Paxter

Abhabongse Janthong

Jun 18, 2020

CONTENTS

Paxter is a document-first text pre-processing mini-language, *loosely* inspired by at-expressions in Racket.

- The language mainly provides a toolchain to parse an input text into a document tree (similarly to a DOM).
- However, the language itself **does not specify** how the parsed tree will be transformed into the final rendered output text. Users have all the freedom to interpret or render the document tree into an output format however they like.
- Alternatively, instead of implementing a document tree renderer by themselves, users may opt-in to use pre-defined **renderers** of document tree renderers also provided by this package.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Installation

Paxter language package can be installed from PyPI via `pip` command (or any other methods of your choice):

```
$ pip install paxter
```

1.1.2 Programmatic Usage

The package is *mainly* intended to be used as a library. To get started, let's assume that we have a source text which contains a document **written in Paxter language syntax**.

```
# Of course, input text of a document may be read from any source,
# such as from a text file loaded from the filesystem, from user input, etc.

source_text = """\
@python##"
    from datetime import datetime

    name = "Ashley"
    year_of_birth = 1987
    current_age = datetime.now().year - year_of_birth
"##\\
My name is @name and my current age is @current_age.
My shop opens Monday@,-@,Friday.
"""
```

Note: Learn more about *Paxter language grammar and features*.

Parsing

First and foremost, we use a **parser** (which is implemented by the class *ParseContext*) to transform the source text into a parsed document tree.

```
from paxter.core import ParseContext

tree = ParseContext(source_text).parse()
```

Note: We can see the structure of the document tree in full by printing out the content of the variable `tree` from above (output reformatted for clarity).

```
>>> tree
FragmentList(
    children=[
        PaxterApply(
            id=Identifier(name="python"),
            options=None,
            main_arg=Text(
                inner='\n    from datetime import datetime\n\n    name = "Ashley"\n    ␣
↪ year_of_birth = 1987\n    current_age = datetime.now().year - year_of_birth\n',
                scope_pattern=ScopePattern(opening='##"', closing='"##'),
                is_command=False,
            ),
        ),
        Text(
            inner="\nMy name is ",
            scope_pattern=ScopePattern(opening="", closing=""),
            is_command=False,
        ),
        PaxterPhrase(inner="name", scope_pattern=ScopePattern(opening="", closing="
↪")),
        Text(
            inner=" and my current age is ",
            scope_pattern=ScopePattern(opening="", closing=""),
            is_command=False,
        ),
        PaxterPhrase(
            inner="current_age", scope_pattern=ScopePattern(opening="", closing="")
        ),
        Text(
            inner=".\nMy shop opens Monday",
            scope_pattern=ScopePattern(opening="", closing=""),
            is_command=False,
        ),
        PaxterPhrase(inner=",", scope_pattern=ScopePattern(opening="", closing="")),
        Text(
            inner="-",
            scope_pattern=ScopePattern(opening="", closing=""),
            is_command=False,
        ),
        PaxterPhrase(inner=",", scope_pattern=ScopePattern(opening="", closing="")),
        Text(
            inner="Friday.\n",
            scope_pattern=ScopePattern(opening="", closing=""),
            is_command=False,
        ),
    ],
```

(continues on next page)

(continued from previous page)

```
    scope_pattern=GlobalScopePattern(opening="", closing=""),
    is_command=False,
)
```

Notice that the source text above also contains what seems like a python code. This is **not** part of the Paxter language grammar in any way; it simply uses the Paxter application command to embed python code, to which we will give meaningful interpretation later.

Rendering

Next step, we use a **renderer** to transform the document tree into its final output. It is important to remember that **the semantics of the document is given depending on which renderer we choose**.

We will use *paxter.renderers.python.RenderContext* already pre-defined by Paxter library package to render the document tree into the final output. One of its useful features is that it will execute python code wrapped by `@python` application command.

```
from paxter.renderers.python import RenderContext, create_unsafe_env

# This dictionary data represents the initial global dict state
# for the interpretation the document tree in python authoring mode.
env = create_unsafe_env({
    '_symbols_': {',': '&thinsp;'},
})

output_text = RenderContext(source_text, env, tree).render()
print(output_text)  # or write to a file, etc.
```

The above code will output the following.

```
My name is Ashley and my current age is 33.
My shop opens Monday&thinsp;-&thinsp;Friday.
```

Note: Learn more about *how to use Python authoring mode* and *how to write custom renderer*.

