Paxter

Abhabongse Janthong

Feb 21, 2021

BEGINNER TUTORIALS

Paxter language helps users write rich-formatting documents with simple and easy-to-understand syntax. Nevertheless, users still have access to the python environment: they can call python functions and evaluate python expressions right inside the document itself, giving users flexibility and power to customize their approach to writing documents.

Paxter package is a document-first, text processing language toolchain, inspired by @-expressions in Racket. Users of the package also has the access to the Paxter language parser API which allows them to implement new interpreters on top of the Paxter language if they so wish.

CHAPTER

ONE

SITE CONTENTS

1.1 Getting Started

1.1.1 Installation

Paxter python package can be installed from PyPI via the following `pip` command. Of course, we can also opt for other methods of python package managements.

```
$ pip install paxter
```

Next, let's write a basic blog entry.

1.1.2 Writing The First Blog Entry

Suppose that we are going to write a blog post under Paxter language syntax as shown in the following. Let's ignore the specific details about the syntax for now as we will discuss them further on the next page.

```
@h1{New Blog!}

Welcome to our new blog website.
@italic{Please keep watching this space for content.}
```

The above content is expected to be rendered into the following result.

Here are a few ways that we can transform the original Paxter source code into the final HTML output.

Method 1: Command Line

Suppose that the Paxter source code (as shown above) is stored within the file called `"new-blog.paxter"`. Once Paxter package is installed, we can run the command `paxter html` to render the HTML output.

```
$ cat new-blog.paxter
@h1{New Blog!}

Welcome to our new blog website.
@italic{Please keep watching this space for content.}

$ paxter html -i new-blog.paxter
<h1>New Blog!</h1><p>Welcome to our new blog website.
<i>Please keep watching this space for content.</i></p>
```

Method 2: Programmatic Usage

A more flexible way to transform Paxter source text into HTML output is to make calls the API functions provided by Paxter library. The easiest way is to do the following.

```
from paxter.quickauthor import run_document_paxter

# The following source text is read from a source file.
# However, in reality, source text may be read from other sources
# such as some databases or even fetched via some content management API.
with open("new-blog.paxter") as fobj:
    source_text = fobj.read()

document = run_document_paxter(source_text)
html_output = document.html()
```

```
>>> print(html_output)
<h1>New Blog!</h1><p>Welcome to our new blog website.
<i>Please keep watching this space for content.</i></p>
```

This approach shown here is merely the very basic usage of Paxter library with *preconfigured* settings. More advanced programmatic usage will be discussed later.

1.2 Quick Blogging

To quickly get started with blogging or writing an article, we introduce the Paxter language syntax as being *preconfigured* by `paxter.author` subpackage which can be used to write rich-formatted content.

Beware

Most of the descriptions about the syntax shown on this page are specific to the preconfigured variation of Paxter language provided by `paxter.author` subpackage. It is actually *not* tied to the core Paxter language specification.

Paxter library (which includes the core Paxter language specification) is designed to be very extensible and customizable. The `paxter.author` subpackage is merely supplementary provided by Paxter library for convenience. It is entirely possible to utilize Paxter library *without* touching any of the `paxter.author` whatsoever, as being demonstrated in the *Under Construction*.

1.2.1 Command: A Basic Building Block

A command is the core building block in Paxter language. It has various syntactical forms, but all of them have the same basic principles:

- Each command always begin with an `@` symbol in the source text.
- Each command always has the **phrase part** which immediately follows the initial `@` symbol.
- Each command may optionally have what is called the **options part**. If it exists, it has the form of `[...]` that follows the phrase part.
- Each command may optionally have what is called the **main argument part**. If it exists, it follows the phrase part or the options part (if the options part exists).

This may sound confusing right now. Hopefully things will get clearer as we discuss the specific syntax on the page *Evaluation Cycle Explained*.

Rule of Thumb

A rule of thumb to remember about the core Paxter language is that it dictates only how a command in the source text should be parsed. It has no bearing on how each parsed command and any other raw text are to be interpreted or evaluated into the desired final output. For descriptions of syntax appeared on this page, this interpretation is done by the supplementary `paxter.author` subpackage.

1.2.2 Bolds, Italics, and Underline

Let's begin with "bolding" part of a source text. We use the command `@bold{...}`, replacing `...` with the actual text to be emphasized. In this particular command, `bold` is the phrase part whereas the emphasized text is the main argument part of the command.

For example, consider the following source text written in Paxter language.

```
This is a very @bold{important part} of the statement.
```

This source text will be transformed to the following HTML output.

```
<p>This is a very <b>important part</b> of the statement.</p>
```

And likewise, for italicized text and underlined text, use the command `@italic{...}` and `@uline{...}` respectively. Notice that we altered the phrase part of the command while the the main argument parts remains the same.

```
This is a very @italic{important part} of @uline{the statement}.
```

This source text will be transformed to the following HTML output.

```
<p>This is a very <i>important part</i> of <u>the statement</u>.</p>
```

Aside: Nested commands

One nice thing about Paxter command is that they are allowed to be nested inside the main argument between the pair of curly braces. For example,

```
This is @italic{so important that @uline{multiple emphasis} is required}.
```

When the above source text get rendered into HTML, we obtain the following result.

```
<p>This is <i>so important that <u>multiple emphasis</u> is required</i>.</p>
```

1.2.3 Monospaced Code

Similarly to what we have seen with `@bold{...}`, `@italic{...}`, and `@uline{...}` from above, we use the command `@code{...}` to encapsulate text to be displayed as monospaced code.

For example, the following source text written in Paxter language

```
Run the @code{python} command.
```

will be evaluated into the HTML output shown below.

```
<p>Run the <code>python</code> command.</p>
```

1.2.4 Multiple Paragraphs

To write multiple paragraphs, simply separate chunks of texts with at least two newline characters (i.e. there must be a blank line between consecutive paragraphs). Each chunk of text will result in its own paragraph. Consider the following example containing exactly three paragraphs.

```
This is @bold{the first paragraph}.
This is the second sentence of the first paragraph.

This is @italic{another} paragraph.

This is the @uline{final} paragraph.
```

The above text in Paxter language will be transformed into the following HTML.

```
<p>This is <b>the first paragraph</b>.
   This is the second sentence of the first paragraph.</p>
<p>This is <i>another</i> paragraph.</p>
<p>This is the <u>final</u> paragraph.</p>
```

Reminder

The implicit paragraph splitting behavior of the source text is preconfigured by the supplementary `paxter.author` subpackage and has nothing to do with the core Paxter language specification.

1.2.5 Headings

To include a heading (from level 1 down to level 6) use the command `@h1{...}` through `@h6{...}` on its own chunk. They must be separated from other paragraph chunks with at least one blank line.

```
@h1{New Blog!}

@bold{Welcome to the new blog!} Let's celebrate!

@h2{Updates}

There is no update.
```

```
<h1>New Blog!</h1>
<p><b>Welcome to the new blog!</b> Let's celebrate!</p>
<h2>Updates</h2>
<p>There is no update.</p>
```

Observe that if the `@h1{...}` and `@h2{...}` were removed from encapsulating the heading text, they would have been rendered as its own paragraph. Try that to see for yourself.

Also, what happens if the command `@h1{...}` accidentally did *not* surround the entire chunk of text? Let's look at this example in which the exclamation mark is located *outside* of the command:

```
@h1{New Blog}!

@bold{Welcome to the new blog!} Let's celebrate!
```

```
<p><h1>New Blog</h1>!</p>
<p><b>Welcome to the new blog!</b> Let's celebrate!</p>
```

Since *not* the entire chunk of heading text is encapsulated by the `@h1{...}` command, Paxter assumes that it is simply just a paragraph. So beware of this kind of errors.

1.2.6 Blockquote

The `@blockquote{...}` command must reside on its own chunk just like a heading command. So the following Paxter source text

```
They said that

@blockquote{I refuse.}
```

would be transformed into the following HTML output.

```
<p>They said that</p>
<blockquote>I refuse.</blockquote>
```

However, suppose that we want to include multiple paragraphs inside the blockquote. We can follow the similar rules as to how to write multiple paragraphs in general: by separating them with at least one blank lines. This is demonstrated in the following example.

```
They said that

@blockquote{
    I refuse.

    Then I regret.
}
```

```
<p>They said that</p>
<blockquote>
    <p>I refuse.</p>
    <p>Then I regret.</p>
</blockquote>
```

The important key to note here is that, each paragraph within the blockquote will be surrounded by a paragraph tag `<p>...</p>` as long as more than one chunk of text exists.

Reminder

This particular behavioral rule is enforced by `paxter.author` mainly for convenience. Again, it has nothing to do with the core Paxter language specification.

Aside: Manual Paragraph Annotation

However, if we wish to force wrap the only paragraph within the blockquote with a paragraph tag, we can manually wrap that part of text with the `@paragraph{...}` command. Let's reconsider the first example of this section again. If we wish to have a paragraph tag surround the text "I refuse.", then we can write as follows.

```
They said that

@blockquote{@paragraph{I refuse.}}
```

And we would get the following HTML output.

```
<p>They said that</p>
<blockquote><p>I refuse.</p></blockquote>
```

By the way, do you remember when an entire chunk of text was contained within a command such as `@h1{...}`? As a result, that particular chunk of text did not get treated with paragraph tag `<p>...</p>`. While this behavior is desirable for heading commands, it is not the case for other inline commands such as `@bold{...}`, `@italics{...}` or `@uline{...}`. For these commands, explicit `@paragraph{...}` is needed.

```
@bold{Bold text without paragraph encapsulation.}

@paragraph{@bold{Bold text paragraph.}}
```

```
<b>Bold text without paragraph encapsulation.</b>
<p><b>Bold text paragraph.</b></p>
```

1.2.7 Links and Images

So far, all of the commands we have seen on this page contains the phrase section followed by the main argument part. Now it is time to introduce other variations of a command syntax, especially those which contain the options part.

To put a link such as a URL on a piece of text, we use the command `@link["target"]{text}` replacing the `"target"` with the string literal containing the actual target URL. The displaying text would still be those in between the curly braces.

Here is an example of the usage of the `@link` command.

```
Click @link["http://example.com"]{here} to go to my website.
```

```
<p>Click <a href="http://example.com">here</a> to go to my website.</p>
```

Next, to insert an image, we use the command `@image["srcpath", "alt"]`. Notice that this command does not have the main argument part. The options part of this commands accepts two arguments: the first one being the string literal containing the URL path to the image and the second one is for the image alternative text. In fact, the second argument is actually *not* required and will default to an empty string. For example,

```
@image["http://example.com/hello.png", "hello"]

@image["http://example.com/bye.png"]
```

The above Paxter text will be rendered into the following HTML.

```
<img src="http://example.com/hello.png" alt="hello" />
<img src="http://example.com/bye.png" alt="" />
```

Notice

If you are thinking that the options part of a command syntax looks eerily similar to function call syntax in python, do take note that this happens by design. We will dive into more details about the structure of command syntax on the page *Evaluation Cycle Explained*.

