

docuLaTeX_{v1.0.0}

José Alberto López López

Contents

1	Introduction	1
2	Installation	1
3	Quick Start	1
4	Main Features	4
4.1	Nest Levels	5
5	Environments	5
5.1	Code Environments	6
5.2	List Environments	9
5.3	Special Description Environments	10
5.4	Other Environments	11
6	Delimiters	12
6.1	Caveats	14
7	Sections	15
7.1	Library Sections	15
8	Code Keywords	17
9	Specific Actions, Errors and Best Practices	17
10	List Environments and Their Elements	18
10.1	Descriptions	19
11	Tables	21
12	Comments	21
13	Informal References	21

14 JSON Configurations	22
15 CLI	26
16 Language Definition Files	29
17 Logs	30
17.1 Line State	30
17.2 Line Comparison	31
18 Known Issues	31
19 TODO	32
20 License	32

Introduction

A handy plain text format for writing code documentation is defined in this project, along with a parser that translates it to $\text{\LaTeX}$ output files, which can produce a final PDF file through $\text{\LuaTeX}$ or $\text{\XeTeX}$. The output PDF will have the same input content, but with a nice and professional format, which is inspired by the manual $\text{\TikZ \& PGF}$ *.

The main purpose is to provide a simple format that allows a quick and clean writing of code documentation, with a professional-looking final format, produced automatically through $\text{\LaTeX}$, but using a much simpler syntax.

Installation

The program is available as a Python package, so it can be easily installed through `pip` or `pip3`, as any other package or library, with a command like the following:

Command 2.1

```
pip3 install doculate
```

Quick Start

Once the package has been installed, it can be used through the `doculate` main command, by providing a `TEXT` file name as input argument, which must be in the same directory where the command is executed (or by prepending its path). The file content must fit the syntax described in this documentation or errors could arise.

Command 3.1

```
doculate input_file.txt
```

Next is an example of the input file content, which has several syntactic components defined by default (some are in Spanish, but they can be redefined). Only ensure indented lines use tabs instead of spaces (a search and replace of `UUUU` to tab character `\t` is recommended).

* This documentation was made with `docuLaTeX`.

Example 3.1

Python

Python language was released in 1991.

It is an interpreted language with a friendly syntax.

Next is a code example written in Python:

Ejemplo:

```
import pandas as pd
from numpy import array
```

```
# Line comment
```

```
x = array([1, 2, 3])
```

```
s = "This is a string"
print(s)
```

```
def my_func(a: int, b: float):
    return a + b
```

Main disadvantages that Python has in contrast with other languages are:

- Speed: it is slower than most of the compiled languages.
- High level: it is a drawback if more control of aspects like memory managing are required.

It is b"important" to mention that, with help of some libraries, the efficiency can be improved considerably.

Keywords

This section covers some Python keywords.

*for

Instruction to execute a i"for" loop.

Sintaxis:

```
for num in range(begin_num, end_num):
```

```
# Actions...

The `break` sentence can be used to q"abort" the loop execution.

*if
*elif
*else

Control structures for conditional sentences.
Their main features are:

-If the `if` condition evaluates to `True`, its content will be executed.
-If the `if` condition evaluates to `False`, and `elif` condition to `True`
  `, `elif` content will be executed.
-If the `if` and `elif` conditions evaluate to `False` or `None`, the `
  else` block will be executed.

Boolean operators `and`, `or` and `not` can be placed in conditions.

*print()

This function allows printing information on screen during program execution.

Opciones:

-`sep`: separator to be printed between each argument.
-`end`: concatenated string at end of the printed content.
-`file`: leads the output string to a file, instead of the terminal.

Its arguments can be separated by commas, for example: `print("Hello", "world
!")`.
```

With the previous file, the `--programming-languages` option can be used to apply a predefined Python syntax highlighting.

Command 3.2

```
doculate input_file.txt --programming-languages Python
--title "Python Example" --author Anonymous
```

After executing this command, some `.tex` files should be generated, from which the final PDF file can be built.

Main Features

The parser processing consists of analyzing the input `TXT` file line by line, to convert or map each line to the equivalent `LaTeX` format.

A line could be identified as some of the following elements:

- Section title.
- Text line.
- Blank line.
- Environment beginning.
 - Code environment content.
 - List, enumeration or description element beginning.
 - List, enumeration or description element content.
 - Math environment content.
 - Table row.
- Code keyword.
- Specific action.
- Best practice.
- Error.
- Code keyword, specific action or error content.
- Paragraph title.
- Line comment.
- Multiline comment beginning.
 - Multiline comment content.
- Multiline comment end.
- Informal reference.

Nest Levels

Each time an environment starts through its identifier, its content must be indented an additional level with respect to its identifier.

Indentations must be inserted using tabs, one for each extra level.

Each tab inserted from the beginning of a line will correspond to an additional nest level, unless the line is in a code environment, in which case it will be ignored; also, if the line pattern is ignorable, like empty or commented lines, the line will be ignored too.

