CMinx

Release 1.0.0alpha

Mar 11, 2020

Contents

Welcome to the documentation for CMinx! CMinx is a tool for extracting API documentation from CMake files.

CHAPTER 1

Table Of Contents

1.1 Quickstart

Note: This tutorial assumes you are familiar with reStructuredText (reST) and CMake.

According to CMinx, the contents of a CMake file can be broken into three categories:

1. Documentation comments - what CMinx extracts
2. Annotation comments - Comments that are not documentation
3. CMake source code - Everything else

To distinguish between the two types of comments CMinx borrows the convention that documentation comments start with an additional comment character. Thus to indicate that a CMake comment is documentation use `#[[[` (the block comment should still end with `#]]`).

The contents of the documentation comments should follow usual Python reST conventions. In other words:

```
#[[[
# Short description. Runs up to the first blank line. So this is still the
# short description.
#
# Longer description goes here and includes all text and paragraphs from
# here forward that are not part of reST directives.
#
# The parameters, keywords, and their types will be pulled out of the longer
# description and placed in separate sections regardless of where they
# appear.
#
# :param name_of_param: Description of what `name_of_param` is used for
#
# The next line is a reST directive and will not be part of the longer
# description.
```

(continues on next page)

(continued from previous page)

```
#
# .. note::
#
#    This is a note, it will show up using reST's native note section
#]]
```

Technically speaking the contents of the comments are dumped more-or-less verbatim into the resulting `*.rst` file so you can use any reST directives and markup you like. That said, you will probably only want to use:

- `:param <name of parameter>:  <description of parameter>`
- `:type <name of parameter>:  <type of parameter named "name of parameter">`
- `:keyword <name of keyword>:  <description of keyword>`

as those are the subset of Python language features CMake actually supports.

1.1.1 Installing

Manually

Run the following commands one at a time:

```
foo@bar:~$ git clone github.com/CMakePP/CMinx.git
foo@bar:~$ cd CMinx/
foo@bar:~/CMinx$ python3 -m venv virt-env #Create our virtual environment
foo@bar:~/CMinx$ source virt-env/bin/activate #Activate virtual environment
(virt-env) foo@bar:~/CMinx$ pip3 install wheel #Necessary dependency to install␣
↪manually with pip
(virt-env) foo@bar:~/CMinx$ pip3 install . #If pip installed
(virt-env) foo@bar:~/CMinx$ python3 setup.py install #If pip not installed
```

With CMake

Run:

```
foo@bar:~$ git clone github.com/CMakePP/CMinx.git
foo@bar:~$ cd CMinx/
foo@bar:~/CMinx$ mkdir build && cd build
foo@bar:~/CMinx/build$ cmake ..
foo@bar:~/CMinx/build$ make install
```

1.1.2 Usage

For each CMake function or variable that you would like to document, prepend it with a block doc-comment. A block doc-comment begins with `#[[[` and ends with `#]]`.

Then run `cminx` on your CMake files, outputting to a directory of your choosing. The help text is printed below for reference:

```
Usage: cminx [-h] [-o OUTPUT] [-r] file [file ...]

positional arguments:
```

(continues on next page)

(continued from previous page)

```
   file                  CMake file to generate documentation for. If
                         directory, will generate documentation for all *.cmake
                         files (case-insensitive)

optional arguments:
  -h, --help             show this help message and exit
  -o OUTPUT, --output OUTPUT
                         Directory to output generated RST to. If not specified
                         will print to standard output. Output files will have
                         the original filename with the cmake extension
                         replaced by .rst
  -r, --recursive        If specified, will generate documentation for all
                         subdirectories of specified directory recursively
```

1.1.3 Example

Here we show an example CMake file, called `example.cmake`, that contains doccomments documenting functions, macros, and variables. The file contents are shown below.