Create Your Function

In order to reuse this parse-and-render setup, we can write a utility function such as in the following:

```
from paxter.core import ParseContext
from paxter.renderers.python import RenderContext, create_unsafe_env

def interp(source_text: str) -> str:
    tree = ParseContext(source_text).parse()
    output = RenderContext(source_text, create_unsafe_env(), tree).render()
    return output
```

1.1.3 Command-Line Usage

As a shortcut, Paxter library package also provides utility via command-line. To get started, read the help message by typing:

```
$ paxter --help
```

To get the parsing result only, we will use `parse` subcommand. Suppose that we have an input file called `intro.paxter` which contains the following text:

```
@python##"
    from datetime import datetime

    _symbols_ = {
        ',': '&thinsp;',
    }
    name = "Ashley"
    year_of_birth = 1987
    current_age = datetime.now().year - year_of_birth
"##\
My name is @name and my current age is @current_age.
My shop opens Monday@,-@,Friday.
```

Then we can look at the intermediate parsed tree result with the following command:

```
$ paxter parse -i intro.paxter
```

If we wish to render the document source text with the default environment dict, then we can use the following command:

```
$ paxter python-authoring -i intro.paxter
```

which will result in

```
My name is Ashley and my current age is 33.
My shop opens Monday&thinsp;-&thinsp;Friday.
```

1.2 Paxter Language Tutorial

Todo: Tutorial is coming soon.

1.3 Python Authoring Mode Tutorial

1.3.1 Tutorials

Todo: Tutorial is coming soon.

1.3.2 API Reference

The following class implements a standard parser which comes with Paxter package library.

class `paxter.renderers.python.`**`RenderContext`**(*input_text: str*, *env: dict*, *tree: paxter.core.data.FragmentList*)

A suite of Paxter document tree renderer.

Users of this renderer may embed and run python code directly from within the Paxter document source file.

`env: dict`
Python execution environment data

`input_text: str`
Document source text

`render`() → str
Transforms the already provided input source text, the initial python execution environment data, and the parsed document tree, into the final output.

`tree: FragmentList`
Parsed document tree

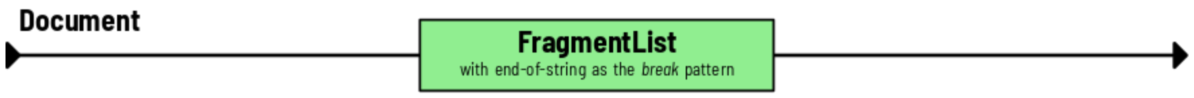
1.4 Custom Renderer Tutorial

Todo: Tutorial is coming soon.

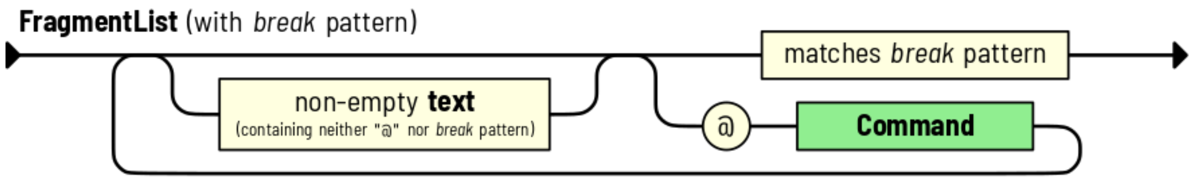
1.5 Syntax Reference

Below are syntax diagrams for Paxter language.

- **Document**: Top-level document; parsing starts here. Once all fragments of the fragment list is parsed, the caret pointer must end exactly at the end of input text.



- **FragmentList:** Consists of an interleaving of raw texts and @-commands, and ends with dynamically designated *break pattern* (which is simply tells where the fragment list stops).