A non-empty line having less initial tabs than the previous one means the end of one or more previous environments or elements content.

Environments

There are 2 main types of environments: list environments and code environments. List environments can be normal bulleted list, enumerations or descriptions. Code environments can be of various types. Examples of the syntax to generate code, list, enumeration and description environments are shown below.

Example 5.1

Previous text.

Ejemplo:

code environment content

More text...

- List element 1.
Continuation/content of list element 1.
- List element 2.
- List element 3.

More text...

1. Enumeration element 1.
Continuation/content of enumeration element 1.
2. Enumeration element 2.
3. Enumeration element 3.

More text...

```
-Element 1: description element 1.
  Continuation/content of description element 1.
-Element 2: description element 2.
-Element 3: description element 3.
```

As can be observed, the identifier that starts an environment (like `Ejemplo:`) is optional for list environments.

The different kinds of available environments, and the identifiers to begin them are listed below.

Code Environments

Identifiers of code or list environments can be redefined or extended through the JSON configurations options: `environments.<id_env>.values` and `environments.<id_env>.add_values` (see *JSON Configurations*).

Example Identifiers: `Ejemplo`, `Ej..`

Examples Identifiers: `Ejemplos`, `Ejs..`

Anti-example Identifiers:

`Antiejemplo`

`Anti-ejemplo`

`Ant. ej.`

`A. ej.`

Anti-examples Identifiers:

`Antiejemplos`

`Anti-ejemplos`

`Ant. ejs.`

`A. ejs.`

Syntax Identifiers: `Sintaxis`.

Code syntax Identifiers: `Sintaxis código`.

Code Identifiers: Código, Cod..

Command Identifiers: Comando, Comandos código.

Configuration Identifiers:

Configuraciones

Configuración

Confs.

Conf.

Definition Identifiers: Definición, Def..

Definitions Identifiers: Definiciones, Defs..

Code example Identifiers:

Código ejemplo

Cod. ej.

Códigos ejemplo

Cods. ej.

Code anti-example Identifiers:

Código antiejemplo

Códigos antiejemplo

Cod. antiej.

Cods. antiej.

Cod. aej.

Cods. aej.

Function Identifiers:

Función	Cód. función
Func.	Cód. func.
Funciones código	Código funciones
Funciones cód.	Cód. funciones
Funcs. cód.	Cód. funcs.
Código función	

Procedure Identifiers:

Procedimiento	Cód. procedimiento
Proc.	Cód. proc.
Procedimientos código	Código procedimientos
Procedimientos cód.	Cód. procedimientos
Procs. cód.	Cód. procs.
Código procedimiento	

Class Identifiers:

Clase	Cls. cód.
Clase código	Código clase
Clases código	Código clases
Clase cód.	Cód. clases
Clases cód.	Cód. clas.
Clas. cód.	Cód. cls.

The JSON configuration IDs of these environments are:

<code>syntax</code>	<code>antiexample</code>	<code>codeantiexample</code>	<code>function</code>
<code>codesyntax</code>	<code>antiexamples</code>	<code>commandcode</code>	<code>procedure</code>
<code>example</code>	<code>code</code>	<code>configuration</code>	<code>definition</code>
<code>examples</code>	<code>codeexample</code>	<code>class</code>	<code>definitions</code>

The title bar color of code environments can be restored through `code_environments_title_colors` configuration. If it's only required to change one or few environments, configuration `add_code_environment` can be used instead, since `code_environments_title_colors` will set the title bar color of all non specified environments to the default `code_title_color` configuration. Code background color can also be modified via `code_background_color` configuration, and its frame lines as well, through `code_frame_color`. Colors can be specified in RGB format, with 3 comma separated whole numbers from 0 to 255; or by name, like its default values shown in *JSON Configurations* section.

The environment name shown in the title bar corresponds to the first value for the configuration list `environments.<id_env>.values`. Currently, new environments can't be defined and added to existing ones, but a trick that can be used to emulate this is changing the first value of the list `environments.<id_env>.values` for an unused environment to a new appropriate name. Since `<id_env>` only acts as identifier at programming level, the environment name shown in the document will be the specified in the new value.

List Environments

As mentioned before, the next list environments identifiers are optional.

List Identifiers: `Lista`.

Enumeration Identifiers: `Enumeración`, `Enum..`

Description Identifiers: `Descripción`.

The content of these environments must be indented one additional level in any case. In addition, the content of each list element, after the first line which has the `TXT` bullet, also must be indented one level.

The JSON configuration IDs for these environments are: `list`, `enumerate`, and `description`

Special Description Environments

Most of the next description environments are designed to describe code components, like options, function arguments, class members, etc.

Options Identifiers: Opciones.

Fields Identifiers: Campos, Llaves, Claves.

Arguments Identifiers:

Argumentos

Args.

Argumentos requeridos

Args. req.

Argumentos opcionales

Args. opt.

Parameters Identifiers:

Parámetros

Params.

Parámetros requeridos

Params. req.

Parámetros opcionales

Params. opt.

