed with the key `prompt-background-color` whose inital value is `gray!15`).

Here a programmation of environment `{PitonREPL}` which does that job (for technical reasons, the `!` is here mandatory in the signature of the environment). It's not possible to process as previously (in the "standard" use of pyluatex) because, of course, the output of `{pythonrepla}` must be treated by piton. Therefore, it's not possible to use the escaping tools (`detected-commands`, `begin-escape`, etc.) in the code.

```
\ExplSyntaxOn
\NewDocumentEnvironment { PitonREPL } { ! O { } } % the ! is mandatory
 {
   \PitonOptions
     {
       prompt-background-color=blue!15 ,
       background-color=none, % for small margins
       #1
     }
   \PyLTVerbatimEnv
   \begin{pythonrepl}
 }
 {
   \end{pythonrepl}
   \lua_now:n
     {
       tex.print("\\begin{Piton}")
       tex.print(pyluatex.get_last_output())
       tex.print("\\end{Piton}")
       tex.print("")
     }
   \ignorespacesafterend
 }
\ExplSyntaxOff
```

Here is an example of use of that new environment `{PitonREPL}`.

```
\begin{PitonREPL}
    def valeur_absolue(x):
        """Renvoie la valeur absolue de x"""
        if x > 0:
           return x
        else:
           return -x

    valeur_absolue(-3)
    valeur_absolue(0)
    valeur_absolue(5)
\end{PitonREPL}
```

```
>>> def absolute_value(x):
...     """Computes the absolute value of x"""
...     if x > 0:
...        return x
...     else:
...        return -x
...
>>> absolute_value(-3)
3
>>> absolute_value(0)
0
>>> absolute_value(5)
5
```

In fact, it's possible to avoid the display of the prompts themselves (that is to say the strings `>>>` and `...`). Indeed, piton provides a style for those elements, called `Prompt`. The initial value of that style is empty, and that's why no action is done for those elements and they are displayed as they are. By using a value which is a function which gobbles its argument, it's possible to require that these prompts are not displayed.

```
\NewDocumentCommand{\Gobe}{m}{}[33]
\SetPitonStyle{ Prompt = \Gobe }
```

L'exemple précédent donne alors :

```
\begin{PitonREPL}
    def absolute_value(x):
        """Computes the absolute value of x"""
        if x > 0:
           return x
        else:
           return -x

    absolute_value(-3)
    absolute_value(0)
    absolute_value(5)
\end{PitonREPL}
```

[33] Here we have defined a function `\Gobe` but, in fact, it already exists in L3 with the name `\use_none:n`.

```
def absolute_value(x):
    """Computes the absolute value of x"""
    if x > 0:
       return x
    else:
       return -x

absolute_value(-3)
```
```
3
```
```
absolute_value(0)
```
```
0
```
```
absolute_value(5)
```
```
5
```

10 The styles for the different computer languages

10.1 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 of Pygments, as applied by Pygments to the language Python.[34]

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

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

10.2 The language OCaml

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

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

Here is an example:

```
let rec quick_sort lst =     (* Quick sort *)
  match lst with
  | [] -> []
  | pivot :: rest ->
      let left  = List.filter (fun x -> x < pivot) rest in
      let right = List.filter (fun x -> x >= pivot) rest in
      quick_sort left @ [pivot] @ quick_sort right
```

10.3 The language C (and C++)

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

Style	Use
`Number`	the numbers
`String.Short`	the characters (between `'`)
`String.Long`	the strings (between `"`)
`String.Interpol`	the elements `%d`, `%i`, `%f`, `%c`, etc. in the strings; that style inherits from the style `String.Long`
`Operator`	the following operators : `!= == << >> - ~ + / * % = < > & . | @`
`Name.Type`	the following predefined types: `bool`, `char`, `char16_t`, `char32_t`, `double`, `float`, `int`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `long`, `short`, `signed`, `unsigned`, `void` et `wchar_t`
`Name.Builtin`	the following predefined functions: `printf`, `scanf`, `malloc`, `sizeof` and `alignof`
`Name.Class`	the names of the classes when they are defined, that is to say after the keyword `class`
`Name.Function`	the name of the Python functions defined by the user *at their point of definition* (with the keyword `let`)
`UserFunction`	the name of the Python functions previously defined by the user (the initial value of that parameter is `\PitonStyle{Identifier}` and, therefore, the names of that functions are formatted like the identifiers).
`Preproc`	the instructions of the preprocessor (beginning par `#`)
`Comment`	the comments (beginning by `//` or between `/*` and `*/`)
`Comment.LaTeX`	the comments beginning by `//>` which are composed by piton as LaTeX code (merely named "LaTeX comments" in this document)
`Keyword.Constant`	`default`, `false`, `NULL`, `nullptr` and `true`
`Keyword`	the following keywords: `alignas`, `asm`, `auto`, `break`, `case`, `catch`, `class`, `constexpr`, `const`, `continue`, `decltype`, `do`, `else`, `enum`, `extern`, `for`, `goto`, `if`, `nexcept`, `private`, `public`, `register`, `restricted`, `try`, `return`, `static`, `static_assert`, `struct`, `switch`, `thread_local`, `throw`, `typedef`, `union`, `using`, `virtual`, `volatile` and `while`
`Identifier`	the identifiers.