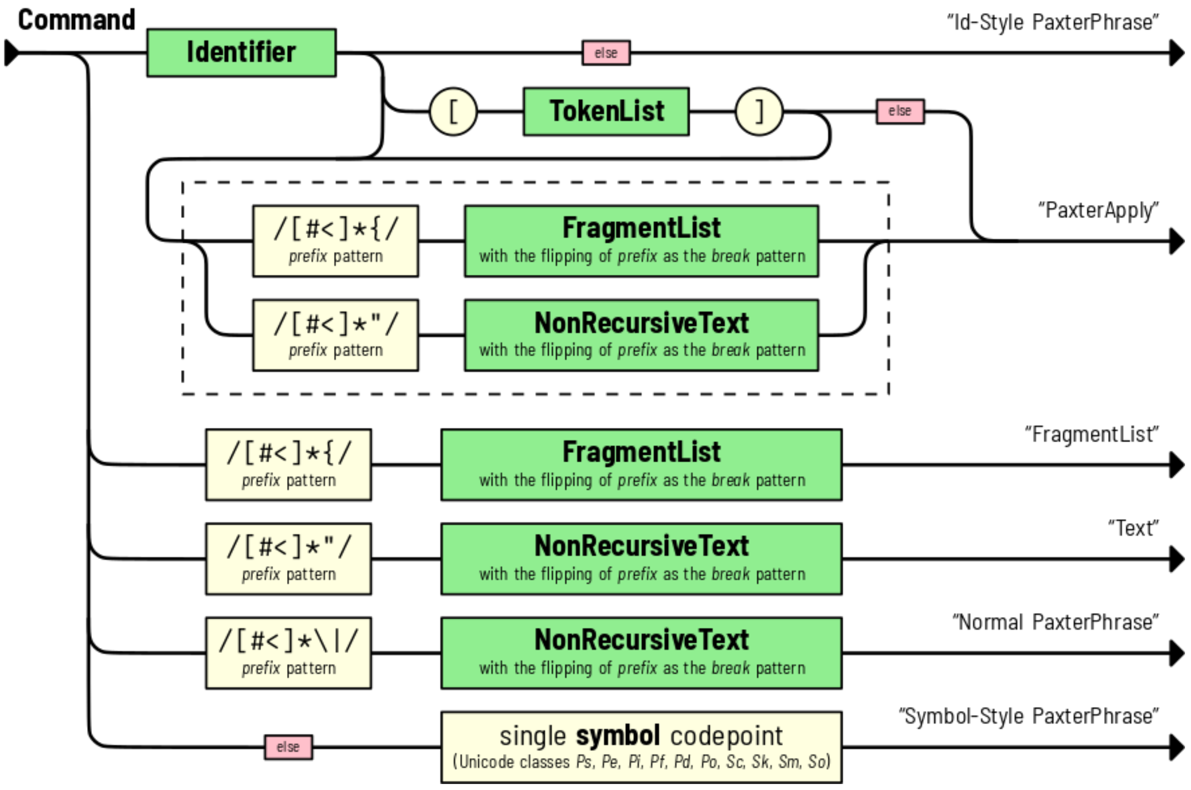

FragmentList (with break pattern)
matches break pattern
non-empty text
(containing neither "@" nor break pattern)
@
Command

For example, if preceding the fragment list is an opening brace pattern `##<#{`, then the break (i.e. closing) pattern for this fragment list would be `}#>##`, which mirrors the opening pattern.

Please note that by construction of the language, the non-empty raw text would never contain the *break pattern*; if it was the case then the parsing of fragment list would have terminated earlier. In other words, we *non-greedily* parses text within the fragment list.

The result of parsing fragment list is a `FragmentList` node type whose children is a list of `Text` or command tokens.

- **Command:** Parses right after the @-symbol for one of 6 possibilities.



Note: The red `else` box in this diagram indicates that such path can be followed only if the next token does not match any other possible paths. Walking through the boxes in itself consumes nothing.