Class Identifiers: Clases.

Members Identifiers: Miembros.

Methods Identifiers: Métodos, Méts..

Attributes Identifiers: Atributos, Atribs..

Values Identifiers:

Valores

Vals.

Posibles valores

Pos. vals.

Constantes

Configuration options Identifiers:

Opciones de configuración

Opc. conf..

Commands Identifiers: Comandos.

Subcommands Identifiers: Subcomandos, Subcom..

Functions Identifiers: Funciones, Funcs..

Procedures Identifiers: Procedimientos, Procs..

The JSON configuration IDs for these environments are:

options	functions
values	members
arguments	methods
attributes	fields
parameters	commands
classes	subcommands
configurationoptions	procedures

Other Environments

Next are alternative environments to insert math expressions, tables or insert $\LaTeX$ code directly. Like list environments identifiers, table identifier is also optional.

Equation Identifiers:

Texto matemático

Mat.

Ecuación

Ec.

Math expressions Identifiers:

`Expresiones matemáticas`

`Ecuaciones`

`Mats.`

`Ecs.`

Aligned math expressions Identifiers:

`Expresiones matemáticas alin.`

`Ecuaciones alin.`

`Alin. Mats.`

`Alin. Ecs.`

`A. Ecs.`

Table Identifiers: `Tabla.`

LaTeX Identifiers: `LaTeX`, `Latex`, `latex`.

Comment Identifiers: `Comentario`, `Comentarios`.

The JSON configuration IDs for these environments are:

<code>math</code>	<code>mathexpressions</code>	<code>comment</code>
<code>table</code>	<code>alignedmathexpressions</code>	<code>latex</code>

Delimiters

A text line can contain one or more inline delimiters, which also can be nested. Inline delimiters allow to set different formats or styles to text pieces.

There are 2 main delimiter types: normal and no nesting. Normal delimiters allow to nest other delimiters inside them, they are useful, for example, to place inline code inside italic text or similar things. No nesting delimiters take its content as is, until close delimiter is found, so they are suitable to insert inline code.

Several delimiters are defined through a distinctive constructor character or string, from which three variants of a delimiter are created: normal, no nesting and no ambiguity normal,

using the default base characters `'`, `"` and `{`, `}`, respectively. For example, the constructor character of the bold delimiter variants is `b`, so its variants are `b'...'`, `b"..."` and `b{...}`.

Normal delimiters are more practical, but they are also more error-prone. When the final character or string, previous to a normal closing delimiter match with a constructor character of another delimiter, the parser will wrongly identify it as a opening delimiter. That is why no ambiguity normal delimiters exist, which have much less probability of falling in those situations.

The different kinds of delimiters are listed below. Unless another specification, they are defined through the explained character constructor form.

Unformatted normal text constructor character: `t`.

Bold constructor character: `b`.

Italics normal: `i'...'`, `i"..."`, `i{...}`; no nesting: `"..."`.

Slanted constructor character: `s`.

Uppercase (*small caps*) constructor character: `sc`.

Quotes constructor character: `q`.

Underline constructor character: `u`.

Code no nesting: ``...``.

Referenced code no nesting: `r`...``. Usable, but not implemented yet.

Special code no nesting: `e"..."`. Code containing character ```.

Special referenced code no nesting: `re"..."`. Usable, but not implemented yet.

Math no nesting: `$...$`.

File name no nesting: `f"..."`.

Directory path no nesting: `d"..."`.

Label no nesting: `l"..."`.

Label reference constructor character: `r`. Usable, but not implemented yet.

Cite no nesting: `c"..."`. Usable, but not implemented yet.

Footnote constructor character: `fn`.

Web link no nesting: `h"..."`.

Keyboard key no nesting: `k"..."`.

Button no nesting: `bu"..."`.

Line comment no nesting: `cc"..."`.

LaTeX no nesting: `L"..."`, `LaT"..."`, `LaTeX"..."`. `LATEX` code injection.

The JSON configurations for delimiters identifiers (`<delim_id>`) are: `italics`, `normal_text`, `bold`, `math`, `slanted`, `small_caps`, `quotes`, `underline`, `code`, `referenced_code`, `special_code`, `referenced_special_code`, `comment`, `file`, `directory`, `label`, `reference`, `cite`, `footnote`, `hyperlink`, `keyboard`, `button` and `LaTeX`.

Both default constructor characters and default base characters can be changed using next configurations:

- `delimiters.<delim_id>.signature_constructor`
- `default_inline_normal_delimiter`
- `default_inline_no_nesting_delimiter`
- `default_unambiguous_normal_delimiters`

New delimiters can also be added to existent types, through one or more constructor characters specified in `delimiters.<delim_id>.add_signature_constructor` configuration.

Or they can be also explicitly added or redefined as arbitrary string pairs, using next configurations:

- `delimiters.<delim_id>.normal`
- `delimiters.<delim_id>.add_normal`
- `delimiters.<delim_id>.no_nesting`
- `delimiters.<delim_id>.add_no_nesting`

In *JSON Configurations* section, several examples of these configurations can be found.