1.2.8 Lists

There are two kinds of list enumerations: numbered list and bulleted list (sometimes known as ordered and unordered lists respectively). To create a numbered list, use the `@numbered_list[...]` command where each argument of the options part represents an item of the list. The textual content for each item must be enclosed by a pair of curly braces like in the following example.

```
@numbered_list[
    {This is the first item.},
    {This is the @italic{second} item.},
    {This is the last item.},
]
```

```
<ol>
    <li>This is the first item.</li>
    <li>This is the <i>second</i> item.</li>
    <li>This is the last item.</li>
</ol>
```

Similarly, for bulleted list, use `@bulleted_list[...]` command with the similar structure.

What happens there are more than one chunk of text separated by a single blank line within one of the items of the list? The paragraph splitting rules for `@blockquote{...}` also applies here, as demonstrated in the following example.

```
@bulleted_list[
    {
        @bold{Rule number one.} Be clear.

        Very clear indeed.
    },
    {@bold{Rule number two.} Be consistent.},
]
```

```
<ul>
    <li>
        <p><b>Rule number one.</b> Be clear.</p>
        <p>Very clear indeed.</p>
    </li>
```

(continues on next page)

(continued from previous page)

```
    <li><b>Rule number two.</b> Be consistent.</li>
</ul>
```

And yes, if there is only one paragraph and the explicit tag is needed, wrap the content with the `@paragraph{...}` command.

1.2.9 Tables

Essentially, a table is a sequence of rows, and each row is a sequence of cells. To construct a table, we use the command `@table[...]` where each argument within the options part must be a command of the form `@table_header[...]` for table header rows or `@table_row[...]` for table data rows. In turn, each cell within a table row would be wrapped in curly braces and presented as an argument inside the options part of `@table_header[...]` or `@table_row[...]`

To demystify this tedious explanation, consider the following example.

```
@table[
    @table_header[{No.}, {Name}, {Age}],
    @table_row[
        {1},
        {FirstnameA LastnameA},
        {21},
    ],
    @table_row[
        {2},
        {FirstnameB LastnameB},
        {34},
    ],
    @table_row[
        {3},
        {FirstnameC LastnameC},
        {55},
    ],
]
```

```
<table>
    <tr>
        <th>No.</th>
        <th>Name</th>
        <th>Age</th>
    </tr>
    <tr>
        <td>1</td>
        <td>FirstnameA LastnameA</td>
        <td>21</td>
    </tr>
    <tr>
        <td>2</td>
        <td>FirstnameB LastnameB</td>
        <td>34</td>
    </tr>
    <tr>
        <td>3</td>
        <td>FirstnameC LastnameC</td>
        <td>55</td>
```

(continues on next page)

(continued from previous page)

```
    </tr>
</table>
```

Paragraph splitting rules also applies to each cell data just like within a blockquote or within an item of a list.

1.2.10 Raw HTML

In HTML, symbols such as `&`, `<`, `>`, and `"` requires **escaping** in order to be properly displayed in the rendered output (in the form of `&amp;`, `&lt;`, `&gt;`, and `&quot;` respectively). For HTML rendering performed by the `paxter.author` subpackage, the escaping of these special characters are automatically done for both convenience and safety reasons.

However, there might be times you wish to include HTML tags or HTML entities such as `<del>...</del>` or `&ndash;`. This can be done using the command of the form `@raw"text"`. For example,

```
Let's count A&ndash;Z.

No, I mean A@raw"&ndash;"Z!

Use <del>...</del> for @raw"<del>"strikethrough@raw"</del>" text.
```

```
<p>Let's count A&amp;ndash;Z.</p>
<p>No, I mean A&ndash;Z!</p>
<p>Use &lt;del&gt;...&lt;/del&gt; for <del>strikethrough</del> text.</p>
```

And this is how the above HTML code is displayed:

Pre-defined Raw HTML

For convenience, the `paxter.author` has already defined a few of common raw HTML strings for use, as shown below.

Command	HTML equivalent	Meaning
`@hrule`	`<hr />`	Thematic break
`@line_break`, `@\`	`<br />`	Line break
`@nbsp`, `@%`	`&nbsp;`	Non-breaking space
`@hairsp`, `@.`	`&hairsp;`	Hair space
`@thinsp`, `@,`	`&thinsp;`	Thin space

Notes about commands in the above table

- Every command shown in the table expects *neither* the options part *nor* the main argument part. This is one valid form of a command in Paxter language.
- Observe that there is a rather unusual form of a command, which consists of the `"@"` symbol immediately followed by another symbol character (namely `@\`, `@%`, `@.`, and `@,`). This is called the **symbol-only form**. The parsing rules of this kind of command is distinct from other commands we have seen up until this point. Specific differences will be discussed on later pages in this documentation.

Here is some usage example.

```
The store opens Monday@,-@,Friday @line_break 9@%AM@,-@,5@%PM.evaluate-and-execute-
↪python-code

@hrule
```

```
<p>The store opens Monday&thinsp;-&thinsp;Friday <br />
   9&nbsp;AM&thinsp;-&thinsp;5&nbsp;PM.</p>
<hr />
```

1.3 Evaluation Cycle Explained

On this page, we are going to see what happens under the hood when a source text in Paxter language got parsed and interpreted. Let's consider evaluating the following source text as our motivating example:

```
Please visit @link["https://example.com"]{@italic{this} website}. @line_break
@image["https://example.com/hello.jpg", "hello"]
```

We are going to assume that we use the function `run_document_paxter()` in order to evaluate the above source text into the final HTML output. This transformation can be divided into three logical steps.

1. Parsing source text
2. Evaluating parsed tree into document object
3. Rendering document object

1.3.1 Step 1: Parsing Source Text

Specifically, the core `paxter.parse` subpackage implements a parser, `ParseContext`, which parses a source text (written in Paxter language) into the parsed tree form. Here is how to use python API to run this step.

```
from paxter.syntax import _ParsingTask

source_text = '''Please visit @link["https://example.com"]{@italic{this} website}.␣
↪@line_break
@image["https://example.com/hello.jpg", "hello"]'''
parsed_tree = _ParsingTask(source_text).tree
```

We can also see the content of the `parsed_tree` if we print them out. However, feel free to skip over this big chunk of output as they are not relevant to what we are discussing right now.

```
>>> print(parsed_tree)
FragmentSeq(
    start_pos=0,
    end_pos=126,
    children=[
        Text(start_pos=0, end_pos=13, inner="Please visit ",␣
↪enclosing=EnclosingPattern(left="", right="")),
        Command(
            start_pos=14,
            end_pos=64,
            phrase="link",
            phrase_enclosing=EnclosingPattern(left="", right=""),
            options=TokenSeq(
```

(continues on next page)

(continued from previous page)

```
                start_pos=19,
                end_pos=40,
                children=[
                    Text(
                        start_pos=20,
                        end_pos=39,
                        inner="https://example.com",
                        enclosing=EnclosingPattern(left='"', right='"'),
                    )
                ],
            ),
            main_arg=FragmentSeq(
                start_pos=42,
                end_pos=63,
                children=[
                    Command(
                        start_pos=43,
                        end_pos=55,
                        phrase="italic",
                        phrase_enclosing=EnclosingPattern(left="", right=""),
                        options=None,
                        main_arg=FragmentSeq(
                            start_pos=50,
                            end_pos=54,
                            children=[
                                Text(
                                    start_pos=50,
                                    end_pos=54,
                                    inner="this",
                                    enclosing=EnclosingPattern(left="", right=""),
                                )
                            ],
                            enclosing=EnclosingPattern(left="{", right="}"),
                        ),
                    ),
                    Text(start_pos=55, end_pos=63, inner=" website",␣
↪enclosing=EnclosingPattern(left="", right="")),
                ],
                enclosing=EnclosingPattern(left="{", right="}"),
            ),
        ),
        Text(start_pos=64, end_pos=66, inner=". ", enclosing=EnclosingPattern(left="",
↪ right="")),
        Command(
            start_pos=67,
            end_pos=77,
            phrase="line_break",
            phrase_enclosing=EnclosingPattern(left="", right=""),
            options=None,
            main_arg=None,
        ),
        Text(start_pos=77, end_pos=78, inner="\n", enclosing=EnclosingPattern(left="",
↪ right="")),
        Command(
            start_pos=79,
            end_pos=126,
            phrase="image",
```

(continues on next page)

(continued from previous page)

```
            phrase_enclosing=EnclosingPattern(left="", right=""),
            options=TokenSeq(
                start_pos=85,
                end_pos=125,
                children=[
                    Text(
                        start_pos=86,
                        end_pos=115,
                        inner="https://example.com/hello.jpg",
                        enclosing=EnclosingPattern(left='"', right='"'),
                    ),
                    Operator(start_pos=116, end_pos=117, symbols=","),
                    Text(start_pos=119, end_pos=124, inner="hello",␣
↪enclosing=EnclosingPattern(left='"', right='"')),
                ],
            ),
            main_arg=None,
        ),
    ],
    enclosing=GlobalEnclosingPattern(),
)
```

Dear Advanced Users

For those who are familiar with the field of Programming Languages, this maybe enough to get you run wild! See the *syntax reference* and the *data definitions for parsed tree nodes* to help get started right away.

1.3.2 Step 2: Evaluating Parsed Tree Into Document Object

The `parsed_tree` from the previous step is then interpreted by a tree transformer from the `paxter.interpret` subpackage. In general, what a parsed tree would be evaluated into depends on each individual (meaning you, dear reader).

Paxter library decides to implement *one possible version* of a tree transformer called `InterpreterContext`. This particular transformer tries to **mimic the behavior of calling python functions** as closest possible. In addition, this transformer expects what is called the **initial environment dictionary** under which python executions are performed. For this particular scenario, this dictionary is created by the function `create_document_env()` from the `paxter.author` subpackage. This environment dictionary contains the mapping of function aliases to the actual python functions and object and it is where the magic happens.