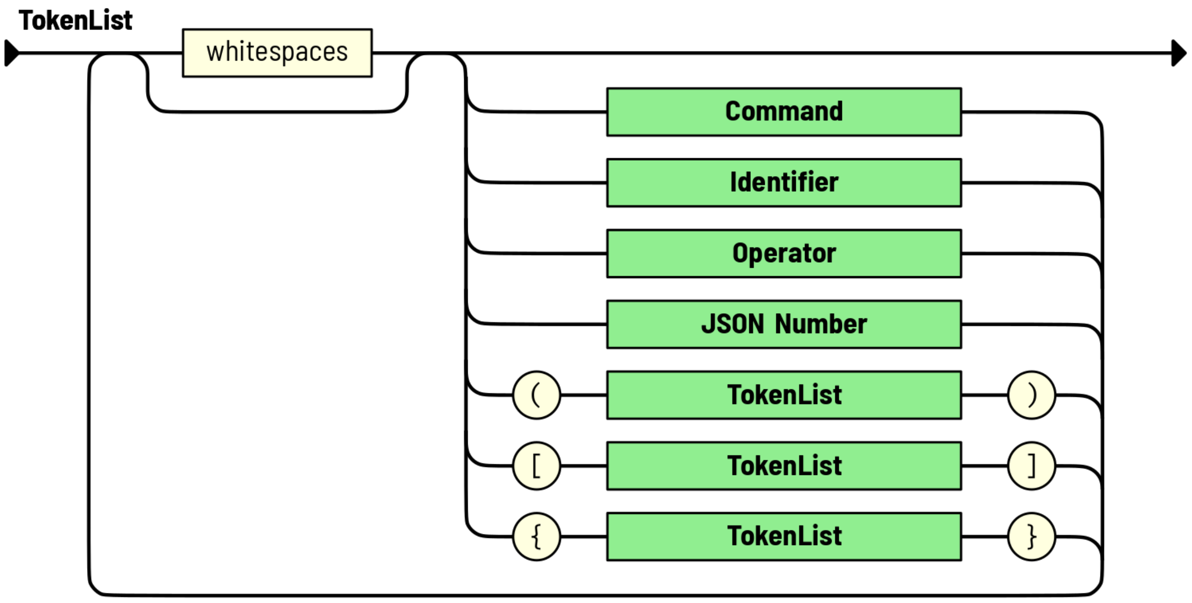
Note: The *prefix pattern* matched before the fragment list or the non-recursive text will be used to determine the break pattern indicating when to stop parsing for the fragment list or the non-recursive text itself, respectively. The break pattern is generally the mirror image of the matched prefix pattern, and can be computed by flipping the entire string as well as flipping each individual character to its mirror counterpart.

Possible results are:

- A `PaxterApply` which consists of an identifier, followed by at least one option section or one main argument section. The option section is a list of tokens enclosed by a pair of square brackets (node is represented with `TokenList`). On the other hand, the main argument section (surrounded by the dashed box in diagram below) is either a fragment list (represented with `FragmentList`) or a non-recursive raw text (represented with `Text`).

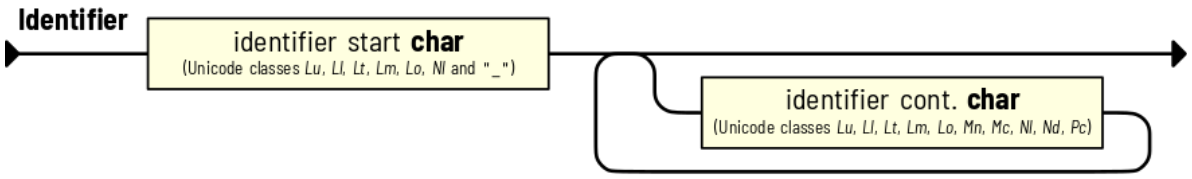
- However, if the token immediately succeeding the identifier neither does match the option section path nor does match the main argument path, the the parsing results in the identifier-style `PaxterPhrase` whose inner phrase content derives from the identifier string.
- If the command begins with the brace prefix pattern, then the parsing yields the `FragmentList` node as a result.
- If the command begins with the quoted prefix pattern, then the parsing yields a regular `Text` node as a result
- If the command begins with the bar prefix pattern, then the parsing outputs the normal `PaxterPhrase` node.
- Finally, if the first token found does not match any of the above scenarios, then a single symbol codepoint is consumed and such character becomes the inner phrase content of symbol-style `PaxterPhrase`.

• **TokenList:** A sequence of zero or more tokens Each token either a command, an identifier, an operator, a number following JSON specification, or a nested token list enclosed by a pair of parentheses `()`, a pair of square brackets `[]`, or a pair of pure braces `{}`. The result is a `TokenList` node type.

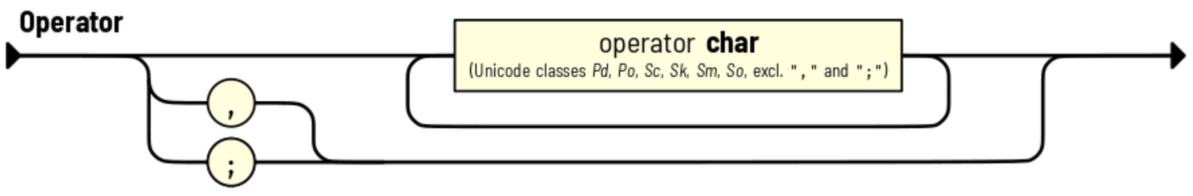


Note: The option section (or the token list) is the only place where whitespaces are ignored (when they appear between tokens).