10.4 The language SQL

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

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

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

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

10.5 The languages defined by \NewPitonLanguage

The command `\NewPitonLanguage`, which defines new computer languages with the syntax of the extension listings, has been described p. 10.
All the languages defined by the command `\NewPitonLanguage` use the same styles.

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

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,hspace,http-equiv,id,ismap,label,lang,link,%
      longdesc,marginwidth,marginheight,maxlength,media,method,multiple,%
      name,nohref,noresize,noshade,nowrap,onblur,onchange,onclick,%
      ondblclick,onfocus,onkeydown,onkeypress,onkeyup,onload,onmousedown,%
      profile,readonly,onmousemove,onmouseout,onmouseover,onmouseup,%
      onselect,onunload,rel,rev,rows,rowspan,scheme,scope,scrolling,%
      selected,shape,size,src,standby,style,tabindex,text,title,type,%
      units,usemap,valign,value,valuetype,vlink,vspace,width,xmlns},%
   tag=<>,%
   alsoletter = - ,%
   sensitive=f,%
   morestring=[d]",
  }
```

10.6 The language "minimal"

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

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

That language is provided for the end user who might wish to add keywords in that language (with the command `\SetPitonIdentifier`: cf. 7.5, p. 23) in order to create, for example, a language for pseudo-code.

10.7 The language "verbatim"

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

Style	Usage
`None...`	

The language `verbatim` doesn't provide any style and, thus, does not do any syntactic formating. However, it's possible to use the mechanism `detected-commands` (cf. part 7.6.4, p. 26) and the detection of the commands and environments of Beamer.

11 History

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

The successive versions of the file `piton.sty` provided by TeX Live are also available on the SVN server of TeX Live:

`https://tug.org/svn/texlive/trunk/Master/texmf-dist/tex/lualatex/piton/piton.sty`

Changes between versions 4.8 and 4.9

New keys `paperclip` and `annotations`.
The package piton is now provided with three TeX files: `piton-code.dtx` (for the code), `piton.tex` (the documentation in English) and `piton-french.tex` (the documentation in French).

Changes between versions 4.7 and 4.8

New key `\rowcolor`
The command `\label` redefined by piton is now compatible with hyperref (thanks to P. Le Scornet).
New key `label-as-zlabel`.

Changes between versions 4.6 and 4.7

New key `rounded-corners`

Changes between versions 4.5 and 4.6

New keys `tcolorbox`, `box`, `max-width` and `vertical-detected-commands`
New special color: `none`

Changes between versions 4.4 and 4.5

New key `print`
`\RenewPitonEnvironment`, `\DeclarePitonEnvironment` and `\ProvidePitonEnvironment` have been added.

Changes between versions 4.3 and 4.4

New key `join` which generates files embedded in the PDF as *joined files*.

Changes between versions 4.2 and 4.3

New key `raw-detected-commands`
The key `old-PitonInputFile` has been deleted.

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

The syntax for the relative and absolute paths in `\PitonInputFile` and the key `path` has been changed to be conform to usual conventions.
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 computer languages with the syntax used by listings. Therefore, it's possible to say that virtually all the computer languages are now supported by piton.

Changes between versions 2.7 and 2.8

The key `path` now accepts a *list* of paths where the files to include will be searched.
New commands `\PitonInputFileT`, `\PitonInputFileF` and `\PitonInputFileTF`.

Changes between versions 2.6 and 2.7

New keys `split-on-empty-lines` and `split-separation`

Changes between versions 2.5 and 2.6

API: `piton.last_code` and `\g_piton_last_code_tl` are provided.

Changes between versions 2.4 and 2.5

New key `path-write`

Changes between versions 2.3 and 2.4

The key `identifiers` of the command `\PitonOptions` is now deprecated and replaced by the new command `\SetPitonIdentifier`.
A new special language called "minimal" has been added.
New key `detected-commands`.

Changes between versions 2.2 and 2.3

New key `detected-commands`
The variable `\l_piton_language_str` is now public.
New key `write`.

Changes between versions 2.1 and 2.2

New key `path` for `\PitonOptions`.
New language SQL.
It's now possible to define styles locally to a given language.

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`.

Other documentation

The document `piton-french.pdf` (provided by the extension piton) contains a French translation of the current documentation. The file `piton-code.pdf` describes the implementation of the package piton (both L3 and Lua parts).

Acknowledgments

Acknowledgments to Yann Salmon and Pierre Le Scornet for their numerous suggestions of improvments.

Index

Contents