Let's look at the contents of the environment dictionary created by the above function `create_document_env()`.

```
from paxter.quickauthor.environ import create_document_env

env = create_document_env()
```

```
>>> env
{'_phrase_eval_': <function paxter.authoring.standards.phrase_unsafe_eval>,
 '_extras_': {},
 '@': '@',
 'for': <function paxter.authoring.controls.for_statement>,
 'if': <function paxter.authoring.controls.if_statement>,
 'python': <function paxter.authoring.standards.python_unsafe_exec>,
```

(continues on next page)

(continued from previous page)

```
'verb': <function paxter.authoring.standards.verbatim>,
'raw': <class paxter.authoring.elements.RawElement>,
'paragraph': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h1': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h2': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h3': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h4': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h5': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'h6': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'bold': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'italic': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'uline': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'code': <classmethod paxter.authoring.elements.SimpleElement.from_fragments>,
'blockquote': <classmethod paxter.authoring.elements.Blockquote.from_fragments>,
'link': <classmethod paxter.authoring.elements.Link.from_fragments>,
'image': <class paxter.authoring.elements.Image>,
'numbered_list': <classmethod paxter.authoring.elements.EnumeratingElement.from_
↪direct_args>,
'bulleted_list': <classmethod paxter.authoring.elements.EnumeratingElement.from_
↪direct_args>,
'table': <classmethod paxter.authoring.elements.SimpleElement.from_direct_args>,
'table_header': <classmethod paxter.authoring.elements.EnumeratingElement.from_
↪direct_args>,
'table_row': <classmethod paxter.authoring.elements.EnumeratingElement.from_direct_
↪args>,
'hrule': RawElement(body='<hr />'),
'line_break': RawElement(body='<br />'),
'\\': RawElement(body='<br />'),
'nbsp': RawElement(body='&nbsp;'),
'%': RawElement(body='&nbsp;'),
'hairsp': RawElement(body='&hairsp;'),
'.': RawElement(body='&hairsp;'),
'thinsp': RawElement(body='&thinsp;'),
',': RawElement(body='&thinsp;')}
```

It is crucial to point out that, all of the commands we have seen so far on the page *Quick Blogging* (e.g. `bold`, `h1`, `blockquote`, `numbered_list`, `table`, and many others) are some keys of the `env` dictionary object as listed above. This is *not* a coincidence. Essentially, Paxter library utilizes the data from this dictionary in order to properly interpret each command in the source text.

Interpreting a Command

The process of interpreting a command is divided into two steps: resolving the phrase and invoking a function call. Let's explore each step assuming the initial environment dictionary `env` (borrowed from above).

1. **Resolve the phrase part.** By default, the phrase part is used as the key for looking up a python value from the environment dictionary `env`. For example, resolving the phrase `italic` from the command `@italic{...}` would yield the value of `env["italic"]` which refers to `Italic.from_fragments()` class method. Likewise, the phrase `link` from the command `@link["target"]{text}` maps to `Link.from_fragments()` under the dictionary `env`.

   Fallback Plan

   However, if the key which is made of the phrase part does not appear in `env` dictionary, then the fallback plan is to use python built-in function `eval()` to **evaluate the entire phrase string** with `env` as the global namespace. This fallback behavior enables a myriad of features in Paxter ecosystem including evaluating an anonymous python expression from right within the source text. In order to encode any string as the phrase of a command, we need to introduce a slightly different syntactical form of a command, which we would cover *in a later tutorial*, but here is a little taste of that:

```
The result of 7 * 11 * 13 is @|7 * 11 * 13|.
```

```
<p>The result of 7 * 11 * 13 is 1001.</p>
```

Noteworthy

The phrase part resolution behavior (as described above) is completely dictated by the default function located at `env["_phrase_eval_"]`. This behavior can be fully customized by switching out the default function and replacing it with another implementation with identical function signature. See *Disable Python Environment (Demo)* to learn how to customize this behavior and see how it affects the entire cycle of command evaluation.

2. **Invoke a function call.** Before we continue, if the original command contains *neither* the options *nor* the main argument parts, then the python object returned from step 1 will not be further process and will immediately become the final output of the command interpretation.

   Otherwise, the available options part and the main argument part will all become input arguments of a function call to the object returned by the previous step. Of course, that python object is expected to be callable in order to work. Particularly,

   - If the main argument part exists, its value will always be the very first input argument of the function call. If the options part also exists, then each of its items (separated by commas) will be subsequent arguments of the function call.
   - If the main argument part does not exist, then all of the items from the options part will be sole input arguments of the function call.

Let's walkthrough these two-step process with a few examples.

Example 1: Non-Callable Command

Let's begin with a basic example. The command `@line_break` on its own would get translated roughly into the following python code equivalent. The final result is stored inside the variable `result`.

```
# Step 1: resolving the phrase
line_break_obj = env['line_break']  # paxter.quickauthor.elements.line_break
# Step 2 is skipped since there is no function call
result = line_break_obj
```

Example 2: Command With Main Argument

Consider the command `@italic{this}`. It would be transformed into the following python equivalent:

```
# Step 1: resolving the phrase
italic_obj = env['italic']  # paxter.quickauthor.elements.Italic.from_fragments
# Step 2: function call
result = italic_obj(FragmentList(["this"]))
```

Notice that the main argument part `{this}` of the command `@italic{this}` gets translated to `FragmentList(["this"])` in python representation. In Paxter's terminology, any component of the command syntax which is enclosed by a pair of matching curly braces would be known as **a fragment list**, and it would be represented as a list of subtype `FragmentList`.

Example 3: Command With Both Options and Main Argument

Let's look at this rather complicated command and its python code equivalent.

```
@link["https://example.com"]{@italic{this} website}
```

```
# Step 1: resolving the phrases
italic_obj = env['italic']  # paxter.quickauthor.elements.Italic.from_fragments
link_obj = env['link']  # paxter.quickauthor.elements.Link.from_fragments

# Step 2: function call
result = link_obj(
    FragmentList([
        italic_obj(FragmentList(["this"])),  # just like previous example
        " website",
    ]),
    "https://example.com",
)
```

There are a few notes to point out:

- The first input argument of the function call to `link_obj` derives from the main argument fragment list, which contains the nested function call to `italic_obj`.
- The target URL `"https://example.com"` appeared in the options part of the `@link` command becomes the second argument in the function call to `link_obj`.

To provide further clarification of how a command in Paxter source text gets translated, consider the following example where a command contains two argument items within its options part.

```
@foo["bar", 3]{text}
```

```
# Step 1: resolving the phrases
foo_obj = env['foo']
# Step 2: function call
result = foo_obj(FragmentList(["text"]), "bar", 3)
```

Python-style keyword arguments are also supported within the options part, and it works in the way we expect.

```
@foo["bar", n=3]{text}
```

```
# Step 1: resolving the phrase
foo_obj = env['foo']
# Step 2: function call
result = foo_obj(FragmentList(["text"]), "bar", n=3)
```

Example 4: Commands With Options Only

In the master example at the beginning of this page, we can see the following `@image` command:

```
@image["https://example.com/hello.jpg", "hello"]
```

Because the main argument part is not present inside the `@image` command, the above source text would be interpreted similarly to the following python code.

```
# Step 1: resolving the phrase
image_obj = env['image']  # paxter.quickauthor.elements.Image
# Step 2: function call
result = image_obj("https://example.com/hello.jpg", "hello")
```

Is there a way to make a function call to the object with zero arguments? Of course. It can be done by writing square brackets containing nothing inside it.

```
@foo[]
```

```
# Step 1: resolving the phrase
foo_obj = env['foo']
# Step 2: function call
result = foo_obj()
```

Beware *not* to use curly braces in place of square brackets as it would have resulted in slightly different interpretation, like in the following.

```
@foo{}
```

```
# Step 1: resolving the phrase
foo_obj = env['foo']
# Step 2: function call
result = foo_obj(FragmentList([]))
```

Motivating Example Revisited

By combining all of the above examples, we can describe the semantics of the motivating example as shown in the following python code (the original source text is reproduced below for convenience):

```
Please visit @link["https://example.com"]{@italic{this} website}. @line_break
@image["https://example.com/hello.jpg", "hello"]
```

```
# Step 1: resolving the phrases
italic_obj = env['italic']  # paxter.quickauthor.elements.Italic.from_fragments
link_obj = env['link']  # paxter.quickauthor.elements.Link.from_fragments
line_break_obj = env['line_break']  # paxter.quickauthor.elements.line_break
image_obj = env['image']  # paxter.quickauthor.elements.Image
```

(continues on next page)

(continued from previous page)

```
# Step 2: function call
document_result = FragmentList([
    "Please visit ",
    link_obj(
        FragmentList([
            italic_obj(FragmentList(["this"])),
            " website",
        ]),
        "https://example.com",
    ),
    ". ",
    line_break_obj,
    "\n",
    image_obj("https://example.com/hello.jpg", "hello"),
])
```

However, the actual python API to replicate the above result is as follows (where `parsed_tree` is the result borrowed from step 1).

```
from paxter.quickauthor.environ import create_document_env
from paxter.interp.task import InterpretingTask

env = create_document_env()
document_result = InterpretingTask(source_text, env, parsed_tree).rendered
```

The result of interpreting the entire source text using `InterpreterContext` is always going to be a fragment list of each smaller pieces of content (which is why the `document_result` in the above code is an instance of `FragmentList` class). Displaying the content of `document_result` gives us the following evaluated result.

```
>>> document_result
FragmentList([
    "Please visit ",
    Link(body=[Italic(body=["this"]), " website"], href="https://example.com"),
    ". ",
    RawElement(body="<br />"),
    "\n",
    Image(src="https://example.com/hello.jpg", alt="hello"),
])
```

1.3.3 Step 3: Rendering Document Object

Reminder Again

In all truthfulness, rendering the `final_result` into HTML string output has *nothing* to do with the core Paxter language specification. In fact, if library users implement their own version of parsed tree evaluator, this particular step would be non-existent.

Rendering the entire `document_result` into HTML string output is simple. Two small steps are required:

1. Wrap the `document_result` with `Document`
2. Invoke the `Document.html()` method.

And here is the python code to do exactly as just said:

```
from paxter.quickauthor.elements import Document

document = Document.from_fragments(document_result)
html_output = document.html()
```

This yields the following final HTML output:

```
>>> print(html_output)
<p>Please visit <a href="https://example.com"><i>this</i> website</a>. <br />
<img src="https://example.com/hello.jpg" alt="hello" /></p>
```

Preset Function

The preset function `run_document_paxter()` introduced in the section *Programmatic Usage* (from Getting Started page) simply performs all three steps as mentioned above in order.

1.4 Interpreting Python Code

Let's assume that we are still using the environment dictionary created by `create_document_env()` *together* with the default interpreter, which is implemented by `InterpreterContext`. Under this particular setup, there are various ways to embed and run python code within the source text itself. We discuss each possibility below.

1.4.1 Executing Python Statements

It would seem at first that there is no way to introduce a new binding from a phrase to a python object into the environment dictionary on an ad-hoc basis (i.e. from within the source text). Actually, we can do so through the `@python"..."` command, by putting actual python statements in-between the pair of quotation marks.

Assigning Variables

Here is one example where a long string is pre-defined once, and reused multiple times.

```
@python"yaa = 'Yet Another Acronym'"
YAA is @yaa and it stands for @yaa.
```

```
<p>YAA is Yet Another Acronym and it stands for Yet Another Acronym.</p>
```

The command phrase `@python` maps to the callable object `python_unsafe_exec()`. What this particular function does is executing the entire python source through the built-in `exec()` function, using the environment dictionary `env` as the global namespace. When the assignment statement `yaa = 'Yet Another Acronym'` gets executed, then the entry `env['yaa']` gets populated with the string `"Yet Another Acronym"`, which is why the command `@yaa` can subsequently be referred to within the source text.