• **Identifier:** Generally follows Python rules for parsing identifier token (with some exceptions). The result is an `Identifier` node type.

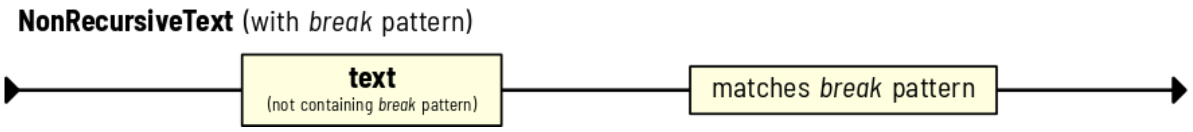

Identifier
identifier start char
(Unicode classes Lu, Ll, Lt, Lm, Lo, Nl and "_")
identifier cont. char
(Unicode classes Lu, Ll, Lt, Lm, Lo, Mn, Mc, Nl, Nd, Pc)

- **Operator:** Greedily consumes as many operator character as possible (with two notable exceptions: a comma and a semicolon, which has to appear on their own). A whitespace may be needed to separate two consecutive, multi-character operator tokens. The result is an *Operator* node type.



- **NonRecursiveText:** Parses the text content until encountering the *break pattern*. As opposed to fragment list, no @-symbol will be recognized as the indicator of the beginning of a command.

  Text extracted through this process will be used as the inner content of either *Text* or *FragmentList* while a command is being parsed.


NonRecursiveText (with break pattern)
text
(not containing break pattern)
matches break pattern

1.6 Core API Reference

Paxter language package provides the following core functionality.

1.6.1 Parsing

This class implements the parser for Paxter language.

class paxter.core.**ParseContext**(*input_text: str*)

Implements recursive descent parser for Paxter language.

Below is how to utilize this class:

```
input_text = 'Hello @name'
tree = ParseContext(input_text).parse()
```

input_text: str

Document source text

parse() → paxter.core.data.FragmentList

Parses the already provided input text starting from the beginning. This method is expensive and should not be called more than once.

1.6.2 Data Definitions

The result of the parsing yields the parsed tree consisting of the following classes.

class `paxter.core.`**`Token`**(*start_pos: int*, *end_pos: int*)

Base class for all types of nodes to appear in Paxter document tree.

`end_pos: int`

The index right after the ending position of the token.

`start_pos: int`

The index of the starting position of the token.

class `paxter.core.`**`Fragment`**(*start_pos: int*, *end_pos: int*)

Bases: `paxter.core.data.Token`

Subtypes of nodes in Paxter document tree that is allowed to appear as elements of *`FragmentList`*.

class `paxter.core.`**`TokenList`**(*start_pos: int*, *end_pos: int*, *children: List[paxter.core.data.Token]*)

Bases: `paxter.core.data.Token`

Node type which represents a sequence of tokens wrapped under a pair of parentheses `()`, brackets `[]`, or braces `{}`. It appears exclusively within the option section of *`PaxterApply`*.

`children: List[Token]`

A list of *`Token`*.

class `paxter.core.`**`Identifier`**(*start_pos: int*, *end_pos: int*, *name: str*)

Bases: `paxter.core.data.Token`

Node type which represents an identifier. It can appear at the identifier part of or within the option section of *`PaxterApply`*.

`name: str`

String containing the name of the identifier

class `paxter.core.`**`Operator`**(*start_pos: int*, *end_pos: int*, *symbol: str*)

Bases: `paxter.core.data.Token`

Node type which represents an operator. It appears exclusively within the option section of *`PaxterApply`*.

`symbol: str`

String containing the operator symbol

class `paxter.core.`**`Number`**(*start_pos: int*, *end_pos: int*, *value: Union[int, float]*)

Bases: `paxter.core.data.Token`

Node type which represents a number recognized by JSON grammar. It appears exclusively within the option section of *`PaxterApply`*.

`value: Union[int, float]`

Numerical value deserialized from the number token

class `paxter.core.`**`FragmentList`**(*start_pos: int*, *end_pos: int*, *children: List[paxter.core.data.Fragment]*, *scope_pattern: paxter.core.scope_pattern.ScopePattern*, *is_command: bool = False*)

Bases: `paxter.core.data.Fragment`

Special intermediate node maintaining a list of fragment children nodes. This usually corresponds to global-level fragments or fragments nested within braces following the @-command.

