lina Documentation

Release 1.0.10

Author

Aug 07, 2023

Contents

Lina is a minimal template system for Python, modelled after Google's CTemplate library. It is designed to provide fast, safe template evaluation to generate code or other text documents.

```
enum DataTypes {
{{#types:list-separator=,NEWLINE}}  {{name}}={{value:hex}}{{/types}}
}
```

evaluated with:

```
types = [{'name':'Vector3i', 'value': 0x301}, {'name':'Vector3f', 'value': 0x302}]
```

will produce:

```
enum DataTypes {
    Vector3i = 0x301,
    Vector3f = 0x302
}
```

CHAPTER 1

Overview

The base class in Lina is *lina.Template* which must be initialized with the template contents. It can be then evaluated to a string using *lina.Template.Render()* and *lina.Template.RenderSimple()*.

Lina has two main directives, *values* and *blocks*. A value is something which is replaced directly by the provided value, while a block is used to iterate over collections. Both blocks and values can be optionally formatted using a `formatter`, which allows for example to turn a string into uppercase inside the template.

Values are escaped using double curly braces:

```
Hello {{name}}!
```

Blocks have an additional prefix before the variable, `#` for the block start and `/` for the block end:

```
{{#users}}Hello {{name}}!{{/users}}
```

This requires to pass an array of named objects:

```
template.Render ({'users':[{'name':'Alice'}, {'name':'Bob'}]})
```

CHAPTER 2

Element access

In some cases, accessing members by names is unnecessary complicated. Lina provides a special syntax to access the *current* element, using a single dot. Using a self-reference, the template above can be simplified to:

```
{{#users}}Hello {{.}}!{{/users}}
```

and rendered with:

```
template.Render ({'users': ['Alice', 'Bob']})
```

or even simpler using *lina.Template.RenderSimple()*:

```
template.RenderSimple (users = ['Alice', 'Bob'])
```

Both self-references as well as values can also access fields of an object. Assuming the `User` class has fields `name`, `age`, the following template will print the user name and age:

```
{{#users}}Hello {{.name}}, you are {{.age}} years old!{{/users}}
```

For an object, use `{{item.field}}`. The field accessor syntax works for both fields as well as associative containers, that is, for Lina, the following two objects are equivalent:

```
u = {'name':'Alice'}
```

and:

```
class User:
    def __init__(self, name):
        self.name = name

u = User ('Alice')
```

It is also possible to directly reference indexed items using `[0]`, `[1]`, etc. For instance, the following template:

```
{{#vectors}}X: {{.[0]}}, Y: {{.[1]}}, Z: {{.[2]}}\n{{/vectors}}
```

rendered with:

```
template.RenderSimple (vectors = [[0, 1, 2], [3, 4, 5]])
```

will produce:

```
X: 0, Y: 1, Z: 2
X: 3, Y: 4, Z: 5
```

CHAPTER 3

Blocks

For blocks, Lina provides additional modifiers to check whether the current block execution is the first, an intermediate or the last one:

```
{{#block}}{{variable}}{{#block#Separator}},{{/block#Separator}}{{/block}}
```

`#First` will be only expanded for the first iteration, `#Separator` will be expanded for every expansion which is neither first nor last and `#Last` will be expanded for the last iteration only. If there is only one element, it will considered both first and last item of the sequence.

If a block variable is not found, or the block is `None`, the block will be not expanded. It is possible to capture this case using `!` blocks, which are only expanded if the variable is not present:

```
{{!users}}No users :({{/users}}
```

Rendered with `template.Render ()`, this will yield `No users :(`. This can be used to emulate conditional statements.