Yet, a burning question arises: what happens if the python source code itself has to contain quotation mark characters when we also use it to delimit the main argument part of the `@python` command itself? Let's try that out!

```
@python"yaa = "Yet Another Acronym""
YAA is @yaa and it stands for @yaa.
```

Attempting to evaluate the above source text yields the following error (omitting traceback for clarity):

```
Traceback (most recent call last):
  ...
  File "<string>", line 1
    yaa =
         ^
SyntaxError: invalid syntax

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  ...
paxter.exceptions.PaxterRenderError: paxter apply evaluation error at line 1 col 2
```

The reason behind this error is that the main argument part of the command was prematurely terminated at the first (closing) quotation mark character it finds. Therefore, the incomplete python statement `yaa =` was parsed, which yields the above error when executed.

The solution around this problem is to additionally enclose the quoted main argument with **an equal number of hash characters** to both ends of the quoted argument. For example, in the source text below, the python source code begins at `#"` and ends at `"#` (though we can also use the `##"`, `"##` pair as well).

```
@python#"yaa = "Yet Another Acronym""#
YAA is @yaa and it stands for @yaa.
```

More Information

The **hash-enclosing rule** is enforced by the core Paxter language specification, and it applies at other locations as well. Learn more on Paxter's ways to escape special characters on the page *Escaping Mechanisms*.

Define New Functions

Continuing on the same line of thinking from above, we could also define python functions using `@python` command and make calls to them using command syntax from within the source text.

For example, we will create a new function that will repeat the main argument a few times.

```
@python##"
def repeat(main_arg, n=2):
    return n * main_arg
"##

@repeat{woof}

@repeat[3]{@bold{hi}}

@repeat[n=4]{@repeat{?}!}
```

```
<p>woofwoof</p>
<p><b>hi</b><b>hi</b><b>hi</b></p>
<p>??!??!??!??!</p>
```

Using Imported Values and Functions

We can also use command syntax to refer to values and functions obtained through the import statement.

```
@python##"
from string import ascii_uppercase
from textwrap import shorten
"##

Letters in English alphabet are @ascii_uppercase.

@shorten[15]#"Good morning world!"#

@shorten["Good evening everyone.", width=20]
```

```
<p>Letters in English alphabet are ABCDEFGHIJKLMNOPQRSTUVWXYZ.</p>
<p>Good [...]</p>
<p>Good evening [...]</p>
```

Did you spot something?

Readers with a pair of eagle eyes will be able to spot that we are liberally using the hash-enclosing rule here at the first `@shorten` command as well, albeit totally unnecessary. This is to illustrate that this hash-enclosing rule works for *any* command (not just `@python`).

1.4.2 Evaluating Python Expressions

Applying all of the knowledge we have learned so far with Paxter library, one way to evaluate a python expression and print its result inside the source text is to do the following two steps:

1. Inside the `@python` command, evaluate the desired expression and assign its result to a variable.
2. Refer to the value of such variable through the command syntax.

Here is an example to demonstrate the above process.

```
@python##"
def add_one(value):
    return value + 1

ninetynine_plus_one = add_one(99)
product = 7 * 11 * 13
"##

The result of 99 + 1 is @ninetynine_plus_one.

The result of 7 * 11 * 13 is @product.
```

```
<p>The result of 99 + 1 is 100.</p>
<p>The result of 7 * 11 * 13 is 1001.</p>
```

Fortunately, there is a much nicer way to evaluate an anonymous python expression and insert its evaluation result right at *that* location: by using the `@|...|` syntax, replacing `...` with the expression itself.

```
@python##"
def add_one(value):
    return value + 1
"##

The result of 99 + 1 is @|add_one(99)|.

The result of 7 * 11 * 13 is @|7 * 11 * 13|.
```

```
<p>The result of 99 + 1 is 100.</p>
<p>The result of 7 * 11 * 13 is 1001.</p>
```

Although this new syntax `@|...|` may seem new, it is actually an alternative form of the **same old syntax command** with the same old semantics. Here are some key points about this syntax and the command syntax in general.

- `@|...|` is still considered a command in Paxter language (no different from other commands we have seen so far up until this point). For this particular syntax, **everything between the pair of bars is the phrase part of the command**. In fact, the command syntax `@foo` is the short form of `@|foo|` (both are syntactically equivalent). This realization also applies to commands with options part and/or main argument part. For example, `@foo[bar]{baz}` can also be written in full form as `@|foo|[bar]{baz}`. Conversely, one may say that the phrase part may be written *without the bars* if the entire phrase string resembles a python identifier form.
- Do you remember the section *Interpreting a Command* from a previous page where we discuss how a command is being interpreted? The very first step is to **resolve the phrase part**. Notice that as part of the *the backup plan*, the entire phrase will be evaluated as a python expression. This is why commands like `@|add_one(99)|` and `@|7 * 11 * 13|` work the way it is.

Let's look at another example in which a command has a function call form but the callable object is not stored inside a simple python identifier.

```
@python##"
import string, textwrap
"##

Letters in English alphabet are @|string.ascii_uppercase|.

@|textwrap.shorten|[15]#"Good morning world!"#

@|textwrap.shorten|["Good evening everyone.", width=20]
```

```
<p>Letters in English alphabet are ABCDEFGHIJKLMNOPQRSTUVWXYZ.</p>
<p>Good [...]</p>
<p>Good evening [...]</p>
```

And below is another example of rather complicated usage of the command syntax (some concepts appeared below have not yet been discussed).

```
@python##"
import statistics
d6_faces = [1, 2, 3, 4, 5, 6]
"##

The expected outcome of rolling a D6 is @|statistics.mean|[@d6_faces].
If we remove the first item from the list (which is @|d6_faces.pop|[0])
then we are left with @|' '.join|[@map[@str, @d6_faces]].
```

```
<p>The expected outcome of rolling a D6 is 3.5.
   If we remove the first item from the list (which is 1)
   then we are left with 2 3 4 5 6.</p>
```

Before we move on, there is one more issue to address: if the phrase part of a command could just be any python expression, then how do we write expressions that contain bar characters themselves (e.g. doing the bitwise or operation and the set union operation)? Note that this kind of problem is very similar the previous problem (discussed earlier on this page) where it was tricky to include quotation mark characters within quoted main argument, remember?

Paxter decides to solve all of these problem in the same way, again, through hash-enclosing rule. For example,

```
The bitwise OR between 5 and 9 is @##|5 | 9|##.

The union of set {1, 2, 4, 8} and {2, 3, 5, 7} is @#|{1, 2, 4, 8} | {2, 3, 5, 7}|#.
```

```
<p>The bitwise OR between 5 and 9 is 13.</p>
<p>The union of set {1, 2, 4, 8} and {2, 3, 5, 7} is {1, 2, 3, 4, 5, 7, 8}.</p>
```

1.5 Disable Python Environment (Demo)

In this demo, we are going to customize the initial environment dictionary in order to prevent arbitrary python code execution whatsoever. Perhaps we as a programmer would like our users to write some content using Paxter language without any risk of arbitrary code execution.

By default, the initial environment dictionary created by `create_document_env()` allows python code execution through two distinct endpoints:

- the `@python` command
- the anonymous python expression evaluation of phrase part of a command (which is dictated by the function `python_unsafe_eval()` located at `env["_phrase_eval_"]` of the environment `env`)

For the first endpoint, we simply remove the command from the environment, whereas for the second endpoint, we replace the function located at `env["_phrase_eval_"]` with another variant that does not make a call to `eval()` built-in function.

```python
from typing import Optional

from paxter.quickauthor.controls import for_statement, if_statement
from paxter.quickauthor.elements import (
  Blockquote, Bold, BulletedList, Code,
  Heading1, Heading2, Heading3, Heading4, Heading5, Heading6,
  Image, Italic, Link, NumberedList, Paragraph, RawElement,
  Table, TableHeader, TableRow, Underline,
  hair_space, horizontal_rule, line_break,
  non_breaking_space, thin_space,
)
from paxter.quickauthor.standards import verbatim


def phrase_safe_eval(phrase: str, env: dict) -> Any:
  """
  Safely evaluates the given phrase of a command.
  If performs the evaluation in the following order.
```

(continues on next page)

(continued from previous page)

```
    1. Looks up the value from ``env['_extras_']`` dict using phrase as key
    2. Looks up the value from ``env`` dict using phrase as key.

    The implementation of this function is borrowed from inspecting
    :func:`paxter.authoring.standards.phrase_unsafe_eval`.
    """
    if not phrase:
        return None
    extras = env.get('_extras_', {})
    if phrase in extras:
        return extras[phrase]
    if phrase in env:
        return env[phrase]
    raise KeyError(f"there is no command with key: {phrase}")


def create_safe_document_env(data: Optional[dict] = None):
    """
    Creates an string environment data for Paxter source code evaluation
    in Python authoring mode, specializes in constructing documents.

    The implementation of this function is borrowed from inspecting
    :func:`paxter.authoring.environ.create_document_env`.
    """
    data = data or {}
    return {
        '_phrase_eval_': phrase_safe_eval,
        '_extras_': {},
        '@': '@',
        'for': for_statement,
        'if': if_statement,
        # 'python': python_unsafe_exec,
        'verb': verbatim,
        'raw': RawElement,
        'paragraph': Paragraph.from_fragments,
        'h1': Heading1.from_fragments,
        'h2': Heading2.from_fragments,
        'h3': Heading3.from_fragments,
        'h4': Heading4.from_fragments,
        'h5': Heading5.from_fragments,
        'h6': Heading6.from_fragments,
        'bold': Bold.from_fragments,
        'italic': Italic.from_fragments,
        'uline': Underline.from_fragments,
        'code': Code.from_fragments,
        'blockquote': Blockquote.from_fragments,
        'link': Link.from_fragments,
        'image': Image,
        'numbered_list': NumberedList.from_direct_args,
        'bulleted_list': BulletedList.from_direct_args,
        'table': Table.from_direct_args,
        'table_header': TableHeader.from_direct_args,
        'table_row': TableRow.from_direct_args,
        'hrule': horizontal_rule,
        'line_break': line_break,
        '\\': line_break,
        'nbsp': non_breaking_space,
```

(continues on next page)

(continued from previous page)

```
    '%': non_breaking_space,
    'hairsp': hair_space,
    '.': hair_space,
    'thinsp': thin_space,
    ',': thin_space,
    **data,
  }
```

And now we may safely evaluate the content written in Paxter language without having to worry that there may be arbitrary python code execution by using the initial environment dictionary created by the function `create_safe_document_env()` from above.

```
from paxter.quickauthor import run_document_paxter

# The following source text is read from a source file.
# However, in reality, source text may be read from other sources
# such as some databases or even fetched via some content management API.
with open("new-blog.paxter") as fobj:
  source_text = fobj.read()

env = create_safe_document_env()  # from above
document = run_document_paxter(source_text, env)
html_output = document.html()
```

1.6 Escaping Mechanisms

Upon tinkering with writing blog posts through `paxter.author` subpackage, we would eventually find out some technical limitations with the command syntax in Paxter language. On this page, we discuss these limitations.