`children: List[Fragment]`

A list of *`Fragment`*

is_command: bool = False
Boolean indicating whether this fragment list begins with @-symbol

scope_pattern: ScopePattern
Information of the enclosing braces pattern

class `paxter.core.`**`Text`**(*start_pos: int*, *end_pos: int*, *inner: str*, *scope_pattern: paxter.core.scope_pattern.ScopePattern*, *is_command: bool = False*)
Bases: `paxter.core.data.Fragment`

Text node type which does not contain nested @-commands. It may be presented as an element of *`FragmentList`*, the main argument of *`PaxterApply`* and *`PaxterPhrase`*, or within the option section of *`PaxterApply`*.

inner: str
The string content

is_command: bool = False
Boolean indicating whether this fragment list begins with @-symbol

scope_pattern: ScopePattern
Information of the enclosing quote pattern

class `paxter.core.`**`PaxterPhrase`**(*start_pos: int*, *end_pos: int*, *inner: str*, *scope_pattern: paxter.core.scope_pattern.ScopePattern*)
Bases: `paxter.core.data.Fragment`

Node type which represents @-command and has one of the following form:

- It begins with a command switch `@` and is immediately followed by a non-empty identifier. It also must unambiguously not be a *`PaxterApply`* (i.e. it is not followed by an option section or main argument section).
- It begins with a command switch `@` and is immediately followed by a wrapped bar section (e.g. `@|...phrase...|`, `@<#|...phrase...|#>`).
- It begins with a command switch `@` and is immediately followed by a single symbol character that is unmistakeably not a quote `"`, a brace `{`, or a bar `|`.

inner: str
The string content of the phrase

scope_pattern: ScopePattern
Information of the enclosing bar pattern

class `paxter.core.`**`PaxterApply`**(*start_pos: int*, *end_pos: int*, *id: paxter.core.data.Identifier*, *options: Optional[paxter.core.data.TokenList]*, *main_arg: Optional[Union[FragmentList, Text]]*)
Bases: `paxter.core.data.Fragment`

Node type which represents @-command which has the following form:

- It begins with a command switch `@`, and is immediately followed by a non-empty identifier.
- Then it may optionally be followed by an option section surrounded by square brackets.
- If options section is present, then it may be followed by a main argument section; however, if options is not present, then it must be followed by the main argument section.

  The main argument section, if present, can either be a *`FragmentList`* (surrounded by wrapped braces such as `{...main arg...}`) or a *`Text`* (surrounded by wrapped quotation marks such as `"...text..."`).

id: Identifier
The identifier part

main_arg: Optional[MainArgument]
The main argument section at the end of expression, or `None` if this section is not present.

options: Optional[TokenList]
A list of tokens for the option section enclosed by `[]`, or `None` if this section is not present.

1.6.3 Exceptions

Here are the list of exceptions raised from this library.

class `paxter.core.exceptions.`**PaxterBaseException**(*message: str*, ***positions: paxter.core.line_col.LineCol*)
Bases: `Exception`

Base exception specific to Paxter language ecosystem.

message: str
Error message

positions: Dict[str, LineCol]
A mapping from position name to `LineCol` position data

class `paxter.core.exceptions.`**PaxterConfigError**(*message: str*, ***positions: paxter.core.line_col.LineCol*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for configuration error.

class `paxter.core.exceptions.`**PaxterSyntaxError**(*message: str*, ***positions: paxter.core.line_col.LineCol*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for syntax error raised while parsing input text in Paxter language. Positional index parameters indicates a mapping from position name to its indexing inside the input text.

class `paxter.core.exceptions.`**PaxterRenderError**(*message: str*, ***positions: paxter.core.line_col.LineCol*)
Bases: *paxter.core.exceptions.PaxterBaseException*

Exception for parsed tree transformation error.

1.6.4 Other Utility Classes

Classes in this subsection is for reference only.

class `paxter.core.`**ScopePattern**(*opening: str*, *closing: str = None*)
Data regarding the opened pattern and the closed pattern of one particular scope.

closing: str = None
The closing pattern enclosing the scope

opening: str
The opening pattern enclosing the scope

class `paxter.core.`**LineCol**(*input_text: dataclasses.InitVar*, *pos: dataclasses.InitVar*)
The starting or ending position of a token within the input text.

col: int
1-index column index value

`line: int`

1-index line number

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

INDEX