CHAPTER 4

Formatters

Lina comes with a few formatters which can be used to modify values or block elements. The value formatters are:

- `width`, `w`: This aligns a value to a specific width. Negative values align to the left. For example: `{{name:w=-8}}` using `name='Ton'` will yield ‘‘ Ton‘‘.
- `prefix` adds a prefix to a value. For example: `{{method:prefix=api_}}` with `method='Copy'` yields `api_Copy`
- `suffix` adds a suffix to a value. For example: `{{method:suffix=_internal}}` with `method='Copy'` yields `Copy_internal`
- `default` provides a default value in case the provided value is `None`. `{{name:default=Unknown}}` with `name=None` yields `Unknown`.
- `upper-case`, `uc` converts the provided value to upper case. `{{func:upper-case}}` with `func=Copy` yields `COPY`.
- `escape-newlines` escapes embedded newlines. `{{s:escape-newlines}}` with `s='foo\nbar'` yields `foo\\nbar`.
- `escape-string` escapes newlines, tabs, and quotes. `{{s:escape-string}}` with `s=a "string"` yields `a \"string\"`.
- `wrap-string` wraps the provided string with quotes. `{{s:wrap-string}}` with `s=string` yields `"string"`. If the value is not a string, it will not be wrapped.
- `cbool` converts booleans to `true` or `false`. `{{enabled:cbool}}` with `enabled=True` yields `true`. If the value is not a boolean, it will be returned as-is.
- `hex` prints numbers in hexadecimal notation. `{{i:hex}}` with `i=127` yields `0x7F`.

Lina provides the following block formatters:

- `indent` indents every line with a specified number of tabs. `{{#block:indent=2}}{{.}}{{/block}}` with `block=[1,2]` yields `\t\t1\t\t2`
- `list-separator`, `l-s` separates block repetitions using the provided value. `{{#block:l-s=,}}{{.}}{{/block}}` with `block=[1,2]` yields `1,2`. `NEWLINE` is replaced with a new line, and `SPACE` with a space.

4.1 Whitespace & special characters

Whitespace in Lina is preserved. If you want to explicitly insert whitespace, you can use `{{_NEWLINE}}` to get a new line character inserted into the stream, and `{{_SPACE}}` to get a blank space. To produce a left brace or right brace, use `{{_LEFT_BRACE}}` and `{{_RIGHT_BRACE}}`.

CHAPTER 5

Contents

5.1 API reference

class lina.**Formatter**(*formatterType*)
Bases: object

Base class for all formatters.

A formatter can be used to transform blocks/values during expansion.

Format(*text*)
Format a value or a complete block.

IsBlockFormatter()
Check if this formatter is a block formatter.

IsValueFormatter()
Check if this formatter is a value formatter.

OnBlockBegin(*isFirst*)
Called before a block is expanded.

Parameters **isFirst** – True if this is the first expansion of the block.

Returns String or None. If a string is returned, it is prepended before the current block expansion.

OnBlockEnd(*isLast*)
Called after a block has been expanded.

Parameters **isLast** – True if this is the last expansion of the block.

Returns String or None. If a string is returned, it is appended after the current block expansion.

class lina.**FormatterType**
Bases: enum.Enum

The formatter type, either Block or Value.

Block = 0

Value = 1

class lina.**IncludeHandler**
Bases: object

Base interface for include handlers.

Get(*name*)

exception lina.**InvalidBlock**(*message*, *position*)
Bases: *lina.TemplateException*

An invalid block was encountered.

exception lina.**InvalidFormatter**(*message*, *position*)
Bases: *lina.TemplateException*

An invalid formatter was encountered.

This exception is raised when a formatter could not be found or instantiated.

exception lina.**InvalidNamedCharacterToken**(*message*, *position*)
Bases: *lina.InvalidWhitespaceToken*

An invalid named character token was encountered.

exception lina.**InvalidToken**(*message*, *position*)
Bases: *lina.TemplateException*

An invalid token was encountered.

exception lina.**InvalidWhitespaceToken**(*message*, *position*)
Bases: *lina.TemplateException*

Only for backwards compatibility. Will be removed in 2.x.

class lina.**Template**(*template*, *includeHandler=None*, *, *filename=None*)
Bases: object

The main template class.