1.6.1 Escaping '@'

As readers have already noticed that '@' symbol has special meaning in Paxter language: it acts as a switch which turns the subsequence piece of source text into a command. Therefore, if Paxter library users wish to include '@' string literal as-is in the final HTML output, an escape of some sort is required.

. . . except that the core Paxter language specification actually does *not* provide a way to escape '@' symbols per se. However, there are a few ways around this.

Method 1: Define Constants For '@'

We will take advantage of being able to run python code within the source text. Specifically, we will define a variable to store the `@` symbol character.

```
@python##"
at = '@'
"##
This is the @bold{at} symbol: @at.
```

```
<p>This is the <b>at</b> symbol: @.</p>
```

But this method would not work when you wish to write an email address or a twitter handle. For this, additional bar-delimiters surrounding the phrase is needed (*see the next section of this page for more information*).

```
@python##"
at = '@'
"##
Email me at @link["mailto:person@example.com"]{person@|at|example.com}
and my twitter handle is @|at|example. Don’t @at me.
```

```
<p>Email me at <a href="mailto:person@example.com">person@example.com</a>
   and my twitter handle is @example. Don’t @ me.</p>
```

Method 2: Using `@verb` Command

The pre-defined `@verb` command (short for **verbatim**) accepts a string argument and returns it as-is. Here is an example of how to author the same document from the previous example.

```
Email me at @link["mailto:person@example.com"]{@verb##"person@example.com"##}
and my twitter handle is @verb"@"example. @verb"Don’t @ me".
```

```
<p>Email me at <a href="mailto:person@example.com">person@example.com</a>
   and my twitter handle is @example. Don’t @ me.</p>
```

Method 3: Using Symbol-Only Command

Recall the *Pre-defined Raw HTML* section from a past page. We have the commands `@\`, `@%`, `@.`, and `@,` as shortcuts for some raw HTML strings. In fact, commands under the symbol-only form may represent other kinds of objects as well. Particularly in `paxter.author` subpackage, we can display the string ‘@’ through the command `@@`.

Suppose we wish to include an email address in a blog post. Here is an example of the source text:

```
Email me at @link["mailto:person@example.com"]{person@@example.com}
and my twitter handle is @@example. Don’t @@ me.
```

The above source text gets transformed into the following HTML output.

```
<p>Email me at <a href="mailto:person@example.com">person@example.com</a>
   and my twitter handle is @example. Don’t @ me.</p>
```

What would happen if we forgot to *double* the `@` symbol? Consider the following example source text.

```
Email me at @link["mailto:person@example.com"]{person@@example.com}
and my twitter handle is @example. Don’t @@ me.
```

Parsing the above source text would yield the following error. Essentially, the `@example` command at line 2 column 27 is an unknown command. (The stack trace may be long and scary. It is totally to skim over it.)

```
Traceback (most recent call last):
  File ".../paxter/src/paxter/evaluate/context.py", line 149, in transform_command
    phrase_value = phrase_eval(token.phrase, self.env)
  File ".../paxter/src/paxter/author/standards.py", line 31, in phrase_unsafe_eval
    return eval(phrase, env)
  File "<string>", line 1, in <module>
NameError: name 'example' is not defined

The above exception was the direct cause of the following exception:
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File ".../paxter/venv/bin/paxter", line 33, in <module>
    sys.exit(load_entry_point('paxter', 'console_scripts', 'paxter')())
  File ".../paxter/venv/lib/python3.8/site-packages/click/core.py", line 829, in __
↪call__
    return self.main(*args, **kwargs)
  File ".../paxter/venv/lib/python3.8/site-packages/click/core.py", line 782, in main
    rv = self.invoke(ctx)
  File ".../paxter/venv/lib/python3.8/site-packages/click/core.py", line 1259, in␣
↪invoke
    return _process_result(sub_ctx.command.invoke(sub_ctx))
  File ".../paxter/venv/lib/python3.8/site-packages/click/core.py", line 1066, in␣
↪invoke
    return ctx.invoke(self.callback, **ctx.params)
  File ".../paxter/venv/lib/python3.8/site-packages/click/core.py", line 610, in␣
↪invoke
    return callback(*args, **kwargs)
  File ".../paxter/src/paxter/__main__.py", line 99, in run_html
    document = run_document_paxter(src_text, env)
  File ".../paxter/src/paxter/author/preset.py", line 34, in run_document_paxter
    evaluate_context = EvaluateContext(src_text, env, parse_context.tree)
  File "<string>", line 6, in __init__
  File ".../paxter/src/paxter/evaluate/context.py", line 40, in __post_init__
    self.rendered = self.render()
  File ".../paxter/src/paxter/evaluate/context.py", line 43, in render
    return self.transform_fragment_list(self.tree)
  File ".../paxter/src/paxter/evaluate/context.py", line 120, in transform_fragment_
↪list
    result = [
  File ".../paxter/src/paxter/evaluate/context.py", line 120, in <listcomp>
    result = [
  File ".../paxter/src/paxter/evaluate/context.py", line 117, in <genexpr>
    self.transform_fragment(fragment)
  File ".../paxter/src/paxter/evaluate/context.py", line 73, in transform_fragment
    return self.transform_command(fragment)
  File ".../paxter/src/paxter/evaluate/context.py", line 153, in transform_command
    raise PaxterRenderError(
paxter.exceptions.PaxterRenderError: paxter command phrase evaluation error at line 2␣
↪col 27: 'example'
```

1.6.2 Escaping Delimiters: Curly Braces, Quotes, and Bars

Under Construction

This section is under construction.

1.7 Dive Into Command Syntax

Under Construction

This section is under construction.

1.8 Codeblock Syntax Highlight (Demo)

Under Construction

This section is under construction.

1.9 Under Construction

Under Construction

- Stages of Paxter language processing
- Customizing evaluation environment dictionary
- Customizing parsed tree evaluator
- Full language syntax tutorial

1.10 Core API Reference

Paxter language package provides the following core functionality.

1.10.1 Parsing

The following class is where the main Paxter language parsing logic happens.

Data Definitions

Results of the Paxter language parsing yields parsed trees which consist of instances of the following data classes.

Other Utility Classes

Other classes related to parsing, presented here for reference only.

1.10.2 Interpretation

The following class implements the basic tree evaluation in Paxter language. Users may want to extend this class to override the tree evaluation.

The evaluated list of fragments will be of the following type

Function decorators

Wrappers for functions in python environments to be used as function decorators.

1.10.3 Exceptions

These are all the exception classes raised from this library package.

class `paxter.exceptions.`**`PaxterBaseException`**(*message: str*, ***positions: CharLoc*)

Bases: `Exception`

Base exception specific to Paxter language ecosystem. Positional index parameters indicates a mapping from position name to its indexing inside the input text.

positions: `dict[str, CharLoc]`

A mapping from position name to `LineCol` position data

message: `str`

Error message

class `paxter.exceptions.`**`PaxterConfigError`**(*message: str*, ***positions: CharLoc*)

Bases: *`paxter.exceptions.PaxterBaseException`*

Exception for configuration error.

class `paxter.exceptions.`**`PaxterSyntaxError`**(*message: str*, ***positions: CharLoc*)

Bases: *`paxter.exceptions.PaxterBaseException`*

Exception for syntax error raised while syntax input text in Paxter language.

class `paxter.exceptions.`**`PaxterRenderError`**(*message: str*, ***positions: CharLoc*)

Bases: *`paxter.exceptions.PaxterBaseException`*

Exception for parsed tree transformation error.

1.11 Authoring API Reference

All of the following functions and classes are not part of the core Paxter language. They are provided only for convenience; it is entirely possible to utilize Paxter package without using any of the following functions. Users are encouraged to read source code of these functions to learn how to reassemble core APIs to suit their needs.

1.11.1 Preset Functions

The following function combines Paxter language parsing together with parsed tree evaluation.

1.11.2 Environment Creations

The following function creates a pre-defined unsafe Python environment dictionary to be used with the evaluation context class.

1.11.3 Evaluation Context Objects

The following instances are available in preset environments.

Object	Alias	Simple Environment	Document Environment
`"@"`	`"@@"`	Yes	Yes
`for_statement`	**for**	Yes	Yes
`if_statement`	**if**	Yes	Yes
`python_unsafe_exec`	**python**	Yes	Yes
`verbatim`	**verbatim**	Yes	Yes
`RawElement`	**raw**	-	Yes
`Paragraph`	**paragraph**	-	Yes
`Heading1`	**h1**	-	Yes
`Heading2`	**h2**	-	Yes
`Heading3`	**h3**	-	Yes
`Heading4`	**h4**	-	Yes
`Heading5`	**h5**	-	Yes
`Heading6`	**h6**	-	Yes
`Blockquote`	**blockquote**	-	Yes
`Bold`	**bold**	-	Yes
`Italic`	**italic**	-	Yes
`Underline`	**uline**	-	Yes
`Code`	**code**	-	Yes
`Link`	**link**	-	Yes
`Image`	**image**	-	Yes
`NumberedList`	**numbered_list**	-	Yes
`BulletedList`	**bulleted_list**	-	Yes
`Table`	**table**	-	Yes
`TableHeader`	**table_header**	-	Yes
`TableRow`	**table_row**	-	Yes

continues on next page

Table 1 – continued from previous page

Object	Alias	Simple Environment	Document Environment
`horizontal_rule`	**hrule**	-	Yes
`line_break`	**line_break** or `"@\"`	-	Yes
`non_breaking_space`	**nbsp** or `"@%"`	-	Yes
`hair_space`	**hairsp** or `"@."`	-	Yes
`thin_space`	**thinsp** or `"@,"`	-	Yes

Control Functions

Standard Functions

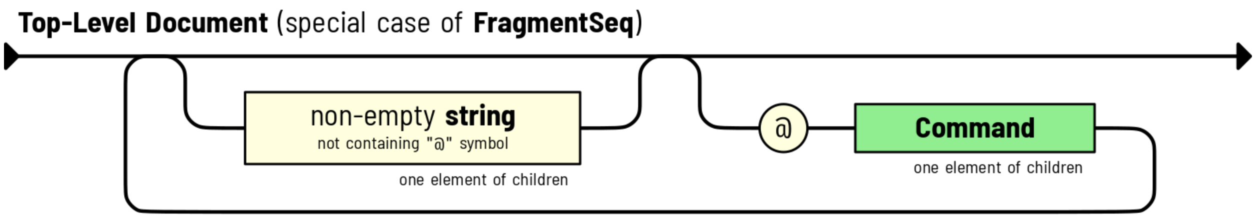
Element Data Classes

1.12 Syntax Reference

Below are syntax descriptions of Paxter language.