example.cmake:

```
#[[
# This is a normal block comment and will not
# be treated as a doccomment.
#]]
include_guard()

#[[[
# This function has very basic documentation.
#
# This function's description stays close to idealized formatting and does not do
# anything fancy.
#
# :param person: The person this function says hi to
# :param me: What my name is
# :type person: string
# :type me: string
#]]
function(say_hi_to person me)
    message("Hi ${person}, I am ${me}")
endfunction()

#[[[
This macro says hi.
This documentation uses a differing format,
but is still processed correctly.

:param person: The person we want to greet.
:type person: string
#]]
macro(macro_say_hi person)
   message("Hi ${person}")
endmacro()
```

(continues on next page)

(continued from previous page)

```
#[[[
# This is an example of variable documentation.
# This variable is a list of string values.
#]]
set(MyList "Value" "Value 2")


#[[[
# This is another example of variable documentation.
# This variable is a string variable.
#]]
set(MyString "String")
```

To generate the documentation, we enter our system shell (example assumes Bash-like shell on a Unix-like system).

Generating documentation in directory `output`:

```
foo@bar:~$ cminx -o output/ example.cmake
Writing RST files to /home/foo/output
Writing for file /home/foo/example.cmake
Writing RST file /home/foo/output/example.rst
```

The resultant file :code: *output/example.rst*:

```
#############
example.cmake
#############

.. function:: say_hi_to(person me)

   This function has very basic documentation.

   This function's description stays close to idealized formatting and does not do
   anything fancy.

   :param person: The person this function says hi to
   :param me: What my name is
   :type person: string
   :type me: string



.. function:: macro_say_hi(person)

   .. warning:: This is a macro, and so does not introduce a new scope.


   This macro says hi.
   This documentation uses a differing format,
   but is still processed correctly.

   :param person: The person we want to greet.
   :type person: string
```

(continues on next page)

(continued from previous page)

```
.. data:: MyList

   This is an example of variable documentation.
   This variable is a list of string values.


   :Default value: ['"Value"', '"Value 2"']

   :type: VarType.List


.. data:: MyString

   This is another example of variable documentation.
   This variable is a string variable.


   :Default value: String

   :type: VarType.String
```

Place this file in a Sphinx source directory and add it to your :code: *toctree* to render it.

1.2 Developer Documentation

The following documentation topics are meant primarily for developers working on CMinx.