Caveats

Caution must be taken to open and close delimiters correctly, especially nested ones (normal type); as well as making sure that there is not ambiguity when closing normal delimiters. For this reason it's recommended to prefer no nesting delimiters, using normal ones only when is really necessary, and also preferring the no ambiguity version.

If one or more characters that are wanted to be interpreted as text, match some delimiters, the string must be escaped with `\`. Otherwise, it's possible that the parser interprets it as a delimiter, causing an error or undesired behavior. For example, character `}` is considered by default both as closing delimiter and as special replacement character, that must be translated

to `.tex` file as `\}`. This is a special case where the character must be escaped when is required to be interpreted as text (unless it is not considered as delimiter by a redefinition), otherwise could be situations where the corresponding open delimiter is previously inserted and the character is erroneously taken as its closing pair.

The parser ignores special string replacements that match some delimiter, giving priority to be treated as delimiters, so the users must escape them manually if they require them to be interpreted as text.

Sections

There are up to six section levels and two final paragraph and subparagraph levels. For example, a document could be divided into: book, part, chapter, section, subsection and subsubsection; using all of its section and paragraph levels, or it could start from a lower level, like chapter, which is the upper level by default.

The root level for sections can be set through the `root_section` configuration. This configuration can be useful to set the format and spacing of section titles in a handy way. For example, in small documents, values `section` or `subsection` can be used to set a continuous change of sections, without page breaks, since the `chapter` default value induce a page break every time a new chapter starts, which has an unnecessary isolating effect for very short sections.

Like Markdown, a section title of level 1 starts with a `#`, then a `##` for titles of level 2, and so on, until reaching level 4 (or level 6 if all levels are used). A paragraph title starts with `#####`, while `#####` is used for subparagraphs, regardless of the current section level in the document.

The character that identifies a section title pattern (`#` by default) can be changed through the `header_mark` configuration.

Library Sections

The symbol `+` (`library_mark` configuration) is used to differentiate section titles of libraries, frameworks, packages, etc., from normal section titles, beginning with a `#`. A library section title can have a special format, with the library name shown in code style by default and optionally a constant prefix and/or suffix, like “Library” or “Framework” words before the library name.

Using `+` is analogous to `#`. By default, the same number of `+` and `#` indicates the same section level. So subsections patterns like `++` or `+++` can be used, for instance, to open sections of sub-libraries, sub-modules, etc., defined inside a principal library section; or to

begin libraries sections inside normal sections.

Another feature of this library syntax is the possibility of stack same-level library names in consecutive lines in the **TXT** file. Then these names will be placed together in the library title, with a specific format, like comma separated or stacked. This can be useful to document libraries with popular alias, so original and alias names are included in the library title; or to include the import identifiers used in code in addition to the library name. With this syntax, an easy and fast search for a library section or any of its alias in the **TXT** file is still possible, since all starts with **+** symbol.

If is not necessary to use library sections or its features are insufficient, normal section format will always be the base option, since any library section format can be manually replicable using normal sections.

Next is an example of a possible usage of library sections:

Example 7.1

```
+ numpy
+ np

Text...

*Numpy specific action.
    Text...

Text...

*numpy_function()
    Text...

Text...

# Graphics

Text...

++ matplotlib

Text...
```

Code Keywords

A keyword or term documentation is identified when a line starts (by default) with the `*` character and it doesn't end with a dot, in which case the line would be identified as a specific action, error or best practice.

What follows the `*` character is placed in the PDF document in code format, with highlighted syntax if applies, and with no need of using code delimiters. The subsequent content related to the term must be indented one level in `TXT` file.

Language keywords are principal elements in a code documentation, so they will always be at the top level of the current section in the document, that is, without nesting them in themselves or in other elements like environments. This also applies for specific actions, errors and best practices. For that reason, the way to establish when a key documentation starts and finish differs from a list or code environment.

The way to detect when a keyword documentation content has started is through identifying its header pattern. From that, the term content must be indented one level, and the way to finish this term pseudo-environment will be only when one of the following situations happens:

- When another keyword, specific action, error or best practice header appears.
- When a new chapter, section, subsection, etc., starts.
- By continuing with text or other kind of content that does not belong to the keyword documentation, by removing the initial indent of the keyword content.

All the content between the beginning of the keyword and any of the previous situations is considered as part of the keyword documentation.

It is neither expected nor encouraged to nest the documentation of a keyword in another keyword or environment, since for that there are description code environments, like `Options`, `Arguments`, etc., where sub-terms or related keywords can be documented inside another principal term.

Specific Actions, Errors and Best Practices

Specific actions, errors and best practices are alternative documentation elements to keywords, which content is related to a concrete subject, as their names suggest. They follow a

similar syntax to keywords: by default they start with `*`, followed by its header, which must end with a dot.

Headers examples of specific actions, errors or best practices could be:

- *Measuring the execution time of a code block.*
- *Conversion error when casting int to string.*
- *Use short sentences to improve readability.*

in which delimiters also can be used to apply format.