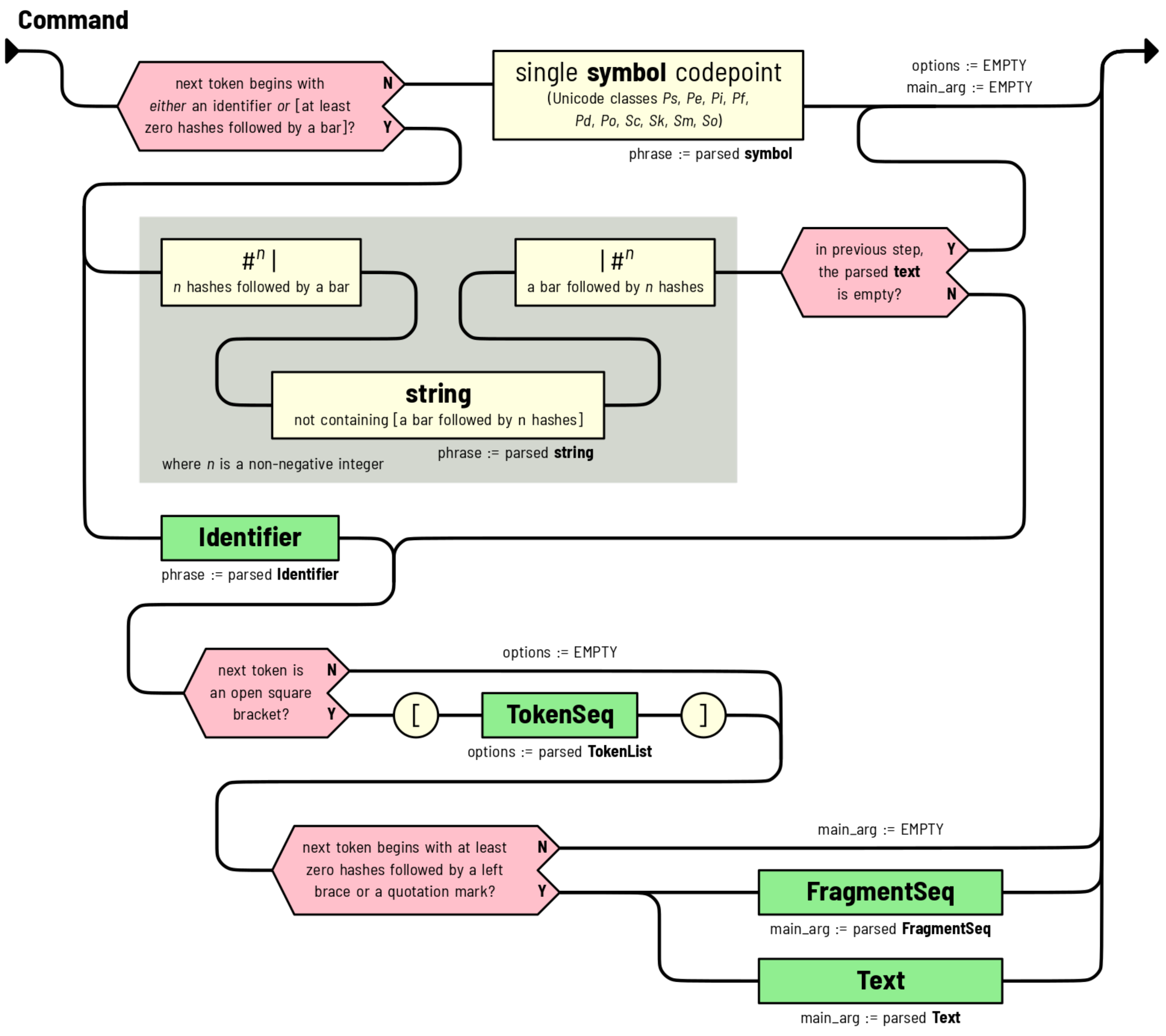
1.12.1 Document Rule

The **starting rule** of Paxter language grammar which is a special case of *FragmentSeq Rule*. The result of parsing this rule is always a `FragmentSeq` node whose children includes non-empty strings (as `Text` nodes), interleaving with the result produced by *Command Rule*.


Top-Level Document (special case of FragmentSeq)
non-empty string
not containing "@" symbol
one element of children
@
Command
one element of children

1.12.2 Command Rule

Rule for parsing the textual content after encountering the `@`-switch symbol character.



Command
next token begins with either an identifier or [at least zero hashes followed by a bar]?
N
Y
single symbol codepoint
(Unicode classes Ps, Pe, Pi, Pf, Pd, Po, Sc, Sk, Sm, So)
phrase := parsed symbol
options := EMPTY
main_arg := EMPTY
#n |
n hashes followed by a bar
| #n
a bar followed by n hashes
string
not containing [a bar followed by n hashes]
phrase := parsed string
where n is a non-negative integer
in previous step, the parsed text is empty?
Y
N
Identifier
phrase := parsed Identifier
next token is an open square bracket?
N
Y
options := EMPTY
[
TokenSeq
]
options := parsed TokenList
next token begins with at least zero hashes followed by a left brace or a quotation mark?
N
Y
main_arg := EMPTY
FragmentSeq
main_arg := parsed FragmentSeq
Text
main_arg := parsed Text

What are the red shapes?

Red shapes appeared in the above diagram indicates a branching path of parsing depending on conditions specified with the shapes.

There are a few possible scenarios.

1. The first token is an identifier. The parsed identifier becomes the phrase part of the `Command`. Then the parser would attempt to parse the options section and the main argument section if they exist.
2. The first token is n hash characters followed by a bar character. Then the parser will attempt to parse for the phrase part non-greedily until a bar character followed by n hash characters are found. If the parsing result of the phrase is not empty, then the parser would continue on trying to parse for the options section and the main argument section.

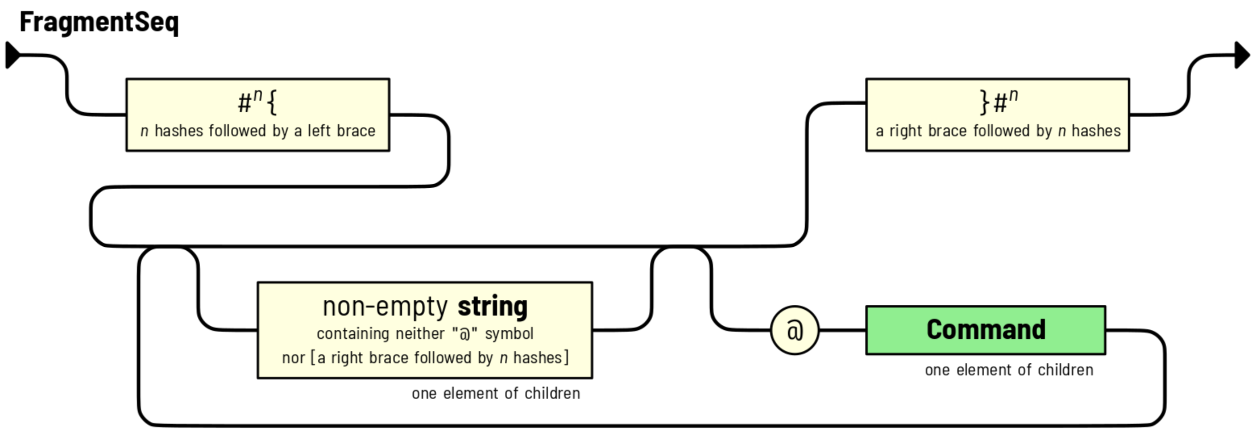
However, if the phrase is empty, then the options section as well as the main argument section are assumed to be empty.

3. The first token is a single symbol character. This would result in such symbol becoming the sole content of the phrase section, while other sections (i.e. options and main argument) are empty.

1.12.3 FragmentSeq Rule

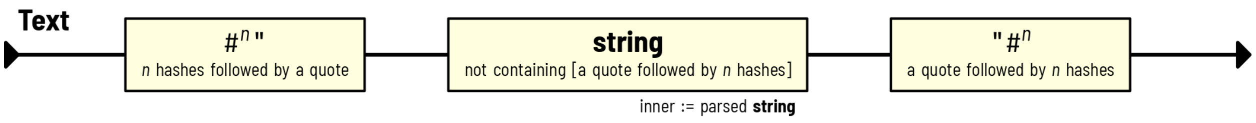
This rule always begins with n hash characters followed by a left brace and ends with a right brace followed by n hash characters, for some non-negative integer n. Between this pair of curly braces is an interleaving of strings (as `Text`) and `Command`, all of which are children of `FragmentSeq` instance.

One important point to note is that each string is parsed non-greedily; each resulting string would never contain a right brace followed by n or more hash characters.


FragmentSeq
#ⁿ{
n hashes followed by a left brace
}#ⁿ
a right brace followed by n hashes
non-empty string
containing neither "@" symbol
nor [a right brace followed by n hashes]
one element of children
@
Command
one element of children

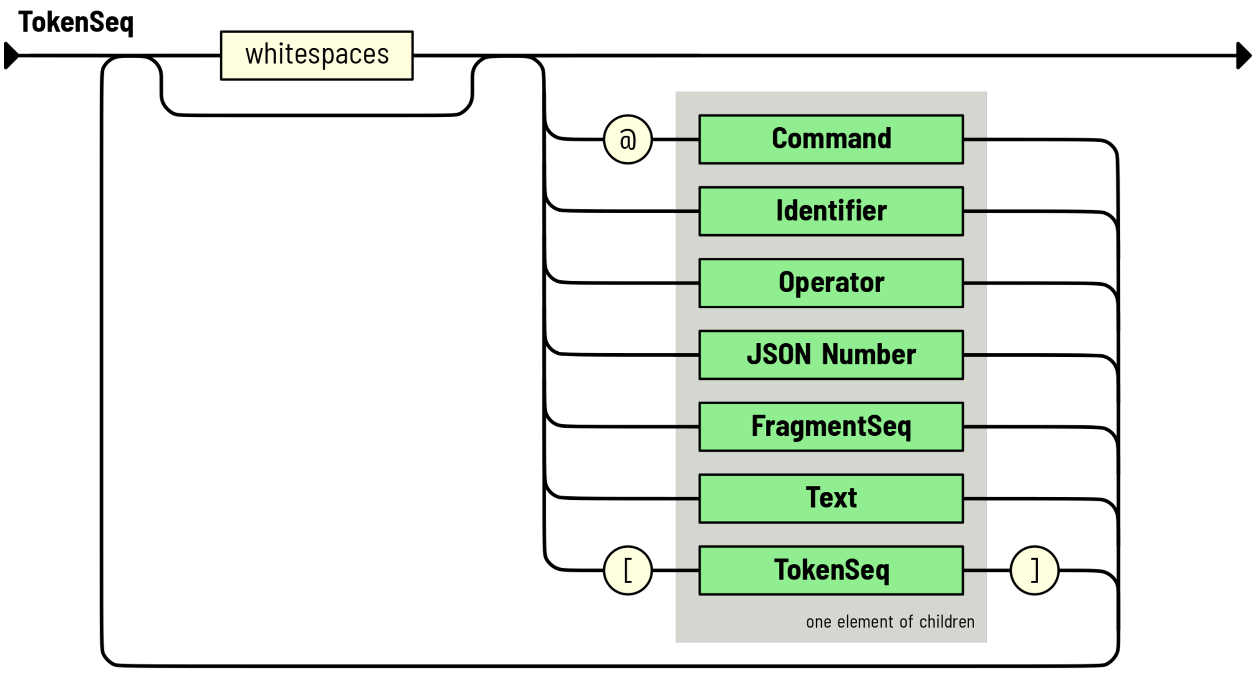
1.12.4 Text Rule

This rule is similar to *FragmentSeq Rule* except for two main reasons. The first reason is that nested `Command` will not be parsed (i.e. `"@"` is not a special character in this scope). Another reason is that, instead of having a matching pair of curly braces indicate the beginning and the ending of the rule, quotation marks are used instead.