1.2.1 Table of Contents

Overview of How CMinx Works

Source File Parsing

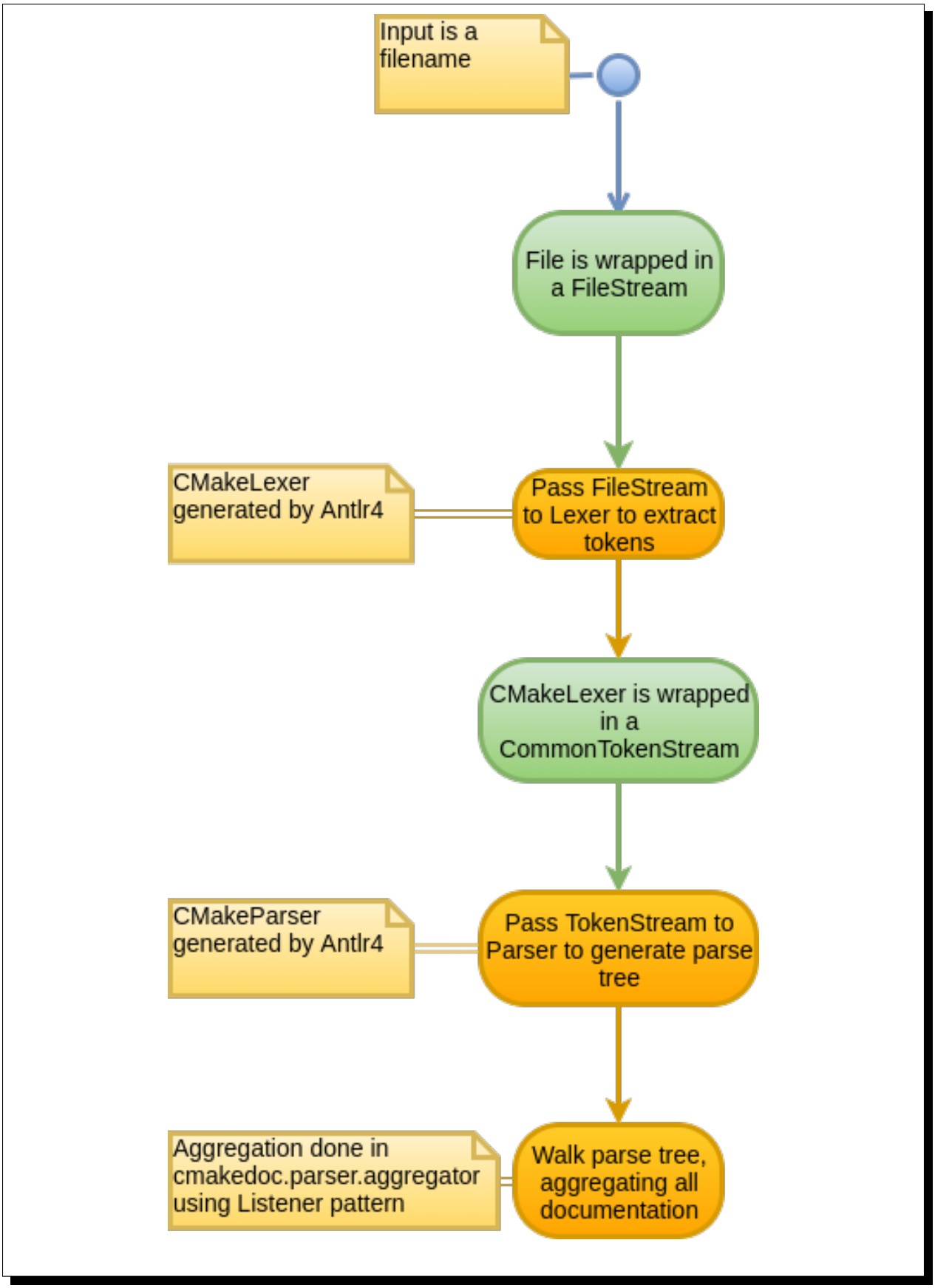
Input is a filename
File is wrapped in a FileStream
CMakeLexer generated by Antlr4
Pass FileStream to Lexer to extract tokens
CMakeLexer is wrapped in a CommonTokenStream
CMakeParser generated by Antlr4
Pass TokenStream to Parser to generate parse tree
Aggregation done in cmakedoc.parser.aggregator using Listener pattern
Walk parse tree, aggregating all documentation

Fig. 1.1: How CMinx parses a CMake source file.