The starting character (`*`) that detects an action, error or best practice header can be changed through `action_error_best_practice_mark` configuration. Currently these 3 elements are indistinguishable in the final document, only being able to differentiate them by the section title that contains them or by their own headers.

Like keywords, their content must be indented one additional level.

List Environments and Their Elements

The content (subsequent lines) of a list element must be nested one additional level respect to the first line, which is the one having the `TXT` bullet.

If there is no environment identifier, neither different `TXT` bullet are used for each kind of list, the pattern of the first element, in conjunction with the configuration values for lists, will decide what kind of list environment element (normal or description) will be used for the rest of the elements.

Because in some situations the first line of a normal or description list element can coincide in their patterns, regardless if that line contains or not a colon, next three forms to differentiate those elements are defined to avoid ambiguities:

1. Set different `TXT` bullets for each kind of list, normal and description.
2. Explicitly start environments by their identifiers: `Lista` or `Descripción`, or their variants.
3. Set the `colon_rule_when_no_identifiers` configuration value to `true`, which tells the parser that if no environment identifier is present, normal lists will never have a colon in its first line, and descriptions will always have it, to avoid ambiguities. This rule leaves the responsibility on the user who always must meet the rule, otherwise undesired behaviors could appear.

One of the above three alternatives must be applied or an error could be raised because ambiguity, or, if the `default_list_elem` is set to true, the normal list environment will always be used by default when no identifier is present.

Descriptions

There are two styles for description lists: `sameline` and `nextline`. First one consists of placing the element to describe along with its description in the same line; whilst in `nextline` format, the description is placed below the element to be described.

Next two examples use the correct syntax to generate the two kinds of description lists, remembering that the PDF format for all elements in a list is determined by the first element in the corresponding TXT list.

Example 10.1

Previous text...

```
%% Nextline description
-Element 1:
    Description content...
-Element 2:
    Description content...
```

Unindented text to finish implicit environment and start next one...

```
%% Nextline description
-Element 1
    Description content...
-Element 2: Description content...
```

Previous text followed by environment identifier (optional)...

Description:

```
%% Sameline description
-Element 1: Description content...
-Element 2:
    Large description content...
-Element 3: Description content...
```

More text...

Sameline descriptions always must have :_□ as element-description separator.

If a description list doesn't have an identifier and is incomplete, that is, some elements don't have a description, and it's required to be identified as description list instead of normal list (removing the bullet), adding two spaces at the end of the first element is enough to achieve that. With this, all the elements in the list will be interpreted as a description elements instead of bulleted ones. For example:

Example 10.2

```
-`visible`□□
-`hidden`
```

Or, a description could be added to the first element as alternative to this trick.

List elements where this syntax is used will be treated as sameline description elements, due sameline format is more suitable for short descriptions. Nextline descriptions are more suitable for long and formal descriptions, therefore when a nextline description is going to be used is encouraged that all elements, or mostly at least, contains its corresponding descriptions. However, a nextline description can also start as sameline description and later it can be turned easily into a nextline description only by changing its first element to nextline format.

If for some reason a TXT nextline description is wanted to be treated as sameline description in the final document, the same two final spaces trick in the first element can be applied.

Example 10.3

```
Opciones:

-`visible`□□
    element description.
-`hidden`
```

Due to the aforementioned reasons, it doesn't exist, nor is it encouraged a way to interpret a sameline description environment as a nextline one in the output document.

By default, the text of the described elements will appear in bold format, but delimiters can be used to apply a different style each in element and its description. For example, code delimiters (``...``) can be applied in environments like `Options`, as it's shown in previous example, since code style is not applied yet by default in such description environments.

Tables

Like list environments, a table can be inserted with or without its identifier (`Tabla`).

Single column tables should have an explicit identifier, or, in implicit environments, each row should end with one or more tabs, otherwise it could not be identified as a table row. This only applies for single column tables, for multi-column tables that property is optional.

Nothing can be nested in a table row or cell. In other words, it can't be anything more indented than the first row in a table environment.

If an empty cell is required, a blank space can be placed in the corresponding cell.

Currently the number of columns is obtained from the first row, so if it contains empty cells, they must be indicated with a blank space. Final and consecutive empty cells of all rows (except the first) can be omitted.

Comments

A line comment token is provided (`%%` by default).

Block comments are also supported, as well as inline comments, through opening and closing block comments delimiters (`%%` and `%%`) by default). In addition, delimiters `cc"..."` allow to insert inline comments only; while comment environments identifiers `Comentario` and `Comentarios` allow to insert block comments only.

Unless block comment delimiters (`%%`, `%%`) start and end in the same line, it is not allowed to begin a block comment in a line where the open delimiter (`%%`) is preceded by any character except blank spaces or tabs, in which case the line comment token should be used. Closing a block comment and then insert non empty space characters on the same line is also forbidden.