Render(*context*)
Render the template using the provided context.

RenderSimple(**items*)
Simple rendering function.

This is just a convenience function which creates the context from the passed items and forwards them to *Template.Render()*.

exception lina.**TemplateException**(*message*, *position*)
Bases: Exception

Base class for all exceptions thrown by Lina.

GetPosition()
Get the position where the exception occurred.

Returns An object with two fields, line and column.

class lina.**TemplateRepository**(*templateDirectory*, *suffix=''*)
Bases: *lina.IncludeHandler*

A file template repository.

This template repository will load files from a specified folder.

Get(*name*)

class lina.**TextStream**(*text*, *, *filename=None*)
Bases: object

A read-only text stream.

The text stream is used for input only and keeps track of the current read pointer position in terms of line/column numbers.

Get()
Get a character.

If the end of the stream has been reached, None is returned.

GetOffset()
Get the current read offset in characters from the beginning of the stream.

GetPosition()
Get the current read position as a pair (line, column).

IsAtEnd()
Check if the end of the stream has been reached.

Peek()
Peek at the next character in the stream if possible. Returns None if the end of the stream has been reached.

Reset()
Reset back to the beginning of the stream.

Skip(*length*)
Skip a number of characters starting from the current position.

Substring(*start*, *end*)
Get a substring of the stream.

Unget()
Move one character back in the input stream.

class lina.**Token**(*name*, *start*, *end*, *position*)
Bases: object

Represents a single token.

Each token may contain an optional list of flags, separated by colons. The grammar implemented here is:

```
[prefix]?[^:}]+(:[^:})+, for example:
{{#Foo}} -> name = Foo, prefix = #
{{Bar:width=8}} -> name = Bar, prefix = None,
                                        flags = {width:8}
```

The constructor checks if the formatter matches the token type. A block formatter can be only applied to a block token, and a value formatter only to a value.

EvaluateNamedCharacterToken(*position*)
Get the content of this token if this token is an escape character token.

If the content is not a valid character name, this function will raise InvalidSpecialCharacterToken.

GetEnd()
Get the end offset.

GetFormatters()
Get all active formatters for this token.

GetName()
Get the name of this token.

GetPosition()
Get the position as a (line, column) pair.

GetStart()
Get the start offset.

IsBlockClose()
Return true if this token is a block-close token.

IsBlockStart()
Return true if this token is a block-start token.

IsInclude()
Return true if this token is an include directive.

IsNamedCharacter()
Return true if this token is a named character token.

IsNegatedBlockStart()
Return true if this token is a negated block-start token.

IsSelfReference()
Return true if this token is a self-reference.

IsValue()

5.2 Release notes

5.2.1 1.0.10

- Added `{{_LEFT_BRACE}}` and `{{_RIGHT_BRACE}}` tokens.

5.2.2 1.0.9

- Applying block formatters to non-blocks and value formatters to non-values raises an error now. Previously, those were silently ignored.

5.2.3 1.0.8

- Packaging changes only.

5.2.4 1.0.7

- The formatter registration has been improved. Instead of a long `if-elif` cascade, it now jumps directly to the right formatter through a dictionary.

5.2.5 1.0.6

- Various build improvements.

5.2.6 1.0.5

- Handling of `None` blocks changed (i.e. set using `block_name = None`.) Instead of treating them as empty, they are now completely ignored. Only blocks initialized to an empty container (`{}`) are now treated as empty.

5.2.7 1.0.4

- Added a new `escape-string` formatter.

5.2.8 1.0.3

- Add support for negated blocks: `{{!block}}`

5.2.9 1.0.2

- Add support for self references via `{{.[0]}}`.

5.2.10 1.0.1

- Allow block formatters to write a suffix
- Add support for field access via `{{item.member}}`.

5.2.11 1.0

Initial public release.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index