Text
#ⁿ"
n hashes followed by a quote
string
not containing [a quote followed by n hashes]
inner := parsed string
"#ⁿ
a quote followed by n hashes

1.12.5 TokenSeq Rule

Following this parsing rule results in a sequence of zero or more tokens, possibly separated by whitespaces. Each token may be a `Command`, an `Identifier`, an `Operator`, a `Number`, a `FragmentSeq`, a `Text`, or a nested `TokenSeq`. This resulting sequence of tokens are children of `TokenSeq` node type.

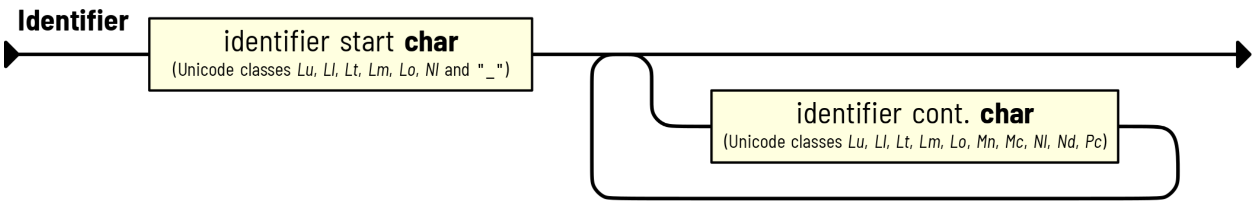


Good To Know

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

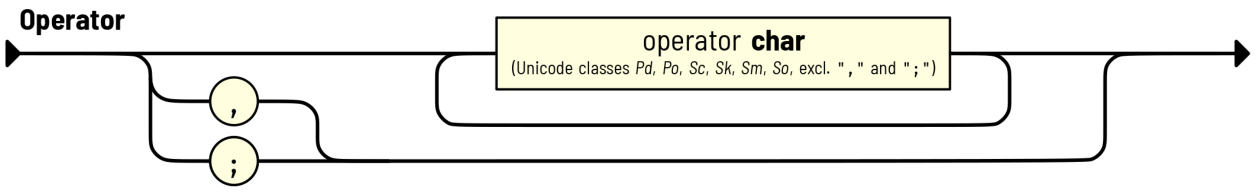
1.12.6 Identifier Rule

This rule generally follows python rules for greedily parsing an identifier token (with some extreme 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)

1.12.7 Operator Rule

Greedily consumes as many operator characters as possible (with two notable exceptions: a common and a semicolon, each of which has to appear on its own). Whitespace characters may be needed to separate two consecutive, multi-character operator tokens. The result is an `Operator` node type.



1.13 Getting Started

Todo: This page is due for removal.

1.13.1 Installation

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

```
$ pip install paxter
```

1.13.2 Programmatic Usage

This package is *mainly* intended to be utilized as a library. To get started, let's assume that we have a document source text written using **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** (implemented by the class `ParseContext`) to transform the source input into an intermediate parsed tree.

```
from paxter.core import ParseContext

parsed_tree = ParseContext(source_text).tree
```

Note: We can see the structure of the parsed tree in full by printing out its content as shown below (output reformatted for clarify).

```
>>> parsed_tree
FragmentList(
    start_pos=0,
    end_pos=236,
```

(continues on next page)

(continued from previous page)

```
    children=[
        Command(
            start_pos=1,
            end_pos=148,
            starter="python",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=Text(
                start_pos=10,
                end_pos=145,
                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',
                enclosing=EnclosingPattern(left='##"', right='"##'),
            ),
        ),
        Text(
            start_pos=148,
            end_pos=161,
            inner="\\\nMy name is ",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        Command(
            start_pos=162,
            end_pos=166,
            starter="name",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=None,
        ),
        Text(
            start_pos=166,
            end_pos=189,
            inner=" and my current age is ",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        Command(
            start_pos=190,
            end_pos=201,
            starter="current_age",
            starter_enclosing=EnclosingPattern(left="", right=""),
            option=None,
            main_arg=None,
        ),
        Text(
            start_pos=201,
            end_pos=223,
            inner=".\nMy shop opens Monday",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        SymbolCommand(start_pos=224, end_pos=225, symbol=","),
        Text(
            start_pos=225,
            end_pos=226,
            inner="-",
            enclosing=EnclosingPattern(left="", right=""),
        ),
        SymbolCommand(start_pos=227, end_pos=228, symbol=","),
```

(continues on next page)

(continued from previous page)

```
        Text(
            start_pos=228,
            end_pos=236,
            inner="Friday.\n",
            enclosing=EnclosingPattern(left="", right=""),
        ),
    ],
    enclosing=GlobalEnclosingPattern(),
)
```

Notice how the source text above also contains what seems like a Python code. This has *nothing* to do with Paxter core grammar in any way; it simply uses the Paxter *command* syntax to *embed* Python code to which we will give a meaningful interpretation later.

Rendering

Next step, we use a built-in **renderer** to transform the intermediate parsed tree into its final output. It is important to remember that **the semantics of the documents depends on which renderer we are choosing**.

We will adopt the **Python authoring mode** whose renderer (implemented by `EvaluateContext`) is already predefined by the Paxter library package to transform the parsed tree into the desired final form. One of its very useful features is that it will execute python code under the `@python` command.

```python
from paxter.author import create_simple_env
from paxter.interpret import EvaluateContext

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

result = EvaluateContext(source_text, env, parsed_tree).rendered
print(result)  # 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 own function

We recommend Paxter library users to by themselves write a utility function to connect all of the toolchains provided Paxter package. This is the minimal example of a function to get you started.

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

def interp(source_text: str) -> str:
    parsed_tree = ParseContext(source_text).tree
```

(continues on next page)

(continued from previous page)

```
    result = RenderContext(source_text, create_unsafe_env(), tree).rendered
    return result
```

1.13.3 Command-Line Usage

As a shortcut, Paxter library package also provided some utilities via command-line program. To get started, red the help message using the following command:

```
$ paxter --help
```

To play around with the parser, you may use `parse` subcommand with an input. Suppose that we have the following input file.

```
$ cat intro.paxter
@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 see the intermediate parsed tree using this command:

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

If we wish to also render the document written in Paxter language under the Python authoring mode with the default environment, then use the following command:

```
$ paxter pyauthor -i intro.paxter -o result.txt
$ cat result.txt
My name is Ashley and my current age is 33.
My shop opens Monday&thinsp;-&thinsp;Friday.
```

However, this command-line option does *not* provide a lot of flexibility. So we recommend users to dig deeper with a more programmatic usage. It may require a lot of time and effort to setup the entire toolchain, but it will definitely pay off in the long run.

1.14 Paxter Language Tutorial

Todo: This page requires revision.

Note: This is a tutotrial for *bare* Paxter language specification. It discusses only the basic Paxter syntax without any associated semantics as the semantics to the intermediate parsed tree is generally given by users of Paxter library.

For a simpler usage of Paxter library package, please also see *Python authoring mode tutorial page*.

Paxter syntax is very simple. In most cases, a typical text is a valid Paxter document, like in the following:

```
Hello, World!
My name is Ashley, and I am 33 years old.
```

However, Paxter provides a special syntax called **@-expressions** (pronounced "at expressions") so that richer information may be inserted into the document. There are 2 kinds of @-expressions, all of which begins with an @-symbol: a command and a short symbol expression.

This @-symbol (codepoint U+0040) is sometimes called a *switch* because it indicates the beginning of an @-expression, and whatever follows the switch determines which kind of @-expression it is.

Next, we dive into each kind of @-expressions.

Note: Consult *Syntax Reference* for a more detailed Paxter language grammar specification.

1.14.1 1. Command

A **command** is the most powerful syntax in Paxter language. It consists of the following 3 sections of information:

```
"@" starter [option] [main_argument]
```

Among these 3 sections, only the starter section is mandatory; the other 2 sections are optional and can be omitted. Additionally, there should *not* be any whitespace characters separating between the switch and the starter section, nor between different sections of the same command.

Starter section

A starter of a command may contain any textual content, surrounded by a pair of bars `|` (U+007C).

Here are examples of a valid command with only the starter section.

```
@|foo|
@|_create|
@||
@|foo.bar|
@|1 + 1|
@|Hello, World!|
```

However, if the content of the starter section takes the form of a valid Python identifier, then the pair of bars may be dropped. So the first 3 examples from above may be rewritten as follows:

```
@foo
@_create
@
```

On the other hand, the textual content of the starter may sometimes contain a bar as part of itself (such as `x || y || z`). Then we may additionally surround the matching pair of bars with an equal number of hashes `#` (U+0023):

```
@#|x || y || z|#
@###|x || y || z|###
```

But the following example will *not* work as expected:

```
@|x || y || z| is a command whose starter content contains exactly just “x ”
followed by regular text “| y || z|”.
```

Obviously, if the starter section begins with *n* hashes followed by a bar, then the textual content itself *cannot* contain a bar followed by *n* or more hashes (otherwise, the starter section would have terminated earlier).