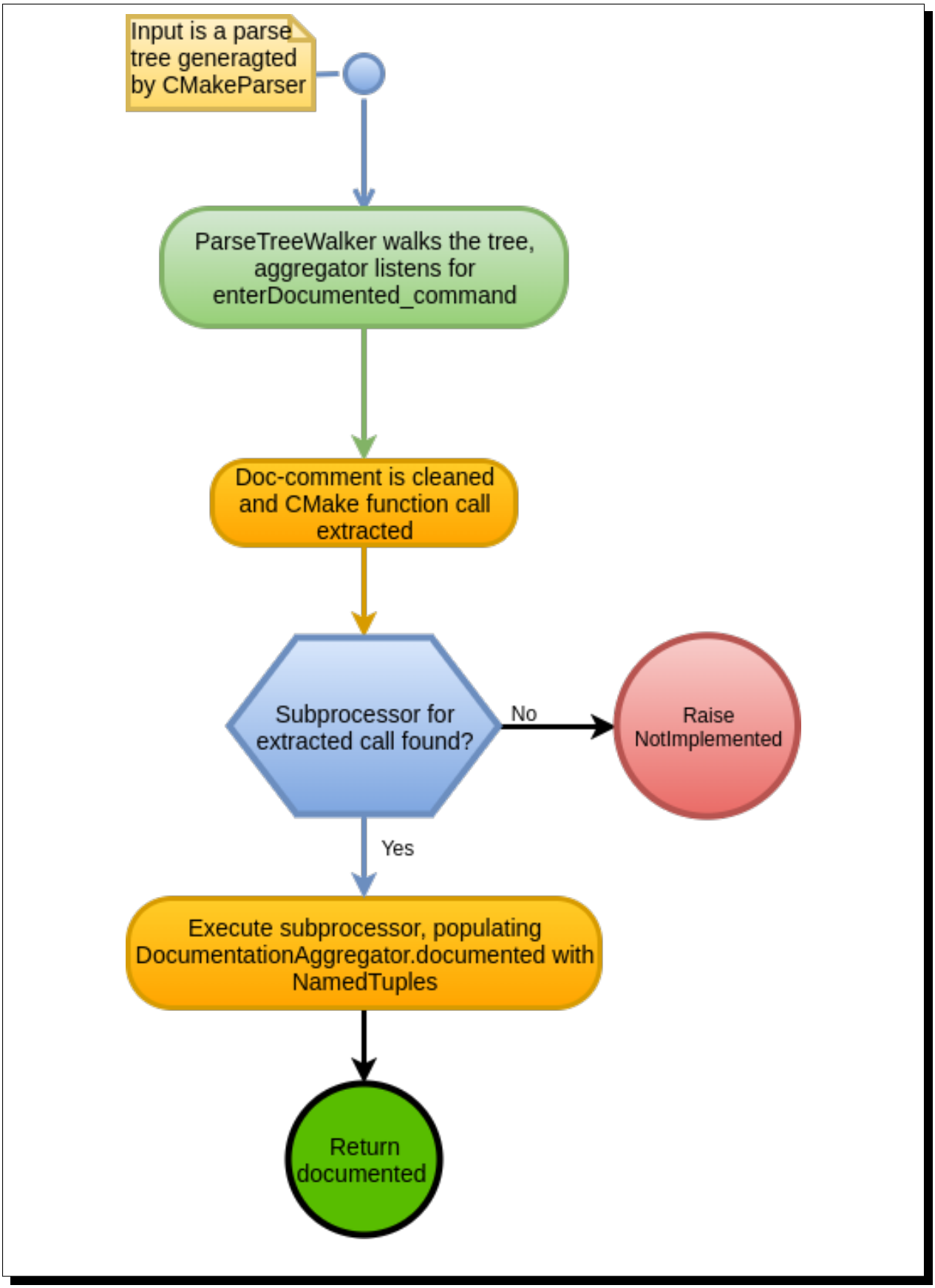
Input is a parse tree generagted by CMakeParser
ParseTreeWalker walks the tree, aggregator listens for enterDocumented_command
Doc-comment is cleaned and CMake function call extracted
Subprocessor for extracted call found?
No
Raise NotImplemented
Yes
Execute subprocessor, populating DocumentationAggregator.documented with NamedTuples
Return documented

Fig. 1.2: How CMinx aggregates documentation from the parse tree.

Parsing

In CMinx parsing of a source file is the role of the Antlr4 parsing runtime, generated from the modified CMake.g4 grammar file.

1. As per the standard usage, the file contents are read into an Antlr4 FileStream, which is then passed to the generated CMake lexer.
2. The lexer generates a token stream, which is then fed into the CMakeParser.
3. The parser generates a tree of parse elements, called contexts, which are then walked over by the ParseTreeWalker.

This process is diagrammatically summarized in Fig. 1.1.

Aggregation

After the parser generates the parse tree, CMinx walks the tree and aggregates the various documentation.