By default, the block comments content is not translated to the `.tex` file as comment. Comments that start and finish on the same line will be kept in the `.tex` file as comments as well. To change this rules, JSON configurations options `parse_block_comments`, `parse_line_comments` and `parse_inline_line_comments` are provided.

Informal References

A simple format for informal references is defined, which could or could not be shown in the final document, by setting the option `parse_informal_references`. This format is designed

to annotate quick references in the `TEXT` file, since a complete `BIBTEX` reference (which is also considered to be included in future) takes more time to elaborate.

An informal or quick reference consists of a line that starts with a pair of empty brackets, followed by arbitrary text that corresponds to the reference content.

If it's indicated to show the informal references in the final document, by default they will be shown exactly as they are in the `TEXT`.

Example 13.1

```
[] LaTeX in 24 Hours, Dilip Datta.  
[] El Universo LaTeX, Rodrigo De Castro Korgi.  
[] https://tex.stackexchange.com/
```

Quick references can be written in the `TEXT` as a reminder to later pass them to a standard format. As future work, in addition to implement a complete `TEXT` format that can be translated into a `BIBTEX` document, it is also considered to make an intermediate format, which only takes one or few lines, where most important `BIBTEX` fields can be defined in a handy way, and also translated to the equivalent `BIBTEX` output.

JSON Configurations

Several configurations options can be defined through a JSON file, which are related to the input `TEXT` syntax components, the output PDF format, among others features. If no configuration file is specified, the process will search for the following default file names in the working directory, using the first one found as configuration file.

- `conf.json`
- `config.json`
- `configuration.json`
- `doculate.json`
- `docuLaTeX.json`

Following is an example of a JSON configuration file with all the existing configurations and its default values, except for environments and delimiters configurations, where only an example for one environment and one delimiter is given. Most of the options are not allowed through CLI, hence if a detailed configuration is required, a JSON file must be used. If `null` configurations are encountered, default values shown below will be used instead, the same applies for missing configurations in the JSON file.

Configurations without a default value in next example means that they are obtained indirectly from other configurations or from a user input. A comment with a short description is placed next to these particular configurations.

Code 14.1

```
{
    // Use the indicated configuration file instead this.
    "config_file": null,
    // Main input, must be specified.
    "txt_input_file": ,
    "output_dir": "./",
    // Same value as "output_dir" by default.
    "output_line_state_log_dir": ,

    // Same value as input file name, from "txt_input_file", by default.
    "title": ,
    "author": "",
    "date": "\\today",

    "main_font_size": "12pt",

    "page_left_margin": "25mm",
    "page_right_margin": "25mm",
    "page_top_margin": "40mm",
    "page_bottom_margin": "30mm",

    "front_page_left_margin": "50pt",
    "front_page_right_margin": "50pt",
    "front_page_top_margin": "1ex",
    "front_page_bottom_margin": "1ex",

    "title_font_size": "200",

    "tableofcontents_name": "Contents",

    "root_section": "chapter",

    "header_mark": "#",
    "library_mark": "+",
    // Same value as "header_mark" by default.
    "paragraph_mark": ,
    "section_marks_unit": "-",
```

```
"term_mark": "\\*",
"action_error_best_practice_mark": "\\*",
"list_mark": "-",
"description_mark": "-",
"informal_reference_mark": "\\[\\]",
"block_comment_delimiters": ["(%", "%)"],
"line_comment_symbol": "%%",

"LaTeX_special_replacements": {
  "\\": "\\textbackslash{}",
  "^": "\\textasciicircum{}",
  "_": "\\_",
  "&": "\\&",
  "%": "\\%",
  "$": "\\$",
  "#": "\\#",
  "{": "\\{",
  "}": "\\}",
  "~": "\\textasciitilde{}",
  "TeX": "\\TeX{}",
  "LaTeX": "\\LaTeX{}",
  "BibTeX": "\\textsc{Bib}\\TeX{}"
},
"LaTeX_code_special_replacements": {
  "\\": "\\\\",
  "%": "\\%",
  "{": "\\{",
  "}": "\\}",
  "#": "\\#"
},
"LaTeX_url_special_replacements": {
  "\\": "\\\\",
  "%": "\\%",
  "#": "\\#"
},

// Dictionary for adding values to "LaTeX_special_replacements".
"add_LaTeX_special_replacements": null,
// Dictionary for adding values to "LaTeX_code_special_replacements".
"add_LaTeX_code_special_replacements": null,
// Dictionary for adding values to "LaTeX_url_special_replacements"
"add_LaTeX_url_special_replacements": null,
```

```
"colon_rule_when_no_identifiers": true,
"default_list_element_when_no_identifiers": true,

"parse_informal_references": true,
"parse_block_comments": false,
"parse_line_comments": true,
"parse_inline_line_comments": true,

"header_pre_chapter_mark": "Chapter ",
"header_pos_chapter_mark": ". ",

"front_page_file_name": "front_page.tex",
"preamble_file_name": "preamble.tex",

// Plain text by default, no syntax highlighting.
"main_programming_language": ,
// List of supported programming languages.
"programming_languages": [],
// List of file (path/)names.
"programming_languages_definition_files": [],
"use_file_name_if_no_main_language": false,

"code_background_color": "gray!10!white",
"code_title_color": "gray!70!black",
"code_frame_color": "gray!70!white",
"code_environments_title_colors": {
  "Sintax": "violet!30!gray",
  "CodeSintax": "violet!30!gray",
  "Example": "green!30!gray",
  "Examples": "green!30!gray!85!black",
  "AntiExample": "red!30!gray",
  "AntiExamples": "red!30!gray!85!black",
  "Code": "blue!30!gray",
  "CodeExample": "green!30!gray",
  "CodeAntiExample": "red!30!gray",
  "CommandsCode": "black!30!gray",
  "Configuration": "cyan!30!gray",
  "Class": "blue!30!gray",
  "Function": "blue!30!gray",
  "Procedure": "blue!30!gray",
  "Definition": "yellow!30!gray",
  "Definitions": "yellow!30!gray!85!black"
},
```

```
// Same fields as "code_environments_title_colors".
"add_code_environments_title_colors": null,

"code_environments_font_size": "small",

"environments": {
  "example": {
    "values": ["Example", "Ex."],
    "add_values": []
  }
},

"default_inline_normal_delimiter": "'",
"default_inline_no_nesting_delimiter": "\"",
"default_unambiguous_normal_delimiters": [{"", "}"],

"delimiters": {
  "bold": {
    // Automatically generates the values shown in "normal" and "no_nesting".
    "signature_constructor": "b",
    "add_signature_constructor": null,
    "normal": [["b'", ""], ["b{", "}"]],
    "add_normal": null,
    "no_nesting": ["b\\\"", "\\\""],
    "add_no_nesting": null
  }
}
}
```