```
@##|good|#|one|##
@##|bad|##|one|##
```

In this example (shown above), the starter of the first command is `good|#|boy` whereas that of the other command cuts short at `bad` (followed by the text `|one|##`).

Note: In a sense, this *bar pattern* (by which we mean the pattern of surrounding some content with a pair of bars plus an equal number of hashes on both ends) will be parsed **non-greedily** (i.e. the parsing of the starter halts as soon as the closing pattern corresponding to the opening pattern encountered earlier is found).

Main argument section

Let’s skip the option section for now and discuss the main argument section of a command first.

As the name suggests, the main argument section of a command contains the most important piece of information to which the command is applied. The main argument can be supplied in one of 2 modes: the fragment list mode (in which the content is wrapped within the *brace pattern*) and the text mode (i.e. the content is wrapped within the *quoted pattern*).

(a) Wrapped fragment list mode

For a fragment list mode as the main argument, the content may contain texts as well as any *nested* `@`-expressions.

The content itself must be surrounded by a pair of curly braces (U+007B and U+007D) called the *brace pattern* (in analogous to the *bar pattern* associated with the starter section of a command). Of course, additionally appending the equal number of hashes to both ends are allowed.

For example,

```
@foo{Hello, @name}
@|font.large|{BUY ONE GET ONE FREE!}
@highlight##{A set of natural numbers: {0, 1, 2, 3, ...}.}##.
```

Similarly to the *bar pattern* from the starter section of a command, if the wrapped fragment list begins with *n* hashes followed by a left curly brace, then the **immediate** inner textual content may *not* contain a right curly brace followed by *n* or more hashes.

In the following example, the outermost command has the starter `foo` and its main argument is in fact `@bar{1###}###`. That is because (1) the curly braces pair surrounding `1###` (marked with “^”) match with each other, and thus (2) the succeeding 3 hashes are not associated with the marked closing curly brace.

```
@foo###{@bar{1###}###}###
           ^    ^
```

(b) Wrapped text mode

Wrapped texts are somewhat similar to wrapped fragment lists, except for 2 major aspects:

- Instead of using a matching pair of curly braces surrounding the inner content, wrapped texts use a pair of quotation marks (U+0022). This is called the *quoted pattern* in analogous to the *brace pattern* for wrapped fragment lists.
- All @-symbol characters within the textual content will *not* be interpreted as the switch for @-expressions. Hence, wrapped texts would *not* contain any nested @-expressions.

This mode of main argument is useful especially when we expect the inner content of the main argument to be from **another domain** where @-symbols are prevalent.

For example, when you want to embed source code from another language:

```
@python_highlight##"

    # Results of the following function is cached
    # depending on its input
    from functools import lru_cache

    @lru_cache(maxsize=None)
    def add(x, y):
        """Adding function with caching."""
        return x + y

"##
```

Again, if the inner content needs to contain a quotation mark, we may add an equal number of hashes to both ends:

```
@alert#"Submit your feedback to "ashley@example.com"."#
```

Option section

The existence of a left square bracket immediately after the starter section of a command *always* indicates the beginning of the option section. The option section itself is a sequence of *tokens* where each token can be one of the following:

- **Another @-expression** of any kind
- **An identifier** (according to Python grammar)
- **An operator** which can be a single comma, a single semicolon, or a combination of all *other* symbol characters (excluding hashes, quotation marks, curly braces, and square brackets)
- **A number** whose syntactical form adheres to JSON grammar for number literal
- **A fragment list** wrapped within the *brace pattern* (which shares the same syntax as already discussed in the main argument section)
- **A text** wrapped within the *quoted pattern* (which shares the same syntax as already discussed in the main argument section)
- **A nested sequence of tokens** itself, surrounded by a matching pair of square brackets (U+005B and U+005D).

Warning: Please note that inside the option section of a command is the only place in Paxter language where whitespace characters between tokens are ignored.

Here are a couple of examples of commands which include the option section:

- For the command `@foo[x="bar", y=2.5, z={me}]{text}`, its option section contains a sequence of 11 tokens:
    1. an identifier `x`
    2. an equal sign operator =
    3. a text token `bar`
    4. a comma operator `,`
    5. an identifier `y`
    6. an equal sign operator =
    7. the number literal `2.5`
    8. a comma operator `,`
    9. an identifier `z`
    10. an equal sign operator =, and
    11. a fragment list containing the text `me`
- For the command `@|foo.bar|[x <- [2]; @baz]`, its option section contains a sequence of 5 tokens:
    1. an identifier `x`
    2. a left arrow operator `<-`
    3. a nested sequence containing the number literal `2` as the only token within it
    4. a semicolon operator `;`, and
    5. a nested command with `baz` as the starter section and with all other sections omitted.

Paxter language syntax gives a lot of freedom for what is allowed within the option section of a command; a programmer-write who writes a renderer to transform Paxter intermediate parsed trees into data of another form has a liberty to add whatever constraints to the syntactical structure within the option section.

1.14.2 2. Single Symbol Expression

This kind of @-expression is in the form of a single symbol character immediately following the @-symbol switch. This single *symbol* character will be the sole content of the single symbol expression.

For example,

```
There is free food today between 3@,-@,5 PM.
```

Warning: If `@#` happens to be the prefix of a full-form @-expressions (such as in `@#|foo|#`), then `@#` by itself is *not* a valid command in special form. It must be **unambiguously** *not* part of full-form command for itself to become a valid command of special form.

1.14.3 Escaping @-Symbol Switches

Paxter language does *not* provide any syntax to escape @-symbol switches of @-expressions. We recommend the library user solve this kind of problem at the interpreter/renderer level instead.

One way to do this is to define the behavior of `@@` (a single symbol expression with `@` symbol following the switch) to be transformed into a single `@` symbol in the rendered output.

```
My email is ashley@@example.com.
```

Another method to work around this problem is to introduce a command called `verbatim` (inspired by the command of the same name in LaTeX) which will output the main input argument as-is.

```
My email is @verbatim"ashley@example.com".
```

1.15 Python Authoring Mode Tutorial

Todo: This page requires revision.

1.15.1 Block Python Code Execution

In Python authoring mode, Python source code may be embedded into the document for execution using `python` command syntax with the code as the main argument. For example,

```
@python##"
    name = "Ashley"
"##
```

In the example document above, once the Python code in the preamble is executed, the value of the variable `name` will be available in the environment for the rest of the document.

Referring to variable from Python code

One way to referring to the value of the variable `name` is to use the command syntax `@name` without any options or main arguments sections. So the following document

```
@python##"
    name = "Ashley"
"##
Hi, @name.
```

will be rendered into

```

Hi, Ashley.
```

Remove unwanted newlines

Notice how the newline character was preserved in the above output. If we wish to remove that newline character, we may put a backslash at the end of that line. So the following document

```
@python##"
    name = "Ashley"
"##\
Hi, @name.
```

yields the following output in Python authoring mode

```
Hi, Ashley.
```

Referring to functions from Python code

We may also define Python functions within the embedded Python source code and refer to them later in the document. The syntax to make a call to a function already defined is a command syntax with the main argument supplied. Here is one example,

```
@python##"
    def surround(text):
        return "(" + flatten(text) + ")"
"##\
This is @surround{sound}.
```

which will return

```
This is (sound).
```

The reason why we need to `flatten` the main argument first is that the fragment list (i.e. the part surrounded by a matching pair of curly braces) returns a list of string tokens (not the string itself), hence it is important to flatten them into a single string first (otherwise an error would have occurred).

Python functions with multiple arguments

When there is more than one argument to the function, the main argument of the command will always be the first argument of the function, and the rest of the function arguments can be supplied to option section of the command (similarly to Python function call syntax):

```
@python##"
    def surround(text, n, left='(', right=')'):
        return flatten(left) * n + flatten(text) + flatten(right) * n
"##\
This is @surround[3]{sound}.
This is @surround[n=3]{sound}.
This is @surround[3, "[", "]"]{sound}.
This is @surround[3, right=""]{sound}.
This is @surround[n=3, left="_", right="_"]{sound}.
```

Here is the result.

```
This is (((sound))).
This is (((sound))).
```

(continues on next page)

(continued from previous page)

```
This is [[[sound]]].
This is (((sound.
This is ___sound___.
```

Notice that we use wrapped text inside the option section in order to supply strings as arguments to the function `surround`.

Additionally, we may also omit the main argument section, and then the entire option section will all be the arguments to the function:

```
@python##"
    def surround(text, n, left='(', right=')'):
        return flatten(left) * n + flatten(text) + flatten(right) * n
"##\
This is @surround["sound",3].
This is @surround["sound",n=3].
```

The above document will be rendered into

```
This is (((sound))).
This is (((sound))).
```

1.15.2 Inline Python Code Evaluation

We may wish to insert the result of the evaluation of Python expression. We can do so by using the command syntax with the bar pattern `@|...|`:

```
The result of 7 × 11 × 13 is @|7 * 11 * 13|.
```

and that would be transformed into

```
The result of 7 × 11 × 13 is 1001.
```

Inline Python code with function call

If a function behind an attribute or key lookup, we may use the bar pattern in conjunction with main arguments and/or options.

```
@python##"
    import statistics
    values = [2, 3, 5, 7]
    funcs = {
        'median': statistics.median
    }
"##\
The average of first 4 primes is @|statistics.mean|[@values].
The median of first 4 primes is @|funcs['median']|[@values].
```

The above document returns the following.

```
The average of first 4 primes is 4.25.
The median of first 4 primes is 4.0.
```

1.15.3 Special Symbol Commands

For the sake of simplicity, we provide an easy way to perform text replacements for symbol-style commands. Simply define a dictionary mapping from each symbol to the substituting results under the variable `_symbol_` inside the Python source code.

```
@python##"
    _symbols_ = {
        '.': '&hairsp;',
        ',': '&thinsp;',
        '@': '@',
    }
"##\
My email is ashley@@example.com.
My office hours is between 7@.-@.9 PM.
```

Here is the result of the above document.

```
My email is ashley@example.com.
My office hours is between 7&hairsp;-&hairsp;9 PM.
```

1.15.4 Special Commands: For and If

For statements within the document for Python authoring mode has the following format

```
@for[<IDENTIFIER> in <EXPRESSION>]{<BODY>}
```

whereas if statements has the 3 following formats

```
@if[<CONDITIONAL>]{<BODY>}
@if[not <CONDITIONAL>]{<BODY>}
@if[<CONDITIONAL> then <THEN_BODY> else <ELSE_BODY>]
```

Here is the document that illustrates how to use these special commands:

```
@python##"
    def is_odd(value):
        return value % 2 == 1
"##\
Odd digits are @flatten{@for[i in @|range(10)|]{@if[@|is_odd(i)|]{ @i}}}.
Even digits are @flatten{@for[i in @|range(10)|]{@if[not @|is_odd(i)|]{ @i}}}.
Digits are @flatten{@for[i in @|range(10)|]{@if[@|is_odd(i)| then " odd" else " even
↪"]}} in this order.
```

and the result would be

```
Odd digits are 1 3 5 7 9.
Even digits are 0 2 4 6 8.
Digits are  even odd even odd even odd even odd even odd in this order.
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

INDEX