1. The walker calls the aggregator methods upon entering or exiting parse rules, such as entering a `documented_command` parse rule.
2. The parse rule enterDocumented_command cleans the doccomment and extracts the documented command. For example, if a function definition is documented, enterDocumented_command will extract the `function` command.
3. The aggregator then locates the subprocessor that corresponds to the extracted command, for example if the extracted command is `function` then the subprocessor would be `process_function()`. This subhandler is then executed with the parse context and cleaned docstring.
4. The documentation aggregator subhandler generates NamedTuples representing the type of documentation generated, such as FunctionDocumentation, complete with all relevant information, and adds them to a *documented* list.
5. From there, Documenter loops over the documentation list, generating equivalent RST via RSTWriter for each type of documentation.

This process is diagrammatically summarized in Fig. 1.2.

CMinx API

This is a list of all functions within the CMinx project. These functions are meant for use by developers. Users should run the `cminx` command.

Modules

cminx.rstwriter	This module and associated classes provide a pure-Python interface for generating reStructuredText.

Continued on next page

Table 1.1 – continued from previous page

cminx.documenter	This file contains the Documenter class, which combines functionality from parser.aggregator and rstwriter to generate RST documentation for CMake files.
cminx.parser.aggregator	
cminx.parser.CMakeLexer	
cminx.parser.CMakeParser	
cminx.parser.CMakeListener	
cminx.parser.CMakeVisitor	

cminx.rstwriter

This module and associated classes provide a pure-Python interface for generating reStructuredText. The generated text may be either consumed by another Python application or directly written to a file-like object. This module does not currently perform syntax validation so it is up to the application developer to not generate invalid RST structures.

Author Branden Butler

License Apache 2.0

class cminx.rstwriter.**Directive**(*name*, *indent=0*, **arguments*)

Use `cminx.RSTWriter.directive()` to construct. Represents an RST directive, such as toctree or an admonition. Does not verify that the directive or its arguments are valid.

build_heading()

Build directive heading format (ex. '.. toctree::') and return.

Returns Correctly formatted directive heading string.

directive(*title*, **arguments*)

Add a sub-directive.

Parameters

- **title** – Name of the new subdirective, eg. toctree or admonition.
- **arguments** – Varargs used as the directive arguments, such as a topic's title.

field(*name: str*, *txt: str*)

Add a field to document tree, adds proper indenting for directives.

Parameters

- **name** – Name of the field
- **txt** – Text of the field

format_arguments()

Format argument list into the correct argument string for use with directives.

Returns A string representing the directive arguments.

get_indents(*num*)

Get the string containing the necessary indent.

Returns A string containing the correct number of whitespace characters, derived from the indent level.

option(*name: str*, *value=''*)

Add an option, such as toctree's maxdepth. Does not verify if valid option

text(*txt*)

Add paragraph to document tree, adds proper indenting for directives as well.

Parameters **txt** – Content of the new paragraph.

to_text()
Return text representation of this document

Returns The completed RST document in string form.

class cminx.rstwriter.**DirectiveHeading**(*title: str*, *indent: str*, *args*)
Represents the unique heading for a Directive (.. :<name>:)

class cminx.rstwriter.**DocTest**(*test_line*, *expected_output*)
Represents an RST DocTest

build_doctest_string()
Populates DocTest.doctest_string with the RST string cooresponding to this DocTest

class cminx.rstwriter.**Field**(*field_name*, *field_text*, *prefix=''*)
Represents an RST field, such as Author

build_field_string()
Populates Field.field_string with the RST string cooresponding to this field

class cminx.rstwriter.**Heading**(*title*, *header_char*)
Represents a section heading

build_heading_string()
Populates Heading.heading_string with the RST string cooresponding to this heading

class cminx.rstwriter.**List**(*items*, *list_type: int*)
Represents one of the two types of RST lists: Enumerated or Bulleted

build_list_string()
Populates List.list_string with the RST string cooresponding to this list

class cminx.rstwriter.**Option**(*name*, *value*, *indent*)
Represents a directive option, such as maxdepth

class cminx.rstwriter.**Paragraph**(*text*, *prefix=''*)
Represents an RST paragraph

build_text_string()
Populates Paragraph.text_string with the RST string cooresponding to this paragraph

class cminx.rstwriter.**RSTWriter**(*title: str*, *header_char: str = '#'*, *section_level: int = 0*)
Base reStructuredText writer. Does not perform verification.