At the moment, JSON configurations are not fully tested and verified to mitigate errors and undesired behaviors, so it's requested not to do anything too crazy.

CLI

After installing the package, the user will be able to use it through a command line interface. The fastest way to do so is by executing the `doculate` command, followed by the name of the input file, placed in the same directory where the command is executed (or prepending its path).

Command 15.1

```
doculate input_file.txt
```

Commands and options available for the package are described below:

doculate

Main package command. With no arguments, it will search a configuration file in the current directory, with any of the names listed in *JSON Configurations* section. If no file is found neither the `txt_input_file` configuration is specified, a message will report it and no action will be performed. As main argument receives the file name to process, which is equivalent to the `txt_input_file` JSON configuration. Upon receiving the input file name with no additional configurations, the corresponding `.tex` file will be generated, along with the preamble `.tex` files, necessary to generate the final PDF file by executing the `preamble.tex` file with Lua $\text{\LaTeX}$ or Xe $\text{\LaTeX}$.

Below the exclusive `doculate` command options are listed:

- `--only-parse-file` flag to only generate the parsed `.tex` file, without preamble files.
- `--only-create-preamble-files` flag to only generate the preamble files.
- `--only-analyze-file` flag to only verify the correctness of the input TXT file, generating a `.log` file to consult each line state interpreted by the parser.
- `--parsed-file-name` sets the output `.tex` file name. By default is the same name as the input file.
- `--preamble-file-name` sets the output `.tex` preamble main file name (by default `preamble.tex`).
- `--line-state-file-name, -s` sets the output line state log file name.

Optional configurations available for `doculate` command and for sub-commands as well are listed below:

- `--help, -h` shows information about available commands and its usage.
- `--config-file, -c` configuration file name to be used. Configurations in this file will replace the equivalent configurations in the default configuration file.
- `--output-dir, -d` output directory where output `.tex` files will be placed.
- `--title, -t` title shown in the final document front page (by default the same name as input file name). Titles with blank spaces must be enclosed in quotes (`"`).
- `--author, -a` author shown in the final document front page. Names with blank spaces must be enclosed in quotes (`"`).

`--programming-languages, -p` space separated programming languages names, used for code syntax highlighting. Currently supported programming languages and tools are Python, C++, bash, L^AT_EX and Git. If some other language is required, or redefining a predefined one, the corresponding language definition files must be indicated in the `--programming-languages-definition-files` option (see *Language Definition Files*).

`--programming-languages-definition-files, -f` language and style definition files to apply code syntax highlighting for specific languages (see *Language Definition Files*).

Next, available `doculate` subcommands are described:

`parse-file, p`

Receives the input `TXT` file and translates it in the equivalent `.tex` file, without generating preamble files. Is equivalent to use the `--only-parse-file` flag with the main command. Its exclusive options are:

`--output-file, -o` output `.tex` file name.

`--output-log` output log file name with line comparison between input and output files. The file only will be generated if this option is present.

`create-preamble, pr`

Receives the `.tex` content file name that will be included in the preamble file (for instance, the output file from `parse-file`) and generates the preamble files. Is equivalent to use the `--only-create-preamble-files` flag with the main command. Its exclusive options are:

`--output-file, -o` output `.tex` preamble main file name.

`analyze-file, a`

Receives the input `.txt` file and perform an analysis that could generate errors if the syntax is not correct. It generates a log file with the state of each line. Is equivalent to use the `--only-analyze-file` flag with the main command. Its exclusive options are:

`--output-file, -o` output log file name.

Some subcommands usage examples are shown below:

Comandos código:

```
doculate parse-file input_file.txt
doculate pr preamble.tex
doculate analyze-file input_file.txt -o line_state.log
```

Language Definition Files

If syntax highlight is required for some particular programming language, or to redefine predefined languages definition, language definition files can be included to indicate how syntax highlighting should be applied. For each language to be defined, two files are required: one to indicate the language keywords and another to define their style.

At the moment there is no JSON configuration options to add or change a language definition, so this must be done in a $\text{\LaTeX}$ specific format, in the aforementioned definition files.

Next, an example of the definition files content for Python language is presented:

`python_language.tex`

Example 16.1

```
\lstdefinlanguage{Python}{
  % Core keywords
  morekeywords = {as, and, break, class, continue, def, del, elif, else,
    for, from, if, import, in, is, not, or, pass, print, return, try, while},
  % Built-ins
  morekeywords = [2]{abs, all, any, basestring, bin, bool, bytearray,
    callable, chr, classmethod, cmp, compile, complex, delattr, dict, dir},
  sensitive = true,
  morecomment = [1]\#,
  morestring = [b]',
  morestring = [b]",
  morestring = [s]{''}{''},
  morestring = [s]{""}{""},
  style = Python
}
```

python_style.tex*Example 16.2*

```
\definecolor{purple-python-keywords}{RGB}{153, 0, 153}
\definecolor{green-python-strings}{RGB}{0, 153, 0}

\lstdefinestyle{Python}{
  keywordstyle = \color{orange!80!black}\bfseries, % Core keywords
  keywordstyle = {[2]\color{purple-python-keywords}}, % Built-ins
  stringstyle = \color{green-python-strings},
  commentstyle = \color{gray}\itshape
}
```

The definition files names must be indicated in `--programming-languages-definition-files` option, separating each file by spaces, and the language name (which is defined in `\lstdefinelanguage` and `\lstdefinestyle` commands) must be placed in the `--programming-languages` option, as is shown in next example:

Command 16.1

```
doculate input_file.txt --programming-languages-definition-files python_language
.tex python_style.tex --programming-languages Python
```

If a more advanced definition is required, the `listings` package documentation can be consulted (<https://ctan.math.washington.edu/tex-archive/macros/latex/contrib/listings/listings.pdf>).

Logs

Line State

The output log file of `analyze-file` subcommand or `--only-analyze-file` option, contains, for each line (in addition to the line itself), the next line state features:

1. Line number (beginning from 1).
2. Environment stack.
3. Identifier (text, blank line, content of, environment, comment, etc.).

4. Nest level (initial tabs number).

Line Comparison

The `parse-file` subcommand output log from `--output-log` option contains, for each TXT line, that original line and its corresponding parsed line to $\LaTeX$.

Known Issues

- In english, the `default_inline_normal_delimiter` configuration should be modified, because the `'` character it's used by default, and there will be errors if expressions like `it's` are used. So, to prevent those errors one of the next recommendations can be chosen: replace the `'` delimiter character by another one like `|`, `!`, etc.; redefine the `normal_text` normal delimiter; or use `it is` instead `it's`.
- Inline code or keywords that end with blank spaces could generate errors that block the PDF generation process.
- Double quotes (`"`) required in JSON configuration values must be escaped (e.g. `"configuration": "\""`).
- Non alphanumeric characters in JSON configuration values, like `%`, `*`, `?`, etc., can produce errors or undesired behaviors if they aren't escaped with `\\`. For example, if an error is generated, instead of `"configuration": "*"'`, try with `"configuration": "\\*"`.
- Implicit list or table environments (with no identifier that mark their beginning) must be indented one additional level. If an implicit environment starts at same level than its previous text, an error or undesired behavior could arise.
- At the moment all the keywords, specific actions, errors and best practices content also must be indented, otherwise errors or undesired behaviors could arise.
- Backtick character (```), even without parsing it in TXT as delimiter and passing it directly to the `.tex` file as plain text, will be interpreted by $\LaTeX$ engine as a inline code delimiter, since $\LaTeX$ use it as short inline code delimiter. It can be interpreted as text by escaping and inserting it directly as $\LaTeX$ code (adding braces to prevent attaching it to a letter): `L"\`{}`.

TODO

Next, the main 5 features to be implemented in future versions are listed:

- Keywords glossary.
- Lists (table of contents) of specific actions, errors and best practices, by section.
- Cross references to sections, keywords, specific actions, errors, best practices, etc.
- Environments arguments, to modify features like the programming language used, style, etc.
- Optional panel to show code environments output.

The full list can be consulted in `TODO.md` file.

License

This project is licensed only for personal, non commercial use. Consult the `LICENSE` file to know the full terms.