build_heading()
Adds overline and underline to RSTWriter.title (character is RSTWriter.header_char) and returns it

Returns A fully constructed Heading object.

bulleted_list(**items*)
Add a bulleted list to the document tree.

Parameters **items** – varargs containing the desired list items.

clear()
Clear all document tree elements (besides required heading)

directive(*name*, **arguments*)
Construct a directive and return it

Parameters

- **name** – Name of the directive being used, such as toctree or admonition.

- **arguments** – Varargs to be used as the directive arguments, such as a topic title.

doctest(*test_line: str*, *expected_output: str*)

Adds a doctest segment to the document tree. A doctest segment appears as an interactive python session. The doctest Python module can then be used to scan for these segments and execute the test_line to ensure the output is the same.

Parameters

- **test_line** – Python code segment being used as the line to be tested.
- **expected_output** – The exact string that is expected to be returned when test_line is evaluated.

enumerated_list(**items*)

Add an enumerated list to the document tree; e.g.

1. Item 1
2. Item 2

Parameters **items** – varargs containing the desired list items.

field(*field_name: str*, *field_text: str*)

Add a field, such as Author or Version, to the document tree.

Parameters

- **fieldname** – Name of the field being added, such as Author.
- **field_text** – Value of the field, such as the author's name.

heading_level_chars = ['#', '*', '=', '-', '_', '~', '!', '&', '@', '^']

Characters to use as heading overline/underline, indexed by section_level

section(*title: str*)

Constructs another RSTWriter and adds it to the document before returning it for use

Parameters **title** – The heading title to be used for the new subsection.

simple_table(*tab*, *column_headings=[]*)

Add a simple table to the document tree. tab is a 2-dimensional list containing the table data. The first index is the row, the second is column. The first column may not contain multiple lines, all other columns may. column_headings is a list where each element is treated as a heading for that specific column, i.e. index 0 will be the heading for the leftmost column.

Parameters

- **tab** – A two-dimensional list representing table data. First index is row, second is column. Each row must have the same number of columns.
- **column_headings** – A single-dimensional list containing column headings, if required. Index zero is the heading for the left-most column. List must be same length as number of columns.

Raises **ValueError** – If tab has inconsistent numbers of columns or column_headings length (if nonzero) does not match number of columns in tab.

text(*txt: str*)

Add a paragraph to document tree.

Parameters **txt** – The content of the new paragraph.

to_text()
Return text representation of this document

Returns The completed RST document in string form.

write_to_file(*file*)
Write text representation of this document to file. File may be a string (path will be opened in write mode) or a file-like object.

Parameters **file** – File to write to. May be a string representing a real filepath on the local filesystem (will be overwritten), or a file-like object.

Raises

- **ValueError** – If file is nothing or empty string.
- **TypeError** – If file is not str or object with 'write()' method

class cminx.rstwriter.**SimpleTable**(*tab*, *headings*)
Represents an RST simple table.

build_table_string()
Populates SimpleTable.table_string with the RST string equivalent of this table

cminx.documenter

This file contains the Documenter class, which combines functionality from parser.aggregator and rstwriter to generate RST documentation for CMake files.

Author Branden Butler

License Apache 2.0

class cminx.documenter.**Documenter**(*file: str*, *title: str = None*)
Generates RST documentation from aggregated documentation, combining parser.aggregator and rstwriter.

process()
Process Documenter.aggregator.documented and build RST document from it.

Returns Completed RSTWriter document, also located in Documenter.writer

process_docs(*docs*)
Loops over document and dispatches each documentation to the respective processor.

Parameters **docs** – List of documentation objects.

process_function_doc(*doc: cminx.parser.aggregator.FunctionDocumentation*)
FunctionDocumentation processor. Generates the RST "function" directive.

Parameters **doc** (FunctionDocumentation) – Documentation for the function

process_macro_doc(*doc*)
MacroDocumentation processor. Generates the RST "function" directive containing a "warning" directive explaining that it is a macro.

Parameters **doc** (MacroDocumentation) – Documentation for the macro

process_variable_doc(*doc*)
VariableDocumentation processor. Generates the RST "data" directive.

Parameters **doc** (VariableDocumentation) – Documentation for the variable.

cminx.parser.aggregator

class cminx.parser.aggregator.**DocumentationAggregator**
Processes all docstrings and their associated commands, aggregating them in a list.

documented = None
All current documented commands

enterDocumented_command(*ctx: cminx.parser.CMakeParser.CMakeParser.Documented_commandContext*)
Main entrypoint into the documentation processor and aggregator. Called by ParseTreeWalker whenever encountering a documented command. Cleans the docstring and dispatches ctx to other functions for additional processing (process_{command}(), i.e. process_function())

Parameters **ctx** – Documented command context, constructed by the Antlr4 parser.

Raises **NotImplementedError** – If no processor can be found for the command that was documented.

process_function(*ctx: cminx.parser.CMakeParser.CMakeParser.Documented_commandContext*, *docstring: str*)
Extracts function name and declared parameters.

Parameters

- **ctx** – Documented command context. Constructed by the Antlr4 parser.
- **docstring** – Cleaned docstring.

process_macro(*ctx: cminx.parser.CMakeParser.CMakeParser.Documented_commandContext*, *docstring: str*)
Extracts macro name and declared parameters.

Parameters

- **ctx** – Documented command context. Constructed by the Antlr4 parser.
- **docstring** – Cleaned docstring.

process_set(*ctx: cminx.parser.CMakeParser.CMakeParser.Documented_commandContext*, *docstring: str*)
Extracts variable name and values from the documented set command. Also determines the type of set command/variable: String, List, or Unset.

Parameters

- **ctx** – Documented command context. Constructed by the Antlr4 parser.
- **docstring** – Cleaned docstring.

class cminx.parser.aggregator.**FunctionDocumentation**(*function*, *params*, *doc*)

doc
Alias for field number 2

function
Alias for field number 0

params
Alias for field number 1

class cminx.parser.aggregator.**MacroDocumentation**(*macro*, *params*, *doc*)

doc
Alias for field number 2

macro
Alias for field number 0

params
Alias for field number 1

class cminx.parser.aggregator.**VarType**
An enumeration.

class cminx.parser.aggregator.**VariableDocumentation**(*varname*, *type*, *value*, *doc*)

doc
Alias for field number 3

type
Alias for field number 1

value
Alias for field number 2

varname
Alias for field number 0

cminx.parser.CMakeLexer

cminx.parser.CMakeParser

cminx.parser.CMakeListener

cminx.parser.CMakeVisitor

Functions

cminx.document(input[, output_path, recursive])	Handler for documenting CMake files or all files in a directory.
cminx.main([args])	CMake Documentation Generator program entry point.

cminx.document

cminx.**document**(*input*, *output_path=None*, *recursive=False*)
Handler for documenting CMake files or all files in a directory. Performs preprocessing before handing off to document_single_file over all detected files. Also generates index.rst files for all directories.

Parameters

- **input** – String locating a file or directory to document.
- **output_path** – String pointing to the directory to place generated files, will output to stdout if None
- **recursive** – Whether to generate documentation for subdirectories or not.

cminx.main

cminx.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

CMake Documentation Generator program entry point.

Parameters **args** – Array of strings containing program arguments, excluding program name. Same format as sys.argv[1:].

Python Module Index

C

Index

M

O

P

R

S

T

V

W